From SPRING to SUMMER

Design, Definition and Implementation of Programming Languages for String Manipulation and Pattern Matching

PROEFSCHRIFT

ter verkrijging van de graad van doctor in de technische wetenschappen aan de Technische Hogeschool Eindhoven, op gezag van de rector magnificus, prof.ir.J. Erkelens, voor een commissie aangewezen door het college van dekanen in het openbaar te verdedigen op dinsdag 30 maart 1982 te 16.00 uur

door

Paul Klint

geboren te 's Gravenhage

Dit proefschrift is goedgekeurd
door de promotoren

Prof.dr. F.E.J. Kruseman Aretz

en

Prof. H. Whitfield, b.s., d.i.c.

Printed at Mathematical Centre, Amsterdam.

CONTENTS

SUMMARY

Written text is an essential element in our culture and various technical means have been invented to aid in its production. Paper and pencil, the typewriter and the typesetter are examples of such inventions.

Continuing this same line of development, computers are nowadays being used to alleviate the writing task. Computerized text processing systems (ranging from word processors for writing and editing simple texts to fully automated newspaper and book printing systems) are rapidly penetrating into all areas of human activity where written text is the primary means of communication.

Historically, the impetus behind the development of computers has always been primarily numerical in nature. This is reflected in the design of most computers and programming languages. However, the increasing use of computers for text processing and other non-numeric tasks makes the purely arithmetic design obsolete.

This thesis concentrates on the programming language aspects of computerized text handling and, to be more precise, on the design and implementation of **string processing languages**. The term 'string processing' refers to the process of inspecting, modifying and transforming texts, i.e. sequences of symbols. It comprises such seemingly disparate activities as text editing, transforming a text with embedded formatting directives into a final layout, and compiling a source program into a string of machine instructions.

A more or less chronological account is given of attempts to solve some of the problems in string processing languages. First of all, two exercises in designing application oriented programming languages are described. This has resulted in the languages SPRING and SUMMER. The lessons learned from the design and use of SPRING have been incorporated in SUMMER. Next, an exercise in the formal definition of the semantics of programming languages is described. The definition and implementation of SUMMER together constitute the final result of the project.

This thesis consists of two parts. Part I traces the historical development in detail and consists of chapters 1 through 7. Part II is devoted to the definition of SUMMER and consists of chapters 8 through 12. The contents of the thesis are now briefly summarized.

Chapter 1 is introductory and gives the necessary motivation and background for the study of string processing languages.

Chapter 2 sketches the language SPRING, a first attempt to design a string processing language. SPRING may be characterized as a **big** language, i.e. it provides a large number of language primitives for solving problems in its envisaged application areas. Attention is drawn to undesirable language features resulting from seemingly logical design choices. Many problems and questions discussed in Chapter 1 were identified as such during this effort.

Chapter 3 is devoted to general design considerations for string processing languages and compares the semantics of various pattern matching models. Attention is paid to different forms of side-effects during a pattern match. This is done by giving an operational, formal definition of the semantics of the various models. As a result of this, a new pattern matching model based on side-effect recovery is developed.

Chapter 4 gives an overview of the language SUMMER, a second attempt to design a string processing language. SUMMER may be characterized as a **small** language, i.e. it consists of a relatively small set of primitive operations together with a modest extension mechanism.

Chapter 5 concentrates on the problem of finding a method for formal language definition that is suitable for the designers as well as the implementors and users of a language. An improved method for the operational definition of programming language semantics is developed and the result of applying this method to SUMMER is illustrated.

Implementation issues are discussed in Chapter 6. The SUMMER compiler and run-time system are described in some detail.

Chapter 7 concludes the first part of this thesis with an evaluation of the research described in it and suggestions for further research.

Part II is devoted to the definition of the SUMMER programming language. It provides both a formal and informal language definition and tutorial examples.

In Chapter 8 the techniques and notational conventions that are used in the definition are introduced. Much attention is paid to the method used for the formal definition of the semantics of SUMMER.

Chapter 9 contains a semi-formal definition of the SUMMER kernel. This is a small subset of the language on which a semantic definition of the whole language can be based. The description of each language feature consists of its syntax, an informal as well as a formal definition of its semantics, and examples.

In Chapter 10 the kernel is extended with useful data types and associated operations, such as reals, arrays, tables, files, bit strings, etc.

Some complete, annotated SUMMER programs are presented in Chapter 11.

Finally, a summary of the syntax is given in Chapter 12.

Readers who are only interested in getting a general impression of the language SUMMER may confine themselves to Chapter 4 and the annotated examples in Chapter 11. Readers who are not interested in the formal definition of the language may skip Chapter 8 (except Sections 8.1 and 8.2), and all subsections of Chapter 9 entitled 'Semantics'.

SAMENVATTING

Geschreven tekst vormt een essentieel element in onze cultuur en het wekt dan ook geen verbazing dat verschillende technische hulpmiddelen uitgevonden zijn om het produceren van geschreven tekst te vereenvoudigen. Potlood en papier, de schrijfmachine en de zetmachine zijn voorbeelden van dergelijke uitvindingen.

Als voortzetting van deze lijn van ontwikkeling worden computers tegenwoordig gebruikt om het produceren van geschreven tekst te vereenvoudigen. Geautomatiseerde tekstverwerkende systemen (van 'word processors' voor het schrijven en redigeren van eenvoudige teksten tot volledig geautomatiseerde systemen voor het drukken van kranten en boeken) dringen momenteel door in allerlei gebieden waar het geschreven woord het voornaamste communicatiemiddel is.

Historisch gezien is de ontwikkeling van computers altijd in hoge mate bepaald door de behoefte om veel en snel te kunnen rekenen. Dit heeft zijn weerslag gevonden in het ontwerp van de meeste computers en programmeertalen. Door het toenemend gebruik van computers voor tekstverwerking en voor de oplossing van andere, niet numerieke, problemen raken de oorspronkelijke, hoofdzakelijk op rekenen gerichte ontwerpen verouderd.

Dit proefschrift is gewijd aan de programmeertaalaspecten van geautomatiseerde tekstverwerking en in het bijzonder aan het ontwerp en de implementatie van 'stringmanipulatietalen'. Dit zijn programmeertalen die gebruikt kunnen worden bij het bouwen van tekstverwerkende systemen. Onder stringmanipulatie wordt hier verstaan het inspecteren of wijzigen van rijen 'symbolen'. In het geval van tekstverwerking zal men als symbolen kiezen de letters, cijfers en leestekens waaruit een te behandelen tekst bestaat. Men kan ook andere basissymbolen kiezen om anderssoortige problemen op te lossen.

Een min of meer chronologisch overzicht wordt gegeven van pogingen om enkele problemen die zich in bestaande stringmanipulatietalen voordoen op te lossen. Het beschreven onderzoek omvat allereerst twee exercities op het gebied van het ontwerp van toepassingsgerichte programmeertalen. Dit heeft geleid tot het ontwerp van de talen SPRING en SUMMER. De lessen die geleerd zijn bij het ontwerp en het gebruik van SPRING zijn verwerkt in het ontwerp van SUMMER. Het beschreven onderzoek omvat verder een exercitie op het gebied van het formeel definiëren van de betekenis ('semantiek') van programmeertalen. Definitie en implementatie van de programmeertaal SUMMER vormen tenslotte het feitelijke eindproduct van dit onderzoek.

Dit proefschrift bestaat uit twee delen. Deel I volgt de ontwikkeling van het onderzoek op de voet en bestaat uit de hoofdstukken 1 t/m 7. Deel II vormt het eindprodukt van het onderzoek en bestaat uit hoofdstukken 8 t/m 12. De inhoud wordt hieronder kort samengevat.

Hoofdstuk 1 is een inleiding op het onderwerp en geeft de noodzakelijke motivering en achtergrond voor de studie van stringmanipulatietalen.

Hoofdstuk 2 schetst de programmeertaal SPRING, een eerste poging tot het ontwerpen van een stringmanipulatietaal. SPRING is een nogal omvangrijke programmeertaal die een groot aantal ingebouwde operaties bevat om problemen op het gebied van tekstverwerking op te lossen. In dit hoofdstuk wordt gewezen op een aantal ongewenste eigenschappen van deze taal die voortkomen uit ogenschijnlijk logische

ontwerpkeuzen. Veel van de problemen en vragen die in het eerste hoofdstuk aan de orde komen werden tijdens dit onderzoek als zodanig onderkend.

Hoofdstuk 3 is gewijd aan algemene ontwerpoverwegingen voor stringmanipulatietalen en aan een vergelijking van de werking van 'patroonherkennings'-modellen. Patroonherkenning is een methode die dient om vast te stellen of een tekst bepaalde eigenschappen heeft, zoals 'is korter dan 83 tekens', of 'bevat het woord "heks"'. Bij deze vergelijking wordt aandacht besteed aan verschillende vormen van neveneffecten die kunnen optreden tijdens een patroonherkenningsoperatie. Als resultaat van deze analyse wordt een nieuw patroonherkenningsmodel gepresenteerd dat een elegante besturing van het al dan niet ongedaan maken van neveneffecten mogelijk maakt.

In hoofdstuk 4 wordt een overzicht gegeven van de programmeertaal SUMMER, een tweede poging tot het ontwerpen van een stringmanipulatietaal. SUMMER is een vrij 'kleine' programmeertaal die bestaat uit een relatief kleine kern van primitieven voor tekstverwerking, naast een uitbreidingsmechanisme om toepassingsgerichte operaties te definiëren.

In hoofdstuk 5 staat de vraag centraal hoe de semantiek van een programmeertaal op dusdanige wijze formeel beschreven kan worden dat zowel de ontwerpers als de implementatoren en gebruikers van een taal, met succes van een dergelijke formele beschrijving gebruik kunnen maken. In dit hoofdstuk wordt een verbeterde methode voor de operationele definitie van de semantiek van programmeertalen ontwikkeld en wordt de toepassing daarvan bij het definiëren van SUMMER geïllustreerd.

In hoofdstuk 6 wordt de implementatie van SUMMER beschreven.

Hoofdstuk 7 besluit het eerste deel van dit proefschrift door de resultaten van het onderzoek samen te vatten en door enkele richtingen voor voortgezet onderzoek aan te geven.

Deel II is gewijd aan de definitie van de programmeertaal SUMMER. Het bevat zowel een informele als een formele definitie van de taal en geeft enkele uitgewerkte voorbeelden.

In hoofdstuk 8 worden de techniek en de notatie uiteengezet die in de definitie gebruikt worden. De feitelijke definitie-methode krijgt hierbij veel aandacht.

Hoofdstuk 9 bevat een semi-formele definitie van een 'kern' van SUMMER. Deze kern is een klein deel van de taal dat voldoende is om de rest van SUMMER in te beschrijven. De definitie van iedere taalconstructie bestaat uit een beschrijving van zijn vorm, een informele en formele definitie van zijn betekenis, en voorbeelden.

In hoofdstuk 10 wordt de kern van SUMMER uitgebreid met een aantal nuttige datatypen met bijbehorende operaties, zoals reële getallen, arrays, associatieve geheugens, databestanden, enzovoorts.

Een aantal volledige, geannoteerde, SUMMER programma's wordt in hoofdstuk 11 gepresenteerd.

Hoofdstuk 12 bevat tenslotte een overzicht van de syntax van SUMMER.

Lezers die alleen een globale indruk van de taal SUMMER willen krijgen kunnen zich beperken tot hoofdstuk 4. Lezers die niet geïnteresseerd zijn in de formele definitie kunnen hoofdstuk 8 overslaan (met uitzondering van de paragrafen 8.1 en 8.2), evenals alle paragrafen van hoofdstuk 9 met als titel 'semantics'.

ACKNOWLEDGEMENTS

The research reported in this thesis was conducted while the author was employed at the Mathematical Centre in Amsterdam.

Several people contributed to this effort.

Design and implementation of both SPRING and SUMMER were done in close cooperation with Marleen Sint. Contributions to the design of SUMMER were made by Jan Heering. Their enthusiasm, patience and friendship were essential to the success of these projects.

Jan Heering, Marleen Sint and Arthur Veen have read drafts of this thesis. They pointed out various errors and made numerous suggestions for improving the style and presentation of it. I am grateful for their support and criticism.

Comments made by Leo Geurts, R.J. Lunbeck, Lambert Meertens and W.L. van der Poel are gratefully acknowledged.

CURRICULUM VITAE

Naam:	Klint, Paul.
Geboren:	8 September 1948 , te 's Gravenhage.
1967:	Diploma gymnasium β, Vossius Gymnasium, Amsterdam.
1970:	Kandidaatsexamen Natuurkunde, Universiteit van Amsterdam.
1973:	Doctoraalexamen Wiskunde, Universiteit van Amsterdam,
1973-heden:	Wetenschappelijk Medewerker, Afdeling Informatica, Mathematisch Centrum, Amsterdam.

Current address of the author:

Mathematisch Centrum
Kruislaan 413
1098 SJ Amsterdam

PART I

From SPRING to SUMMER

1. INTRODUCTION

1.1. Subject of this thesis

Written text is an essential element in our culture and therefore various technical means have been invented to aid in its production. Paper and pencil, the typewriter and the typesetter are examples of such inventions.

Continuing this same line of development, computers are nowadays being used to alleviate the writing task. Computerized text processing systems (ranging from word processors for writing and editing simple texts to fully automated newspaper and book printing systems) are rapidly penetrating into all areas of human activity where written text is the primary means of communication.

Historically, the impetus behind the development of computers has always been mostly numerical in nature. This is reflected in the design of most computers and programming languages. However, the increasing use of computers for text processing and for other non-numeric tasks makes the purely arithmetic design obsolete.

This thesis concentrates on the programming language aspects of computerized text handling and, to be more precise, on the design and implementation of **string processing languages**. The term 'string processing' refers to the process of inspecting, modifying and transforming texts, i.e. sequences of symbols. It comprises such seemingly disparate activities as text editing, transforming a text with embedded formatting directives into a final layout, and compiling a source program into a string of machine instructions.

In motivating the study of string processing languages we shall first consider three typical applications for which a string processing language would be a prime choice as implementation language. At the same time, we shall try to fit the problems and language requirements that are typical for string processing applications into a general scheme. It is not our intention to contend that the solutions proposed and the techniques used are the only ways to solve these problems. There are indeed many programs that solve them without relying on higher level concepts in their implementation language. In such programs the method of procedural extension is used to realize higher level concepts. What we do contend, however, is that the concepts proposed here follow in a natural way from the various applications.

Typical application 1: count the frequency of occurrence of all words in a text and print an alphabetically sorted list of the results. This is a prototype of many simple editing and text processing problems. A program to perform this task will presumably consist of the modules: **Read word**, **Tally** and **Sort and Print**.

Read word isolates the next 'word' from the input and fails if no more words are available. This requires a simple lexical recognition capability to distinguish letters, digits and punctuation marks. **Tally** compares the word just read with the words in a table containing all previously read words. If the word occurred before, its frequency is incremented in the table, otherwise a new table entry is created with frequency set to one. This requires table lookup and automatic storage allocation. Note that neither the maximum length of a word nor the maximum number of different words is known in advance. **Sort and Print** sorts the table and prints it. This requires a sorting facility and simple string synthesis functions to produce output in tabular form.

Typical application 2: format a text containing embedded formatting directives. A text formatting program might contain the modules: **Read input**, **Manage text streams**, **Adjust** an **Hyphenate**.

Read input reads input text and recognizes embedded formatting directives. In a simple system, this requires recognition power at the lexical level. More sophisticated systems might support input specifications for the formatting of mathematical formulas, tables, block diagrams, etc. In that case more complex patterns must be recognized in the input text. **Manage text streams** supervises the output stream. Various areas in the 'current' output page (like headers, text columns and footnotes) are usually filled independently. This is implemented most naturally by storing the information related to them in separate data structures. This requires data structures allowing their components to grow dynamically. **Adjust** distributes the spaces embedded in a text line so as to obtain right adjusted margins. This can be done in several ways and it depends on the particular implementation which language features are needed. One implementation might, for example, represent a line as a linked list of words with each word containing a relative distance to the previous word. If the amount of blank space in a line becomes too large, Adjust calls **Hyphenate**. The latter subdivides words into syllables. Hyphenation is used when a given word fits the current output line only partially. This requires table lookup in tables with hyphenation prefixes and suffixes or in tables containing words with exceptional hyphenation points.

Typical application 3: compile a source program in some programming language into machine code. The modules **Lexical analyzer**, **Syntax analyzer** and **Code generator** can be found in most traditional compilers.

A **Lexical analyzer** reads the input stream character by character and constructs from these characters the basic symbols (such as integers, identifiers and keywords) of the programming language. This requires lexical level recognition power. The **Syntax analyzer** performs the syntactic analysis of the stream of symbols produced by the lexical analyzer. For each type of context-free grammar there exists an associated recognizer and the precise form and efficiency of such a recognizer depends on the kind of grammar. Each recognition function should be able to handle the case that its input string is **not** recognized, i.e. that the recognition fails. The output of the syntax analyzer is the parse tree that corresponds to the source program. The construction of parse trees requires dynamically allocated data structures. The **Code generator** transforms parse trees into executable machine code. The requirements depend in this case on the particular implementation method chosen.

Before embarking on yet another effort to design a programming language it is worthwhile to answer the question as to how well existing languages satisfy the typical requirements of string processing or, if they are inadequate in this respect, in what way they can be extended so as to meet them in a more satisfactory manner. This is done in Section 1.3 below. As a preparation for this the reader is first, in Section 1.2, familiarized with some basic notions that are used frequently in subsequent chapters. Problems in existing string processing languages are illustrated in Section 1.4 by means of some SNOBOL4 programs. Section 1.5 contains a list of questions that can serve as a basis for the evaluation of string processing languages, while at the same time suggesting the direction of future developments.

This thesis gives a more or less chronological account of attempts to solve some of the problems in string processing languages. It consists of two parts.

Part I traces the historical development in detail. Chapter 1 is introductory and gives the necessary motivation and background for the study of string processing languages.

Chapter 2 is mainly of historical interest and is not essential for understanding subsequent chapters. It describes the language SPRING, a first attempt to design a string processing language. SPRING may be characterized as a **big** language, i.e. it provides a large number of language primitives for solving problems in its envisaged application areas. Attention is drawn to undesirable language features resulting from seemingly logical design choices. Many problems and questions discussed in Chapter 1 were identified as such during this effort.

Chapter 3 is devoted to general design considerations for string processing languages and compares the semantics of various pattern matching models. Attention is paid to different forms of side-effects during a pattern match. This is done by giving an operational, formal definition of the semantics of the various models. As a result of this, a new pattern matching model, based on side-effect recovery, is developed.

Chapter 4 gives an overview of the language SUMMER a second attempt to design a string processing language. SUMMER may be characterized as a **small** language, i.e. it consists of a relatively small set of primitive operations together with a modest extension mechanism.

Chapter 5 concentrates on the problem of finding a method for formal language definition that is suitable for the designer as well as the implementors and users of a language. An improved method for the operational definition of programming language semantics is developed and the result of applying this method to SUMMER is illustrated.

Implementation issues are discussed in Chapter 6. The SUMMER compiler and run-time system are described in some detail.

Chapter 7 concludes the first part of this thesis by evaluating the research described in it and by outlining several areas for further research.

Part II contains a complete definition of the SUMMER programming language. It consists of a definition of the language (both formal and informal), gives examples of the various language constructs and discusses some annotated programs.

In this thesis we are not concerned with the social implications of text processing and office automation. The interested reader is referred to the literature for a discussion of this issue. [Mowshowitz81] discusses the different approaches to the study of social issues in computing. [Weizenbaum76] analyzes the influence of technology (and in particular computer science) on our society and exposes (mis)conceptions among computer scientists regarding the tasks that can ultimately be delegated to computers.

1.2. Basic operations on strings

Agreement is necessary on what we shall mean by **strings** and **string processing** before a characterization of string processing languages is possible. A **string** is defined as a sequence of **string-items** (to be defined below), such that:

- The sequence is linearly ordered and of arbitrary (finite) size.
- Individual string-items in the sequence can be selected by means of indexing. For a sequence of length N, the items in the sequence have indices $0, \ldots, N-1$ respectively.
- An equality relation is defined on the set of string-items. This relation extends in a natural way to the set of strings.

This definition is deliberately general and does not use any particular property of string-items, apart from the assumption that an equality relation is defined on the set of string-items. It allows, for instance, strings of integers, strings of reals, strings of strings of integers, and so on. Most of the time, however, we shall be dealing with strings consisting of **characters**, i.e. entities corresponding to letters, digits and other symbols which can be displayed on a printing device. Unless otherwise stated, all strings are assumed to consist of characters and in the examples literal character strings will be enclosed in single quotes (like *'metaphysics'*).

String processing will be understood to encompass the totality of operations to synthesize and analyze (parse, recognize) strings.

The most primitive operations on strings are **concatenation** and **substring selection**. A dyadic operator denoted by '||' will be used for string concatenation; it 'glues' two strings together. For example,

'meta' || *'physics'*

has the new string *'metaphysics'* as value.

Substring selection extracts a substring from a given string. For example,

substring('metaphysics', 7, 3)

produces the new string *'sic'* by extracting a substring of size 3 starting at position 7 from *'metaphysics'*. Remember that the characters in a string have indices $0, 1, \ldots, N-1$, where N is the number of characters in the string.

Less primitive recognition operations, as can be found in SNOBOL4, operate on a single common string ('the subject string') starting at a certain index in that string ('the cursor position'). These recognition operations appear in two varieties. The first variety consists of operations and predicates which depend only on the current value of the cursor. Typical examples are:

- Increment the cursor by 7. This operation fails if the resulting cursor is not a legal index in the current subject string.
- Is the current value of the cursor equal to 3?

The second variety consists of operations and predicates which depend both on the current value of the cursor and on the characters in the subject string. Examples are:

- Does *'metaphysics'* occur as substring in the subject string, starting at the current cursor position?
- Can the cursor be moved to the right in such a way that it is only moved past letters? And if so, which letters?

These operations can either **succeed** if their predicate is true (and perhaps change the value of the cursor or deliver a value or both) or **fail** if the predicate is false. These

examples show the need for failure handling in string processing languages (see 1.3.3).

After these preparations, a list of recognition operations follows for reference purposes. These operations are presented in a more or less abstract form, without commitment to specific syntactic or semantic details. More detailed descriptions of these operations will appear in subsequent chapters.

LEN(*n*) increments the cursor by *n* (see Figure 1.1) and fails if the new cursor falls outside the subject string.

LEN(2): *'route 66.'* → *'route 66.'*
↑ ↑
1 3

Figure 1.1. Example of *LEN*.

TAB(*n*) moves the cursor to index *n* and fails if that new index falls outside the subject string (see Figure 1.2). Note, that this operation depends on the specific index convention chosen.

TAB(7): *'route 66.'* → *'route 66.'*
↑ ↑
1 7

Figure 1.2. Example of *TAB*.

RTAB(*n*) moves the cursor to position *length*(*subject*) − *n* − 1, where *length*(*subject*) gives the number of characters in the subject string (see Figure 1.3). The operation fails if the desired cursor position falls outside the subject string.

RTAB(5): *'route 66.'* → *'route 66.'*
↑ ↑
1 3

Figure 1.3. Example of *RTAB*.

POS(*n*) succeeds if the value of the cursor is equal to *n* and fails otherwise (see Figure 1.4).

POS(1): *'route 66.'* → *'route 66.'*
↑ ↑
1 1

Figure 1.4. Example of *POS*.

RPOS(*n*) succeeds if the value of the cursor is equal to *length*(*subject*) − *n* − 1, and fails otherwise.

SPAN(*S*) moves the cursor past the largest number of characters (but at least one), all of which must occur in *S* (see Figure 1.5) and fails otherwise. Note that functions

SPAN and *BREAK* (see below) use their argument string S as a set of acceptable characters.

```
SPAN('0123456789'): 'route 66.' → 'route 66.'
                          ↑          ↑
                          6          8
```

Figure 1.5. Example of *SPAN*.

BREAK(S) moves the cursor (zero or more positions) to the right until it points to the first character that occurs in S (see Figure 1.6), and fails otherwise.

```
BREAK('86420'): 'route 66.' → 'route 66.'
                      ↑          ↑
                      1          6
```

Figure 1.6. Example of *BREAK*.

1.3. Why are string processing languages special?

We shall now consider three major aspects of string processing languages in more detail:

- **Bookkeeping**. How can a record be kept of the progress of the recognition process?
- **Recognition strategies**. What is the best method to determine the structure of a given string?
- **Failure handling**. What should be done if a string cannot be recognized?

1.3.1. Bookkeeping

A general way to formulate many parsing problems is to divide the problem into a number of **recognition steps** of the form

$$S \rightarrow S'$$

in which S (the string to be recognized) is mapped on a new string S' on which the next step operates. In other words, each step delivers a new string value for the next step to work on, and each step begins its recognition task by looking at the leftmost character of its input string. An important special case occurs if successive steps operate strictly from left to right. In that case, all recognition steps operate on substrings of the original input string and each step delivers a tail of its input as result to the next step. In both the general and the special case, a completely **functional** (e.g. LISP-like) formulation of the recognition process can be achieved. This approach is attractive, but has several disadvantages, to wit:

- The need to explicitly mention the string on which each step operates has an adverse effect on the size of programs.

- If one attempts to exploit the special case, only strict left-to-right scanning can be formulated, since the characters in the initial string that occur left of the start of each substring are lost.
- It is not easy to implement the functional model efficiently.

Another way of looking at the recognition process is to assume that there is one common string on which all operations work starting at different cursor positions. The form of a recognition step then becomes

$$<S, C_1> \rightarrow <S, C_2>$$

where S stands for the fixed string to be recognized and C_1 and C_2 stand for the cursor position before and after the step. This can be expressed by introducing the notion of a **current subject** consisting of a string S and a cursor position C in S. All recognition steps operate on the string S starting at cursor position C. This approach has the advantage of obviating the need to mention the subject string explicitly each time a new step is performed as well as of providing cursor management. In other words, the notation is made more concise at the expense of introducing a global entity, which acts as 'current focus of activity'.

In order to limit the field of discussion, we will only pursue the second approach in this thesis. Some consequences of the functional approach can be found in [Morris80]. As to the choice made, it is interesting to note that it is hard to find a notion of a 'current focus of activity' in any existing general purpose programming language.

1.3.2. Recognition strategy

Parsing a string amounts to recognizing some given structure in it. A natural way of expressing such structures is by means of a **grammar**. There exist many kinds of grammar with varying descriptive power (see for example [Aho72]). In practice, most grammars have an associated algorithm to recognize strings belonging to it. In the design of a string processing language, a decision must be made regarding the descriptive power and recognition strategy that will be supported by the language. One can either restrict the class of admissible grammars to those having an efficient recognition algorithm, or one can allow arbitrary context-free grammars and use a general, but less efficient parsing method. The latter will be done in this thesis, since the problems involved are interesting and have only been partially explored. Having chosen a recognition method, the conciseness of recognition algorithms is, in general, enhanced by providing a shorthand notation for it. In this way, the details of the algorithm (like shifting to a new state or reading the next input symbol) can be omitted for each recognition step.

Backtracking will be used as the recognition method for arbitrary context-free grammars. Backtracking [Golomb65] is a programming technique for organizing search processes that are based on trial-and-error. It amounts to imposing a tree-structure on the search space and traversing the tree in a predetermined order. Backtracking can be applied to parsing as follows. Initially, it is **assumed** that a given input sentence can be derived from the grammar rule

```
<s> ::= <r> .
```

where <s> is the start symbol of the grammar and <r> is the right hand side of the grammar rule for <s>. This assumption can either be verified in a trivial way (if <r>

is simple, e.g. a terminal symbol of the grammar) or the recognition process must prepare itself for the verification of a more complex assumption. To this end, new assumptions are made that correspond to the constituents of <r>. If all these assumptions turn out to be true, the initial assumption was true. If an assumption turns out to be false, there are two cases:

- There exists an alternative for it. In this case an attempt is made to verify the alternative. For example, the assumption that an <addition-operator> will occur in the input sentence may turn out to be true if either a '+' or '-' symbol is encountered.
- There exist no alternatives for the current assumption. In this case, the 'parent' assumption was false, but it may in its turn have alternatives.

Several subsidiary questions must be answered when the particular backtracking method chosen is to be specified completely. A first question that arises concerns the **order** in which alternatives are attempted. A method is said to be **deterministic** if the order in which alternatives are attempted is reproducible. In **nondeterministic** methods alternatives are attempted in an arbitrary order. Again, in order to narrow the field of discussion, we shall restrict our attention entirely to deterministic methods. A second question to be answered has to do with the **moment** at which the search space is established. Is it fixed statically at the start of the search process or can it be modified dynamically during the search? We shall consider both possibilities. A final question concerns the precise **structure** of the search space. Does it have the structure of a tree, a directed acyclic graph or perhaps even an arbitrary graph? We shall mostly encounter tree-like structures.

Further aspects of backtracking (as used in SNOBOL4) are discussed in Section 1.4.

1.3.3. Failure handling

The outcome of the entire recognition process is dependent on the outcome of each individual recognition step. Since each step may discover the subject string to have an unexpected form, failure handling is an important issue. For each step there are two possibilities:

- The step succeeds and this fact together with more detailed information (the recognized part of the subject string, the new cursor value) have to be made available to subsequent steps.
- The step fails and the kind of failure has to be indicated.

How the success or failure of an individual step affects the overall recognition process, depends on the particular recognition strategy chosen.

A short remark on failure handling is appropriate in anticipation of discussions on this topic in Chapters 2 and 4. When considering the combinations of language features dealing with failure handling and flow of control, one has the following choices:

1) Include 'Boolean' values in the language, which can be used to remember the outcome of logical operations, and let the flow of control constructs be dependent on these Boolean values. All recognition functions should then be Boolean functions; success or failure of each function is delivered as the result of its invocation and subsidiary results (such as the new cursor value) can then

be delivered using call-by-reference parameters.

2) Let all 'values' in the language consist of (value, signal)-pairs; the flow of control constructs use the signal-part of each value and all other constructs use the value-part. The signal-part of a value can thus be inspected at any moment after the value has been computed. Since it may be desirable for the evaluation of an expression to terminate as soon as one of its subexpressions fails, all operations in the language should be defined in such a way that they immediately terminate when one of their arguments is a value containing a signal-part indicating previous failure.

3) All operations generate a 'failure signal', which is used to drive the flow of control constructs. In contrast to the previous case, where failure signals can be remembered for later use, in this case they are transient entities: failure signals are not part of a value and should be immediately intercepted when they are generated.

4) Include both Boolean values and a general exception handling mechanism in the language. The flow of control constructs can then operate on Boolean values and all other 'abnormal' conditions can be taken care of by the exception handling mechanism.

Alternative 1) is the obvious choice if recognition functions have to be embedded in a conventional programming language. It has the disadvantage that many additional if-statements are required to test the outcome of each recognition function. Alternative 2) is interesting since it allows differentiation between sources of failure (by specifying different values in the signal-part) without introducing complicated flow of control primitives needed for general exception handling. In Chapter 2 we will discuss a restricted form of an alternative 2) expression evaluation mechanism. Alternative 3) is a compromise between expressive power and simplicity: it incorporates exception handling for one kind of exception (failure signals) but does not require complicated flow of control primitives in the language. This alternative will play an important role in our studies. Alternative 4) is the most general, but at the same time the most complicated form of expression evaluation. It will not be considered here to avoid the many unsolved problems associated with general exception handling. See, for instance, [Goodenough75] or [Luckham80] for a discussion of this issue.

1.3.4. Existing languages and string processing

By combining the language requirements encountered in Section 1.1 with the more detailed characteristics of string processing given above, we arrive at the following list of language requirements for string processing:

R1. Recognition power at the syntactic level. If recognition of arbitrary context-free grammars is desired, then some form of backtracking should be available in the language. The notion of a 'subject string' should be available.

R2. Failure handling, i.e. language constructs for (restricted) exception handling.

R3. Data structures that can be allocated dynamically and that may grow dynamically.

Other obvious requirements that apply to **all** kinds of programming languages, such as modularity and adequate control structures, are taken for granted and will not be considered here.

Two general observations will place these requirements in perspective. First of all, it should be noted that **all** envisaged applications **could** be implemented using FORTRAN, assembly language, etc. However, the introduction of special language features for string processing can result in a programming language that is much more suited to string processing applications than other languages that are not 'optimized' for this particular application.

Secondly, one should bear in mind that we have chosen to investigate problems related to backtracking. Backtracking is just another programming technique, but manifests itself differently when integrated with other constructs in a programming language. This becomes particularly clear if side-effects are taken into account. The incorporation of backtracking facilities into a programming language makes it possible to define explicitly the interaction between backtracking and the operations that may cause side-effects (e.g. assignment statements). This cannot be achieved if backtracking is added on top of an existing programming language by, for example, procedural extension.

There are also more specific reasons for designing a new language instead of choosing an existing one. Only the chief shortcomings of PASCAL [Wirth71] and ALGOL68 [VanWijngaarden76] will be discussed here; a discussion of SNOBOL4 is postponed to Section 1.4.

There are five major obstacles to using PASCAL for string processing. First, the size of PASCAL data structures is fixed statically and this conflicts with requirement R3. Secondly, the programmer has to be aware of the life-time of some data structures; these must allocated and de-allocated explicitly. Thirdly, the size of strings is part of their type, i.e. two strings of different length have different type and cannot, for example, be assigned to the same variable. Several attempts (see for instance [Sale79]) have been made to eliminate this problem, but none seems successful. Fourthly, it is not easy to incorporate any form of failure or exception handling into the language. Finally, backtracking and more specifically the control of side-effects during backtracking are difficult, if not impossible, to implement in PASCAL.

There are three major obstacles if one tries to use ALGOL68 for string processing. First, the programmer is responsible for the allocation of objects on the heap. This is a nuisance since, typically, procedures deliver objects that have a longer life-time than the procedure itself and such objects must therefore be explicitly allocated on the heap. The other two obstacles are the same as the ones mentioned for PASCAL: the difficulty of implementing failure handling and backtracking.

1.4. Problems in string processing languages

There are several problems in existing string processing languages and most of them are a consequence of side-effects occurring during the recognition process. These problems will now be illustrated by introducing an absolute minimum of SNOBOL4 [Griswold71] (being the best known string processing language) and by giving some SNOBOL4 examples that exhibit these problems.

1.4.1. A short introduction to SNOBOL4

In SNOBOL4 the recognition steps are described by a **pattern** and the recognition process is called **pattern matching**. A pattern defines a set of acceptable strings and acts as a predicate that succeeds or fails when it is presented with a string that is or is

not in the set of acceptable strings. A pattern may also perform arbitrary computations while deciding whether a given string is acceptable or not. The general form of a SNOBOL4 statement is:

<label> <subject> <pattern> '=' <replacement> <goto>

A <label> identifies a statement and allows other statements to 'jump' to that statement. A <subject> followed by a <pattern> indicates the beginning of a pattern match to determine whether the subject string contains a substring that is in the set of acceptable strings defined by the pattern. If so, the matched substring is replaced by the <replacement> string and execution proceeds at the statement associated with success. Otherwise, no replacement takes place and execution proceeds at the statement associated with failure. The labels of the successor statements for success and failure are given in the <goto> field. Most parts of a SNOBOL4 statement are optional. Apart from the two examples that follow, we shall only consider statements in which all fields except the subject and pattern field are empty.

Example 1:

L X SPAN('0123456789') :S(P)F(Q)

Here, *L* is the <label>, *X* is the <subject>, *SPAN('0123456789')* is the <pattern> and *:S(P)F(Q)* is the <goto>. The result of executing the above statement is a jump to label *P* if the subject string *X* contains a span of digits or a jump to label *Q* otherwise.

Example 2:

L PACT 'multi−lateral' = 'impossible'

Replaces the first occurrence of the string *'multi−lateral'* in *PACT* by the string *'impossible'*. Does nothing if the pattern fails, since no 'failure' label was given in the <goto> field.

All these pattern matches are **unanchored,** i.e. the pattern as a whole is attempted at all cursor positions in the subject string.

1.4.2. Compound patterns

Compound patterns are constructed from primitive ones (i.e. the literal string, *SPAN*, *BREAK*, etc.) by means of **pattern concatenation** and **pattern alternation**. The construction of compound patterns is performed **before** the pattern match is started. This leads to two evaluation moments: **pattern construction time** and **pattern matching time.**

The concatenation of two patterns P_1 and P_2 is written as

P_1 P_2

(i.e. P_1 followed by P_2 separated by one or more space characters) This constructs a new pattern that will apply P_1 followed by P_2. For example,

YEAR 'AD' SPAN('0123456789')

succeeds if *YEAR* contains the string *'AD'* followed by digits.

The alternation of two patterns P_1 and P_2 is written as

$P_1 \mid P_2$

and constructs a new pattern that will succeed if either P_1 succeeds, or P_1 fails but P_2 succeeds. For example,

YEAR 'UNKNOWN' | SPAN('0123456789')

succeeds if *YEAR* contains **either** the string *'UNKNOWN'* **or** a span of digits and

X ('d' | 'b') 'ea' ('n' | 'r' | 'd')

will succeed if *X* contains *'dean'*, *'dear'*, *'dead'*, *'bean'*, *'bear'*, or *'bead'* as substring.

In fact, compound patterns represent **and/or goal-trees** (see [Nilsson71]) and a pattern match succeeds if (part of) the tree has been 'traversed successfully'. Figure 1.7 shows the and/or tree corresponding to the last example.

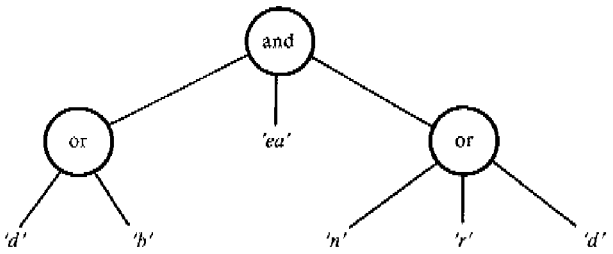

Figure 1.7. And/or goal tree.

If the root of the tree is an 'and' node (representing pattern concatenation), **all** immediate subtrees of the root must have been traversed successfully before the pattern match succeeds. If the root of the tree is an 'or' node (representing pattern alternation), only **one** immediate subtree of the root must have been traversed successfully before the pattern match succeeds. In the last case the pattern may have untried alternatives, i.e. unattempted immediate subtrees of the root. All subtrees of an alternative node are always attempted starting at the same cursor position.

The tree is traversed by means of **backtracking**; this is a structured form of trial-and-error (see 1.3.2). When one attempt to traverse a subtree fails, the aforementioned untried alternatives may lead to a different, but successful traversal of the tree.

1.4.3. Side-effects during pattern matching

The SNOBOL4 patterns introduced so far cannot have side-effects: the values of variables in the program cannot be modified during the traversal of the tree if only pattern concatenation and pattern alternation are used. However, several other operations are available in SNOBOL4 and these can have side-effects. Three of them are: **immediate value assignment**, **conditional value assignment** and **unevaluated expressions**.

Immediate value assignment is written as

P $ V

and constructs a new pattern that will assign to variable *V* the part of the subject string that is recognized by pattern *P*. This assignment is performed immediately, i.e. at the moment that the immediate value assignment operation is encountered in the pattern tree. For example

'AD 1984' SPAN('0123456789') $ YEAR

assigns the string *'1984'* to the variable *YEAR*, and

'1984 BC' (SPAN('0123456789') $ YEAR) ' AC'

fails, but also assigns *'1984'* to *YEAR*.

Conditional value assignment is written as

P . V

and constructs a new pattern that will assign to variable *V* the part of the subject string that is recognized by pattern *P*. Assignment is **only** performed at the end of a successful pattern match. For example,

'1984' SPAN('0123456789') . YEAR

assigns *'1984'* to *YEAR*, but

'1984 BC' (SPAN('0123456789') . YEAR) 'AC'

fails and does **not** assign a new value to *YEAR*.

Finally, let *E* be an arbitrary SNOBOL4 expression. **Unevaluated expressions**, written as

**E*

construct a new pattern that will evaluate the expression *E* at the moment the new pattern is encountered during the match. The value of *E* is then used as the pattern to be recognized. For example,

*X (SPAN('0123456789') $ Y) 'AA' *Y*

succeeds if *X* contains two identical spans of digits separated by the string *'AA'*. Note that, in this example, side-effects are used that were the result of previous operations in the pattern match, namely the immediate value assignment to the variable *Y*. In general, the evaluation of an unevaluated expression may itself cause side-effects.

With the introduction of these operations, the program state can be influenced during a pattern match by:

- immediate value assignments
- cursor movements caused by recognition operations
- side-effects caused by the evaluation of unevaluated expressions.

Note that conditional value assignment can only affect the state at the completion of a successful pattern match.

1.4.4. Problems with the SNOBOL4 approach

A more elaborate example will give the reader some feeling for the complexity that can result from the application of SNOBOL4 pattern matching operations. Let *P* be the pattern defined by

$$((LEN(2)\ \$\ X)\ ('CD' \mid 'EF')\ .\ Y\ *X\ *Y) \mid (LEN(3)\ .\ Y)$$

and assume that the variables *X* and *Y* both have initial value *'ZZZ'*. Considering the pattern match

$$'ABCDABZZZ'\ \ P\ \ ,$$

which values will be assigned to *X* and *Y* after execution it? The following intermediate steps provide the answer.

1) $LEN(2)$ immediately assigns *'AB'* to *X*.
2) (*'CD'* | *'EF'*) conditionally assigns *'CD'* to *Y*, i.e. assignment is not performed but remembered.
3) $*X$ evaluates to *'AB'*, and this pattern succeeds.
4) $*Y$ evaluates to *'ZZZ'* (the initial value of *Y*!), and this pattern succeeds.
5) The pattern match succeeds and the conditional value assignment to *Y* (which was remembered in step 2, above) is performed.
6) At the completion of the pattern match, *X* has value *'AB'* and *Y* has value *'CD'*.

The problems with the SNOBOL4 approach can now be summarized as follows:

- Side-effects during the pattern match in combination with immediate/conditional value assignment lead to opaque programs in which left-to-right textual order of the program source text need not correspond to the actual order of evaluation.
- Backtracking is completely automatic and cannot be controlled by the programmer. This may either lead to gross inefficiencies or to undesired or unexpected behavior of programs.
- There are two different vocabularies in the language. One for expression evaluation and another for pattern matching (see [Griswold80]).

1.5. A checklist for string processing languages

After this inventory of string processing operations and associated problems in string processing languages one can compose a list of questions that can serve as a basis for the characterization of string processing languages. As with any questionnaire, the questions asked largely determine the answers one gets. The list of questions given here is based on a particular view of the way in which string processing languages should develop. This point of view will be explained in more detail in Chapter 3.

1.5.1. Treatment of the subject

- Can the subject be defined explicitly?
- What is the scope of the subject? Is it the whole program, one procedure or one statement?
- Can more than one subject be defined? And if so, are subjects defined consecutively or simultaneously?
- Which data types can the subject have? Possibilities are character string, character file, integer array and perhaps others.

1.5.2. Recognition strategy

One can distinguish several recognition strategies, such as the ones used for the recognition of regular expressions and LL(k) or LR(k) languages, and the techniques used for recursive descent and backtrack parsing. Only recursive descent parsing and backtrack parsing will be considered in this thesis. There are two reasons for making this restriction. The first reason is historical, since initially SNOBOL4 was taken as a starting point and backtrack parsing is the only recognition strategy available in that language. The second reason is that backtrack parsing allows the recognition of a wider class of languages than is possible with, for example, LR(1) parsers. In general, it might be a better idea to make the recognition strategy invisible at the programming language level and to let the implementation choose the best strategy for a given problem. This line of development is interesting but falls outside the scope of the current work.

With respect to backtrack parsers, the following questions can be asked:

- Are side-effects possible during the recognition process?
- How are side-effects treated in case of failure? See the last point below.
- How is flow of control backtracking organized, i.e. how is the next alternative selected? One can distinguish between *ad hoc* and systematic flow of control backtracking. In the former case, the programmer has to indicate each alternative explicitly while in the latter case, alternatives are determined in some systematic, implicit manner. Systematic backtracking may either be completely automatic or the programmer may have the possibility of exercising more detailed control over the backtracking process.
- How is data backtracking organized, i.e. how is determined which values program variables should have after an attempt failed? Here one can distinguish *ad hoc* and systematic backtracking in the same way as above.

1.6. References for Chapter 1

[Aho72] Aho, A.V. & Ullman, D., *The Theory of Parsing, Translation and Compiling*, Volumes I and II, Prentice-Hall, 1972.

[Golomb65] Golomb, S.W. & Baumert, L.D., "Backtrack programming", *Journal of the ACM*, **12** (1965) 4, 516-524.

[Goodenough75] Goodenough, J.B., "Exception handling: issues and a proposed notation", *Communications of the ACM*, **18** (1975) 12, 683-696.

[Griswold80] Griswold, R.E. & Hanson, D.R., "An alternative to the use of patterns in string processing", *Transactions on Programming Languages and Systems*, 2 (1980) 2, 153-172.

[Griswold71] Griswold, R.E., Poage, J.F. & Polonsky, I.P., *The SNOBOL4 Programming Language*, Second Edition, Prentice-Hall, Englewood Cliffs, N.J., 1971.

[Luckham80] Luckham, D.C. & Polak, W., "ADA exception handling: an axiomatic approach", *Transactions on Programming Languages and Systems*, 2 (1980), 225-233.

[Morris80] Morris, J.H., Schmidt, E. & Wadler, Ph., "Experience with an applicative string processing language", *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, 1980, 32-46.

[Mowshowitz81] Mowshowitz, A., "On approaches to the study of social issues in computing", *Communications of the ACM*, **24** (1981), 146-155.

[Nilsson71] Nilsson, N.J., *Problem-solving Methods in Artificial Intelligence*, McGraw-Hill, 1971.

[Sale79] Sale, A.H.J., "Strings and the sequence abstraction in Pascal", *Software Practice and Experience*, **9** (1979), 671-683.

[VanWijngaarden76] Van Wijngaarden, A., *et al*, *Revised Report on the Algorithmic Language ALGOL68*, Springer-Verlag, Berlin, 1976.

[Weizenbaum76] Weizenbaum, J., *Computer Power and Human Reason*, W.H. Freeman, San Francisco, 1976.

[Wirth71] Wirth, N., "The programming language PASCAL", *Acta Informatica*, **1** (1971) 1, 35-63.

2. AN OVERVIEW OF THE LANGUAGE SPRING

2.1. Introduction

SPRING was designed as an implementation language for general text processing systems. The input specifications for such systems may be very complex, since mathematical texts, tabular material and block diagrams must, in principle, be handled. The language had therefore to provide a concise method for recognizing in its input complex layout descriptions. Also, powerful output primitives were required for the construction of elaborate layouts and for the use of arbitrary type fonts.

These overall requirements together with deficiencies we had observed in SNOBOL4, led us to the following, more detailed design goals:

- General purpose pattern matching and string manipulation facilities.
- Pattern matching and string synthesis primitives of 'equal' power. SNOBOL4 contains extensive pattern matching facilities but only a few primitives for string synthesis. By giving pattern matching and string synthesis equal emphasis, we hoped to be able to design a more properly balanced language.
- Uniform treatment of patterns and procedures. Patterns and procedures have much in common but are treated as completely different entities in most string processing languages. We aimed at unification of both concepts (with the ultimate goal of finding a single new concept encompassing the essential properties of both).
- Decent control structures to promote structured programming and to facilitate correctness considerations.

An oversimplified characterization of SPRING might be 'a generalized, structured version of SNOBOL4 with enhanced string synthesis capabilities'. This characterization does injustice to both SNOBOL4 and SPRING but emphasizes the conceptual basis from which SPRING was developed. SPRING is a **big, inextensible** language: it contains many primitives for pattern matching and string synthesis but it does not allow the user to add his own operations or redefine existing ones in an easy manner.

This chapter provides a global overview of the language SPRING and gives an impression of the range of problems that can be solved by SPRING programs. It also draws the reader's attention to several interesting problems that arose during the design of the language. Apart from identifying these problems, the material presented here is largely historical in nature and is not essential to the understanding of following chapters.

Sections 2.2 and 2.3 give an overall idea of expression evaluation, control structures and values in SPRING. Section 2.4 contains a more detailed description of operations on 'three dimensional' character strings and Section 2.5 describes pattern matching facilities. Several examples are given in Section 2.6. A critical evaluation of the language in Section 2.7 concludes the chapter.

2.2. Expression evaluation and control structures

SPRING is an expression oriented language. An expression consists of **operands** separated by **operators**. Expressions return a value but have the additional property that they can **fail** or **succeed** (see 1.3.3). This success or failure is caused by certain operators (like the relational operators or the pattern match operator) or procedures.

Both built-in and user-defined procedures may fail and will report this failure to their caller. Note that expression evaluation continues when failure is detected, and that failure is reported to the enclosing expression.[1]

Control structures are driven by the success or failure of expression evaluation. For example, in the statement

if e_1 **then** e_2 **else** e_3 **fi**

expression e_1 is evaluated first. If this evaluation is successful, e_2 is evaluated next. Otherwise, e_3 is evaluated next.

2.3. Values and variables

SPRING supports values of the following types:

- integer
- block ('three dimensional' character string)
- array (one dimensional row of arbitrary values)
- table (associative memory)
- file (sequential character stream)
- procedure
- pattern.

Of these types, only blocks (see Section 2.4) and patterns (see Section 2.5) will be described in detail.

Variables must be declared but are typeless, i.e. values of different types can be assigned to them during their lifetime. The scope of variables is either global or local. Global variables are accessible from the whole program, while local variables are only accessible from the procedure or pattern in which they are declared. Procedures can only be declared at a global level.

2.4. Blocks

Written text is two-dimensional in nature, but during the 'cut and paste' process of composing it a third dimension comes into play. Text elements like columns in a table, drawings or photographs, subscripts or superscripts in formulae, etc. are stacked on top of each other in order to bring about the final result. In SPRING an attempt was made to provide string manipulation facilities for describing this 'cut and paste' activity. The string manipulation primitives are based on the concept of 'three dimensional blocks' of characters as introduced in [Gimpel70]. The horizontal and vertical dimensions correspond to the width and height of a piece of paper, while the third ('normal') dimension is used to describe the printing of several characters on the same position of the paper. In principle, such blocks can be concatenated, sliced and replicated in all three dimensions. Operations on blocks return a tree-shaped data structure as 'value'. Printing such a value involves traversing the tree from left to right. During the traversal a projection along the normal axis is performed, hence the normal direction corresponds to overprinting. Concatenation of blocks is denoted by

1) Compare this with the SUMMER evaluation model (see Chapter 4), in which evaluation of an expression is aborted as soon as a subexpression fails.

'||' (horizontal), '==' (vertical) and '++' (normal). Some examples of block-valued expressions and their printed image are shown in Figure 2.1.

Expression	Printed image
'a' || 'b'	*ab*
'a' == 'b'	*a* *b*
'0' ++ '/'	*0* */*

Figure 2.1. Examples of simple block expressions.

In multi-dimensional concatenation the **alignment** of the blocks being concatenated can be specified with the aid of special operators. For example, '=<' indicates vertical concatenation with left sides aligned, '=>' indicates vertical concatenation with right sides aligned, and '==' indicates centered, vertical concatenation. Three examples are shown in Figure 2.2.

Expression	Printed image
'ab' =< 'c'	*ab* *c*
'ab' => 'c'	*ab* *c*
'abc' == 'd'	*abc* *d*

Figure 2.2. Examples of aligned blocks.

Apart from these basic construction operators, several built-in functions exist for the creation of three dimensional blocks with given dimensions.

Finally, it is possible to create two types of block ('iterated' and 'duplicated') which are repeated depending on the context in which they occur. Both the number of repetitions and the direction of repetition are context controlled. The meaning of these blocks strongly depends on the fact that the creation and printing of a block is a two stage process, since 'ancestor' information is needed to determine the size and form of their final printed image. The tree-like data structure that is the result of block creation establishes, in a natural way, a 'parent' and 'grandparent' relation between blocks (i.e. nodes in the tree). A block creation expression that contains several occurrences of a binary operator at the same nesting level is represented by a node with more than two descendants. This is illustrated in Figure 2.3. The parent and grandparent of the terminal node containing *'c'* are marked with P and GP respectively.

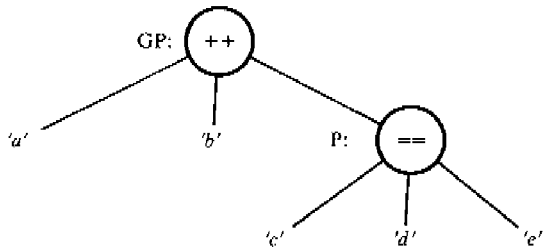

Figure 2.3. Tree representation of $'a'++'b'++('c'=='d'=='e')$.

A last point to be kept in mind before discussing context controlled blocks is that printing a block is also a two pass process. In the first pass, the overall dimensions of all blocks are determined by traversing the tree and accumulating size information. In the second pass, blocks are printed and the overall size information is used to determine the number of duplications of context controlled blocks.

Expression	Printed image
'underlines' ++ it('_')	*underlines*
('abc' == 'd') ++ it('_')	*abc* *d*
pole1 := it('|'); *pile1 := 'x' == 'x';* *pole1 || pile1 || pole1*	|x| |x|
pile2 := 'y' == 'y' == 'y'; *pole1 || pile2 || pole1*	|y| |y| |y|

Figure 2.4. Examples of iterated blocks.

Iterated blocks are duplicated in the **two** directions orthogonal to the concatenation direction of the parent of the iterated block in the tree. The number of duplications is determined by the overall dimensions of the parent. An iterated block is created by the procedure call

it(*B*)

where *B* is the block to be duplicated. Some examples of iterated blocks are shown in Figure 2.4. In the second example, the underline character '_' is duplicated in both horizontal and vertical direction.

It is instructive to visualize the tree structure that is built for such complex block operations. This is done in Figure 2.5. Note that *('abc' == 'd')* determines the overall size of the printed result and that the '++' operator determines the direction of iteration of the argument of *it*.

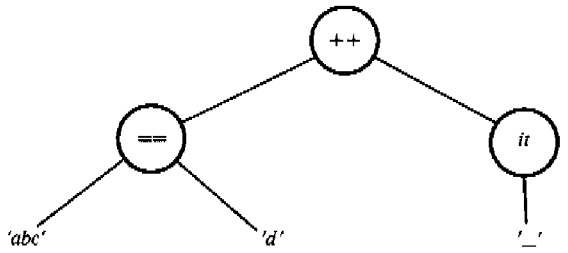

Figure 2.5. Tree representation of *('abc' == 'd') ++ it('_')*.

Replicated blocks are duplicated in the concatenation direction of their parent, but the number of duplications is determined by the overall dimensions of the grandparent of the replicated block. A replicated block is created by

rep(*B*)

where *B* is the block to be duplicated. Some examples are shown in Figure 2.6. Here, *hor*(20) creates a block with size 20 in the horizontal direction and size 0 in the vertical and normal direction.

Expressions:

*line*1 := *'Introduction'* || *rep('.')* || *'1'*;
*line*2 := *'Final thoughts'* || *rep('.')* || *'13'*;
hor(20) == *line*1 == *line*2

Printed image:

Introduction. 1
Final thoughts. . . .13

Figure 2.6. Example of replicated blocks.

2.5. Patterns [2]

A pattern match is initiated by the **match operator** '?', which has a string (or text file) as left operand ('the subject') and a pattern as right operand. The simplest form of a pattern is a string. For example,

'abc' ? 'ab'

determines whether the subject contains the string *'ab'*.[3] The match operator can **fail** or **succeed** and delivers a value in both cases: the subject, in case of failure, and the **assembled scan value** (see below) in case of success.

More complex patterns are built by means of **subsequentiation** (' ') and **alternation** ('|'). Subsequentiation and alternation are identical to pattern concatenation and alternation in SNOBOL4.

The assembled scan value is the horizontal, centered concatenation (see 2.4) of **contributions** made during the pattern match. Two types of contribution exist. **Internal contributions** (denoted by $=P$, where P is a pattern valued expression) contribute a value which is equal to the part of the subject that is matched by P. For instance,

'b' ? =('a' | 'b' | 'c')

contributes *'b'*. **External contributions** (denoted by $/S$, where S is a string valued expression) just contribute the value S. For instance,

'b' ? ('a' | 'b' | 'c')−− /'xyz'

contributes *'xyz'* and

'b' ? =('a' | 'b' | 'c') −− /'xyz'

contributes *'bxyz'*.

Closely related to the above contribution operators are the subject assignment and contribution assignment operators. **Subject assignment** (denoted by $P=:V$, where P is a pattern valued expression and V is a variable) assigns to V the part of the subject that is matched by P. For instance,

'b' ? (('a' | 'b' | 'c') =: x)

assigns *'b'* to x.

Subject assignment is identical to conditional value assignment in SNOBOL4. **Contribution assignment** (denoted by $P/:V$, where P is a pattern valued expression and V is a variable) assigns to V the contributions made by P and does not add these contributions to the assembled scan value. For instance,

'b' ? (=('a' | 'b' | 'c') −− /'xyz') /: x

assigns *'bxyz'* to x, but does not deliver a scan value.[4]

As already mentioned in the previous chapter, a rather delicate question must be answered as to the precise moment at which such assignments are effectuated: this

---

2) A more extensive description of pattern matching in SPRING is given in [Klint78].

3) In fact a distinction is made between **unanchored** and **anchored** pattern matching. Only the latter will be discussed here.

4) Compared with the pattern matching model used in SUMMER one can consider contributions and contribution assignment as methods of returning a value from a pattern matching procedure.

may be immediately after the successful application of the left hand operand of the assignment operator (independently of the success or failure of the overall pattern match), or when the whole pattern match succeeds. SPRING provides two forms of all operators for which this duality exists. Assignment operators can be either conditional ('=:' and '/:') or immediate ('!=:' and '!/:').

To complete this overview of pattern matching facilities, four points will now be discussed: unevaluated expressions, actions, built-in functions and patterns with arguments and local variables.

Unevaluated expressions (denoted by $*P$, where P is a pattern) are similar to unevaluated expressions in SNOBOL4 and provide a means of delaying the evaluation of a pattern until it is actually needed during a pattern match. This is useful for the definition of recursive patterns and for the definition of patterns which are context dependent. For example,

$$p := ('b' \ -- \ *p \mid 'a')$$

defines a pattern that matches *'a'*, *'ba'*, *'bba'*, ··· .

Conditional and **immediate actions** (denoted by $@E$ and $!@E$ respectively, where E is an arbitrary expression) are pieces of program which are evaluated but whose value is not used by the pattern matching process, i.e. they are evaluated for their side-effects and not for their value. For example,

'ab' ? 'a' -- @print('an action') -- 'b'

will print *'an action'*.

Many **built-in functions** are available and most of them have already been described in the previous chapter. Examples are *span*, *any*, *len*, *break*, *arb* (which recognizes a string of zero or more characters, depending on the success or failure of the pattern following it), *rpt* (which applies a pattern repeatedly until it fails), *pos*, *rpos*, *tab*, *rtab*, *reverse* and *replace*.

To make procedures and patterns as similar as possible,[5] patterns may have **arguments** and **local variables**. The use of patterns with local variables is illustrated by the following pattern *'palindrome'* which recognizes a restricted class of palindromes:

```
palindrome := pat { var head;
                    break('.')!=:head -- '.' -- *reverse(head) -- rpos(0)
                  }
```

This pattern will recognize all strings of the form

<left-part> '.' <right-part>

where <left-part> and <right-part> are each others reversal. It is tacitly assumed that neither <left-part> nor <right-part> contains a full-stop (e.g. the character '.'). The built-in function *rpos*(0) is used to ensure that the pattern match proceeds to the last (rightmost) character of the subject string.

5) Real similarity between procedures and patterns has not been achieved in SPRING. In SUMMER (see Chapter 4), the similarity is 'perfect', since the notion of patterns does not exist at all.

The use of patterns with arguments is exemplified by the pattern '*sequence*':

sequence := **pat**(*atom*, *separator*){ *atom* −− *rpt*(*separator* −− *atom*)}

which recognizes sequences consisting of one or more 'atoms' separated by 'separators'.[6] The use of *sequence*, in a pattern that recognizes variable declarations of the form

variable identifier$_1$, . . . , *identifier*$_n$;

might look like:

'variable' −− *sequence*(*identifier*, *','*) −− *';'*

2.6. Some examples

A few examples will illustrate the use of the primitives mentioned so far. A series of pattern definitions for the recognition of identifiers is:[7]

```
1 empty   := '';
2 letter  := 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ';
3 digit   := '0123456789';
4 letgit  := letter || digit;
5 id      := any(letter) −− (span(letgit) | empty);
```

Now the pattern match *stringvar* ? *id* will succeed if the value of *stringvar* begins with a string having the form of an *id*.

A naive procedure (not covering all exceptional cases) for the conversion of integers to roman numerals is (see [Gimpel76]):

```
1  proc roman(n){
2    var t;
3    if succeed n := (n ? =rtab(1) −− (len(1) =: t))
4    then
5      return(
6        replace(roman(n), 'IVXLCDM', 'XLCDM**') ||
7        ('0,1I,2II,3III,4IV,5V,6VI,7VII,8VIII,9IX,' ? t−−=break(',',))
8      )
9    fi
10 };
```

In line 3 the number *n* is split into a left part which is assigned to *n*, and a rightmost digit which is assigned to *t*. In line 6 the left part *n* is converted to roman and multiplied by ten (in roman). In line 7 the rightmost digit is converted to roman and appended to the result.

A final example illustrates the power of integrated pattern matching and multi-dimensional string concatenation. The following grammar recognizes parenthesized

6) In fact, '*atom*' and '*separator*' in the above expression should be preceded by an unevaluated expression operator (i.e. monadic '*'). This is needed to achieve the correct binding time of these variables. These (very intricate) problems will not be addressed here, but see Section 2.7.

7) In the following examples, line numbers are added to simplify the description. These are, however, not part of the actual programs.

list expressions and converts these expressions into a two-dimensional representation of the list:

```
1  blank  := span(' ') | '';
2  list     := pat[var result, atom;
3              ( span(letter) =: result |
4              '(' -- rpt(*list/:atom -- @(result := result |< atom)) --
5                @(result := it('-') == result) -- ')'
6              ) -- blank -- /(' ' || ('|' == result) || ' ')
7            ];
```

Figure 2.7 gives an example of the result of applying this pattern. In line 3 of the above program, an atomic list is recognized and the associated name is assigned to *result*. In lines 4-5, a parenthesized list with sublists is recognized. During each repetition the next sublist is concatenated to the result with all tops aligned (this is done by |<). In line 5, a row of dashes (equal to the width of the result, formed by *it*('-')) is placed on top of the result. Finally, in line 6 a bar is placed on top of the result (this is done by ==) and a blank is appended to the left and the right of *result*, after which it is contributed to the overall result.

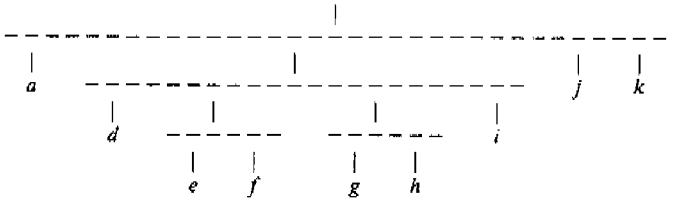

Figure 2.7. The result of applying the pattern *list* to *'(a (d (e f) (g h) i) j k)'*

2.7. SPRING in retrospect

What are the merits of SPRING as a programming language and to what extent does it meet the design goals stated in Section 2.1?

The following observations can be made about the programming language aspects of SPRING:

1) The interaction between expression evaluation and pattern matching is not satisfactory. Most of the problems involved can be traced back to the problem of different evaluation moments. Depending on the type of an expression and its syntactic position, one can distinguish the following five moments at which a SPRING expression can be evaluated:

- Normal expression evaluation.
- Pattern construction.

- Pattern matching.
- Conditional expression evaluation.
- Immediate expression evaluation.

The four additional evaluation moments as compared to other languages are due to two basic dichotomics in the language model: the distinction between expression evaluation and pattern matching and the distinction between conditional and immediate operations. Although the model is consistent in itself, it does not of course relieve the poor programmer from having to worry about all those different evaluation moments. In summary, one can say that the language as a whole is too complex and is not based on a simple conceptual framework. Also, the distinction between immediate and conditional evaluation is unsatisfactory, since in many cases the evaluation order is not reflected properly by the program text. This can be seen in

$$'abc' \ ? \ ('a' = :x) \ -- \ ('b' != :x) \ -- \ ('c' = :x)$$

where the values *'b'*, *'a'* and *'c'* are assigned to *x* in this order.

2) The lack of data structures other than arrays and tables is annoying. There is, however, no fundamental reason why data structures could not have been added.

3) SPRING uses a consistent but awkward notation. It was probably unwise to use so many built-in operators consisting of operator symbols lacking mnemonic value.

4) It turns out to be difficult to implement the language efficiently.

Regarding the design goals one can make the following observations:

1) SPRING provides general purpose pattern matching and string manipulation facilities.

2) The synthesis facilities based on the 'block' concept are very powerful but cannot easily be extended to arbitrary type fonts consisting of characters with different sizes.

3) A uniform treatment of patterns and procedures has not been achieved.

4) Control structures are simple but adequate.

5) Inherent inefficiencies in the implementation of the language make it less suited to applications in a production environment (although it is still being used for this purpose).

2.8. References for Chapter 2

[Gimpel70] Gimpel, J.F., "Blocks — a new datatype for SNOBOL4", *Communications of the ACM*, **15** (1972) 6, 438-447.

[Gimpel76] Gimpel, J.F., *Algorithms in SNOBOL4*, John Wiley, 1976.

[Klint78] Klint, P., "Pattern matching in SPRING", in: Van Vliet, J.C. (ed), Colloquium Capita Datastructuren, Mathematical Centre Syllabus 37, 1978, 65-83.

3. DESIGN CONSIDERATIONS FOR STRING PROCESSING LANGUAGES

3.1. Introduction

The application of backtracking in pattern matching and the integration of pattern matching operations into string processing languages form the central themes of this chapter.

Backtracking can be used to solve certain classes of pattern matching problems elegantly. However, the effects of backtracking on the global program state are often difficult to understand. Are side-effects postponed until the backtracking process as a whole has succeeded (which restricts the class of problems that can be solved), or are such modifications performed immediately and undone on failure? Several methods are being used to control the interaction between the backtracking process and its side-effects. Most methods have defects. In this chapter an attempt will be made to improve on this state of affairs.

The analysis of backtracking will proceed in three stages. First, in Section 3.2 some representative pattern matching functions and operators are defined. These functions are sufficiently powerful that they exhibit the same problems with regard to side-effects and backtracking as occur in, for example, SNOBOL4. They are, at the same time, sufficiently simple to allow a concise formal definition of their semantics and an analysis of the problems at hand. In other words, these functions are used to **model** (i.e. imitate on a smaller scale) certain features of existing programming languages.

Secondly, we have to choose a method for describing the semantics of backtracking processes. After comparing several methods found in the literature, an improved description method is introduced, which is based on operational semantics. This is the subject of Section 3.3.

Thirdly, the description method is used to compare two different pattern matching models. In Section 3.4.2 the **immediate/conditional model** is described. This is a generalization of the SNOBOL4 model [Griswold71] which has been used in SPRING (see Chapter 2). In Section 3.4.3 the **recovery model** is introduced. This new model is based on the recovery block concept [Randell75] which has been used for the construction of fault-tolerant software. It is an attractive alternative to existing backtracking models, since it combines simplicity and consistency with adequate expressive power and ease of implementation.

Finally, Section 3.5 is devoted to the question of how pattern matching and normal expression evaluation can be integrated and how the domain of pattern matching can be extended beyond the domain of strings. In the following chapters these ideas will be explored further.

3.2. Some representative pattern matching functions and operators

We first introduce a uniform terminology with the aid of which the problems inherent in the two pattern matching models can be discussed. A **pattern** defines a set of **acceptable** strings. It is a predicate which succeeds or fails when presented with a string that is or is not an element of the set of acceptable strings. Arbitrary (terminating) computations may be performed while determining whether a given string is acceptable or not. A **pattern match** is the process of deciding whether a given string S is acceptable to a given pattern P or not. The notation $S\ ?\ P$ will be used to

denote a pattern match. All (sub)patterns in P will operate on the same subject string S. An implicit index (the **cursor**) holds the position in the subject string where each subpattern should start its recognition task.

Three types of primitive patterns will be used: **string literals**, **unevaluated expressions** and **actions**. These are informally defined as follows:

String literal: T

Try to match a given string value T in the subject string starting at the current cursor position.

Unevaluated expression: $*E$

Evaluate expression E and use the result as a pattern.

Action: $act(E)$

Evaluate expression E for its side-effects. This primitive pattern always succeeds and does not cause any movement of the cursor (i.e. it matches the empty string).

These primitive patterns can be combined to form more complicated patterns by means of the pattern construction functions **alternation**, **subsequentiation** and **subject assignment**. Note that we will attach different meanings to alternation and subsequentiation in the two pattern matching models to be considered. Here, their meaning is the one used in the immediate/conditional model. Their meaning in the recovery model is given in Section 3.4.3. The pattern construction functions are informally defined as follows:

Alternation: $P \mid Q$

If pattern P fails, attempt pattern Q; if P succeeds, remember Q in case failure occurs later on (this can only happen if the alternation is part of a larger pattern).

Subsequentiation: $P -- Q$

Attempt pattern Q after a successful match of pattern P.

Subject assignment: $sbas(P,V)$

The part of the subject string successfully matched by pattern P is assigned to variable V. (This resembles the '.' and '$' operators in SNOBOL4.) Note that subject assignments are more restricted than primitive action patterns: the latter may contain assignments of any value to any variable, but the former only allow assignment of a matched part of the subject string to an explicitly mentioned variable.

This collection of functions and operators exhibits most features found in pattern matching languages. Later on, some additional functions that are typical for each individual model will be introduced.

3.3. Description methods for pattern matching

Before we embark on our attempt to compare both models, it is necessary to select appropriate semantic description tools. Four methods are discussed:

SET: patterns are characterized by the (possibly infinite) set of strings recognized by them.

ALG: patterns are described by algebraic transformations on (subject, cursor) values.

COR: patterns are described by collections of recursive coroutines

OPS: patterns are described by operational semantics.

3.3.1. Patterns defined by sets of strings

SET is the oldest method for describing patterns and was used during the design of SNOBOL3 [Farber64]. It is possible to view patterns as generative grammars and to associate a set of strings with every pattern constructed from simple strings by sequentiation and alternation. This method has several disadvantages. A first difficulty stems from the fact that alternation is not commutative, i.e. it applies its left and right operands in left-to-right order. (This could be repaired by using **ordered** sets of string.)

More severe difficulties arise, however, with several primitive patterns. It is extremely difficult, if not impossible, to associate a unique set of strings with patterns containing primitives like $POS(n)$, $TAB(n)$ or $BREAK(s)$, since the strings that are matched by such primitives depend on the patterns in which they occur.

3.3.2. Patterns defined by algebraic transformations

This method (based on [Gimpel73] and [Stewart75]) describes the meaning of a pattern as an algebraic transformation of (subject string, cursor) values. The meaning M of a pattern P is defined by a function $M(P)(S, c)$, where S is the subject string and c is an integer indexing S. c is called the **pre-cursor** position. The value of $M(P)(S,c)$ is an ordered sequence of integers, indexing S, which are called the respective **post-cursor positions**. Cursor positions can take on the values $0, 1, \ldots, length(S)$, where $length(S)$ denotes the length of the subject string S. As soon as the meaning of the elementary scanning functions has been expressed in terms of transformations on sequences of pre-cursor values to sequences of post-cursor values, one can simply use composition of cursor sequences to define the meaning of arbitrary patterns.

For instance, the meaning of a string literal T can be described by

$$
\begin{aligned}
M(T)(S,c) = \ & \textbf{if } substring(S,c,length(T)) = T \\
& \textbf{then} \\
& \quad \{c+length(T)\} \\
& \textbf{else} \\
& \quad \{\} \\
& \textbf{fi}\ .
\end{aligned}
$$

In other words, take a substring from S, starting at cursor position c, that has the same length as T. If such a substring exists and it is equal to T, the string literal is said to match and the result is the post-cursor sequence $\{c + length(T)\}$, i.e. the cursor moved past string T. Otherwise the match fails and this is indicated by the empty post-cursor sequence $\{\}$.

Alternation of patterns P_1 and P_2 can be described by

$$M(P_1 \mid P_2)(S,c) = M(P_1)(S,c) \oplus M(P_2)(S,c)$$

where $\oplus$ denotes the concatenation of two sequences.

Subsequentiation of patterns P_1 and P_2 can be described by

$$M(P_1 \text{ -- } P_2)(S,c) = M(P_2)(S, M(P_1)(S, c)) .$$

It is tacitly assumed here that $M(P)$ has been generalized so as to operate on cursor sequences, i.e. the function $M(P)$: $S \times N \rightarrow 2^N$ has been generalized to $M(P)$: $S \times 2^N \rightarrow 2^N$, where S denotes the set of subject strings and N denotes the natural numbers.

We now apply these definitions in an example. If P and S are defined by

P = *'ab'* | *'aab'* | *'a'*
S = *'aab'*

then the application of pattern P to the string S can be characterized by

$M(P)(S, 0) = \{3, 1\}$
$M(P)(S, 1) = \{3, 2\}$
$M(P)(S, 2) = \{\}$
$M(P)(S, 3) = \{\}$.

Continuing in the same spirit, one can give concise definitions for the elementary scanning functions as defined in Chapter 1:

$M(LEN(n))(S, c)$ = **if** $c + n \leqslant$ *length*(S) **then** $\{c + n\}$ **else** $\{\}$ **fi**
$M(POS(n))(S, c)$ = **if** $c = n$ **then** $\{c\}$ **else** $\{\}$ **fi**
$M(RPOS(n))(S, c)$ = **if** $n =$ *length*$(S) - c$ **then** $\{c\}$ **else** $\{\}$ **fi**
$M(TAB(n))(S, c)$ = **if** $n \geqslant c$ & $n \leqslant$ *length*(S) **then** $\{n\}$ **else** $\{\}$ **fi**
$M(RTAB(n))(S, c)$ = **if** *length*$(S) - n \geqslant c$ & $n \leqslant$ *length*(S)
then $\{$*length*$(S) - n\}$
else $\{\}$
fi

This method is suited to the description of the recognition properties of patterns but it does not lend itself to the description of side-effects occurring during a pattern match.

3.3.3. Patterns defined by recursive coroutines

SL5 is a language that provides programmable backtracking based on recursive coroutines. For a description of this method the reader is referred to [Doyle75], [Druseikis75] or [Griswold76]. The latter reference also contains an overview of SL5. There is a straightforward relationship between SL5 coroutines and the more familiar detach/resume operations in SIMULA [Dahl70]. SL5 has been used to describe patterns: a separate coroutine is associated with each component in the pattern and signaling between coroutines is used to control the pattern matching process.

The recursive coroutine method is powerful in that it allows the description of both the recognition process itself as well as the side-effects caused by it. Although the primitives used are powerful, they are not generally known and rather complex, and this would lead us to the undesirable situation that a complex problem would have to be described using complex primitives. We therefore explore a much simpler description method.

3.3.4. Patterns defined by operational semantics

The previous methods are either based on high-level concepts (coroutines) or do not address the problems of environment modification during a pattern match. In this section we shall develop a description technique based on an operational method for defining the semantics of programming languages. In this way we obtain a precise, operational definition of pattern matching semantics which is not based on complex primitives but nevertheless completely describes all aspects of pattern matching.

To describe a pattern matching model completely two entities must be specified:

PAT A grammar describing all patterns.

match A function describing the meaning of those patterns.

To structure the following discussion we further identify:

EXP A subset of *PAT* containing the syntax rules that describe the expressions that may occur in a pattern.

eval A function describing the meaning of expressions. Obviously, *eval* may be considered as an auxiliary function of *match*.

In the next section the functions *match* and *eval* will be described in a simple, but powerful programming language.

3.4. A comparison of two backtracking models

3.4.1 Common definitions for the two models

Some global variables are shared by *PAT*, *match*, *EXP* and *eval*. It is not essential to do so, but it makes the resulting descriptions more concise. This global information consists of:

subject_string:
the current subject string for each pattern matching process being described. For convenience, we introduce the set *CURS* of legal indices (i.e. cursor positions) in the current subject string.

INITIAL_ENV:
the program environment (i.e. the values of variables) at the start of the pattern matching process.

CURRENT_ENV:
the initial environment modified by expressions evaluated during the pattern match.

Environments consist of (*name*, *value*) pairs and for reasons of simplicity we assume here that all names are global. An identifier *id* with value *val* is added to environment *ENV* by the operation:

ENV.bind(*id*, *val*).

If *id* occurred already as the name-part in some pair in that environment, only the value-part of that pair is modified. The value associated with an identifier *id* in the environment *ENV* is retrieved by:

ENV.binding(*id*).

We require *id* to be the name-part of some pair in *ENV*. Finally, complete environments can be copied by the operation

$$ENV_2 := copy(ENV_1)$$

which assigns a copy of environment ENV_1 to ENV_2.

Another aspect common to both models is the expression language *EXP*. In the remainder of this paragraph, *EXP* and its formal definition will be explained. The syntax[1] of *EXP* is given by:

```
<expression>      ::= { <assignment> ';' }* .
<assignment>      ::= <identifier> ':=' <right-hand-side> .
<right-hand-side> ::= <string-literal> | <identifier> [ '+' <string-literal> ] .
```

An <expression> thus consists of zero or more <assignment>s **separated** by semicolons. Each <assignment> consists of an <identifier> followed by an assignment operator (':=') and a <right-hand-side>. A <right-hand-side> consists either of a <string-literal>, an <identifier>, or an <identifier> followed by a plus sign followed by a <string-literal>.

The semantics of <expression>s is as follows. The <assignment>s in an <expression> are treated from left to right. An <assignment> can have three different kinds of <right-hand-side>:

Case 1: <identifier>$_1$:= <identifier>$_2$: 'Bind' the value of <identifier>$_2$ to <identifier>$_1$ in the current environment.

Case 2: <identifier>$_1$:= <identifier>$_2$ + <string-literal>: Concatenate the (string) value of <identifier>$_2$ and the <string-literal> to obtain a new string value. This new string value is bound to <identifier>$_1$ in the current environment.

Case 3: <identifier> := <string-literal>: The <string-literal> is bound to <identifier> in the current environment.

For instance, the evaluation of the expression

$$x:='abc';y:=x;z:=y+'def'$$

will add the pairs (*x*,*'abc'*), (*y*,*'abc'*) and (*z*,*'abcdef'*) to the current environment (provided that no previous bindings existed for the variables *x*, *y* and *z*).

The formal definition of the semantics of *EXP* is given in Figure 3.1. The notation for formal definitions as used there will be used throughout. Its most unusual feature are the **parse expressions** of the form[2]

```
'{' <identifier> '==' <syntax-rule> '}' .
```

These parse the string value of a given identifier and at the same time extract substrings from it. To this end, parts of the <syntax-rule> can be labelled with tags. Each tag corresponds to a variable declared elsewhere in the program (such as *exprs* and *id*1 in the above definition) and upon a successful parse of the string value, each

1) See 8.1 for a definition of the syntax notation used.

2) Do not confuse the syntactic equality operator used here and the block concatenation operator as described in Section 2.4. Both are denoted by the symbol '==='.

such variable receives, as value, the substring recognized by the part of the syntax rule following the tag. If the tag labels a repetitive syntax construct (as with *exprs* above), the corresponding variable receives an array of strings (one string for each repetition of the construct) as value. Obviously, parse expressions can succeed or fail. If it is known in advance that a parse expression will always succeed (as is the case in the lines marked with #a# and #b# in Figure 3.1), the expression need not be contained in an if-statement and can be used for the sole purpose of extracting substrings.

Our notation is fully described in part II of this thesis and an informal description can be found in Chapter 5. In the following sections we assume it to be self-explanatory.

```
proc eval(s)
(   var e, exprs, id1, id2, rhs, str, val;
    if {{ s == exprs: { <assignment> ';' }* }}
    then
      for e in exprs
#a# do {{ e == id1:<identifier> ':=' rhs:<right-hand-side> }} ;
         if {{ rhs == id2:<identifier> }}
         then
           val := CURRENT_ENV.binding(id2)
         elif {{ rhs == id2:<identifier> '+' str:<string-literal> }}
         then
           val := CURRENT_ENV.binding(id2) || str
         else
#b#        {{ rhs == str:<string-literal> }} ;
           val := str
         fi;
         CURRENT_ENV.bind(id1, val)
      od
    else
      ERROR
    fi
);
```

Figure 3.1. Formal definition of *eval*.

3.4.2. The immediate/conditional model

3.4.2.1. Overview

In this model (a generalization of the SNOBOL4 model) the moment at which operations are performed can be controlled at the programming language level. The meaning of the functions *act* (action) and *sbas* (subject assignment) may depend on the moment they are carried out. There are two versions of these functions. The **immediate** versions *imact* and *imsbas* are performed at the moment they are encountered during the pattern match; the global environment is neither saved nor restored on failure. The **conditional** versions *cdact* and *cdsbas* are remembered when they are

encountered during the pattern match but are performed only after the successful completion of the complete pattern match. Remembered conditional functions are forgotten if the pattern match fails.

Conditional functions do not add recognition power, because only immediate modifications of the environment can influence the direction in which the recognition process proceeds. The recognition power of pattern matching is thus based on the existence and power of immediate functions. One can, for instance, build context dependent patterns by combining (immediate) subject assignment and unevaluated expressions.

An example may illustrate why it is, in principle, desirable to use such a powerful model.

Suppose pattern matching is used to parse a program. Parsing of certain constructs in the program, like variable declarations, must have an immediate effect on the parsing of the remainder of the program. Later occurrences of variables can then be compared with their declaration.

Immediate operations allow the recognition of context-sensitive languages. One way of recognizing the (context-sensitive) language $A^n B^n C^n$ is:[3]

```
row_of_A  :=  'A' -- *row_of_A | '' ;
abc       :=  imsbas(row_of_A,as) -- imact(n:=length(as)) --
              *replicate('B',n) -- *replicate('C',n) ;
```

Here, *row_of_A* recognizes arbitrarily long sequences of *'A'*s. Pattern *abc* first attempts to recognize a row of *'A'*s which is immediately assigned to variable *as*. Next, the length of *as* is computed and assigned to variable *n*. Finally, two new patterns are constructed (and attempted) consisting of *n 'B'*s and *n 'C'*s respectively.

The above example illustrates the usefulness of immediate operations. But under which circumstances is it desirable to perform operations conditionally? An extreme example is in the code generation phase of a compiler: all code generation operations would be remembered until the whole program had been parsed. After the program was found to be syntactically correct, the code generation operations would be carried out. If, on the other hand, syntax errors were detected, all code generation operations would be discarded. A less extreme example is the parsing of (locally) ambiguous or non-LL(1) languages. If a construct can be identified only after having been parsed in its entirety, it may be necessary to postpone all operations associated with it until that time. This may, for instance, arise in a language allowing multiply labelled statements: in that case an identifier is either the next label or the beginning of the actual statement. The same phenomenon also occurs for case statements allowing an arbitrary number of expressions to be associated with each case.

We now list some advantages and disadvantages of the immediate/conditional model. The advantages are:

---

3) In order to present a non-trivial example, we have taken the liberty of using a slightly more powerful EXP language than defined in 3.4.1; <right-hand-side>s consisting of function calls and functions (like *length* and *replicate*) were not defined there but have an obvious meaning.

- Information about failure can be passed to higher levels; it is thus possible to construct patterns that do not require rescanning of the subject string.
- The programmer has explicit control over the moment operations are performed. This is also a disadvantage however, since it complicates the model.
- The model can be implemented efficiently.

Disadvantages of the immediate/conditional model are:

- Mixing of conditional and immediate operations is problematical. First, the program text no longer reflects the order of events, which greatly reduces its readability. Secondly (and worse), conditional operations are performed in the environment as it exists at the end of the successful pattern match in which they occur. This is the price paid for an efficient implementation: the environment is not saved when a conditional operation is encountered during the pattern match. Consider, for example:

  $P := \text{'ab'} -\!- \mathit{imact}(n:=\text{'X'}) -\!- \text{'c'} -\!- \mathit{cdact}(\mathit{res}:=n) -\!- \mathit{imact}(n:=n+\text{'Y'});$

  During the pattern match *'abc' ? P* the following steps are performed:

  1) The string *'ab'* is recognized.
  2) The operation $n:=\text{'X'}$ is performed.
  3) The string *'c'* is matched.
  4) The operation $\mathit{res}:=n$ is remembered.
  5) The operation $n:=n+\text{'Y'}$ is performed. This amounts to assigning the string *'XY'* to variable *n*.
  6) The pattern match succeeds and all remembered operations (in this example only $\mathit{res}:=n$) are performed in left to right order.

  Variable *res* thus receives *'XY'* as final value, but *'X'* instead of *'XY'* would in many respects have been a more reasonable outcome.
- Each implementation of the immediate/conditional model has to solve the non-trivial problem of not imposing restrictions on the number of simultaneously remembered conditional operations. If an implementation does impose such restrictions, it may not be possible to perform pattern matches in which large numbers of conditional operations occur (such as compiling a whole program in one pattern match). In practice, this would have the highly undesirable effect of forcing the programmer to use immediate operations exclusively and of seriously limiting the usefulness of the backtracking facility.
- It is not visible whether a pattern modifies the global environment on failure or not.

A final illustration of the intricacies of the immediate/conditional model follows. Considering

```
x  := 'A'; y := 'B';
p1 := 'a' -- imact(x:=x+'X') | 'ab' -- cdact(x:=x+y);
p2 := imact(y:=y+'Y') -- 'c';
p  := p1 -- p2;
```

the match *'abc' ? p* succeeds and results in $x = \text{'AXBYY'}, y = \text{'BYY'}$, while *'aef' ? p* fails and results in $x = \text{'AX'}, y = \text{'BY'}$. These results are baroque, to say the least.

3.4.2.2. Formal description

In the immediate/conditional model, the grammar *PAT* is described by the syntax rules:

```
<pattern> ::= <pattern-primary> [ ( '|' | '--' ) <pattern-primary> ] .
<pattern-primary> ::=
      '(' <pattern> ')' |
      <string-literal> |
      <identifier> |
      '*' <identifier> |
      <compound> |
      IMSBAS '(' <pattern-primary> ',' <identifier> ')' |
      CDSBAS '(' <pattern-primary> ',' <identifier> ')' |
      IMACT '(' <expression> ')' |
      CDACT '(' <expression> ')' .
<compound> ::= '<' <integer-constant> ',' <expression> ',' <pattern> '>'.
```

In order to simplify the formal definition, we assume without loss of generality, that all syntactic ambiguities are eliminated by the proper use of parentheses, e.g. $P \mid ((Q--R)--S)$ should be used instead of $P \mid Q--R--S$. In the examples, the non-parenthesized version will be used (for better readability) and the operator '--' is assumed to have higher priority than the operator '|'.

Roughly speaking, the state a pattern match is in is fully[4] specified by the following three parameters:

- The pattern that has to match in order to complete the pattern match successfully.
- The cursor position in the subject string where the above pattern should match.
- An expression that is equivalent to all remembered conditional operations. Evaluating this expression has the same effect as evaluating all remembered conditional operations in left to right order.

The meaning of a pattern can now be expressed in terms of its effects on the above three entities.

More formally, the semantics of patterns is given by the function

$$match: CURS \times EXP \times PAT \rightarrow CURS \times EXP \times PAT^0 \cup FAIL$$

where PAT^0 is equal to $PAT \cup \{null\}$ and *null* is the pattern that always fails. Note that *null* can only come into existence as the result of applying *match*, but that *match* is not defined on it. *match* attempts to transform the (cursor, expression, pattern) triple (C, E, P) into a new triple (C', E', P') as follows. If pattern P matches successfully, the cursor is moved to C', all conditional operations performed during this part of the match are appended to expression E (resulting in E'), and P' is a pattern consisting of all untried alternatives of P. If P fails, *match* should deliver the value *FAIL*. The function *match* to be described below achieves this by performing an *freturn* (failure return) operation. This is equivalent to returning the Boolean value

4) The existence of the global variables *subject_string*, *INITIAL_ENV* and *CURRENT_ENV* is not considered here.

false which can subsequently be tested for by the caller of *match*.

Apart from <compound>s, all notions in the grammar given above correspond to some primitive pattern. An implicit semantic property of the immediate/conditional model made it, however, necessary to add a new type of pattern: As was illustrated previously, attempting 'untried alternatives' of a pattern requires a mechanism to restore a previous state of the pattern match. To achieve this, <compound>s have been introduced as explicit representations of the pattern matching state.

The formal definition uses several auxiliary functions. Functions like *mk_alt* and *mk_comp* construct new elements of *PAT* from given components. For example, let *subject_string* have *'abcdef'* as value, then

```
p := mk_string(0,3);
q := mk_string(3,6);
a := mk_alt(p, q);
```

will assign the pattern *'abc'|'def'* to *a*. The correspondence between such functions and the alternatives in the grammar *PAT* is:

mk_alt(p,q)	→	*p* | *q*
mk_subs(p,q)	→	*p* −− *q*
mk_comp(c,e,p)	→	*<c,e,p>*
mk_string(from,to)	→	<string-literal>
mk_cdsbas(p,id)	→	*cdsbas(p,id)*
mk_imsbas(p,id)	→	*imsbas(p,id)*

Note that *mk_string* constructs a new string by extracting the characters with indices *from*, *from* + 1, . . . , *to* − 1 from the global *subject_string*. Also note that *mk_alt*, *mk_subs*, *mk_comp*, *mk_imsbas* and *mk_cdsbas* do not construct a new pattern if one of their arguments is *null*, e.g.

mk_alt(p,null)	→	*p*
mk_alt(null,q)	→	*q*
mk_subs(p,null)	→	*null*
mk_subs(null,q)	→	*null*
mk_comp(c,e,null)	→	*null*
mk_imsbas(null,id)	→	*null*
mk_cdsbas(null,id)	→	*null*

Similar functions exist for the grammar *EXP*:

mk_assign(id,v)	→	*id* := *v*
mk_expr(e1,e2)	→	*e1* ; *e2*

These functions are used to construct 'remembered expressions' as required for the modeling of conditional subject assignment (see below).

We now present the formal definition followed by a detailed explanation.

```
    proc match(curs, expr, pat)
    ( var curs1, curs2, expr1, expr2, P, P1, Q, Q1;
      var rpat1, rpat2, str, id;

#a# if {{ pat == P:<pattern-primary> '|' Q:<pattern-primary> }}
      then
        if [curs1, expr1, P1] := match(curs, expr, P)
        then
          return([curs1, expr1, mk_comp(curs, expr, mk_alt(P1, Q))])
        elif [curs2, expr2, Q1] := match(curs, expr, Q)
        then
          return([curs2, expr2, mk_comp(curs, expr, Q1)])
        else
          freturn
        fi
#b# elif {{ pat == P:<pattern-primary> '--' Q:<pattern-primary> }}
      then
        if [curs1, expr1, P1] := match(curs, expr, P)
        then
          if [curs2, expr2, Q1] := match(curs1, expr1, Q)
          then
            rpat1 := mk_comp(curs1, expr1, Q1);
            rpat2 := mk_subs(mk_comp(curs, expr, P1), Q);
            return([curs2, expr2, mk_alt(rpat1, rpat2)])
          else
            if P1 ~= null
            then
              if [curs2, expr2, Q1] := match(curs, expr, mk_subs(P1, Q))
              then
                return([curs2, expr2, Q1])
              fi
            fi
          fi
        fi;
        freturn
#c# elif {{ pat == '(' P:<pattern> ')' }}
      then
        return(match(curs, expr, P))
#d# elif {{ pat == str:<string-literal> }}
      then
        if curs1 := litmatch(curs, str)
        then
          return([curs1, expr, null])
        else
          freturn
        fi
#e# elif {{ pat == id:<identifier> }}
      then
        P := INITIAL_ENV.binding(id);
```

```
        if [curs1, expr1, P1] := match(curs, expr, P)
        then
          return([curs1, expr1, P1])
        else
          freturn
        fi
#f# elif {{ pat == '*' id:<identifier> }}
      then
        P := CURRENT_ENV.binding(id);
        if [curs1, expr1, P1] := match(curs, expr, P)
        then
          return([curs1, expr1, P1])
        else
          freturn
        fi
#g# elif {{ pat == '<' curs1:<integer-constant> ','
                     expr1:<expression> ',' P:<pattern> '>' }}
      then
        if [curs2, expr2, P1] := match(integer(curs1), expr1, P)
        then
          return([curs2, expr2, P1])
        else
          freturn
        fi
#h# elif {{ pat == IMSBAS '(' P:<pattern-primary> ',' id:<identifier> ')' }}
      then
        if [curs1, expr1, P1] := match(curs, expr, P)
        then
          CURRENT_ENV.bind(id, mk_string(curs, curs1));
          rpat1 := mk_comp(curs, expr, mk_imsbas(P1, id));
          return([curs1, expr1, rpat1])
        else
          freturn
        fi
#i# elif {{ pat == CDSBAS '(' P:<pattern-primary> ',' id:<identifier> ')' }}
      then
        if [curs1, expr1, P1] := match(curs, expr, P)
        then
          expr2 := mk_expr(expr1, mk_assign(id, mk_string(curs, curs1)));
          rpat1 := mk_comp(curs, expr, mk_cdsbas(P1, id));
          return([curs1, expr2, rpat1])
        else
          freturn
        fi
#j# elif {{ pat == IMACT '(' expr1:<expression> ')' }}
      then
        eval(expr1);
        return([curs, expr, null])
#k# elif {{ pat == CDACT '(' expr1:<expression> ')' }}
```

```
        then
          return([curs, mk_expr(expr, expr1), null])
    #1# else
          ERROR
        fi
      );
```

The cases #a# to #1# correspond to the various alternatives in the grammar *PAT*. Each case will now be discussed in more detail:

a) If the pattern is an alternation of the form $P \mid Q$, an attempt is first made to match the pattern P. If successful, it delivers a new cursor, expression and pattern. The new cursor value corresponds to the point to which the pattern P has proceeded in the subject string. The new expression consists of the old expression augmented by the conditional operations encountered during the evaluation of P. The new pattern (P 1) corresponds to the remaining (untried) alternatives of P. The pattern P 1 $\mid Q$ is delivered by the evaluation of $P \mid Q$. If the match of P fails, an attempt is made to match Q. If the latter fails, $P \mid Q$ fails. If it succeeds, again, a new cursor, expression and pattern are delivered. In this case the pattern Q 1 is delivered by the evaluation of $P \mid Q$.

b) If the pattern is a subsequentiation of the form P —— Q, the operations performed are of a similar nature as the ones performed for alternation.

c) If the pattern consists of a pattern enclosed in parentheses, the enclosed pattern is attempted.

d) If the pattern is a <string-literal>, it should occur in the subject string, starting at the current cursor position. This is taken care of by the function *litmatch*(*s*). If, for example, the subject string is *'abcd'*, then *litmatch*(1,*'bc'*) will deliver the value 3, while *litmatch*(2,*'bc'*) will fail.

e) If the pattern is an <identifier>, the corresponding value in the **initial environment** is used as pattern. In this way the notion of **pattern construction**, as it exists, for example, in SNOBOL4 is modeled. Before a pattern match starts, the expression describing the pattern is used to build a data structure which is subsequently used to control the recognition process. In the formal description, this is reflected by the use of the initial values of the variables occurring in the pattern.

f) If the pattern is an unevaluated expression of the form $*x$, the value of x in the current environment is used as pattern. This introduces the possibility of context sensitive patterns, that are modified dynamically during the pattern match.

g) It was already explained above that <compound-pattern>s are a way of representing the pattern matching state. They consist of a cursor value, an expression describing remembered conditional operations, and a pattern. Evaluation of a <compound-pattern> amounts to evaluating its pattern component, starting at its cursor value. The role of the expression component is further explained in points i) and k) below.

h) If the pattern is an immediate subject assignment of the form *imsbas*(P,x), an attempt is made to match pattern P. If this is successful, the part of the subject recognized by P is immediately assigned to variable x.

i) If the pattern is a conditional subject assignment of the form *cdsbas*(P,x), an attempt is made to match pattern P. If this is successful, an assignment statement is constructed that will eventually (i.e. at the end of the pattern match) assign the

recognized part of the subject string to variable x.

j) If the pattern is an immediate action, it is evaluated.

k) If the pattern is a conditional action, it is appended to the current list of conditional operations.

l) In all other cases the pattern is syntactically incorrect.

It is instructive to trace the series of events taking place during a simple pattern match (see Figure 3.2). For reasons of readability, arguments and results that are equal to the empty string are denoted by a single space character.

```
match(0, ,('a'|'ab')--'e')
  match(0, ,('a'|'ab'))
    match(0, ,'a'|'ab')
      match(0, ,'a')
      returns [1, , ]
    returns [1, , <0, ,'ab'>]
  returns [1, , <0, ,'ab'>]
  match(1, ,'e')
  fails
  match(0, ,<0, ,'ab'>--'e')
    match(0, ,<0, ,'ab'>)
      match(0, ,'ab')
      returns [2, , ]
    returns [2, , ]
    match(2, ,'e')
    returns [3, , ]
  returns [3, , ]
returns [3, , ]
```

Figure 3.2. Trace of pattern match *'abe' ? ('a'|'ab')--'e'*.

3.4.3. The recovery model

3.4.3.1. Overview

This new model was inspired by the concept of recovery blocks. In [Randell75] it is shown how the reliability of a program can be increased by inserting **acceptance tests** at appropriate places in the program. Whenever such a test fails, the program state is restored to a state at a previous well-defined point, and an attempt is made to perform the same computation using an alternative algorithm. This process is repeated until either the result satisfies the acceptance test or no more alternative algorithms are available. In the latter case, the failure has to be handled at a higher, more global, level in the program. To this end the program is split up into (nested) recovery blocks. Each recovery block starts with a 'begin of recovery block' statement indicating that a roll-back to this point may take place, and ends with an acceptance test. The body of the recovery block consists of the various alternative algorithms to be tried.

This approach clearly bears a certain resemblance to backtracking. Given the problems with the immediate/conditional model, two further observations led to the idea that backtracking could be completely replaced by the recovery block method:

- It is desirable for the programmer to have complete control over the scope of backtracking.
- A failing pattern component should only be allowed to change the local (innermost) but not the global environment. In this way, the side-effects of failing patterns can be localized.

It is possible to interpret the recovery model in such a way that the above two requirements are satisfied and that the disadvantages of the immediate/conditional model disappear. First, alternation and subsequentiation are stripped of their capability to remember untried alternatives. Secondly, a single operation is defined for the control of both backtracking and environment modification. For this purpose the construction:

try $<id_1, \ldots, id_n> p_1, \ldots, p_m$ **until** p_0 **yrt**

is introduced, where $p_1, \ldots, p_m$ are patterns to be tried successively and p_0 is a pattern to be applied after the successful matching of one of the patterns $p_1, \ldots, p_m$. If $p_0, p_1, \ldots, p_m$ fail, the environment is restored to the state it was in at the moment the **try** construction was entered, **except for the values of the variables** $id_1, \ldots, id_n$. These variables can be used as a communication channel or **window** between the patterns $p_0, p_1, \ldots, p_m$. In this way, each alternative can inspect information supplied by previous (necessarily unsuccessful) attempts. Note, that the variables in the window have to be declared elsewhere; the occurrence of a variable in a window only establishes the fact that its value will not be recovered.

Since all language aspects related to backtracking have been concentrated in the try construct, the alternation and subsequentiation of patterns (represented by '|' and '--' in the immediate/conditional model) can now be expressed by the ordinary Boolean operators *and* and *or*. We shall, however, continue to use the same notation in order to make the comparison between the two models easier.

The recovery model is superior to the immediate/conditional model in several respects:

- It is possible to save information gathered during a failing pattern match in a way which is more structured than in the immediate/conditional model.
- It is much simpler than the immediate/conditional model: the number of operators is smaller and no unexpected effects can occur by mixing immediate and conditional operations. In the immediate/conditional model the programmer has to be aware of unwanted 'backing into previous alternatives'. Consider the pattern:

```
'[' --
    ( expr |
      imact(error: = 'invalid index')
    ) --
']'
```

applied to the string *'[a+b}'*. On encountering the invalid symbol '}', automatic backtracking causes the (inappropriate) error message to be assigned

to the variable *error*. This cannot occur in the recovery model, since the programmer has to ask for backtracking explicitly. In the recovery model, one can formulate the above problem in two ways. The error message alternative can either be discarded, leading to

```
'[' --
    ( expr |
      act(error: ='invalid index')
    ) --
']'
```

(where '|' is used as the ordinary Boolean *or* operator and the action is only evaluated if *expr* fails), or it can be remembered explicitly

```
'[' --
    try expr,
        act(error: ='invalid index')
    until ']'
    yrt
```

□ A reasonably efficient implementation is possible using a **cache** containing the modifications to the enclosing environment. Such a cache mechanism evenly distributes the work of saving and restoring the program state, i.e. the amount of work increases in proportion to the number of modifications to the original state. The absence of automatic backtracking renders this implementation method feasible. It is crucial that the beginning and end of a try construct are known so that the period during which modifications have to be remembered is well-defined.

An example will illustrate the main properties of the recovery model. Consider

```
p := ('a' | 'ab') -- 'c' .
```

Both *'abc' ? p* and *'abd' ? p* fail, since backtracking is not automatic. If, on the other hand, *p* is defined by

```
p := try <> 'a', 'ab' until 'c' yrt
```

then *'abc' ? p* will succeed.

3.4.3.2. Formal description

In the recovery model, the pattern language *PAT* is described by the following syntax rules

```
<pattern> ::= <pattern-primary> [ ( '|' | '--' ) <pattern-primary> ] .
<pattern-primary> ::=
      '(' <pattern> ')' |
      <string-literal> |
      <identifier> |
      '*' <identifier> |
      SBAS '(' <pattern-primary> ',' <identifier> ')' |
      ACT '(' <expression> ')' |
      TRY <window> {<pattern> ','}+ UNTIL <pattern> YRT .
<window> ::= '<' [ <identifier> ',' ]* '>' .
```

The syntax rules for <expression>s and their meaning have already been described in Section 3.4.1. The semantics of patterns is described by the function

$$match: CURS \times PAT \rightarrow CURS \cup FAIL.$$

This function attempts to match a given pattern starting at a given cursor position and delivers a new cursor position. In contrast to the immediate/conditional model, it is not necessary to accumulate expressions describing conditional operations, nor to remember untried alternatives. We now present the semantics of pattern matching in the recovery model.

```
proc match(curs, pat)
( var curs1, curs2, P, P1, Q, Q1, pats, val;
  var str, expr, id, windowvars, SAVED_ENV;

#a# if {{ pat == P:<pattern-primary> '|' Q:<pattern-primary> }}
    then
      if curs1 := match(curs, P)
      then
        return(curs1)
      elif curs2 := match(curs, Q)
      then
        return(curs2)
      else
        freturn
      fi
#b# elif {{ pat == P:<pattern-primary> '--' Q:<pattern-primary> }}
    then
      if curs1 := match(curs, P)
      then
        if curs2 := match(curs1, Q)
        then
          return(curs2)
        else
          freturn
        fi
      fi;
      freturn
#c# elif {{ pat == '(' P:<pattern> ')' }}
    then
      return(match(curs, P))
#d# elif {{ pat == str:<string-literal> }}
    then
      if curs1 := litmatch(curs, str)
      then
        return(curs1)
      else
        freturn
      fi
#e# elif {{ pat == id:<identifier> }}
    then
```

```
        if curs1 := match(curs, INITIAL_ENV.binding(id))
        then
          return(curs1)
        else
          freturn
        fi
#f# elif {{ pat == '*' id:<identifier> }}
      then
        if curs1 := match(curs, CURRENT_ENV.binding(id))
        then
          return(curs1)
        else
          freturn
        fi
#g# elif {{ pat == SBAS '(' P:<pattern-primary> ',' id:<identifier> ')' }}
      then
        if curs1 := match(curs, P)
        then
          CURRENT_ENV.bind(id, mk_string(curs, curs1));
          return(curs1)
        else
          freturn
        fi
#h# elif {{ pat == ACT '(' expr:<expression> ')' }}
      then
        eval(expr);
        return(curs)
#i# elif {{ pat == TRY '<' windowvars: { <identifier> ',' }* '>'
                    pats:{<pattern> ','}+
                    UNTIL Q: <pattern> YRT }}
      then
        SAVED_ENV := copy(CURRENT_ENV);
        for P in pats
        do if curs1 := match(curs, P)
           then
             if curs2 := match(curs1, Q)
             then
               return(curs2)
             fi
           fi;
           for id in windowvars
           do SAVED_ENV.bind(id, CURRENT_ENV.binding(id)) od;
           CURRENT_ENV := copy(SAVED_ENV);
        od;
        freturn
#j# else
        ERROR
      fi
  );
```

Cases #c#, #d#, #e#, #f#, #g#, #h# and #j# have direct counterparts in the immediate/conditional model and need no further explanation. Only the remaining cases will be discussed.

a) If the pattern is an alternation of the form $P \mid Q$, P is attempted first. If successful, the new cursor value is returned. If not successful, Q is attempted. Again, if Q is successful, the cursor delivered by Q is returned. In all other cases $P \mid Q$ fails. It is interesting to compare the simplicity of this alternation operator with the much more complex alternation operator in the immediate/conditional model.

b) If the pattern is a subsequentiation, it is handled in a similar fashion. Again, compare this operator with its counterpart in the immediate/conditional model.

i) If the pattern is a **try** construct of the form

try $< id_1, \ldots, id_n > p_1, \ldots, p_m$ **until** p_0 **yrt**

the current environment is copied first. The patterns $p_1, \ldots, p_m$ are attempted next from left to right. If one of them matches **and** p_0 matches, the try construct as a whole succeeds. When either one of $p_1, \ldots, p_m$ fails or p_0 fails, the environment is restored **except** for the variables $id_1, \ldots, id_n$ whose values are retained. It is assumed here that all variables in the window have a well-defined value at the moment the try construct is entered.

3.5. Unification of pattern and expression language

In the preceding sections a careful analysis was made of two backtracking models. By choosing the right primitive operations we were able to develop a simple, but powerful backtracking method. Is it possible to introduce further simplifications? To do this, we first explicitly formulate our basic assumptions:

- A pattern matching model is characterized by a language *PAT* of patterns (with associated definition function *match*) and a language *EXP* of expressions (with associated definition function *eval*).
- The definition function *match* assumes the existence of a subject string and keeps track of the progress of the match by means of an integer-valued cursor. The only assumption regarding the data type of the subject string is made by the function *litmatch* that does the actual string matching.

These assumptions immediately lead to two guidelines for making further simplifications:

- Unify the languages *EXP* and *PAT*.
- Remove all dependencies on strings as the domain of pattern matching.

The unification of the languages *EXP* and *PAT* amounts to removing the **linguistic dichotomy** in pattern matching languages, so eloquently described in [Griswold80]. This unification can be achieved by eliminating the language *PAT* completely. This is done by modeling all operations in *PAT* by operations in *EXP* and by extending *EXP* when necessary in the process. Typical examples of this modeling are:

Operation in *PAT*:		Modeled in *EXP* by:
pattern match fails or succeeds	→	add success/fail mechanism to *EXP*.
string-literal	→	*lit* (string-literal), where *lit* is a built-in string matching procedure defined in *EXP*.
alternation	→	Boolean *or*.
subsequentiation	→	Boolean *and*.
action	→	expression.
unevaluated expression	→	procedure call.
subject assignment	→	ordinary assignment.
try construct	→	add try construct to *EXP*.

It is surprising that this redefinition of *EXP* also eliminates all dependencies on string pattern matching: the special role played by the subject string and cursor is no longer cared for and the recovery mechanism (embodied in the try construct) is sufficiently powerful that it permits the saving and restoring of variables with arbitrary values. Such variables could be used to describe the progress of a pattern match in some arbitrary, user-defined, domain.

These ideas form the conceptual basis for subsequent chapters.

3.6. References for Chapter 3

[Dahl70] Dahl, O-J, Myhrhaug, B. & Nygaard, K., "SIMULA Information, Common Base Language", Norwegian Computing Centre, S-22, 1970.

[Druseikis75] Druseikis, F.C., "The design of transportable interpreters" (dissertation), S4D49, University of Arizona, 1975.

[Doyle75] Doyle, J.N., "A generalized facility for the analysis and synthesis of strings, and a procedure-based model of implementation" (thesis), S4D48, University of Arizona, 1975.

[Farber64] Farber, D.J., Griswold, R.E. & Polonsky, I.P., "SNOBOL, a string manipulation language", *Journal of the ACM*, **11** (1964) 1, 21-30.

[Gimpel73] Gimpel, J.F., "A theory of discrete patterns and their implementation in SNOBOL4", *Communications of the ACM*, **16** (1973) 2, 91-100.

[Griswold71] Griswold, R.E., Poage, J.F. & Polonsky, I.P., *The SNOBOL4 Programming Language*, Second Edition, Prentice-Hall, Englewood Cliffs, N.J., 1971.

[Griswold76] Griswold, R.E. & Hanson, D.R., "An overview of the SL5 programming language", SL5 project document S5LD1b, The University of Arizona, Tucson, Arizona, October 9, 1976.

[Griswold80] Griswold, R.E. & Hanson, D.R., "An alternative to the use of patterns in string processing", *Transactions on Programming Languages and Systems*, **2** (1980) 2, 153-172.

[Randell75] Randell, B., "System structure for software fault tolerance", in: *Proceedings of the International Conference on Reliable Software*, SIGPLAN Notices, **10** (1975) 6, 437-449.

[Stewart75] Stewart, G.F., "An algebraic model for string patterns", *Second Symposium on Principles of Programming Languages*, ACM, 1975, 167-184.

4. AN OVERVIEW OF THE SUMMER PROGRAMMING LANGUAGE[1]

4.1. Introduction

The language SUMMER has been designed for the solution of problems in text processing and string manipulation. It consists of a relatively small kernel which has been extended in several directions. The kernel supports:

- integers
- strings
- classes
- procedure and operator definitions
- success-directed evaluation
- control structures
- recovery of side-effects.

It has been extended with:

- reals
- files
- arrays (sequences of values)
- tables (associative memories)
- pattern matching
- string synthesis.

Pattern matching has been completely integrated with the success-directed expression evaluation mechanism. It will be shown that the operations in the kernel are sufficient to allow generalization of pattern matching in two directions:

- Simultaneous pattern matches can be expressed, which mutually affect each other.
- Pattern matching can be generalized to domains other than strings.

In the following sections the novel features of SUMMER and the motivation for their inclusion in the language will be discussed. Furthermore, a simplified version of the pattern matching extension is described in some detail.

4.2. Success-directed evaluation and control structures

The expression evaluation mechanism of SUMMER is somewhat unusual and merits special attention. Expressions consist of a juxtaposition of operators (like the addition operator: '+' or the string concatenation operator: '||') and operands (like the numeric constant 10, the string constant *'abc'*, the identifier *x* or the procedure call *p*(10,*x*)). Some operations can deliver only a value, but others can potentially **fail**. The syntactic form of an expression completely determines the steps to be taken when an operation in a subexpression fails. The evaluation of simple expressions like

1) This chapter is a revised version of [Klint80].

```
a + fibonacci(7)
x := duplicate(c, 3) || ' times'
x > y
x := duplicate(an_identifier(s), 5)
```

is aborted immediately when an operation in a subexpression fails and failure is signalled to constructs surrounding the expression. In the last example, failure of the expression *an_identifier(s)* may abort evaluation of the whole expression before *duplicate* even gets called. Note that failure is a transient entity and must be captured at the moment it occurs. The evaluation of more complex constructs in the case of a failing subexpression depends again completely on the syntactic form of each construct. There are three cases:

1) The construct is capable of handling the failure itself. This is the case if the failing expression *E* occurs in contexts like:

```
if E then . . . else . . . fi
while E do . . . od
E | . . .                    {Boolean (McCarthy) or operator}
```

2) The construct is not capable of handling the failure itself, but is (perhaps dynamically) enclosed in a construct with that capability, like:

```
E & . . .                    {Boolean (McCarthy) and operator}
return(E)                    {return value from a procedure}
```

In this way failure can be passed to the caller of the procedure in which the failing expression occurs (see below).

3) Neither of the above two cases applies. This results in abnormal program termination with the error message 'Unanticipated failure'. In

```
x := read(input); print(x);
```

the call to the read procedure may fail (on end of file). This failure will not be detected by the program itself and hence execution of the program will be aborted.

This expression evaluation scheme was designed to be concise and powerful, but at the same time an attempt was made to protect the programmer against unanticipated or unwanted failure.

Conciseness is obtained in two ways.

In the first place, by computing a value and a failure signal in the same expression. This allows, for example

```
while line := read(input) do . . . od
```

instead of

```
while ~eof(input)
do line := read(input);
   if io_errors(input) then . . . fi;
   . . .
od;
```

Secondly, by disregarding the **source of failure** and focusing attention on the **absence of failure** (i.e. success) during the evaluation of the expression. From now on we will use 'succeeds' as a synonym for 'does not fail and delivers a value'. Consider:

```
if (read(input) || read(input)) ~= expected
then
  error('Bad input')
fi
```

where '~=' denotes the inequality comparison between strings and *expected* has the expected input string as value. Three sources of failure can be identified here: the two read operations and the inequality test can fail. The programmer, however, is in most cases interested only in the fact that the input file does not conform to his expectations. This is more manifest in the above formulation than in:

```
l1 := read(input);
if eof(input)
then
  error('Bad input')
else
  l2 := read(input);
  if eof(input) | (l1 || l2 ~= expected)
  then
    error('Bad input')
  fi
fi
```

In principle, this argument works in two directions: since the source of failure may be lost, the programmer may be misled about the **actual** source of failure. It is our experience that this seldom happens, and in all cases where the distinction is important, it can be expressed easily.

Protection is achieved by prohibiting unanticipated failure. This turns out to be a frequent source of run-time errors, which always correspond to 'forgotten' or 'impossible' failure conditions. A direct consequence of this protection scheme is that one can write **assertions** (i.e. expressions which should never fail) in a program. A run-time error occurs if such an assertion is false.

Another noteworthy consequence of this evaluation mechanism is its ability to let a procedure report failure to any procedure which called it (in)directly. This effect is obtained by adhering to the programming convention that procedure bodies have the form E_1 & $\cdots$ & E_n. If one of the expressions E_i fails, this failure is passed to the caller of the current procedure. If that calling procedure has the same form, it will not itself handle the failure but pass it on to its caller. In this way, low-level procedures need not be aware of failure at all and high-level procedures can detect the failure and take appropriate measures. Some programming languages (e.g. ADA, CLU) have special facilities for handling exceptions of this kind; in SUMMER they can be handled by the standard expression evaluation mechanism.

4.3. Recovery of side-effects

For the solution of problems such as heuristic searching, or parsing languages with context-sensitive or non-LL(1) grammars, it is often necessary to **attempt** a solution and to recover from its side-effects if that attempt is not successful. Many schemes have been proposed for the formulation of such **backtracking** algorithms, but most involve either opaque control structures or allow unsatisfactory control over modifications of the program state (i.e. global variables).

SUMMER provides a special language construct for recovering from the side-effects caused by the evaluation of a failing expression. This construct has the form

try E_1 , E_2 , . . . , E_n **until** E_0 **yrt**

and is, to a first approximation, equivalent to

$(E_1 \ \& \ E_0) \ | \ \cdots \ | \ (E_n \ \& \ E_0)$

Before the evaluation of each $(E_i \ \& \ E_0)$ starts, the complete program state (values of all variables, status of all input/output operations) is saved. If the evaluation of the subexpression succeeds, the saved program state is discarded and the try construct as a whole succeeds. If the evaluation fails, the saved program state is restored and evaluation of $(E_{i+1} \ \& \ E_0)$ is attempted in a similar manner. The try construct fails if none of the subexpressions succeeds. Completely automatic backtracking can be achieved by nesting try constructs. This simple scheme is well suited to the formulation of problems occurring in pattern matching applications as will be seen in 4.5.2.

There are two exceptions to the rule that the whole program state is restored when an expression contained in a try construct fails:

- Operations on an input/output stream that corresponds to an interactive terminal are not recovered. In many situations it is not desirable to recover these streams, and in some cases the meaning of such a recovery may be non-obvious or confusing. In SUMMER these streams can be used to interactively control and monitor the backtracking process.
- The local variables of the procedures in which the try construct occurs are not recovered. In this way information about the reason for failure can survive the failure itself. In Section 3.4.3 a similar effect was achieved using a 'window' of variables whose values were never recovered.

It is obvious that saving and restoring complete program states would lead to intolerable inefficiencies when implemented naively. Fortunately, there exists an implementation technique that eliminates most of the run-time overhead involved. The **recovery cache**, which was originally invented to increase software reliability [Randell75], has been adapted to act as a device for monitoring program state modifications in those situations where it may be necessary to restore a previous program state (i.e. while evaluating expressions contained in a try construct). A recovery cache consists of (name, value) pairs. The name part may refer to simple variables, array elements, class components or input/output streams. A new cache is created when the evaluation of a try construct begins, and from that moment, all assignments to variables are monitored. Whenever an assignment is about to be made to a variable whose name does not yet occur in the cache, its name and its old value **before** the assignment are entered into the cache. Assignments to components of structured objects (arrays, class instances) and modifications of input/output streams are

registered in a similar way. The recovery cache is discarded completely if the evaluation of the try construct succeeds, but in the case of failure, the information in the cache is used to restore the program to its state at the moment that the cache was created (i.e. the try construct was entered). Since recovery caches may be nested, 'discarding' may mean: merging the information in the current cache with that in the previous one. In this manner, the information in the previous cache is still sufficient to describe all modifications which have been made since that cache was created.

4.4. Procedures, operators and classes

In this section the remaining features of the SUMMER kernel are summarized.

Procedures have a fixed number of parameters, which are, in principle, passed by value. Procedures may either return zero or more values, or they may fail. The former is achieved by a return-expression of the form *return* or *return*(*expression*). The latter is achieved by a return-expression of the form *freturn* (failure return). Returning a failing expression is equivalent to a failure return (e.g. *return*(3 > 4) is equivalent to *freturn*).

An **operator** is defined by associating a user-defined operator symbol with a procedure which has one or two parameters.

Classes are the only available data structuring mechanism and have been inspired by the class concept in SIMULA [Dahl70]. A class declaration describes the properties of a group of related entities. One particular entity is described by **instantiating** (i.e. making an **instance** of) the class declaration and filling in the specific properties of that entity. For example, all entities of the class *person* may have a *name* and an *age*. An individual person can be described by an instance of the class *person* with the appropriate details (e.g. 'John', 36) filled in. This scheme equally applies to built-in classes like *integer* and *string*, as well as to user-defined classes like the class *person* just described. **All values in SUMMER are thus instances of some class.**

Nothing has so far been gained when comparing classes with, for example, records in PASCAL. The major difference between the two stems from the fact that PASCAL records are **passive** (i.e. a record resembles a variable and is only a container of values) and **unprotected** (i.e. all components of a record are freely accessible) while classes are not. A class can be considered to be **active**, since it may contain locally declared procedures to manipulate the information in each instance of that class (e.g. increment a person's age) or to perform computations based on that information. A class is **protected** since the access to the individual information in the class instance is completely controlled by its class definition. A simple definition of *person* might be:

```
class person(name, age)
begin fetch name, age;
      store age;
end person;
```

Here *name* and *age* may be used from the outside of the class instance, but only *age* may be modified, i.e. may occur on the left hand side of an assignment. Some examples of the use of this class are:

```
friend := person('John', 36);
print( friend.name);
friend.age := friend.age + 1;
```

A more restrictive version of *person* might be:

```
class person(name, age)
begin fetch name, age;
      store age : grow_older;
      proc grow_older(new_age)
      ( if new_age >= age
        then
          age := new_age
        else
          print('Did you find the elixir of life?')
        fi
      );
end person;
```

This definition prescribes that all assignments to *age* are channeled through the procedure *grow_older* which ensures the monotonicity of age. Instances of this class are used in precisely the same manner as in the examples given previously.

A second example of class declarations has to do with the generation of unique labels of the form 'L1', 'L2' and so on. This might, for example, be used in a compiler. A declaration for *UniqueLabels* looks like:

```
class UniqueLabel(prefix, start)
begin fetch generate, reset;
      var progress;
      proc generate()
      ( progress := progress + 1;
        return(prefix || string(progress))
      );
      proc reset() ( progress := start );

init: progress := start
end UniqueLabel;
```

Some applications of this class are:

```
L := UniqueLabel('L000', -1);
M := UniqueLabel('M', 10);
print(M.generate);          # prints 'M11' #
print(L.generate);          # prints 'L0000' #
print(M.generate);          # prints 'M12' #
M.reset;
print(M.generate);          # prints 'M11' #
```

Summarizing, a class declaration consists of:

- A class name and a list of formal parameters. The class name is used as name for the creation procedure for instances of the class. The actual parameters are used to provide initial values for that instance.
- Fields, which are used either to contain information related to the instance (e.g. the *name* and *age* fields in an instance of the class *person*), or to hold information local to the class instance (the variable *progress* in an instance of *UniqueLabel*). Permissions for accessing and/or modifying the fields of a class instance from the outside have to be stated explicitly by using **fetch** and **store**.

The fields of a class are accessed by means of the 'dot' notation. The type of the left operand in a field selection is used to disambiguate 'overloaded' fields, for which definitions occur in more than one class.

Some additional features exist in the language to accommodate the use of classes. In sequences like

$$a := S.x;\ b := S.y;\ c := S.z(10)$$

it is convenient if the prefix '*S*.' can be factored out. PASCAL uses the construct

with *record_variable* **do begin** . . . **end**

for this purpose. All field references that occur inside '**begin** . . . **end**' are automatically prefixed with *record_variable*. In this notation the above example would become:

with *S* **do begin** $a := x;\ b := y;\ c := z(10)$ **end**

Unfortunately, this is not sufficient for the applications we have in mind, where it is not unusual for many procedures to operate on the same class instance. This is illustrated by a set of parsing procedures that operate on one subject string. The PASCAL approach has the disadvantage that this common class instance must either be passed as an argument to all procedures involved or must be assigned to a global variable; all procedure bodies must in that case be enclosed in a **with** construct. This problem can be circumvented as follows. The SUMMER construct[2]

scan *S* **for** *E* **rof**

introduces a new incarnation of a common variable ('*subject*') each time the construct is encountered at run-time and assigns the class instance *S* to it. All occurrences of fields from the class to which *S* belongs are now prefixed with the common global variable in the same way as is done in PASCAL. The **scan** construct is more general, however, in the sense that it not only affects *E* itself, but also all procedures called directly or indirectly as a result of the evaluation of *E*, while in PASCAL the effect is restricted to the expressions which are statically enclosed in the body of the **with** construct. If the scan construct is used in a nested fashion, the previous value of the common global variable is saved and restored properly on exit from the current scan construct. This also applies to the case when the scan construct is left prematurely by means of a return statement. In summary, the scan construct introduces a restricted form of **dynamic binding**.

2) Inspired by the 'scan S using E' construct in Icon [Griswold79].

4.5. A pattern matching extension

4.5.1. String Pattern Matching

We will now show how a string pattern matching system can be build on top of the SUMMER kernel. Pattern matching is done on a string *text* which is indexed by an integer *cursor*. For the sake of this discussion a very simple system will be defined, which only supports the following operations:

text: Gives the value of *text*.

cursor: Gives the current value of *cursor*.

lit(*S*): Literally recognize the string *S*. If *S* occurs as substring in *text* at the current cursor position, deliver *S* as value and move the cursor beyond *S*. Otherwise report failure.

break(*S*): Recognize a substring of *text* that starts at the current cursor position and consists entirely of characters not occurring in *S* and is followed by a terminating character which does occur in *S*. If such a substring exists, return it (without the terminating character) and move the cursor to the terminating character. Fail if such a substring does not exist.

span(*S*): Recognize the longest non-empty substring of *text* that starts at the current cursor position and consists entirely of characters which occur in *S*. If such a substring exists, then return it as value and move the cursor beyond it. Fail if such a substring does not exist.

The following class definition implements this pattern matcher:

```
class scan_string(text)
begin fetch lit, break, span, text, cursor;
      var cursor;
      proc lit(s)
      ( if cursor + s.size <= text.size & text.substr(cursor, s.size) = s
        then
          cursor := cursor + s.size ;
          return(s)
        else
          freturn
        fi
      );
      proc break(s)
      ( var newcursor := cursor;
        while newcursor < text.size
        do if contains(s, text[newcursor])
           then
             var result := text.substr(cursor, newcursor - cursor);
             cursor := newcursor;
             return(result)
           fi;
           newcursor := newcursor + 1
        od;
        freturn
      );
```

```
    proc span(s)
    ( var newcursor := cursor;
      while newcursor < text.size & contains(s, text[newcursor])
      do
        newcursor := newcursor + 1
      od;
      if newcursor > cursor
      then
        var result := text.substr(cursor, newcursor - cursor);
        cursor := newcursor;
        return(result)
      else
        freturn
      fi
    );
    proc contains(s, c)
    ( var c1;
      for c1 in s
      do
        if c1 = c then return fi;
      od;
      freturn;
    );
init: cursor := 0;
end scan_string;
```

The following example illustrates how identifiers starting with the letter 'X' can be recognized:

```
proc identifier(s)
( var t := scan_string(s);
  return(t.lit('X') & (t.span(letter_or_digit) | t.lit('')))
)
```

In these examples we assume *letter* = *'abcdefghijklmnopqrstuvwxyz'*, *digit* = *'0123456789'* and *letter_or_digit* = *letter* || *digit*. Note that the normal Boolean operators *and* ('&') and *or* ('|') are used for combination. Hence there will be no backtracking or reversal of effects if the match fails. The expression *t.lit('')* always succeeds and covers the case when the identifier consists of a single *'X'*. This example can be rewritten in a more concise form if we use the scan construct:

```
proc identifier(s)
( scan scan_string(s)
  for
    return(lit('X') & (span(letter_or_digit) | lit('')))
  rof
)
```

A final example may illustrate the use of the value returned by pattern matching procedures. The problem is to extract all letters from a given string. For example *'a,b,c'* gives *'abc'*. A procedure to achieve this can be defined as follows:

```
proc extract_letter(s)
( var result := '';
  scan scan_string(s)
  for
    while break(letter)
    do result := result || span(letter) od
  rof;
  return(result)
)
```

In SUMMER, pattern matching and backtracking have been completely separated. **It came as a shock to us to discover that the majority of pattern matching problems which we had solved previously by means of implicit backtracking, could be solved without any backtracking at all!** Many problems of practical importance can be solved using LL(k) or LR(k) techniques and are at worst only locally ambiguous. Using completely automatic backtracking as a parsing technique is rather wasteful under such circumstances and this suggests that the close interaction between pattern matching and backtracking, which can be found in several languages, should be reconsidered. See Chapter 3 for an extensive discussion of this topic.

How can pattern matching with automatic backtracking be obtained? Consider the expression:

(lit('ab') | lit('a')) & lit('bc')

In the pattern matcher developed above, the alternative *lit('a')* is discarded as soon as a substring starting with *'ab'* is encountered, since we are using McCarthy *and* and *or* operations {4.2}. The string *'abc'* cannot be recognized in this way. But if we rewrite the expression as

```
try lit('ab'), lit('a')
    until lit('bc')
yrt
```

then the side-effect recovery mechanism implicit in the try construct automatically restores the initial cursor value and attempts the second alternative if *lit('bc')* fails the first time. No special attention need be paid to the cursor: it is an ordinary variable which is saved and restored automatically by the try construct just as any other variable.

4.5.2. Generalized pattern matching

In most pattern matching systems there is only one subject string involved in the pattern match. In our scheme, this restriction can be removed without introducing any new concepts as an example will show. The following (rather artificial) problem is to ensure that two strings $S1$ and $S2$ conform to the following rules:

- $S1$ is of the form $c_1;c_2;\ \cdots\ ;c_n;$ where c_i is a (perhaps empty) sequence of arbitrary characters except for the character ';'.
- For a given $S1$, $S2$ has the form $d_1d_2\ \cdots\ d_n$, and either $d_i = c_i$ or $d_i = reverse(c_i)$ holds. Acceptable values for $S2$ with $S1$ equal to *'ab;cde;f;'* are *'abcdef'*, *'abedcf'*, *'bacdef'* and *'baedcf'*.

The following program checks whether a given *S1* and *S2* satisfy this relation:

```
s1 := scan_string(S1);
s2 := scan_string(S2);
scan s1
for
  while (n := break(';')) & lit(';')
  do
    if ~ scan s2 for lit(n) | lit(reverse(n)) rof
    then
        error('check fails')
    fi
  od
rof;
if s1.cursor = S1.size & s2.cursor = S2.size
then
  print('check succeeds')
else
  error('check fails')
fi
```

Each *scan_string* object maintains its own cursor. The innermost scan construct operates each time on the same *scan_string* instance *s2* whose cursor value gets modified. This allows the innermost pattern match to continue where it left off the previous time.

From the preceding paragraphs it will be clear that pattern matching as presented here, does not depend on the fact that strings are used as the basic unit of recognition. One can, for example, easily imagine pattern matching on an array of strings. The 'cursor' must then be replaced by a pair of values to maintain the current position, and basic scanning procedures like *xlit*, *ylit*, *xspan* and *yspan* must be defined. It may be expected that a system like ESP^3 [Shapiro74] can be defined in a straightforward manner using the primitives from the SUMMER kernel.

4.6. Related work

SUMMER was inspired by and profited from ideas in SNOBOL4 [Griswold71] and SL5 [Griswold76]. The method adopted for the integration of pattern matching and expression evaluation (see Chapter 3) was inspired by Icon [Griswold79].

The major ideas introduced by SUMMER are the evaluation model which prohibits unanticipated failure, the recovery from side-effects in failing expressions, the use of recovery caches as an implementation technique and the separation of pattern matching and backtracking which allows more general pattern matching in domains other than strings.

4.7. References for Chapter 4

[Dahl70] Dahl, O-J, Myhrhaug, B. & Nygaard, K., "SIMULA Information, Common Base Language", Norwegian Computing Centre, S-22, 1970.

[Griswold71] Griswold, R.E., Poage, J.F. & Polonsky, I.P., *The SNOBOL4 Programming Language*, Second Edition, Prentice-Hall, Englewood Cliffs, N.J., 1971.

[Griswold76] Griswold, R.E. & Hanson, D.R., "An overview of the SL5 programming language", SL5 project document S5LD1b, The University of Arizona, Tucson, Arizona, October 9, 1976.

[Griswold79] Griswold, R.E. & Hanson, D.R., "Reference Manual for the Icon Programming Language", TR 79-1, The University of Arizona, Tucson, Arizona, 1979.

[Klint80] Klint, P., "An overview of the SUMMER programming language", *Conference Record of the 7th Annual ACM Symposium on Principles of Programming Languages*, ACM, 1980, 47-55.

[Randell75] Randell, B., "System structure for software fault tolerance", in: *Proceedings of the International Conference on Reliable Software*, SIGPLAN Notices, **10** (1975) 6, 437-449.

[Shapiro74] Shapiro, L.G., "ESP^3: a language for the generation, recognition and manipulation of line drawings", Thesis, TR 74-04, University of Iowa, 1974.

5. FORMAL LANGUAGE DEFINITIONS CAN BE MADE PRACTICAL[1]

> '. . . *The metalanguage of a formal definition must not become a language known to only the priests of the cult. Tempering science with magic is a sure way to return to the Dark Ages.*' [Marcotty76]

5.1. The problem

Programming languages are being designed using pre-scientific methods. There is of course no substitute for experience, taste, style and intuition, but a scientific design methodology to support them is lacking. Methods for describing programming languages are somewhat more developed, but most definitions are either ambiguous and inaccurate, or excessively formal and unreadable. In general, a language definition method should:

- help the language **designer** by giving insight into the language he or she is designing, and by exposing interactions that might exist between language features. The definition should at the same time be a pilot implementation of the defined language, or it should at least be convertible into one. It is assumed here, that design and definition can best be carried out simultaneously.
- help the language **implementor** by providing him with an unambiguous and complete definition that is capable of 'executing' small programs in cases where the implementor is not sure about all implications of a particular language feature.
- help the **user** by providing him with a precise definition in a language with which he is not too **unfamiliar**.

These three goals impose different and to a certain extent contradictory requirements on the definition method to be used. In particular, it seems difficult to combine precision and readability in one method, since a precise definition has to use some formalism into which the reader has to be initiated and such a definition will have a tendency to become long and unreadable. This chapter is devoted to an experiment with a language definition method that may be considered as a tentative step towards satisfying the above requirements.

The **defined** language is (of course) SUMMER. The definition method is similar in spirit to the SECD method [Landin64], i.e. it is an operational language definition method which uses recursive functions and syntactic recognition functions that associate semantic actions with all constructs in the grammar of the language. In the method presented in this chapter, readability has been considerably enhanced by using a few imperative constructs and by introducing a very concise notation for parsing and decomposing the source-text of programs in the defined language. SUMMER, extended with such parsing and decomposing operations, is used as **definition language**. The definition is thus circular (see Sections 5.2.1 and 5.3).

A complete description of the definition method can be found in part II of this thesis. The next section gives only a birds-eye view of the description method and

1) This chapter is a revised version of [Klint81a].

shows some illustrative examples from the SUMMER definition. In Section 5.3 the method as a whole and its application to SUMMER are assessed.

5.2. The method

5.2.1. Introduction

An **evaluation process** or **interpreter** (with the name '*eval*') will be defined that takes an arbitrary, but syntactically correct, source text ('the source program') as input and either computes the result of the execution of that program (if it is a legal program in the defined language), or detects a semantic error, or does not terminate. In the latter two cases, no meaning is attached to the program. The evaluation process operates directly on the text of the source program. During this process a global **environment** is inspected or updated. An environment is a mapping from identifiers in the source program to their actual values during the evaluation process. In this way environments determine the meaning of names in the source program and are used to describe concepts such as variables, assignment and scope rules.

A fundamental question arises here: in which language do we write the definition? Several choices can be made, such as the formalism used in denotational semantics ([Gordon79], which boils down to a mathematical notation for recursive functions and domains) or the Vienna Definition Language ([Wegner72], which is a programming language designed for the manipulation of trees). This is not the right place to discuss the merits of these formalisms, but none has the desired combination of properties described in the previous paragraph. Instead of designing yet another definition language, the defined language itself (this is SUMMER in the examples given in this chapter) will be used as definition language. This choice has the obvious disadvantage that the definition is circular, but it has the practical advantage that readers who have only a moderate familiarity with the defined language will be able to read the definition without great difficulty. An extensive discussion of circular language definitions can be found in [Reynolds72]. It should be emphasized that there is no **fundamental** reason for making the definition circular. The definition method described here would also work if, for instance, ALGOL68 were used as definition language. In any case, it is essential that the definition language has powerful string operations and allows the creation of data structures of dynamically determined size. This requirement, for example, makes PASCAL less suited as definition language. Choosing SUMMER as definition language gave us the opportunity of investigating the suitability of that language in the area of language definition (see Section 5.3).

In the following sections the definition method and an example of its application (in the SUMMER definition) are described simultaneously. In Section 5.2.2 some aspects of the use of SUMMER as a metalanguage are discussed. The definition method can be subdivided into the definition of semantic domains (Section 5.2.3) and of the evaluation process (Section 5.2.4). Further detailed examples from the SUMMER definition are given in Section 5.2.5.

5.2.2. SUMMER as a metalanguage

This paragraph focuses on some aspects of SUMMER that are used in the formal definition, but were not covered in Chapter 4. Most of the constructs to be used in the definition have some similarity with constructs in, for instance, PASCAL and are assumed to be self-explanatory. Only less obvious constructs that are essential to the understanding of the definition are mentioned here.

SUMMER is an object-oriented language with pointer semantics. This means that an object can be modified by assignment and that such modifications are visible through all access paths to that object. For example,

$s := stack(10);$
$t := s;$

assigns one and the same *stack* object to the variables s and t, and

$s.push(v)$

pushes the value of v onto this stack. As a side-effect the stack is modified in such a way that subsequent operations on s or t may perceive the effect of that modification. In the formal definition this is relevant to the concepts 'state' and 'environment', which are modified in this way.

The language is dynamically typed, i.e. the type of variables is not fixed statically (as in PASCAL) but is only determined during the execution of the program (as in LISP or SNOBOL4). Moreover, generic operations on data structures are allowed. If an operation is defined on several data types, then the procedure to be executed when that operation occurs is determined by the type of the (left) operand of that operation.

Control structures and data structures are self-explanatory except possibly **arrays and for-expressions.**

Arrays are vectors of values, indexed by $0, \ldots, N-1$, where N is the number of elements in the array. If A is an array then the operation $A.size$ will yield the number of elements in the array. A new array is created by

$[V_0, \ldots, V_{N-1}]$

or

$array(N, V)$

In the former case, an array of size N is created and initialized to the values $V_0, \ldots, V_{N-1}$. In the latter case, an array of size N is created and all elements are initialized to the value V. Array denotations are also allowed on the left-hand side of assignments. This provides a convenient notation for multiple assignments. For example,

$x := 10; y := 20; z := 30$

is equivalent to

$[x, y, z] := [10, 20, 30]$

and, more generally,

$x_0 := a[0]; \cdots ; x_k := a[k]$.

is equivalent to

$[x_0, \ldots, x_k] := a$

The general form of a for-expression is:

for V **in** G **do** S **od**

where V is a variable, G is an expression capable of generating a sequence of values VAL_i and where S is an arbitrary statement. For each iteration the assignment $V := VAL_i$ is performed and S is evaluated. As used in the formal definition, the value of G is either an array (in which case consecutive array elements are generated) or G is an array on which the operation *index* is performed (in which case all indices of consecutive array elements are generated). For example, in

$a := [144, 13, 7];$
for x **in** a **do** *print*(x) **od**

an array object is assigned to the variable a and the values 144, 13 and 7 will be printed, while

for i **in** a.*index* **do** *print*(i) **od**

will print the values 0, 1 and 2. Further examples of for-expressions will be found in the following paragraphs.

5.2.3. Semantic domains

A semantic domain is a set, whose elements either describe a primitive notion in the defined language (like 'variable' or 'procedure declaration') or have some common properties as far as the language definition is concerned. The relationship between these domains is given by a series of domain equations.

In this paragraph the domains in the SUMMER definition are briefly described. The abstract properties of these domains are given in part II. Here, they are only introduced informally. First, the domain equations are given. Next, the meaning of each domain is described.

The relationship between the domains *BASIC-INSTANCE*, *COMPOSITE-INSTANCE*, *INSTANCE*, *STORABLE-VALUE*, *DENOTABLE-VALUE*, *PROCEDURE*, *CLASS*, *LOCATION*, *STATE* and *ENVIRONMENT* is as follows:

BASIC-INSTANCE	= *INTEGER* ∪ *STRING* ∪ *UNDEFINED*
COMPOSITE-INSTANCE	= *CLASS* × *ENVIRONMENT*
INSTANCE	= *BASIC-INSTANCE* ∪ *COMPOSITE-INSTANCE*
STORABLE-VALUE	= *INSTANCE*
DENOTABLE-VALUE	= *STORABLE-VALUE* ∪ *PROCEDURE* ∪ *CLASS* ∪ *LOCATION*
PROCEDURE	= *PROCEDURE-DECLARATION* × *ENVIRONMENT*
CLASS	= *IDENTIFIER* × *CLASS-DECLARATION*
STATE	= *LOCATION* → (*STORABLE-VALUE* ∪ {*unused*})
ENVIRONMENT	= *IDENTIFIER* → *DENOTABLE-VALUE*

Here, *IDENTIFIER*, *PROCEDURE-DECLARATION* and *CLASS-DECLARATION* are the sets of string values that can be derived from the syntactic notions <identifier>, <procedure-declaration> and <class-declaration> in the

SUMMER grammar. *BASIC-INSTANCE* is the domain of primitive values in the language. *COMPOSITE-INSTANCE* is the domain of user-defined values. *STORABLE-VALUE* is the domain of values which can be assigned to variables in the source program. *DENOTABLE-VALUE* is the domain of values which can be manipulated by the evaluation process. The domains *PROCEDURE* and *CLASS* describe declarations for procedures an classes respectively. The domain *LOCATION* is used to model the notion 'address of a cell capable of containing a (single) value'. *STATE* is the domain that consists of functions which map locations onto actual values or '*unused*'. *ENVIRONMENT* is the domain of functions which map names onto denotable values.

STRING, *INTEGER* and *UNDEFINED* are the domains modeling the values and operations for the built-in types 'string', 'integer' and 'undefined' respectively. *UNDEFINED* is the domain consisting of undefined values. All variables are initialized to an undefined value. Operations are defined on elements in *STRING*, *INTEGER* and *UNDEFINED* that model the primitive operations on the data types 'string', 'integer' and 'undefined'.

PROCEDURE is the domain of procedures. Each element of this domain describes a procedure declaration and contains a literal copy of the text of the procedure declaration itself and an environment that reflects all names and values available at the point of declaration.

CLASS is the domain of classes. Each element of this domain describes one class declaration and contains the name of the class and a literal copy of the text of the class declaration. *COMPOSITE-INSTANCE* is the domain of class instances. All values that are created by a SUMMER program are instances of some class (this has been explained informally in Section 4.4.). A composite instance consists of the name of the class to which it belongs, the literal text of the declaration of that class and an environment that has to be used to inspect or update components from the instance. Operations are defined on elements in *PROCEDURE*, *CLASS* and *INSTANCE* to manipulate the components of an element in these domains. For completeness, these domains are mentioned here, but they will not be used in the remainder of this chapter. A complete definition appears in Section 8.3.3.

ENVIRONMENT is the domain of environments. Environments take care of the binding between names and values and the introduction of new scopes (i.e. ranges in the program where names may be declared). In general, operations defined on environments modify the environment to which they are applied.

The definitions given in following sections are centered around operations on elements of these semantic domains, but we will see relatively few of them in the examples. Operations will be explained only when they occur in an example.

5.2.4. Evaluation process

Before turning our attention to the evaluation process (which defines semantics), a few words must be said about the definition of syntax. In the definition method to be used the role of a syntax definition is twofold:

- to define the grammar of the defined language, and

- to unravel a source text in order to define a meaning for its constituent parts.

These two aspects of a syntax definition are now considered in turn.

An extended form of BNF notation is used to describe the syntax of the defined language. The extensions aim at providing a concise notation for the description of repeated and optional syntactic notions. A syntactic notion suffixed with '+' means one or more repetitions of that notion. A notion suffixed with '*' stands for zero or more repetitions of that notion. The notation

{ notion separator } replicator

i.e. a *notion* followed by a *separator* enclosed in braces followed by a *replicator*, is used to describe a list of notions separated by the given separator. A replicator is either '+' or '*'. The replicator '+' indicates that the list consists of one or more notions. The list begins and ends with a notion. The replicator '*' indicates that the list consists of zero or more notions.

An optional syntactic notion is indicated by enclosing it in square brackets, e.g. '[*notion*]'. The terminal symbols of the grammar are either enclosed in single quotes (for example: ',' or ':=') or written in upper case letters if the terminal symbol consists solely of letters (both 'IF' and "*if*" may, for instance, be used to denote the terminal symbol **'if'**). Where necessary, parentheses are used for grouping.

Some parts of a syntax rule may be labelled with a <tag>; their meaning will become clear below.

The evaluation process is described in SUMMER extended with **parse expressions**[2] of the form

'{{' <identifier> '==' <syntax-rule> '}}'

which provide a concise notation for parsing and extracting information from the text of the source program. A parse expression succeeds if the identifier on the left hand side of the '==' sign has a string as value and if this string is of the form described by the <syntax-rule> on the right hand side of the '==' sign. All <tag>s occurring in the <syntax-rule> should have been declared as variables in the program containing the parse expression, in this case the evaluation process. Substrings of the parsed text recognized by the syntactic categories that are labelled with a <tag> are assigned to the variable that corresponds to that <tag>. Consider, for example, the following program fragment:

```
if {{ e == WHILE t:<test> DO b:<body> OD }}
then
  put('While expression recognized')
fi
```

The parse expression will succeed if *e* has the form of a 'while expression'. The literal text of the <test> is then assigned to variable *t* and the text of the <body> is assigned to variable *b*.

---

2) There is no **fundamental** reason for introducing this language extension. However, the disadvantage of introducing such an *ad hoc* extension is more than compensated by the fact that we use a notation which is sufficiently similar to BNF notation to be almost self-explanatory. The effect of introducing a language extension as proposed here is interesting in its own right but falls outside the scope of the current discussion.

If the recognized part of the text is a list or repetition, then an array of string values is assigned to the variable. In the case of a list of notions separated by separators, the latter are omitted and only the notions occurring in the list are assigned to (consecutive) elements in the array. This is exemplified by:

```
if {{ e == VAR list:{<identifier> ','}+ }}
then
  put('A variable declaration containing:');
  for id in list do put(id) od
fi
```

The parse expression succeeds if *e* has the form of a 'variable declaration' (i.e. the keyword **'var'** followed by a list of <identifier>s separated by commas) and in that case an array of string values corresponding to the <identifier>s occurring in the declaration is assigned to the variable *list*, which is subsequently printed.

Parse expressions may be used in **if**-expressions or may stand on their own. In the latter case, the string to be parsed has to be of the form described by the parse expression. In this way, parse expressions can be used to decompose a string with a known form into substrings.

This concludes our digression on the definition of syntax and we turn now our attention to the evaluation process that defines semantics. In the case of the SUMMER definition, the overall structure of this evaluation process is:

```
var ENV;
var STATE;
var varinit;
proc ERROR
    ...;
proc eval(e)
( var value, signal, ...;
  if {{ e == <program-declaration> }}
  then
    ...
    return([value, signal])
  fi;
  if {{ e == <variable-declaration> }}
  then
    ...
    return([value, signal])
  fi;
  ...
  if {{ e == <empty> }}
  then
    ...
    return([value, signal])
  fi;
);
```

The variable *ENV* has as value the current environment, and *STATE* has as value the current state. The variable *varinit* has as value a string consisting of the text of all <variable-initialization>s in the current <block>.

The procedure *ERROR* is called when a semantic error is detected during evaluation. In this case, the whole evaluation process is aborted immediately. The main defining procedure is *eval*, which selects an appropriate case depending onto the syntactic form of its argument *e*. Some examples of these various cases will be given in Section 5.2.5. Note that each of these cases involves a complete syntactic analysis of the string *e*. The evaluation process is initiated by creating an initial, empty environment *ENV* and by calling *eval* with the text of the source program as argument. If the evaluation process is not prematurely terminated (by the detection of a semantic error) the result of the evaluation of the source program can be obtained by inspecting the resulting environment *ENV*.

The definition of SUMMER has been profoundly influenced by the success-directed evaluation scheme in the language: an expression can either **fail** or **succeed**. The meaning of failure is that evaluation of the 'current' expression is abandoned and that evaluation is continued at a point where a 'handler' (i.e. <if-expression>, <while-expression>) occurs to deal with the failing case. A similar situation exists for <return-expression>s, which terminate the evaluation of (possibly nested) expressions. Both language features can thus essentially influence the flow-of-control in a program.

How are these properties of SUMMER reflected in the definition? The procedure *eval* delivers as result an array of the form [*value*, *signal*], where *value* is the actual result of the procedure and *signal* is a success/failure flag that indicates how *value* should be interpreted. The signal is used to describe the occurrence of failure and/or <return-expression>s and may have the following values:

N: evaluation terminated normally.

F: evaluation failed.

NR: normal return; a <return-expression> was encountered during evaluation.

FR: failure return; a failure return was encountered during evaluation.

The signal is tested after each (recursive) invocation of *eval*. In most cases *eval* performs an immediate return if the signal is not equal to *N* after the evaluation of a subexpression. Exceptions to this rule are of two kinds:

- The semantics of certain constructs is such that the flow of control is intentionally influenced by the success or failure of expressions (e.g. <test>s in <if-expression>s). In *eval* this corresponds to appropriate reactions to *N* and *F* signals. Aborting the evaluation of the 'current' expression, which is necessary if failure occurs in a deeply nested subexpression, can be achieved by passing an *F* signal upwards until it reaches an incarnation of *eval* that can take appropriate measures.
- The semantics of the <return-expression> is such that the execution of the procedure in which it occurs is terminated and that execution is to be continued at the place of invocation. This is reflected by the signal values *FR* and *NR*, that are only **generated** by <return-expression>s and are only **handled** by the semantic rules associated with procedure calls. The latter rules turn *NR* into *N* and *FR* into *F* before the evaluation process is resumed at the point where it left off to perform the (by then completed) procedure call. All other semantic rules return immediately when an *NR* or *FR* signal occurs.

5.2.5. Some examples

5.2.5.1. If expressions

An <if-expression> corresponds to the if-then-else statement found in most programming languages. If evaluation of the <test> immediately contained in the <if-expression> terminates successfully, the <block> following **then** is evaluated. If a <return-expression> was encountered during evaluation of the <test>, then the evaluation of the <if-expression> as a whole is terminated. Otherwise, the <test>s following subsequent **elif**s are evaluated until

- one such evaluation terminates successfully (the <block> in the following **then**-part is then evaluated), or
- a <return-expression> is encountered during evaluation of the <test> (the evaluation of the <if-expression> as a whole is then terminated), or
- the list of <test>s is exhausted.

In the last case, the <if-expression> may contain an **else**-part and if so the <block> following **else** is evaluated. The formal definition is:

```
1   if {{ e == IF t:<test> THEN b:<block>
2            elifpart: (ELIF <test> THEN <block>)*
3            elsepart: [ELSE <block>] FI }}
4   then
5     var v, sig;
6     [v, sig] := eval(t);
7     if sig = N
8     then
9       return(eval(b))
10    elif sig = FR | sig = NR
11    then
12      return([v, sig])
13    else
14      var oneelif;
15      for oneelif in elifpart
16      do {{ oneelif == ELIF t:<test> THEN b:<block> }} ;
17         [v, sig] := eval(t);
18         if sig = N
19         then
20           return(eval(b))
21         elif sig = FR | sig = NR
22         then
23           return([v, sig])
24         fi
25      od;
26      if {{ elsepart == ELSE b:<block> }}
27      then
28        return(eval(b))
```

```
29          else
30            return([a_undefined, N])
31          fi
32        fi
33      fi;
```

The parse expression in lines 1-3 decomposes the string value of *e* into several parts. In line 6 the <test> of the <if-expression> is evaluated. If this evaluation produces the signal *N*, the <block> following **then** is evaluated. The occurrence of the signals *NR* or *FR* (denoting the occurrence of a <return-expression>) terminates the evaluation of the <if-expression> (lines 10, 21). The loop in lines 15-25 iterates over the successive <test>s and describes the semantics as explained above. If all <test>s fail, the (optional) **else**-part is evaluated in lines 26-31.

For a better understanding of the above definition, it may be useful to note that parts of the source program are parsed **repeatedly** during **one** evaluation of a given <if-expression>. For example, the <block> following an **elif** is parsed both in lines 2 and 16. (This explains, by the way, why the parse expression in line 16 need not be contained in an if statement, see Section 5.2.4.). In general, the source text of the <if-expression> is parsed **each** time that it is evaluated.

5.2.5.2. Variable declarations

A <variable-declaration> introduces a series of new variables into the current environment, i.e. names of locations whose contents may be inspected and/or modified. The declaration may contain <expression>s whose values are to be used for the initialization of the declared variables. In the formal definition, this is described by appending all variable initializations in the current <block> to the variable *varinit* and by evaluating the string value of that variable before the evaluation of the subsequent <expression>s in the <block>. The formal definition of <variable-declaration>s is:

```
1       if {{ e == VAR varlist:{<variable-initialization> ',' }+ ';' }}
2       then
3         var name, onevar;
4         for onevar in varlist
5         do if {{ onevar == name:<identifier> ':=' <expression> }}
6            then
7              varinit := varinit || v || ';'
8            else
9              {{ onevar == name:<identifier> }}
10           fi;
11           ENV.bind(name, STATE.extend(a_undefined))
12        od;
13        return([a_undefined, N])
14      fi;
```

In line 1, *e* is decomposed into an array of strings which have the form of a <variable-initialization>. These string values are considered in succession in the loop in lines 4-12. If the <variable-initialization> contains an initializing expression, that

expression is appended to *varinit* (line 7) using the string concatenation operator '||'. Finally, the state *STATE* is extended with a location containing an undefined value, and that new location is bound, in the current environment *ENV*, to the identifier being declared. Note that, in line 9, *v* is known to have the form of an <identifier>.

5.2.5.3. Blocks

A <block> introduces a new scope to be used for the declaration of new variables and constants. It consists of a (perhaps empty) list of declarations followed by a sequence of expressions separated by semicolons. A <block> is evaluated as follows:

- Evaluate all declarations (this can never fail).
- Evaluate all variable-initializations resulting from the evaluation of the declarations. If this evaluation is not completed successfully, the evaluation of the <block> is terminated.
- Evaluate the sequence of expressions in the <block>. SUMMER forbids the failure of an expression inside a sequence of expressions. Only the last expression in a sequence is allowed to fail; this failure is passed upwards to enclosing language constructs. If a <return-expression> is encountered during evaluation of one of the <expressions>s, the evaluation of the <block> is terminated.

The formal definition is:

```
1   if {{ e == decllist: <variable-declaration>*
2                exprlist: {[<expression>] ';'}* }}
3   then
4     var decl, expr, ENV1, i, varinit1, sig;
5     ENV1 := ENV;
6     ENV.new_inner_scope;
7     varinit1 := varinit;
8     varinit := '';
9     for decl in decllist do [v, sig] := eval(d) od;
10    [v, sig] := eval(varinit);
11    varinit := varinit1;
12    if sig ~= N then ENV := ENV1; return([v, sig]) fi;
13    for i in exprlist.index
14    do [v, sig] := eval(exprlist[i]);
15       case sig
16       of N:          % nothing to do %,
17          F:          if i ~= exprlist.size - 1 then ERROR fi,
18          NR: FR:     ENV := ENV1; return([v, sig])
19       esac
20    od;
21    ENV := ENV1;
22    return([v, sig])
23  fi;
```

This definition is a simplified version of the one given in part II. In lines 5-8 local copies are made of *E* and *varinit* and new values are assigned to them. In lines 9-12

the list of <variable-declaration>s in the <block> and the resulting <variable-initialization>s are evaluated. In lines 13-20 the list of <expression>s in the <block> is evaluated. Note how failure of an expression in the middle of the list is treated (line 17, see above).

5.3. Assessment

The formal language definition presented in the previous section will now be assessed. It is tempting to try to get statements like:

Users can answer 87% of their questions on language issues within five minutes if they have access to a formal language definition of the kind described in this chapter.

or

35% of all run-time errors in user programs are directly related to anomalies in the language definition.

In the absence of such results and without methods of obtaining them, we have to live with qualitative and more or less speculative observations.

A rough indication of the **conciseness** of the definition can be obtained by comparing various sizes as they apply to the SUMMER definition:

formal definition	20 pages
reference manual	100 pages
implementation	200 pages

These figures show that the implementation is ten times larger than the formal definition. This is not surprising, since the implementation has to be efficient while the formal definition does not have to be. In this light the 'a-language-is-defined-by-its-implementation' approach can be rephrased as: '**if** a language is defined by its implementation, **then** that implementation had better be small'.

The definition is **precise** and **complete**, in the sense that **all** semantic operations associated with a particular language construct **have** to be specified to allow the construction of an **executable** version of the definition. The number of **operational details**, i.e. details in the definition which stem from the chosen definition method and are not a reflection of details of the defined language, are surprisingly small. This is a consequence of the choice of the definition language (which should have powerful data types and string manipulation operations) and the choice of high-level environment manipulation primitives which correspond directly to operations in the defined language and which are not (yet) perverted by implementational details. SUMMER, extended with parse expressions, seems a quite reasonable vehicle for language definition. However, it is not possible to make 'continuation-style' (see [Gordon79]) definitions, since higher-order functions are lacking.

It is difficult to give an objective judgement as to the **readability** of the definition, but we have observed that only a moderate effort (of a few days) is required on the part of a programmer without any training in formal semantics, and without any previous exposure to the language, to learn SUMMER using only the (annotated) formal definition.

The advantages and disadvantages of the formal definition for designer, implementor and user will now be discussed in some detail.

The advantages for the **designer** are:

- Anomalies in the design are magnified. It is a general rule that ill-formed entities can only be described by ill-formed descriptions or by descriptions which list many exceptional cases. It is easier to locate such exceptions or anomalies in a concise formal definition than in an ambiguous natural language definition or in a bulky implementation. In the SUMMER definition, for example, a very specific operation on environments is needed ('partial-state-copy') to accommodate the definition of just one language feature ('try-expression'). It turned out that a slight modification of that feature would at the same time simplify the definition and improve the feature.
- Exhaustive enumeration of language features. A formal definition method forces the designer to enumerate all language features in the same framework and this may help him to find omissions in the design.
- Interactions between language features can be studied. In the SUMMER definition, for example, the designer is forced to decide what happens when a <return-expression> is evaluated during the evaluation of any other expression. There is, however, no guarantee that all interactions can be found, since the formal definition may still contain hidden interactions between language features. The use of auxiliary functions in the definition is an aid in making interactions explicit. One may even apply techniques such as calling graph analysis and data flow analysis to the definition to discover clusters of interacting features and to establish certain properties of the definition.
- An executable formal definition can be tested and used. This may help eliminate clerical and gross errors from the definition. An executable definition allows the designer to play with (toy) programs written in the language he is designing. There is, however, a problem with circular definitions: some implementation of the defined language has to exist before the definition itself can be made executable.

Disadvantages for the **designer** are:

- A considerable effort is required to construct a formal definition.
- A general problem is that there are no canned, satisfactory definition methods available and that the designer has to begin with either creating a new method or adapting and extending an existing one.

Advantages for the **implementor** are:

- Unambiguous language definition.
- The implementor may stumble over a certain combination of features. Such cases can be executed both by the implementation and by the definition and the results can be compared.

Disadvantages for the implementor are:

- The implementor must be familiar with the definition method or become acquainted with it. This is only a minor effort if one compares it with the total effort required to implement the language.

- It is non-trivial to derive an implementation strategy from the language definition. This is a problem shared by all 'abstract' language definitions, in which no attempt is made to use primitives in the definition with a direct counterpart in an implementation. This leads to the conclusion that such abstract definitions should be accompanied by an 'annotation for implementors', which states where well-known implementation techniques can be used and where certain optimizations are possible.

Advantages for the **user** are:

- Unambiguous and concise language definition.
- The user is used to reading programs and the formal definition can be read as such. In the case of a circular definition, the formal definition may be considered as a very informative example program.

Disadvantages for the **user** are:

- The user must be exposed to the definition method.
- A formal definition is harder to read than a 'natural language' definition.
- In the case of the SUMMER definition, the circularity may be confusing for the naive user.

In retrospect, it seems justified to conclude that the method presented in this chapter is a first step in satisfying the requirements given in Section 5.1. However, many problems remain to be investigated. Does the method given lend itself to mathematical analysis? How can the 'complexity' of a language be derived from its definition? Is it possible to 'optimize' the executable version of definitions? Attempts in this direction can be found in [Jones80]. What is the relationship between this definition method and extensible languages? Answers to these questions will provide more insight into the structure of programming languages and the methods of defining them.

5.4. References for Chapter 5

[Gordon79] Gordon, M.J.C., *The Denotational Description of Programming Languages*, Springer-Verlag, 1979.

[Jones80] Jones, N.D., *Semantics-directed Compiler Generation*, Springer-Verlag, 1980.

[Klint81a] "Formal language definitions can be made practical", in: De Bakker, J.W. & Van Vliet, J.C. (editors), *Algorithmic Languages*, IFIP, North-Holland, 1981, 115-132.

[Landin64] Landin, P.J., "The mechanical evaluation of expressions", *Computer Journal*, **6** (1964), 308-320.

[Marcotty76] Marcotty, M., Ledgard, H.F. & Bochmann, G.V., "A sampler of formal definitions", *Computing Surveys*, **8** (1976) 191-276.

[Reynolds72] Reynolds, J.C., "Definitional interpreters for higher-order languages", *Proceedings ACM Annual Conference*, 1972, 717-740.

[Wegner72] Wegner, P., "The Vienna Definition Language", *Computing Surveys*, **4** (1972), 5-63.

6. IMPLEMENTATION

6.1. Introduction

The implementation of a high level programming language still requires a major effort. Depending on the complexity of the language to be implemented and on the requirements imposed on the final product, one is faced with a task that may take from a few man months to several man years of labour. If the primary purpose of a research project is language **design**, one generally does not want to spend too much time on language **implementation** and consequently some balance has to be found between the speed with which programs in the new language are executed and the effort needed to achieve this.

The implementation of SUMMER which is described in this chapter evolved over several years and required approximately two man years of effort. This implementation allows the execution of SUMMER programs of any size and gives appropriate messages for syntactic and semantic errors. Apart from providing a few simple tools (for symbolic tracing and the gathering of run-time statistics) no effort was made to assist the user in creating, modifying or maintaining SUMMER programs.

In order to minimize implementation time without sacrificing execution time efficiency completely, we proceeded as follows:

- SUMMER programs are translated to abstract machine programs. The latter are subsequently executed by an interpreter.
- Existing algorithms and techniques were used whenever possible.
- A high level implementation language was used. Assembly language was resorted to only when absolutely necessary.
- Simple but less efficient algorithms were preferred to complex but efficient ones.
- Facilities for measurements and internal consistency checking were made an integral part of the system.
- The implementation was made as flexible as possible so as to allow for easy experimentation with new language features and new implementation techniques.
- Communication with the user is performed in terms of entities that are known to the user at the source program level.

The overall organization of the SUMMER system is sketched in Figure 6.1. A compiler (written in SUMMER) transforms the text of a SUMMER program into an equivalent abstract machine program. This abstract machine program is then combined with a library of run-time routines (written in C [Kernighan78]) to form a final executable version of the original SUMMER program. The run-time library contains, amongst other things, an **interpreter** for abstract machine instructions.

Several aspects of the SUMMER implementation have been described elsewhere and will not be discussed any further:

- The merits of a system organization as outlined above and the tradeoffs between compilation and interpretation have been described in [Klint81b].

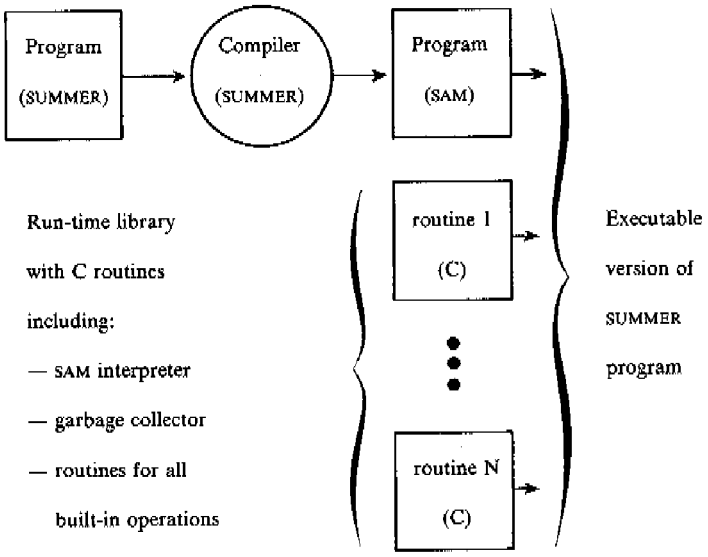

Figure 6.1. Overall organization of the SUMMER system.

- Various forms of abstract machine instructions (i.e stack-oriented vs. two- and three-address instructions) were compared in [Klint79a].
- A technique for maintaining run-time line numbers has been published in [Klint79b].

This chapter aims at giving a general impression of the SUMMER implementation. First of all, the SUMMER Abstract Machine (SAM) is described. Next, we show how the compiler generates code for it. Both descriptions are intentionally brief. We have used as many conventional techniques as possible and the reader is assumed to be familiar with implementation techniques for high level languages. We shall only highlight **new** or **interesting** techniques or algorithms. Readers interested in more details are referred to the amply annotated listings and (internal) documentation of the SUMMER system itself.

6.2. The SUMMER Abstract Machine

The SUMMER Abstract Machine (SAM) provides a largely conventional, stack-oriented, intermediate level architecture tailored to the execution of SUMMER programs. The instruction set is summarized in Table 6.1. The machine has four special purpose registers. The **stack pointer** (*SP*) points to the top of the expression evaluation stack. It is used implicitly by many instructions. We shall use the notation *SP*[*n*], with $n \geq 0$, to denote elements on the stack; *SP*[0] denotes the top element,

$SP[1]$ denotes the element below the top element, and so on. The **current fail label** (see Section 6.2.1) points to the top of the stack of fail labels used to implement failure handling and flow of control. The **current cache** (see Section 6.2.2) points to the top of the stack of recovery caches used to implement the **try**-expression in SUMMER. The **current subject** points to the top of the stack of subject strings used to implement the **scan**-expression in SUMMER.

All values in a SUMMER program are represented in SAM by **mvalues**, which have a uniform layout and occupy one or more consecutive memory cells. The first cell of each mvalue contains a tag that identifies the class to which it belongs. Following cells may contain pointers to other mvalues. A number of classes (integer, string, real, array, table) are primitive in SAM , i.e. they are directly supported by SAM. For operations on other classes, calls to built-in routines are generated. Variables in a SUMMER program are represented in SAM by a pointer to an mvalue. This level of indirection enables SUMMER variables to possess values of varying size and type.

A SAM program consists of three parts as shown in Figure 6.2. The first part contains class and field declarations describing the structure and access rights of the class declarations in the SUMMER program. The second part consists of declarations for global variables and procedures. The third part consists of declarations for the string constants occurring in the generated code. Some examples of SUMMER expressions and their translation into SAM instructions are shown in Figure 6.3.

Three groups of instructions (for failure handling, side-effect recovery and operations on classes) are interesting and deserve a more detailed description. This is the topic of the following three paragraphs.

SAM INSTRUCTION SUMMARY

Push simple values onto the expression evaluation stack

INT	*n*	Push integer constant *n*.
REAL	*r*	Push real constant *r*.
GLOB	*n*	Push value of *n*-th global variable.
ASGLOB	*n*	Assign $SP[0]$ to *n*-th global variable.
LOC	*n*	Push value of *n*-th local variable.
ASLOC	*n*	Assign $SP[0]$ to *n*-th local variable.
LOAD	*l*	Push the address of a constant at location *l*.
NULLSTR		Push the empty string.
UNDEF		Push the value *undefined*.
VOID		Decrement SP.

Relational and arithmetical instructions

EQ	Test whether $SP[1] = SP[0]$. On failure: jump to current fail label. On success: replace $SP[0]$ and $SP[1]$ by $SP[0]$, i.e. $SP[1] := SP[0]$; $SP := SP - 1$. The following relational operators behave similarly. →

NE		Test whether $SP[1] \neq SP[0]$.
LT		Test whether $SP[1] < SP[0]$.
GT		Test whether $SP[1] > SP[0]$.
LE		Test whether $SP[1] \leq SP[0]$.
GE		Test whether $SP[1] \geq SP[0]$.
ADD		Perform $SP[1] + SP[0]$. $SP[0]$ and $SP[1]$ are replaced by their sum, i.e. $SP[1] := SP[0] + SP[1]$; $SP := SP - 1$.
SUB		Perform $SP[1] - SP[0]$.
MUL		Perform $SP[1] * SP[0]$.
DIV		Perform $SP[1] / SP[0]$.
IDIV		Perform $SP[1] \% SP[0]$ (integer division).
NEG		Perform $- SP[0]$ (unary minus).
CONC		Perform $SP[1] \mid\mid SP[0]$ (string concatenation).

Instructions related to arrays and tables

ARINIT	n,m	Array initialization. Create an array of length n and initialize the first m ($m \leq n$) elements using the m values on top of the stack. Push the address of the resulting array onto the stack.
TABINIT	n	Table initialization. Create a table of length n and push the result onto the stack. Initialization of table entries is done by the instruction TABELEM below.
TABELEM	n	Table element assignment.
IND		Index operation in array or table. $SP[0]$ is the array or table instance. $SP[1]$ is the value of the index. Both entries are replaced by the value of the indexed element.
ASIND		Assign to array or table element. $SP[0]$ is the array or table element. $SP[1]$ is the value of the index. $SP[2]$ is the value to be assigned to the indexed element. Only the latter remains on the stack.
XAR	n	Replace the array value in $SP[0]$ by its first n elements.

Procedure calls

CALL	n,m	Call the n-th procedure with m actual parameters. The actual parameters are on top of the stack.
RETURN		Return a value from a procedure call and deallocate stack space for actuals and locals.
FRETURN		Failure return from procedure call. Deallocate stack space for actuals and locals and simulate a GOFL instruction in the caller.

→

Failure handling (see Section 6.2.1) and flow of control

NEWFL	*l*	Define new fail label *l*.
OLDFL		Restore previous fail label.
GOFL		Jump to current fail label.
GO	*l*	Jump to label *l*.
GOCASE		Case table jump. *SP*[1] is the case table to be used. It contains (index-value, program-label) pairs. *SP*[0] is the index value for the table selection.

Recovery caches (see Section 6.2.2)

NEWRC		Install new recovery cache.
OLDRC		Discard current cache and reinstall previous one (if any).
RESRC		Restore from current cache.

Instructions for manipulating the current subject string

NEWSUBJ		Install *SP*[0] as new current subject.
OLDSUBJ		Restore previous subject.
SUBJECT		Push value of current subject.

Instructions related to classes (see Section 6.2.3)

NEWCLASS	*name*	Create new instance of class *name*.
CLOC	*n*	Push *n*-th local of class instance.
ASCLOC	*n*	Assign to *n*-th local of class instance.
FLD	*n*,*name*	Fetch field *name* from class instance *SP*[*n*].
ASFLD	*n*,*name*	Assign to field *name* from class instance *SP*[*n*].
IFLD	*n*,*name*	Fetch from field ignoring associations.
IASFLD	*n*,*name*	Store in field ignoring associations.
SELF		Push current class instance.

Declarative instructions

LAB	*l*	Declare label *l*.
DCLGLOB	*name*	Declare global variable *name*.
DCLSTR	*name*,*n*,$b_1, \ldots, b_n$	Declare string constant *name* consisting of *n* characters $b_1, \ldots, b_n$.
PROC	*p*,*lnames*,*nf*,*nl*,*bline*,*eline*	Start declaration of procedure with name *p*.

lnames	the names of the local variables in *p*.
nf	number of formal parameters of *p*.
nl	number of local variables of *p*.
bline	line number of first line of *p*'s declaration in source text.

→

	eline	line number of last line of *p*'s declaration in source text.
PROGRAM	*p*,*lnames*,*nf*,*nl*,*bline*,*eline*	Start declaration of main program.
PROCEND		End of procedure declaration.
SUBR	*p*	SUMMER program uses library routine *p*.
CLASSES		Start class description tables.
ENDCLASSES		End class description tables.
FIELDS		Start field description tables.
ENDFIELDS		End field description tables.

Miscellaneous instructions

HALT		Halt instruction.
NOOP		No operation.
LINE	*n*	Static line number increment.
ALINE	*n*	Absolute line number.
ERROR	*name*	Generate error message with fixed name.
		Allowed names are:
	ER_case:	index does not occur in case table.
	ER_assert:	assertion failed.
	ER_ret:	procedure does not return a value.

Table 6.1. SAM instruction summary.

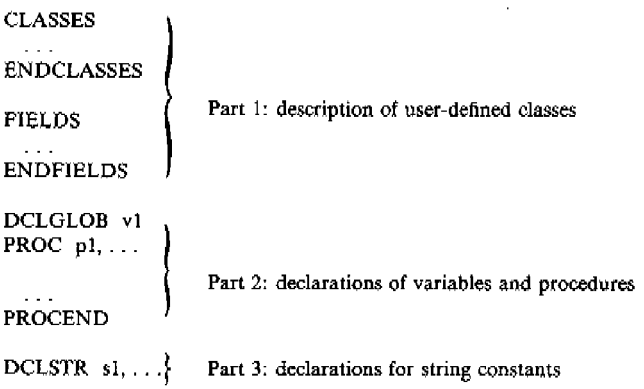

Figure 6.2. General form of SAM program.

SUMMER expression	SAM instructions	
(Assume that a and b are local variables numbered 0, 1.)		
$a := b + 2.5$	LOC	1
	REAL	2.5
	ADD	
	ASLOC	0
$a[10] := x * 2$	GLOB	x
	INT	2
	MUL	
	LOC	0
	INT	10
	ASIND	
$q(a, 13)$	LOC	0
	INT	13
	CALL	q,2

Figure 6.3. SAM instructions generated for some SUMMER expressions.

6.2.1. Failure handling

Each operation in a SUMMER program may fail; correspondingly each SAM instruction may fail. If failure occurs, the execution of the SAM program is continued at a point determined by the structure of the SUMMER program; all intermediate results that were placed on the expression stack by the sequence of instructions containing the failing instruction must then be removed. We use 'sequence' here in the sense of a series of instructions with the same failure continuation label. There are three options when designing an instruction set that has to accommodate this kind of failure handling:

1) Include in each instruction both a failure continuation label and the value of the stack pointer at the entry of the current instruction sequence.

2) Introduce a dedicated (abstract machine) register for maintaining the current failure label and the value of the stack pointer at the entry of the current instruction sequence.

3) Add an instruction for encoding the tree-structure of (nested) failure labels to the abstract machine. Since this tree-structure is known at compile time, a fair amount of computation can be saved in this way which is otherwise spent in maintaining the nesting of failure labels dynamically.

The first alternative is easy to implement, but leads to a substantial increase in the size of the generated code. The third alternative is the most efficient one, but complicates both compiler and abstract machine. As a compromise, the second alternative was adopted in SAM. Account is taken of the nesting of failure labels by a stack of

(continuation label, stack-top-at-sequence-entry) pairs. The **current fail label** register points to the pair on top of the **fail label stack**. The instructions for manipulating the fail label stack can now be defined precisely:

NEWFL *l*

Pushes the pair (l, SP) onto the fail label stack.

OLDFL

Removes the top of the fail label stack. This instruction is used at the end of instruction sequences. If the OLDFL instruction is reached, the execution of the sequence as a whole succeeds and the previous fail label is restored.

GOFL

Go to the current fail label, i.e. pop the pair (l, SP') off the fail label stack, adjust the expression stack by assigning SP' to the stack pointer SP and jump to continuation label l.

Fail labels are only pushed onto the fail label stack by NEWFL instructions. They are (implicitly) popped by:

- OLDFL (at the successful completion of the evaluation of an expression).
- GOFL (used for implementing the SUMMER operators '&' and '|', and for implementing **try**-expressions).
- FRETURN
- Failing expressions (the fail label stack is, for instance, popped if the instruction EQ fails).

Figure 6.4 shows how these instructions are used to implement **if**- and **while**-expressions.

6.2.2. Side-effect recovery

Side-effect recovery, as required by the **try**-expression in SUMMER, is realized in SAM by means of recovery caches. Since try-expressions may occur in a nested fashion, it is necessary to maintain a stack of active caches. Each cache is implemented as a linear list of (address, mvalue) pairs. All operations that modify (parts of) an mvalue must first consult the current cache (if any) to check whether the address of the memory location to be modified already occurs in the cache. If it does not, the address and old contents of the location are added to the cache.

There are three instructions for manipulating recovery caches:

NEWRC

Creates a new cache.

OLDRC

'Discards' {4.3} the current cache and reinstalls the previous one (if any).

RESRC

Restores from cache. Uses the information in the cache stack to restore the program to a previous state.

Figure 6.5 shows how these instructions are used to implement **try**-expressions. Despite the simplicity of this scheme and the (inefficient) linear search that is used to search the cache, no appreciable overhead due to the use of **try**-expressions has been observed.

SUMMER expression	SAM instructions	
if	NEWFL	*F*1
$x = b$	GLOB	x
	LOC	1
	EQ	
	VOID	
then	OLDFL	
$x := 3$	INT	3
	ASGLOB	x
	VOID	
	GO	*L*1
else	LAB	*F*1
$y := 4$	INT	4
	ASGLOB	y
	VOID	
fi	LAB	*L*1
while	LAB	*L*2
	NEWFL	*F*2
$x > y$	GLOB	x
	GLOB	y
	GT	
	VOID	
do	OLDFL	
$x := x - a$	GLOB	x
	LOC	0
	SUB	
	ASGLOB	x
	VOID	
od	GO	*L*2
	LAB	*F*2

Figure 6.4. SAM instructions generated for **if**- and **while**-expression.

6.2.3. Operations on classes

The life of a class instance can be subdivided into three distinct phases:

1) Creation of the new instance. This is performed by executing the instruction

NEWCLASS *cname*

which allocates space for a new instance of class *cname* and leaves a pointer to it on top of the expression stack. Such an instance-pointer is used explicitly or implicitly by all instructions related to classes (see below).

SUMMER expression	SAM instructions	
try	NEWRC	
	NEWFL	*F*1
p()	CALL	*p*,0
	VOID	
	GO	*L*1
,	LAB	*F*1
	RESRC	
	NEWFL	*F*2
q()	CALL	*q*,0
	VOID	
	GO	*L*1
	LAB	*F*2
	RESRC	
	GOFL	
until	LAB	*L*1
r()	CALL	*r*,0
	VOID	
	OLDFL	
yrt	OLDRC	

Figure 6.5. SAM instructions generated for a **try**-expression.

2) Access to the instance. Instructions for field selection (FLD, ASFLD) expect an instance-pointer as one of their arguments. These instructions contain an encoded **symbolic** field name which is looked up in the field declaration tables (in Part 1 of the SAM program, see Figure 6.2) in order to validate the field selection and to select the procedure to be invoked for the actual access. An instance-pointer is kept in a (dedicated) local variable location, when executing instructions inside a class declaration. In this way, certain instructions (CLOC, ASCLOC) can access and modify fields of the instance.

3) Death of the instance. The lifetime of an instance is completely determined by its accessibility. When a request to create a new instance cannot be satisfied because memory space is exhausted, a garbage collector removes all inaccessible instances and compacts the memory made free. The garbage collector is based on techniques described in [Hanson77].

6.3. Compiler

The tasks of the SUMMER compiler are threefold:

1) Check that the input program satisfies the rules of the context-free syntax.

2) Enforce context-sensitive syntax rules (this includes the requirement that variables should be declared, the checking of scope rules, etc.)

3) Generate SAM instructions for the input program.

Steps 1 and 2 are performed by the **parser** and step 3 is performed by the **code generator**. The organization of the compiler is shown in Figure 6.6.

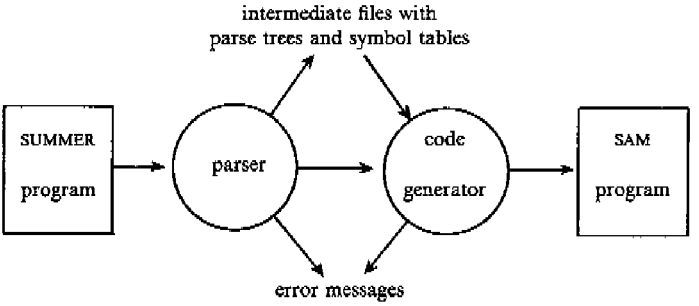

Figure 6.6. Organization of the SUMMER compiler.

The parser uses recursive descent for the syntactic analysis of statements and declarations and a bottom-up parsing method for analyzing expressions. Both parsing methods use an error recovery scheme that was first used in a PASCAL compiler [Hartmann77, Amman78]. This scheme works extremely well for SUMMER since all language constructs have a 'closed' form and distinct closing delimiters. It was decided to use a table-driven, bottom-up expression parser to simplify the recognition of user-defined operators.

The parser produces a series of intermediate files which contain a parse tree of the SUMMER program and symbol tables. These intermediate files serve as input for the code generator.

The code generator rearranges the information in the intermediate files and performs a pre-order traversal of the parse tree to generate SAM instructions.

Further details concerning the organization of the SUMMER compiler can be found in [Sint80].

6.4. References for Chapter 6

[Ammann78] Ammann, U., "Error recovery in recursive descent parsers", in: Amirchachy, M. & Neél, D., (editors), *Le Point sur la Compilation*, Institut de Recherche d'Informatique et d'Automatique, Le Chesnay, France, 1978.

[Hanson77] Hanson, D.R., "Storage Management for an Implementation of SNOBOL4", *Software Practice and Experience*, 7 (1977), 179-192.

[Hartmann77] Hartmann, A.C., *A Concurrent Pascal Compiler for Minicomputers*, Springer-Verlag, Berlin, 1977.

[Kernighan78] Kernighan, B.R. & Ritchie, D.M., *The C Programming Language*, Prentice-Hall, 1978.

[Klint79a] Klint, P., "How inefficient are stack-oriented abstract machines?", Mathematical Centre Report IW 123/79.

[Klint79b] Klint, P., "Line Numbers Made Cheap", *Communications of the ACM*, **10** (1979) 22, 557-559.

[Klint81b] Klint, P., "Interpretation Techniques", *Software Practice and Experience*, **11** (1981), 963-973.

[Sint80] Sint, H.J., "Organization of the SUMMER compiler", Internal Memorandum, Mathematical Centre, 1980.

7. EPILOGUE

7.1. Looking backward

We have come a long way since we began considering string processing in Chapter 1. A first attempt at designing a new string processing language (SPRING, see Chapter 2) resulted in a poor language but gave considerable insight into the problems involved. In Chapter 3, formal techniques were used to analyze the pattern matching method of SNOBOL4. Special attention was paid to the confusion that can arise from side-effects due to failing attempts during a pattern match. As a result of this analysis, a new model for side-effect recovery was designed and formally described.

This model, together with new insights of how string processing should be incorporated into a programming language, led to the design of the language SUMMER (Chapter 4). In order to define the semantics of SUMMER formally, an improved method for operational language definitions was developed in Chapter 5.

Finally, the language was implemented and a general outline of this implementation was given in Chapter 6.

In the current chapter we evaluate the results of this research and indicate some directions for further investigation.

7.1.1. SUMMER as a language

Language design and compromise are almost synonymous. A language designer has to reconcile simplicity, expressive power, orthogonality, economy of concepts, anticipated application area, tradition, style, taste, implementation considerations, experience, enthusiasm and available time (to mention but a few) with each other when designing a new language. Compromises are thus inevitable. Evidence of this kind of compromise can also be found in SUMMER. Some weak points of SUMMER are:

- Certain useful language features (like procedure variables and advanced array operations) were omitted in favor of simplicity.
- A very general notion of 'subject' was included in the language. This generality has (so far) not been fully exploited.
- Try-expressions do not recover the values of local variables in the 'current' procedure. This inelegant way of obtaining information from an attempt that failed was dictated by implementation considerations. An implementation technique for a more elegant solution (not recovering the values of an explicitly stated list of variables) was not discovered until it was too late.
- The mechanism to control access to class instances (fetch- and store-associations) is not sufficiently simple and elegant. This could not be improved due to lack of time.
- The string scanning functions in SUMMER are based on the functions for lexical scanning as found in SNOBOL4. It would have been better to design a new set of higher level primitives for string scanning.

- Some cosmetic changes should be made to the syntax.
- The system of dynamic types used is adequate. However, a slightly more restrictive type system would make static type checking much easier. In such a system, variables may still have values of arbitrary type, but as soon as a variable has received a value of a certain type, only values of that particular type may be assigned to it.

Some strong points of SUMMER are:

- Expression evaluation and pattern matching have been completely unified, resulting in a substantial reduction in the number of language primitives.
- Incorporation of failure handling into the expression evaluation mechanism leads to a concise notation, since the same expression can either compute a result or produce a failure signal.
- The language is based on a consistent view of side-effect recovery, thus eliminating the problems with immediate/conditional side-effects found in SNOBOL4. It turned out that the 'recovery cache' could be used effectively to implement side-effect recovery.
- The use of 'classes' as a data abstraction mechanism was a good choice. Amongst other things, this allowed us to cast the notion of 'subject string' and 'string pattern matching' into a more general framework.

Despite the mentioned shortcomings, experience shows that SUMMER is a convenient, easy to learn, language. The language has been used for the implementation of assemblers, compilers, preprocessors, a parser generator, systems for automatic type inference and data flow analysis, and, surprisingly, also for the implementation of simulators for various computer architectures such as for the Manchester data flow machine and for the architecture of the conventional microprogramming level.

7.1.2. The SUMMER implementation

The SUMMER implementation is reliable, gives good error messages, but is slow. Users tend to complain about long compilation times. The compiler is slow because it is written in SUMMER itself and because its two constituent parts communicate with each other via **symbolic** intermediate files. The compiler loses much time in converting between internal and external data representations. Long compilation times may be a serious problem for large programs, since SUMMER lacks separate compilation facilities.

In most cases, the efficiency of SUMMER programs seems to be acceptable. Generally speaking, one can say that programs which run slowly are those which perform low level operations and do not use the higher level facilities offered by the language.

7.1.3. Use of a formal definition

The semantics of SUMMER were formally described after the language had been completely designed. A considerable effort was required to develop a formal description method and to produce a formal description of the language. This effort, though much larger than anticipated, payed off: much insight was gained into the structure of the language and into errors or omissions in the design. Three lessons can be learned from this:

- Language design and formal definition should proceed hand in hand.
- The language implementation should be derived (in some automatic way) from the formal definition. In that way one can avoid inconsistencies or incompatibilities between definition and implementation.
- Several language definition methods first convert the program to be defined to an intermediate form that is more suitable to operate on. The 'parse expressions' used in the formal definition of SUMMER operate directly on the source text of the program to be defined, thus eliminating the extra conversion step and simplifying the definition.

7.2. Looking forward

The research described in this thesis can be continued in the direction of both **executable language definitions** and **programming environments**.

Executable language definitions are a valuable tool for the language designer as was explained in Chapter 5. The method used in the SUMMER definition could be improved in several ways:

- The method is not suited to the formulation (or proof) of properties of a given language definition. This can be cured by eliminating some operational aspects from the method.
- The method leads to intolerably inefficient implementations. There are two major sources of this inefficiency. First of all, statements are parsed every time they are executed. This can be avoided in several ways: one can either translate the source text to an intermediate form or maintain a 'cache' of pieces of source text that have already been parsed. Secondly, a very general and expensive technique is used to implement environment modifications. Special properties of a language (for example, the property that environments can be implemented on a stack) are not exploited. It is a non-trivial task to extract such optimization information from a given language definition.

A **programming environment** is an interactive computer system dedicated to the development and documentation of programs. When SUMMER was being designed the idea was, that, for the sake of portability, its interface with the host operating system should be kept as simple as possible. Thinking on a dedicated SUMMER environment started only after the design of the language was complete. It quickly turned out that the original austere file system interface was inadequate for use in an integrated environment. More specifically, one would like to exploit the 'class' mechanism not only locally in programs, but also at the level of external files. Besides leading to an integration of 'internal' and 'external' data types, this further suggested the use of SUMMER as a command language. At the same time, we noted a strong similarity between the language of the symbolic debugger in the SUMMER system and SUMMER itself.

Although the analogies were strong in both cases, it was also evident that SUMMER could not play the role of a unified command/programming/debugging language without extensive modification. The advantage to be gained was clear: a highly uniform programming environment. But the problems involved seemed many and we therefore decided to concentrate, not on the modification of SUMMER, but, more generally, on the basic principles underlying **monolingual** systems, i.e. systems in which

the command language, the programming language, and the language of the symbolic debugger are identical. This has resulted in [Heering81].

Having laid the foundations, the next step will be the implementation of a proof-of-concept monolingual environment. It will contain many features of SUMMER in generalized form.

7.3. References for Chapter 7

[Heering81] Heering, J. & Klint, P., "Towards monolingual programming environments", Mathematical Centre Report IW 185/81.

PART II

SUMMER Reference Manual

PREFACE FOR PART II

The second part of this thesis is devoted to the definition of the SUMMER programming language. It provides both a formal and informal language definition and tutorial examples.

In Chapter 8 the techniques and notational conventions that are used in the definition are introduced. Much attention is paid to the method used for the formal definition of semantics. Chapter 9 contains a semi-formal definition of the SUMMER kernel. This is a small subset of the language on which the semantic definition of the whole language can be based. The description of each language feature consists of its syntax, an informal as well as a formal definition of its semantics, and examples. In Chapter 10 the kernel is extended with useful data types and associated operations, such as reals, arrays, tables, files, bit strings, etc. Some complete, annotated SUMMER programs are presented in Chapter 11. And, finally, a summary of the syntax is given in Chapter 12.

Readers who are only interested in getting a general impression of the language may confine themselves to Chapter 4 in Part I and the annotated examples in Chapter 11. Readers who are not interested in the formal definition of the language may skip Chapter 8 (except Sections 8.1 and 8.2), and all subsections of Chapter 9 entitled 'Semantics'.

8. PRELIMINARIES TO THE DEFINITION OF SUMMER

In this chapter we introduce the techniques that will be used to describe the syntax and semantics of the SUMMER programming language. Section 8.1 introduces an extended form of BNF notation that will be used to describe the syntax. Section 8.2 defines the lexical primitives of SUMMER. In section 8.3 the method used for the description of semantics is introduced and, at the same time, the semantic primitives used in the SUMMER definition are defined. In section 8.4 some peculiarities and shortcomings of the definition are discussed.

8.1. Syntactic considerations

An extended form of BNF notation will be used to describe the syntax of SUMMER. It aims at providing a concise notation for the description of repeated and optional syntactic notions. A syntactic notion suffixed with '+' means one or more repetitions of that notion. A notion suffixed with '*' stands for zero or more repetitions of that notion. The notation

{ *notion separator* } *replicator*

i.e. a *notion* followed by a *separator* enclosed in braces and followed by a *replicator*, is used to describe lists of notions separated by the given separator. A separator should be a terminal symbol of the grammar. A replicator is either '+' or '*'. The replicator '+' indicates that the list consists of one or more notions. The list begins and ends with a notion. The replicator '*' indicates that the list consists of zero or more notions.

An optional syntactic notion is indicated by enclosing it in square brackets, e.g. '[*notion*]'. The terminal symbols of the grammar are represented either by their constituent characters enclosed in single quotes (for example: ',' or ':=') or by upper case letters if the terminal symbol consists solely of lower case letters (for example: both 'IF' and "*if*" may be used to denote the terminal symbol '**if**'). Single quote characters that occur in a terminal symbol are duplicated. Where necessary, parentheses are used for grouping.

A description of this formalism in its own notation is:

```
<grammar>           ::= <rule>+ .

<rule>              ::= <rule-name> '::=' <rule-body> '.' .

<rule-body>         ::= {<primary>* '|' }+ .

<primary>           ::= ( <terminal-symbol> | <rule-name> |
                          <option> | <list> | <compound>
                        ) [ '+' | '*' ] .

<option>            ::= '[' <rule-body> ']' .

<list>              ::= '{' <primary> <terminal-symbol> '}' ( '+' | '*' ) .

<compound>          ::= '(' <rule-body> ')' .

<terminal-symbol>   ::= <upper-case-letter>+ |
                        '"' <arbitrary-ascii-character>+ '"' .

<rule-name>         ::= '<' (<lower-case-letter> | ' - ')+ '>'.
```

The syntactic notions <upper-case-letter>, <lower-case-letter> and <arbitrary-ascii-character> are not further defined here, but have an obvious meaning.

In the description of the semantics an extended version of this syntax notation is used; this is further explained in section 8.3.4.

8.2. Lexical considerations

The ASCII character set is used as the basic character set of the language.

The lexical units of a program are: <delimiter>, <identifier> (including keywords, see below), <integer-constant>, <real-constant>, <string-constant> and <operator-symbol>. Lexical units may be separated by zero or more **layout symbols**: space (SP), horizontal tab (HT), newline (NL) or comment. A comment consists of a comment symbol ('#'), zero or more arbitrary characters except the comment symbol, followed by a comment symbol. At least one layout symbol is required between adjacent <identifier>s, <integer-constant>s and <real-constant>s. Except where a layout symbol occurs inside a <string-constant> or is required as separator, it may be removed from a program without changing the semantics of the program.

A <delimiter> is used as separator in lists of language constructs or to enclose (lists of) language constructs. The following <delimiter>s are defined:

<delimiter> ::= ',' | ';' | ':' | '(' | ')' | '[' | ']' .

An <identifier> is used as a name and has the form:

<identifier> ::= <letter> (<letter> | <digit> | '_')* .

The <identifier>s listed below are **keywords** and have a special significance in the language; they can not be redeclared by the programmer:

array	else	monadic	subclass
assert	end	od	subject
begin	esac	of	succeeds
case	fails	op	table
class	fetch	proc	then
code	fi	program	try
const	for	return	undefined
default	freturn	rof	until
do	if	scan	var
dyadic	in	self	while
elif	init	store	yrt

An <integer-constant> is used to denote an instance of the class **integer** {10.2} and has the form:

<integer-constant> ::= ['+' | '-'] <digit>+ .

A <real-constant> is used to denote an instance of the class **real** {10.3} and has the form:

```
<real-constant>    ::= <integer-constant> <real-exponent> |
                       [<integer-constant>] '.' <digit>+ [<real-exponent>] .

<real-exponent>    ::= 'e' <integer-constant> .
```

A <string-constant> is used to denote an instance of the class **string** {10.4} and has the form:

```
<string-constant>  ::= <single-quote> <string-item>* <single-quote> .

<string-item>      ::= <any-character-from-limited-ascii-set> |
                       '\' ('b' | 'n' | 't' | '\') |
                       '\' <digit> <digit> <digit> |
                       <single-quote> <single-quote> .
```

A <string-constant> consists of zero or more characters from a limited set (i.e. all ASCII characters except the characters: single quote (''), backslash ('\') and newline (NL)) enclosed in single quote characters. The single quote character itself can be obtained by writing two adjacent single quote characters. There are two ways to associate a printable representation with non-printable characters. The escape sequences '\b', '\n', '\t' and '\\' are used to denote respectively the characters backspace (BS), newline (NL), horizontal tab (HT) and backslash ('\'). Arbitrary non-printable characters can be denoted by '*abc*', where '*abc*' is the three-digit octal representation of the desired character in the ASCII character set.

An <operator-symbol> is used to denote a built-in or user-defined **operator** {9.1.4} and must adhere to:

```
<operator-symbol> ::= '_' (<letter> | <digit>)+ '_' |
                      ( '*' | '+' | '-' | '/' | ':' | '<' |
                        '=' | '>' | '|' | '&' | '~' | '!' |
                        '\' | '@' | '?' | '$' | '%'
                      )+ .
```

The recognition of <operator-symbol>s is described in 9.1.6.

8.3. Semantic considerations

8.3.1. Description method

An **evaluation process** or **interpreter** (with the name '*eval*') will be defined that takes an arbitrary, but syntactically correct, source text ('the source program') as input and either computes the result of the execution of that program (if it is a legal SUMMER program), or detects a semantic error, or does not terminate. In the latter two cases no meaning is attached to the program. The evaluation process operates directly on the text of the source program. During this process a global **environment** is inspected or updated. An environment is a mapping from identifiers in the source program to their actual values during the evaluation process. In this way environments determine the meaning of names in the source program and are used to describe concepts such as variables, assignment and scope rules.

A fundamental question arises here: in which language do we write the definition? Several choices can be made, such as the formalism used in denotational semantics ([Gordon79], which boils down to a mathematical notation for recursive functions and domains) or the Vienna Definition Language ([Wegner72], which is a programming language designed for the manipulation of trees). The merits of these formalisms will not be discussed here, but they have one disadvantage in common: yet another language and yet another notation have to be introduced. Since this conflicts with our aim of providing a concise, precise and readable definition, it was decided to describe the semantics of SUMMER in a subset of SUMMER itself. This choice has the obvious disadvantage that the definition is **circular**, i.e. language and metalanguage coincide. It will become clear later that we make a meticulous distinction between notions in the definition language (e.g. '1', an integer as used in the formal definition) and notions in the defined language (e.g. '*a_integer*(1)', an expression in the definition language that describes the integer '1' as it may occur in a program in the defined language). At first sight, the reader may find parts of the formal definition needlessly complicated and perhaps even confusing. In many cases, this apparent complexity is due to our making a careful distinction between the two language levels involved in the definition.

The choice of SUMMER as definition language has the practical advantage that readers who have only a moderate familiarity with the language will be able to read the definition without great difficulty. It should be noted that there is no **fundamental** reason for making the definition circular. The definition method used here would also work if, for example, ALGOL68 were used as definition language. In any case, it is essential that the definition language has powerful string operations and allows the creation of data structures of dynamically varying size. This requirement makes, for example, PASCAL less suited as definition language. Choosing SUMMER as definition language gives us the opportunity to investigate the suitability of that language in the area of language definition (see Chapter 5). [Reynolds72] contains an extensive discussion of circular language definitions.

A final remark should be made regarding the **operational** aspects of the definition method. As explained above, the meaning of each source program S is defined by the result of 'executing' the interpreter *eval* with S as input. However, the parts of S that are not visited during this execution are not checked for their semantic correctness. Thus there is no guarantee that S is free of semantic errors.

The following sections describe some aspects of the use of SUMMER as a metalanguage, the semantic domains used by the evaluation process, and the evaluation process itself.

8.3.2. SUMMER as a metalanguage

This paragraph focuses on some aspects of SUMMER that are used in the formal definition, but have not yet been covered in Chapter 4. Most of the constructs to be used in the definition have some similarity to constructs in, for instance, PASCAL and are assumed to be self-explanatory. Only less obvious constructs that are essential for the understanding of the definition are mentioned here.

SUMMER is an object-oriented language with pointer semantics. This means that an object can be modified by assignment and that such modifications are visible through all access paths to that object. For example,

$s := stack(10);$
$t := s;$

assigns one and the same *stack* object to the variables s and t, and

$s.push(v)$

pushes the value of v onto this stack. As a side-effect, the stack is modified in such a way that subsequent operations on s and t can perceive the effect of that modification. In the formal definition this is relevant to the concepts 'state' and 'environment', which are modified in this way.

The language is dynamically typed, i.e. the type of variables is not fixed statically (as in PASCAL and ALGOL60) but is only determined during the execution of the program (as in LISP and SNOBOL4). Moreover, generic operations on data structures are allowed. If an operation is defined on several data types, then the procedure to be executed when that operation is encountered is determined by the type of the (left) operand of that operation.

Control structures and data structures are self-explanatory except possibly for **arrays** and **for-expressions.**

Arrays are vectors of values, indexed by $0, \ldots, N-1$, where N is the number of elements in the array. If A is an array then the operation $A.size$ will yield the number of elements in the array. A new array is created by

$[V_0, \ldots, V_{N-1}]$

or

$array(N, V)$.

In the former case, an array of size N is created and initialized to the values $V_0, \ldots, V_{N-1}$. In the latter case, an array of size N is created and all elements are initialized to the value V. Arrays are also allowed on the left-hand side of assignments. This provides a convenient notation for multiple assignments. For example,

$x := 10; y := 20; z := 30;$

is equivalent to

$[x, y, z] := [10, 20, 30]$

and, more generally,

$x_0 := a[0]; \ldots ; x_k := a[k];$

is equivalent to

$[x_0, \ldots, x_k] := a$

The general form of a for-expressions is:

for V **in** G **do** S **od**

where V is a variable, G an expression capable of generating a sequence of values VAL_i and where S is an arbitrary statement. For each iteration the assignment $V := VAL_i$ is performed and S is evaluated. As used in the formal definition, the value of G is either an array (in which case consecutive array elements are generated)

or G is an array on which the operation *index* is performed (in which case all indices of consecutive array elements are generated). For example, in

a := [144, 13, 7];
for x **in** a **do** *print*(x) **od**

an array object is assigned to the variable a and the values 144, 13, 7 will be printed, while

for i **in** *a.index* **do** *print*(i) **od**

will print the values 0, 1, 2. Further examples of for-expressions can be found in the following paragraphs.

8.3.3. Semantic domains

A semantic domain is a set, whose elements either describe a primitive notion in the defined language (like 'variable' or 'procedure declaration') or have some common properties as far as the evaluation process is concerned. First, the relationship between these domains is given by a series of **domain equations**. Next, we give an operational definition of each domain by specifying its abstract properties.

The relationship between the domains *BASIC-INSTANCE*, *COMPOSITE-INSTANCE*, *INSTANCE*, *STORABLE-VALUE*, *DENOTABLE-VALUE*, *PROCEDURE*, *CLASS*, *LOCATION*, *STATE* and *ENVIRONMENT* is as follows:

$$
\begin{aligned}
\mathit{BASIC\text{-}INSTANCE} &= \mathit{INTEGER} \cup \mathit{STRING} \cup \mathit{UNDEFINED} \\
\mathit{COMPOSITE\text{-}INSTANCE} &= \mathit{CLASS} \times \mathit{ENVIRONMENT} \\
\mathit{INSTANCE} &= \mathit{BASIC\text{-}INSTANCE} \cup \mathit{COMPOSITE\text{-}INSTANCE} \\
\mathit{STORABLE\text{-}VALUE} &= \mathit{INSTANCE} \\
\mathit{DENOTABLE\text{-}VALUE} &= \mathit{STORABLE\text{-}VALUE} \cup \mathit{PROCEDURE} \cup \mathit{CLASS} \cup \mathit{LOCATION} \\
\mathit{PROCEDURE} &= \mathit{PROCEDURE\text{-}DECLARATION} \times \mathit{ENVIRONMENT} \\
\mathit{CLASS} &= \mathit{IDENTIFIER} \times \mathit{CLASS\text{-}DECLARATION} \\
\mathit{STATE} &= \mathit{LOCATION} \rightarrow (\mathit{STORABLE\text{-}VALUE} \cup \{\mathit{unused}\}) \\
\mathit{ENVIRONMENT} &= \mathit{IDENTIFIER} \rightarrow \mathit{DENOTABLE\text{-}VALUE}
\end{aligned}
$$

Here, *IDENTIFIER*, *PROCEDURE-DECLARATION* and *CLASS-DECLARATION* are the sets of strings that can be derived from the syntactic notions <identifier>, <procedure-declaration> and <class-declaration> in the SUMMER grammar. All values in SUMMER are an instance of some class. The domain *INSTANCE* describes all those values. *BASIC-INSTANCE* is the domain of the primitive values in the language. *COMPOSITE-INSTANCE* is the domain of user-defined values. *STORABLE-VALUE* is the domain of values which can be assigned to variables in the source program. *DENOTABLE-VALUE* is the domain of values which can be manipulated by the evaluation process. The domains *PROCEDURE* and *CLASS* describe declarations for procedures and classes. The domain *LOCATION* is used to model the notion 'address of a cell capable of containing a (single) value'. *STATE* is the domain of functions which map locations onto actual values or *unused*. *ENVIRONMENT* is the domain of functions which map names onto denotable values. These semantic domains will now be described in more detail.

The domain *BASIC-INSTANCE* describes the instances of basic, built-in classes in the defined language; it can be subdivided into three parts: the subdomains *INTEGER*, *STRING* and *UNDEFINED*, which are now described in turn.

INTEGER is the domain of the built-in class integer. The following operations are defined on it:

a_integer(intval), with *intval* an integer value of the definition language, creates a new element of *INTEGER*.

i.intval, with $i \in$ *INTEGER*, gives the value of the integer part of i. Note that for all integers n the equality *a_integer(n).intval* $= n$ holds.

is_integer(i), with $i \in$ *DENOTABLE-VALUE*, succeeds if and only if $i \in$ *INTEGER*. (More precisely, *is_integer* is a function from *DENOTABLE-VALUE* to {**succeeds**, **fails**}. Here, **succeeds** denotes successful expression evaluation (as used in SUMMER) and may be interpreted as '*true*'; **fails** denotes evaluation of a failing expression and may be interpreted as '*false*'.)

i.has(oper), with $i \in$ *INTEGER* and *oper* a string, succeeds if *oper* is an operation defined for *INTEGER* values, and is false otherwise.

i.operation(oper, actuals), with $i \in$ *INTEGER*, *oper* a string and actuals an array of *STORABLE-VALUE*s, performs the operation with the name *oper* on *INTEGER* value i with *actuals* as actual parameter list, and delivers a *STORABLE-VALUE* as result. The list of operations defined on integers and the semantics of these operations are described in 10.2.

STRING is the domain of the built-in class string. The following operations are defined on it:

a_string(stringval), with *stringval* a string, creates a new element of *STRING* consisting of the characters in *stringval*.

s.stringval, with $s \in$ *STRING*, gives the value of the string part of s. Note that for all strings t the equality *a_string(t).stringval* $= t$ holds.

is_string(s), with $s \in$ *DENOTABLE-VALUE*, succeeds if and only if $s \in$ *STRING*.

s.has(oper), with $s \in$ *STRING* and *oper* a string, succeeds if an operation with the name equal to the string value of *oper* is defined for values in *STRING* and is false otherwise.

s.operation(oper, actuals), with $s \in$ *STRING*, *oper* a string and *actuals* an array of *STORABLE-VALUE*s, performs the operation with name *oper* on the *STRING* value s with *actuals* as list of actual parameters, and delivers a *STORABLE-VALUE* as result. The list of operations defined on strings and the semantics of these operations are described in 10.4.

UNDEFINED is the domain consisting of undefined values. Undefined values can only be used in a few situations; they may be used in assignments, tests for equality or inequality, or they may be used as actual parameters. All variables are initialized to an undefined value; in that way undefined values serve to detect errors due to the use of otherwise uninitialized variables. The following operations are defined on it:

a_undefined creates a new *UNDEFINED* value.

is_undefined(u), with $u \in$ *DENOTABLE-VALUE*, succeeds if and only if $u \in$ *UNDEFINED*.

u.has(oper), with $u \in$ *UNDEFINED* and *oper* a string, is always false.

u.operation(oper, actuals), with $u \in$ *UNDEFINED*, *oper* a string and *actuals* an array of *STORABLE-VALUE*s, always results in a semantic error.

COMPOSITE-INSTANCE is the domain of instances of user-defined classes. A composite-instance consists of an element of *CLASS* (to be defined below) that describes the class to which the instance belongs and an environment that has to be used to inspect or update components in the instance. The following operations are defined:

a_composite_instance(c, e), with $c \in$ *CLASS* and $e \in$ *ENVIRONMENT*, creates a new element $i \in$ *COMPOSITE-INSTANCE*, such that *i.class_decl* $= c$ and *i.env* $= e$.

is_composite_instance(i), with $i \in$ *DENOTABLE-VALUE*, succeeds if and only if $i \in$ *COMPOSITE-INSTANCE*.

i.class_decl and *i.env*, with $i \in$ *COMPOSITE-INSTANCE*, give the values of the class declaration and environment part of i.

i.same_as(j), with $i, j \in$ *COMPOSITE-INSTANCE*, succeeds if and only if i and j are the same element of *COMPOSITE-INSTANCE*, i.e. they are the result of the same invocation of *a_composite_instance*.

INSTANCE is the domain of values that may occur during the evaluation of a program. It simply consists of the union of the domains *BASIC-INSTANCE* and *COMPOSITE-INSTANCE*. For convenience, the following operations will be used:

is_instance(i), with $i \in$ *DENOTABLE-VALUE*, succeeds if and only if $i \in$ *INSTANCE*.

is_basic_instance(i), with $i \in$ *DENOTABLE-VALUE*, succeeds if and only if $i \in$ *INTEGER*, or $i \in$ *STRING* or $i \in$ *UNDEFINED*.

STORABLE-VALUE is the domain of all values that may be assigned to variables in a program. It is identical to the domain *INSTANCE*; this stresses the fact that all values that can be created by a program can also be assigned to a variable.

DENOTABLE-VALUE is the domain of values that can be manipulated by the evaluation process itself. It consists of the union of the domain *STORABLE-VALUE* (described above) and the domains *PROCEDURE*, *CLASS* and *LOCATION*, which are now described in turn.

PROCEDURE is the domain of procedures. Each element of this domain describes one procedure declaration and contains a literal copy of the text of the procedure declaration itself and an environment that reflects all names and values available at the point of declaration. The following operations are defined:

a_proc(procdecl, e), with *procdecl* $\in$ *PROCEDURE-DECLARATION* and $e \in$ *ENVIRONMENT*, creates a new element $p \in$ *PROCEDURE*, such that *p.text* $=$ *procdecl* and *p.env* $= e$.

is_proc(*p*), with $p \in$ *DENOTABLE-VALUE*, succeeds if and only if $p \in$ *PROCEDURE*.

p.text and *p.env* give the value of the text and environment part of *p*.

CLASS is the domain of classes. Each element of this domain describes one class declaration and contains the name of the class and a literal copy of the text of the class declaration. The following operations are defined on it:

a_class(*id*, *classdecl*), with *id* $\in$ *IDENTIFIER* and *classdecl* $\in$ *CLASS-DECLARATION*, creates a new element $c \in$ *CLASS*, such that *c.name* = *id* and *c.text* = *classdecl*. {Note that, strictly speaking, the class name is already contained in the literal text of its declaration. For convenience, i.e. to easily describe the *type* function {9.2.2,10.11.5}, an additional *name* component is used.}

is_class(*c*), with $c \in$ *DENOTABLE-VALUE*, succeeds if and only if $c \in$ *CLASS*.

c.name and *c.text* give the value of the name and declaration text part of *c*.

{An aside on the asymmetric treatment of procedures and classes is appropriate here. A <class-declaration> can only appear at the most 'global' level, i.e. immediately contained in a <summer-program>. A <procedure-declaration>, however, may appear either at this most global level or may be contained in a <class-declaration>. This implies that for <procedure-declaration>s the declaring environments must be distinguished, while for <class-declaration>s always the most global environment must be used.}

The following operations are defined on *LOCATION*s and *STATE*s:

s.extend(*v*), with $s \in$ *STATE* and $v \in$ *STORABLE-VALUE*s, associates an unused location *l* with *v*, i.e. location *l*, originally associated with the value *unused*, is associated with value *v* (until later modification occurs). Location *l* is returned as value.

s.contents(*l*), with $s \in$ *STATE* and $l \in$ *LOCATION*, returns the value $v \in$ *STORABLE-VALUE* associated with location *l*. Location *l* should not be associated with *unused*.

s.modify(*l*, *v*), with $s \in$ *STATE*, $l \in$ *LOCATION* and $v \in$ *STORABLE-VALUE*, changes the value associated with location *l* to the value *v*. Location *l* should not be associated with *unused*. Location *l* is returned as value of this operation.

is_loc(*l*), with $l \in$ *DENOTABLE-VALUE*, succeeds if and only if $l \in$ *LOCATION*.

Inspection of the contents of a location does not affect either its own contents or that of any other location. Modification of the contents of a location does not affect the contents of any other location. Note that the above operations respect the following rules:

is_loc(*s.extend*(*v*))
s.contents(*s.extend*(*v*)) $\equiv$ *v*
s.contents(*s.modify*(*l*, *v*)) $\equiv$ *v*

ENVIRONMENT is the domain of environments. Environments administer

the binding between names and values and the introduction of new scopes, i.e. ranges in the program where new names may be declared (9.1). Unless stated otherwise, the following operations modify the environment to which they are applied:

e.bind(n,v), with $e \in$ *ENVIRONMENT*, $n \in$ *IDENTIFIER* and $v \in$ *DENOTABLE-VALUE*, binds name n to value v. This operation results in a semantic error if the name n is redeclared, i.e. a *bind* operation has already been performed for name n, without intervening *new_inner_scope* or *new_proc_scope* operation.

e.binding(n), with $e \in$ *ENVIRONMENT* and $n \in$ *IDENTIFIER*, gives the value previously bound to name n. A semantic error occurs if n has no binding.

e.has_binding(n), with $e \in$ *ENVIRONMENT* and $n \in$ *IDENTIFIER*, succeeds if and only if name n has been bound to a value.

e.names, with $e \in$ *ENVIRONMENT*, returns an array of strings which represent all identifiers for which a *bind* operation has been performed in the environment e. Note that $n \in e.names \equiv e.has\_binding(n)$ holds.

e.new_inner_scope, with $e \in$ *ENVIRONMENT*, marks the start of a new inner scope. This operation is used to delimit scopes in order to allow redeclarations of variables. The reverse operation ('*previous_inner_scope*') is not needed, since environments are always copied before a new scope is entered.

e.new_proc_scope, with $e \in$ *ENVIRONMENT*, marks the start of a new inner scope that coincides with a procedure boundary. This operation is needed to distinguish the local variables of the 'current' procedure from all other variables in the program. This distinction is necessary for proper environment restoration in try expressions (see below).

e.name_copy, with $e \in$ *ENVIRONMENT*, creates a copy of environment e. If one looks at environments as sequences of name-value pairs, then this operation creates a new sequence consisting of precisely the same name-value pairs, with names identical to the names in the original, and with value-parts that refer to the same values. This has the effect that modifications made to values referred to by value-parts in the original and in the copied environment, are visible in both environments. However, bindings added to one of the environments will not be part of the other.

e.partial_state_copy, with $e \in$ *ENVIRONMENT*, creates a copy of environment e in the same manner as described for *name_copy*, except that, under certain circumstances, the value referred to by a value-part is **copied.** The latter is applicable to the pairs that satisfy the following two requirements:

1) The value-part refers to a location (say, l).

2) The binding between the name-part (say, n) and the value-part occurred **before** the last *new_proc_scope* operation.

For each such pair, n is bound to a new location l' with contents v', where v' is a copy of v if $v \in$ *COMPOSITE-INSTANCE*, and v' is the same as v if $v \in$ *BASIC-INSTANCE*. The local variables of the

current procedure are distinguished by *partial_state_copy*. This is necessary for a proper treatment of <try-expression>s {9.2.8}.

8.3.4. Evaluation process

The evaluation process is described in SUMMER extended with **parse expressions**[1] of the form

'{{' <identifier> '==' <tagged-rule-body> '}}'

which provide a concise notation for parsing and extracting information from the text of the source program. The precise definition of a <tagged-rule-body> is as follows:

<tagged-rule-body> ::= ([<tag> ':'] <primary>)* .

<tag> ::= <lower-case-letter> (<lower-case-letter> | <digit>)* .

These syntax rules extend the syntax notation given in Section 8.1.

A parse expression succeeds if the identifier on the left-hand-side of the '==' sign has a string as value and if this string has the form described by the <tagged-rule-body> on the right of the '==' sign. All <tag>s occurring in the <tagged-rule-body> should have been declared as variables in the program containing the parse expression, in this case the evaluation process. Substrings of the parsed text recognized by the syntactic categories on the right-hand-side of a ':' symbol are assigned to the variable that occurs on its left-hand-side. Consider, for example, the following program fragment:

```
if {{ e == WHILE t:<test> DO b:<body> OD }}
then
  put('While expression recognized')
fi
```

The parse expression will succeed if *e* has the form of a 'while expression'. The literal text of the <test> is then assigned to variable *t* and the text of the <body> is assigned to variable *b*.

If the recognized part of the text is a list or repetition, an array of string values is assigned to the variable. In the case of a list of notions separated by separators, the latter are omitted and only the notions occurring in the list are assigned to (consecutive) elements in the array. This is exemplified by:

```
if {{ e == VAR list:{<identifier> ','}+ }}
then
  put('Variable declaration contains:');
  for id in list do put(id) od
fi .
```

The parse expression succeeds if *e* has the form of a 'variable declaration' (i.e. the

1) There is no **fundamental** reason for introducing this rather *ad hoc* language extension. However, the disadvantage of introducing it is more than compensated by the fact that it is sufficiently similar to BNF notation to be almost self-explanatory. The effect of introducing a language extension as proposed here is interesting in its own right but this falls outside the scope of the current discussion.

keyword 'var' followed by a list of <identifier>s separated by commas) and in that case an array of string values corresponding to the <identifier>s occurring in the declaration is assigned to the variable *list*, which is subsequently printed.

```
var ENV;
var ENVglobal;
var STATE;
var varinit;
proc ERROR . . . ;
      # Immediately aborts the evaluation process #
proc eval_call(procname, actuals) . . . ;
      # Evaluates procedure calls; this includes the #
      # creation of new class instances. #
proc eval_field_selection(access_type, object, field, actuals) . . . ;
      # Evaluates a field selection. #
proc has_field(access_type, object, field) . . . ;
      # Utility procedure to determine whether a given #
      # field selection can be performed. #
proc expand_super_class(c) . . . ;
      # Utility procedure used for class declarations. #
proc eval_array_init(sz, def, initexpr) . . . ;
      # Evaluates array initializations. #
proc eval_table_init(sz, def, initexpr) . . . ;
      # Evaluates table initializations. #
proc eval(e)
( var v, sig, . . . ;
  if {{ e == <summer-program> }}
  then
    . . .
    return([v, sig])
  fi;
  if {{ e == <variable-declaration> }}
  then
    . . .
    return([v, sig])
  fi;
  . . .
  if {{ e == <empty> }}
  then
    . . .
    return([v, sig])
  fi;
);
```

Figure 8.1. General organization of evaluation process.

Parse expressions may be used in **if**-expressions or may stand on their own. In the latter case, the string to be parsed has to be of the form described by the parse expression. In this way, parse expressions can be used to decompose a string with a known form into substrings.

Now we turn our attention to the evaluation process. Its overall structure is given in Figure 8.1. Five utility procedures (*require_constant_expression*, *equal*, *string_equal*, *substring* and *dereference*) are not shown there.

The variables *ENV* and *ENVglobal* have as respective values the current environment and the environment at the moment that all global declarations have been evaluated. The variable *STATE* has the current state as value. The variable *varinit* has as value a string consisting of the text of all <variable-initialization>s in the current <block>.

The main procedure is *eval*, which selects an appropriate case depending on the syntactic form of its argument *e*. The details of these various cases will be given in the next chapter. The evaluation process is initiated by creating an initial empty environment *ENV* and by calling *eval* with the text of the source program as argument. If the evaluation process is not terminated prematurely (by the detection of a semantic error), the result of the evaluation of the source program can be obtained from the resulting environment *ENV*. It is assumed that *eval* is initially called with a syntactically correct SUMMER program, i.e. a string that has the form of a <summer-program>.

The definition of SUMMER has been profoundly influenced by the success-directed evaluation scheme in the language: an expression can either **fail** or **succeed**. The meaning of failure is that evaluation of the 'current' expression is abandoned and that evaluation is continued at a point where a 'handler' (i.e. <if-expression>, <while-expression>) occurs to deal with the failure case. A similar situation exists for <return-expression>s, which terminate the evaluation of a procedure call and thereby abandon the evaluation of (possibly nested) expressions. Both language features can thus influence the flow-of-control in a program.

How are these properties of SUMMER reflected in the definition? The procedure *eval* delivers as result an array of the form [*value*, *signal*], where *value* is the actual result of the procedure and *signal* is a success/failure flag that indicates how *value* should be interpreted. The signal is used to describe the occurrence of failure and/or <return-expression>s and may have the following values:

N: evaluation terminated normally.

F: evaluation failed.

NR: normal return; a <return-expression> was encountered during evaluation.

FR: failure return; a failure return was encountered during evaluation.

The signal is tested after each (recursive) invocation of *eval*. In most cases *eval* performs an immediate return if the signal is not equal to *N* after the evaluation of a subexpression. Exceptions to this rule are of two kinds:

- The semantics of certain constructs is such that the flow of control is intentionally influenced by the success or failure of expressions (e.g. <test>s in <if-expression>s). This corresponds in *eval* to appropriate reactions to *N* and *F* signals. Aborting the evaluation of the 'current' expression, which is necessary if failure occurs in a deeply nested subexpression, can be achieved by passing

an *F* signal upwards until it reaches an incarnation of *eval* that can take appropriate measures.

- The semantics of the <return-expression> is such that the execution of the procedure in which it occurs is terminated and that execution is to be continued at the place of invocation. This is reflected by the signal values *FR* and *NR*, that are only **generated** by <return-expression>s and are only **handled** by the semantic rules associated with procedure calls. The latter rules turn *NR* into *N* and *FR* into *F* before the evaluation process is resumed at the point where it left off to perform the (by then completed) procedure call. All other semantic rules return immediately when an *NR* or *FR* signal occurs.

8.4. Features not specified in the definition

Chapters 8, 9 and 10 form a nearly complete definition of the SUMMER programming language. There are, however, a few language features that are left unspecified in the formal definition. These features are now briefly summarized.

Chapters 8 and 9 define a **kernel** of SUMMER, i.e. a small subset of the language that allows a semantic description of the whole language using only primitives in this kernel. In chapter 10 these primitives are used to define more elaborate data types. To reduce the size of the definition of the kernel, the availability of three data types (classes) is assumed in the definition of the kernel which are defined only informally in chapter 10. These classes are: *real* {10.3}, *array* {10.5} and *table* {10.7}.

The definition does not formally define the **priorities of operators**. It is assumed that all expressions are fully parenthesized in order to establish the relative priorities of monadic and dyadic operators. This is further discussed in Sections 9.2.14, 9.2.15, 9.2.16 and 9.2.17.

The **operating system interface** is not specified. This is apparent in the way arguments are passed from the operating system to SUMMER programs {9.1} and in the way SUMMER programs communicate with the operating system if certain operations on files have to be performed. Except for opening and closing files, all file operations can be modeled by means of string operations. Only two additional semantic primitives (open and close file) would then be needed to describe files completely. For reasons of simplicity, these two primitives have not been included in the current definition.

8.5. References for chapter 8

[Gordon79] Gordon, M.J.C., *The Denotational Description of Programming Languages*, Springer-Verlag, 1979.

[Liskov77] Liskov, B., Snyder, A., Atkinson, R. & Schaffert, C., "Abstraction mechanisms in CLU", *Communications of the ACM*, **20** (1977) 8, 564-576.

[Reynolds72] Reynolds, J.C., "Definitional interpreters for higher-order languages", *Proceedings ACM Annual Conference*, 1972, 717-740.

[Wegner72] Wegner, P., "The Vienna Definition Language", *Computing Surveys*, **4** (1972), 5-63.

9. A SEMI-FORMAL DEFINITION OF THE SUMMER KERNEL

9.1. Declarations

Declarations introduce new names into the current environment and generally associate a value with those names. The effect of a declaration is limited by the **scope** in which it occurs. A scope is a part of a program that can be derived from one of the syntactic notions <summer-program>, <procedure-declaration>, <operator-declaration>, <class-declaration> or <block>. Scopes can be nested. All names defined in one scope must be distinct, but names in different scopes may be the same. In the formal definition a new scope is introduced by the operations *new_inner_scope* and *new_proc_scope* {8.3.3}.

9.1.1. Summer program

9.1.1.a. Syntax

<summer-program> ::=
    (<variable-declaration> | <constant-declaration> |
      <procedure-declaration> | <operator-declaration> |
      <class-declaration> | <operator-symbol-declaration>
)*
    <program-declaration> .

<program-declaration> ::=
    PROGRAM <identifier> '(' [<identifier>] ')' [<expression>] .

9.1.1.b. Pragmatics

All syntactically correct programs are derived from the non-terminal <summer-program>, the start symbol of the grammar. The evaluation of a <summer-program> proceeds in three steps:

1. Evaluate all <variable-declaration>s {9.1.2}, <constant-declaration>s {9.1.3}, <procedure-declaration>s {9.1.4}, <operator-declaration>s {9.1.4}, <class-declaration>s {9.1.5} and <operator-symbol-declaration>s {9.1.6} immediately contained in the <summer-program>. We will refer to the resulting environment as the **global environment**. It is described by *ENVglobal* in the formal definition.

2. Evaluate all non-empty <variable-initialization>s {9.1.2} in <variable-declaration>s immediately contained in the <summer-program>. The text of these <variable-initialization>s has been accumulated in the previous step and is available as the value of *varinit*.

3. If the <program-declaration> has a formal parameter (it may have at most one) then obtain, in a way left unspecified in this definition, an array of string values that correspond to the actual parameters of the invocation of the <summer-program> from the command level of the operating system. If, for example, some <summer-program> *P* with formal parameter *args* is invoked by the operating system command line:

    P −x abc 1 3

    then *args* gets a value as if the assignment

args := ['−*x*', '*abc*', '*1*', '*3*']

had been performed. Note that the command syntax is operating system dependent.

4. Evaluate the <expression> (9.2) part of the <program-declaration> in the environment established in step 1 and the state initialized in steps 2 and 3 above.

9.1.1.c. Semantics

```
if {{ e == decllist: ( <variable-declaration> | <constant-declaration> |
                       <procedure-declaration> | <operator-declaration> |
                       <class-declaration> | <operator-symbol-declaration>
                     )*
          progdecl: <program-declaration> }}
then
  var body, decl, name, progargs, v, sig;
  varinit := '';
  for decl in decllist
  do [v, sig] := eval(decl);
     if sig ~= N then ERROR fi
  od;
  for name in ENV.names
  do var b := ENV.binding(name);
     if is_proc(b) then b.env := ENV fi
  od;
  ENVglobal := ENV;
  [v, sig] := eval(varinit);
  if sig ~= N then ERROR fi;
  {{ progdecl == PROGRAM <identifier> '(' progargs:[<identifier>] ')'
                 body:[<expression>]
  }};
  ENV.new_proc_scope;
  if {{ progargs == <identifier> }}
  then
    ENV.bind(progargs, STATE.extend(get_program_arguments))
  fi;
  [v, sig] := eval(body);
  return([v, sig]);
fi;
```

Notes

1) The list of declarations is evaluated from left to right.

2) The environment component of all *PROCEDURE* values in the current environment is adjusted, in order to resolve forward references. This is fully discussed in Section 9.1.4.

3) Regarding the treatment of signals other than N, <variable-initialization>s are treated differently when they occur in a <summer-program> (above), in an <identifier-or-call> related to a class creation procedure {9.2.2}, or in a <block> {9.2.11}.

4) The parse-expression '{{ *progdecl* == PROGRAM ... }}' always succeeds. Its only purpose is to extract components from the <program-declaration>.

5) The function *get_program_arguments* delivers the arguments of the program as described previously. It is not further specified in this definition.

9.1.2. Variable declarations

9.1.2.a. Syntax

<variable-declaration> ::= VAR { <variable-initialization> ',' }+ ';' .

<variable-initialization> ::= <identifier> [':=' <expression>] .

9.1.2.b. Pragmatics

A <variable-declaration> introduces a series of new variables, i.e. names of locations whose contents may be inspected and/or modified, into the current environment. The declaration may contain <expression>s {9.2} whose value is to be used for the initialization of the declared variables. If the <variable-declaration> is immediately contained in a <summer-program>, then these initializing expressions are evaluated prior to the evaluation of the <program-declaration> {9.1.1} in that <summer-program>. If the <variable-declaration> is immediately contained in a <class-declaration> {9.1.5}, then these initializations are evaluated prior to the evaluation of the **init**-part of that <class-declaration> {9.2.2}. Otherwise, the initializing expressions are evaluated prior to the evaluation of the <expression> part of the <block> {9.2.11} in which the <variable-declaration> occurs. In the formal definition this is described by appending variable initializations to the string value of variable *varinit* and by evaluating that value at appropriate moments (i.e. before the evaluation of a <program-declaration>, the **init**-part of a <class-declaration>, or the <expression> part of a <block>).

9.1.2.c. Semantics

```
if {{ e == VAR varlist:{<variable-initialization> ',' }+ ';' }}
then
  var name, onevar;

  for onevar in varlist
  do if {{ onevar == name:<identifier> ':=' <expression> }}
     then
       varinit := varinit || onevar || ';'
     else
       {{ onevar == name:<identifier> }}
     fi;
     ENV.bind(name, STATE.extend(a_undefined))
  od;
  return([a_undefined, N])
fi;
```

Notes

1) *varinit* is initialized to the empty string before the evaluation of a <summer-program> {9.1.1}, of a class creation procedure {9.2.2}, or of a block {9.2.11}.

2) The operator '||' denotes concatenation of strings {10.4}.

9.1.2.d. Examples

1) **var** *x*;

2) **var** *x*, *y*, *z*;

3) **var** *x* := 3, *y*, *z* := *x* + 5;

9.1.3. Constant declarations

9.1.3.a. Syntax

<constant-declaration> ::= CONST { <constant-initialization> ',' }+ ';' .

<constant-initialization> ::= <identifier> ':=' <constant-expression> .

9.1.3.b. Pragmatics

A <constant-declaration> introduces a series of new constants, e.g. names with which one, unalterable, value is associated, into the current environment.

9.1.3.c. Semantics

if {{ *e* == CONST *constlist*:{<constant-initialization> ',' }+ ';' }}
then
  var *cexpr*, *name*, *oneconst*, *v*, *sig*;

  for *oneconst* **in** *constlist*
  do {{ *oneconst* == *name*:<identifier> ':=' *cexpr*:<constant-expression> }};
    require_constant_expression(*cexpr*);
    [*v*, *sig*] := *eval*(*cexpr*);
    if *sig* ~= *N* **then** *ERROR* **fi**;
    ENV.bind(*name*, *v*);
  od;
  return([*a_undefined*, *N*])
fi;

Notes

1) The function *require_constant_expression* {9.2.18} ensures that a given <expression> only contains constants.

2) The evaluation of <constant-expression>s is defined in 9.2.18 and is identical to the evaluation of 'ordinary' <monadic-expressions> {9.2.16} and <dyadic-expressions> {9.2.17}.

9.1.3.d. Examples

1) **const** *a* := 1;

2) **const** *a* := 1, *b* := 3 * (*a* + 1), *c* := *'abc'* ;

9.1.4. Procedure and operator declarations

9.1.4.a. Syntax

<procedure-declaration> ::=
PROC <identifier> <formals> [<expression>] ';' .

<operator-declaration> ::=
OP <operator-symbol> <formals> [<expression>] ';' .

<formals> ::= '(' { <identifier> ',' }* ')' .

9.1.4.b. Pragmatics

Both procedure and operator declarations associate a piece of program with a certain name or operator symbol. This association is made at the moment of declaration. How this piece of program can be invoked later on is described in sections 9.2.2, 9.2.16 and 9.2.17.

9.1.4.c. Semantics

if {{ *e* == PROC *id*:<identifier> <formals> [<expression>] ';' }} |
{{ *e* == OP *id*:<operator-symbol> <formals> [<expression>] ';' }}
then
ENV.bind(*id*, *a_proc*(*e*, *undefined*));
return([*a_undefined*, *N*])
fi;

Notes

1) An *a_proc* object with undefined environment component is created. This environment component will be replaced by a well-defined environment in either of two ways. If the declared procedure is global (i.e. if its declaration is immediately contained in the <summer-program> {9.1.1}), the undefined environment component is replaced by the global environment before the evaluation of the <program-declaration>. In this way forward references are dealt with. If the declared procedure is, on the other hand, contained in a <class-declaration> {9.1.5}, the undefined environment component is replaced by a well-defined environment at the moment that an instance of the class is created {9.2.2}.

9.1.4.d. Examples

1)
```
proc fac(n)
( if n = 0
  then
    return(1)
  else
    return(n * fac(n - 1))
  fi
);
```

{The procedure *fac* computes the factorial function.}

2) **proc** *positive*(*x*) *return*($x > 0$);

{The procedure *positive* succeeds if its argument is greater than zero and fails otherwise. A typical use of this procedure is:

if *positive*(*x*) **then** *put*('*x is positive*') **fi**

}

3) **op** +? (*x*) *return*($x > 0$);

{The user-defined monadic operator '+?' has the same effect as the procedure *positive* of example 2 above. A typical use is:

if +? *x* **then** *put*('*x is positive*') **fi**

}

9.1.5. Class declarations

9.1.5.a. Syntax

<class-declaration> ::=
CLASS <identifier> <formals>
BEGIN <subclass-declaration>
<fetch-associations> <store-associations>
(<variable-declaration> | <constant-declaration> |
<procedure-declaration> | <operator-declaration>
)*
[INIT ';' <block>]
END <identifier> ';' .

<subclass-declaration> ::= [SUBCLASS OF <identifier> ';'] .

<fetch-associations> ::= [FETCH <associations> ';'] .

<store-associations> ::= [STORE <associations> ';'] .

<associations> ::= { <association> ',' }+ .

<association> ::= <field-identifier> [':' <identifier>] .

<field-identifier> ::= <identifier> | <operator-symbol> .

9.1.5.b. Pragmatics

Classes form a data abstraction mechanism and provide the only means of declaring new data types. A <class-declaration> introduces a new data type or **class** and all operations which may be performed on objects or **instances** of that class. An instance of a class may be looked upon as consisting of a number of **fields**. Fields may either be passive (and act as simple variables) or be active (and act as procedures).

The <class-declaration> may contain declarations for variables, constants, procedures and operators. All entities so declared can be used freely inside the class, but access from the outside is completely controlled by <fetch-associations> and <store-associations>, which determine the names that are visible outside the class. The former specify which parts of the class instance may be used ('fetched'), the latter

specify which components may be modified ('stored in'), i.e. may occur on the left hand side of an assignment operator {9.2.17}. Both kinds of associations allow the specification of a procedure in the <class-declaration> which will perform the actual access. The process of inspecting or modifying the value of one of the fields of a class is called **field selection** {9.2.14}.

A new instance of a class is created by invoking a **class creation procedure**, which is derived from the class declaration, as described in section 9.2.2.

A new class may inherit properties from a previously declared class; this is indicated by a <subclass-declaration>. Roughly speaking, the declaration of a class whose name occurs in a <subclass-declaration> is literally substituted for that <subclass-declaration>. If a <class-declaration> C contains a non-empty <subclass-declaration> and that <subclass-declaration> contains the <identifier> C', then C is said to be a **subclass** of C', or conversely, C' is said to be a **superclass** of C. In principle, C inherits all of the properties of C', unless they are explicitly redefined in C. This notion of inheritance is realized by adding (parts of) the <class-declaration> of C' to the <class-declaration> of C. The procedure *expand_super_class* performs this transformation.[1] Two additional names for parts of a <class-declaration> will be used in the following algorithm: **class-declaration-part** will be used to denote all <variable-declaration>s, <constant-declaration>s, <procedure-declaration>s and <operator-declaration>s contained in the <class-declaration>; **class-initialization-part** will be used to denote the <block> immediately following the '**init**' keyword in the <class-declaration>. The following steps describe the process of superclass expansion in detail:

1. **Formal parameter correspondence**. The formal parameters of C should be an **extension** of the formal parameters of C': if the number of formal parameters of C and C' is N and N' respectively, then $N' \leq N$ should hold and the names of the first N', corresponding, parameters of C and C' should be the same.

2. **Check compatibility of declarations**. If a name A is declared in the class-declaration-parts of both C and C', then those declarations should be similar, i.e. A should in both cases be declared by the same kind of declaration. This forbids, for instance, that variables are redeclared as procedures and vice versa. This restriction ensures that the result of step 6 (see below) is well-defined.

3. **Superclass transformation**. If the declaration of C' contains a non-empty <subclass-declaration> then the declaration of C' should first be transformed as described in this paragraph.

4. **Combine associations.** <fetch-associations> and <store-associations> are inherited from C', unless they are redefined in C. An <association> A occurring in C' is said to be redefined if and only if the <field-identifier> contained in A, also occurs as <field-identifier> in either <store-associations> or <fetch-associations> of C. All <associations> not redefined in the <fetch-associations> of C' are added to the <fetch-associations> of C. All <associations> not redefined in the <store-associations> of C' are added to the <store-associations> of C.

1) The formal definition of this procedure is not given.

5. **Combine declaration parts.** Prefix the class-declaration-part of C with the class-declaration-part of C' from which all redefined declarations have been removed.

6. **Combine init parts.** Prefix the class-initialization-part of C with the class-initialization-part of C'.

9.1.5.c. Semantics

```
if {{ e == CLASS cname1:<identifier> <formals>
          BEGIN <subclass-declaration>
                <fetch-associations> <store-associations>
                ( <variable-declaration> | <constant-declaration> |
                  <procedure-declaration> | <operator-declaration>
                )*
                [INIT ':' <block>]
          END cname2:<identifier> ';' }}
then
  var e1;

  e1 := expand_super_class(e);
  if ~string_equal(cname1, cname2) then ERROR fi;
  ENV.bind(cname1, a_class(cname1, e1));
  return([a_undefined, N])
fi;
```

Notes

1) The result of *expand_super_class* is a string that has the form of a <class-declaration> containing an empty <subclass-declaration>.

9.1.5.d. Examples

1)
```
class complex(re, im)
begin fetch re, im;
      store re, im;
end complex;
```

{Defines the class *complex* with fields *re* and *im*. Both fields may be inspected ('fetched') and modified ('stored'). A typical use of this class is:

```
c := complex(1.4, 3.7);
x := c.re
```

First, a new instance {9.2.2} of class *complex* is assigned to variable *c*. Next, the field *re* is fetched from that instance. The net effect is that 1.4 is assigned to variable *x*.}

2)
```
class stack(n)
begin fetch push, pop;
      var sp, space;
      proc push(x)
      ( if sp = n
        then
          freturn     # stack overflow #
```

```
        else
          space[sp] := x; sp := sp + 1;
          return(x)
        fi
      );
      proc pop()
      ( if sp = 0
        then
          freturn      # stack underflow #
        else
          sp := sp - 1; return(space[sp])
        fi
      );
init: sp := 0; space := array(n, undefined)
end stack;
```

{Defines the class *stack* with operations *push* and *pop*. Note that only these fields are accessible from the outside. A typical use of this class is:

```
s := stack(10);
s.push(1); s.push(2);
x := s.pop; y := s.pop
```

which assigns, ultimately, 2 to x and 1 to y.}

3)
```
class random_access_stack(n)
begin subclass of stack;
      fetch access;
      proc access(i)
      ( if i >= 0 & i < sp
        then
          return(space[i])
        else
          freturn      # out of range #
        fi
      );
end random_access_stack;
```

{Declares the class *random_access_stack*: a kind of stack that not only defines the operations *push* and *pop*, but also defines the operation *access* to inspect the value of an arbitrary element on the stack. This declaration uses a <subclass-declaration> to extend the class *stack* given in the previous example with the new operation. The above declaration is completely equivalent with the following declaration:

```
class random_access_stack(n)
begin fetch push, pop, access;
      var sp, space;
      proc push(x)
      ( if sp = n
        then
          freturn      # stack overflow #
        else
```

```
            space[sp] := x; sp := sp + 1;
            return(x)
          fi
        );
        proc pop()
        ( if sp = 0
          then
            freturn      # stack underflow #
          else
            sp := sp - 1; return(space[sp])
          fi
        );
        proc access(i)
        ( if i >= 0 & i < sp
          then
            return(space[i])
          else
            freturn      # out of range #
          fi
        );
    init: sp := 0; space := array(n, undefined)
    end random_access_stack;
```

This second declaration of *random_access_stack* is the result of applying *expand_super_class* to the first declaration of *random_access_stack* given above.}

9.1.6. Operator symbol declarations

9.1.6.a. Syntax

<operator-symbol-declaration> ::=
(MONADIC | DYADIC) { <operator-symbol> ',' }+ ';' .

9.1.6.b. Pragmatics

An <operator-symbol-declaration> indicates that the <operator-symbol>s contained in it denote user-defined monadic or dyadic operators. An <operator-symbol-declaration> is only required for <operator-symbol>s that are used before they are declared (in an <operator-declaration> {9.1.4}). Since arbitrary sequences of <operator-symbol>s can follow each other without intervening layout symbols, it is necessary to have rules for disambiguating certain combinations of <operator-symbol>s. For example, should

$$x + - y$$

be interpreted as

$$x + (-y)$$

with + dyadic and − monadic, or as

$$x + - y$$

with + − dyadic? The following rules describe the decomposition of adjacent <operator-symbol>s. An <operator-symbol> occurring at the syntactic position of a <monadic-operator> is decomposed in one or more 'monadic' <operator-symbol>s. An <operator-symbol> occurring at the position of a <dyadic-operator> is decomposed into one 'dyadic' <operator-symbol> and zero or more 'monadic' <operator-symbol>s. These rules are sufficient because there are no postfix operators in SUMMER.

The decomposition rules are now described in detail. For each <operator-symbol> that occurs at the position of a <monadic-operator> (or <dyadic-operator>) the longest initial substring *S* of the <operator-symbol> is taken such that:

1. *S* is of the form '_' (<letter> | <digit>)+ '_', or
2. *S* is a <monadic-operator> (or <dyadic-operator>), and either *S* occurs as <operator-symbol> in an <operator-declaration> with one (or two) <formals>, or *S* occurs as <operator-symbol> in a <operator-symbol-declaration> that contains the keyword **'monadic'** (or **'dyadic'**).

If no such substring exists, the original <operator-symbol> is not identified. If *S* is a proper substring of the <operator-symbol>, the tail of the <operator-symbol> should be decomposable into one or more <monadic-operator>s (in both cases).

9.1.6.c. Semantics

<operator-symbol-declaration>s only affect the lexical structure of a program but have no associated semantics.

Notes

1) An <operator-symbol-declaration> may only occur at the outermost level of declarations, i.e. immediately contained in a <summer-program>.
2) An <operator-symbol> may not occur in both 'monadic' and 'dyadic' <operator-symbol-declaration>s.

9.1.6.d. Examples

1) **monadic** +?, **, @ ;

   {Declares the three <operator-symbol>s '+?', '**' and '@' as monadic operators. After this declaration, the expression *a***b* will be interpreted as *a**(**b*).}

2) **dyadic** <=>, **, *_matvec_* ;

   {Declares the three <operator-symbol>s '<=>', '**' and '*_matvec_*' as dyadic operators. After this declaration, the expression *a***b* will be interpreted as *a***(*b*).}

9.2. Expressions

SUMMER is an expression oriented language: most language constructs can be derived from the syntactic notion <expression> and can deliver a value. The main rules related to expressions are:

<expression> ::= <dyadic-expression> .

<monadic-expression> ::= (<monadic-operator>)* <primary> .

<monadic-operator> ::= '~' | '-' | <operator-symbol> .

<dyadic-expression> ::=
    <monadic-expression> (<dyadic-operator> <monadic-expression>)* .

<dyadic-operator> ::=
    '+' | '-' | '*' | '/' | '%' | '||' | '<' | '<=' | '>' | '>=' |
    '=' | '~=' | ':=' | '&' | '|' | <operator-symbol> .

<primary> ::= <unit> (<subscript> | <select>)* .

<unit> ::=
    <constant> | <identifier-or-call> | <return-expression> |
    <if-expression> | <case-expression> | <while-expression> |
    <for-expression> | <scan-expression> | <try-expression> |
    <assert-expression> | <parenthesized-expression> |
    <array-expression> | <table-expression> .

Note that the rules for <monadic-operator> and <dyadic-operator> are ambiguous, i.e. their first alternatives are all subsumed by the last one that refers to an <operator-symbol>. Nonetheless, this distinction has been made to stress the privileged position of certain operators (regarding, for instance, syntactic priority). See also Sections 9.2.16 and 9.2.17.

A more detailed description of <expression>s follows.

9.2.1. Constants

9.2.1.a. Syntax

<constant> ::= <string-constant> | <integer-constant> | <real-constant> .

9.2.1.b. Pragmatics

<constant>s provide an alternative, more convenient, notation for the values of some built-in classes. When a constant is encountered during evaluation, a new instance of the corresponding class is created.

9.2.1.c. Semantics

The semantics of <integer-constant>s is:

if {{ *e* == *expr*:<integer-constant> }}
then
  return([*a_integer*(*expr*), *N*])
fi;

The semantics of <string-constant>s is:

```
if {{ e == expr:<string-constant> }}
then
  expr := substring(expr, 1, expr.size − 2);
  return([a_string(expr), N])
fi;
```

The semantics of <real-constant>s is:

```
if {{ e == expr:<real-constant> }}
then
  return(eval_call('real', a_string(expr)))
fi;
```

Notes

1) The first and last character (i.e. the surrounding quotes) of a <string-constant> are first deleted before a new instance of class *string* is created.

2) The semantics of the class *real* are not given as part of the definition of the SUMMER kernel {8.4}. Here, it is only established that <real-constant>s are denotations for instances of the class *real*; the semantics of the class *real* itself is described informally in the next chapter {10.3}.

9.2.1.d. Examples

1) *'star\twars'*

2) 314

3) $-3.14e+5$

9.2.2. Identifiers and procedure calls

9.2.2.a. Syntax

<identifier-or-call> ::= <identifier> [<actuals>] .

<actuals> ::= '(' { <expression> ',' }* ')' .

9.2.2.b. Pragmatics

When an <identifier> occurs in an <expression> {9.2}, it depends on the existence and kind of binding of that <identifier> what will be the result of its evaluation. If the <identifier> is bound to a value in *INSTANCE* or *LOCATION*, that value will be the result. This value is **not** dereferenced {9.3.1}, i.e. if the value is a location, that location itself and not its contents are the result. If the <identifier> is bound to a value in *PROCEDURE* or *CLASS*, the <identifier> is evaluated as a procedure call (see below). If none of the above cases applies, a field selection {9.2.14} from the current value of *subject* is performed {9.2.9}.

A procedure call serves the purpose of temporarily suspending the current evaluation and starting the evaluation of the procedure associated with the <identifier> in the <identifier-or-call>. The <identifier> may be followed by a (possibly empty) list of <actual>s. The <expression>s in <actuals> are evaluated from left to right. The dereferenced value of each <expression> is passed as actual parameter to the called procedure. Since that value may be an instance of a class that

allows modification of its instances (say, an array {10.5}), modifications made to that instance are visible in the calling context. However, the contents of locations bound to local variables of the caller can never be modified by the called procedure. In CLU [Liskov77], this parameter passing mechanism has been named **call-by-sharing**.

The procedure call itself is handled by *eval_call*. If the name of the procedure to be called is either *string* {10.4}, or *integer* {10.2}, or *undefined* {8.3}, or *type* {10.11.5}, then the call is treated separately by *eval_standard_procedure* as can be seen in the formal definition. Otherwise, there are two cases, depending on whether the <identifier> in the procedure call is bound to a value in *PROCEDURE* or in *CLASS*.

In the former case, the procedure has to be evaluated in the environment of its declaration. To this end, the current environment is saved, a copy of the declaration environment of the procedure is made the current environment and the <formals> in the associated <procedure-declaration> are bound from left to right to the values of the <actuals> in the call. Next, the body of the procedure is evaluated and the result of this evaluation is the result of the procedure call. Finally, the previous (saved) environment is restored. Note how in the formal definition (see below), an *FR* signal is turned into an *F* signal, and how an *NR* signal is turned into an *N* signal; this was already discussed in Section 8.3.4.

If the <identifier> in the procedure call is bound to a value in *CLASS*, a new instance of that class must be created. This amounts to creating a new composite-instance that contains an, appropriately adjusted, copy of the declaration environment of that class. To this end, the following steps are taken. First, the current environment is saved. Next, a copy of the declaration environment is created and the <formals> are bound to actual values as described above. In addition to this, all declarations in the associated <class-declaration> are evaluated and a new instance is created, containing the name of the class and the environment just constructed. The new instance is bound to the identifier *self* in the new environment. If the <class-declaration> contains an init-part, then the <block> following the '**init**' keyword is evaluated in the new environment. Finally, the original environment is restored and the instance is returned as the result of the procedure call.

9.2.2.c. Semantics

The semantics of an <identifier-or-call> is:

```
if {{ e == id:<identifier> act:[<actuals>] }}
then
  var actual_vals, exprlist;
  if {{ act == '(' exprlist:{<expression> ',' }* ')' }}
  then
    var i, v, sig;
    actual_vals := array(exprlist.size, undefined);
    for i in exprlist.index
    do [v, sig] := eval(exprlist[i]);
       if sig ~= N then return([v, sig]) fi;
       actual_vals[i] := dereference(v);
    od;
  else
```

```
    actual_vals := array(0, undefined);
  fi;
  if ENV.has_binding(id)
  then
    var idbinding := ENV.binding(id);
    if is_instance(idbinding) | is_loc(idbinding)
    then
      if {{ e == <identifier> }}
      then
        return([idbinding, N])
      else
        ERROR
      fi
    elif is_proc(idbinding) | is_class(idbinding)
    then
      return(eval_call(id, actual_vals))
    else
      ERROR
    fi
  else
    return(eval_field_selection('fetch', ENV.binding('subject'), id, actual_vals))
  fi
fi;
```

The definition of *eval_call* is:

```
proc eval_call(procname, actuals)
( var procdef;
  if procname = 'string' | procname = 'integer' |
    procname = 'undefined' | procname = 'type'
  then
    return([eval_standard_procedure(procname, actuals), N])
  fi;
  procdef := ENV.binding(procname);
  if is_proc(procdef)
  then
    var body, ENV1, formals, i, v, sig;
    {{ procdef.text ==
          PROC <identifier> '(' formals: {<identifier> ','}* ')'
          body: [<expression>] ';'
    }};
    if actuals.size ~= formals.size then ERROR fi;
    ENV1 := ENV;
    ENV := procdef.env.name_copy;
    ENV.new_proc_scope;
    for i in actuals.index
    do ENV.bind(formals[i], STATE.extend(actuals[i])) od;
    [v, sig] := eval(body);
    ENV := ENV1;
```

```
    case sig
    of N: NR:  return([v, N]),
       F: FR:  return([v, F])
    esac
  else
    # is_class(procdef) #
    var formals, decls, initexpr, ENV1, varinit1;
    var name, instance, d, v, sig, expr, i;

    {{ procdef.text ==
        CLASS <identifier> '(' formals:{<identifier> ','}* ')'
        BEGIN <fetch-associations> <store-associations>
        decls:  ( <variable-declaration> | <constant-declaration> |
                  <procedure-declaration> | <operator-declaration>
                )*
        initexpr:  [INIT ':' <block>]
        END <identifier> ';'
    }};
    ENV1 := ENV;
    ENV := ENVglobal.name_copy;
    ENV.new_inner_scope;
    varinit1 := varinit;
    varinit := '';
    if actuals.size ~= formals.size then ERROR fi;
    for i in actuals.index
    do ENV.bind( formals[i], STATE.extend(actuals[i])) od;

    for d in decls
    do [v, sig] := eval(d);
       if sig ~= N then ERROR fi
    od;
    for name in ENV.names
    do var b := ENV.binding(name);
       if is_proc(b) & b.env = undefined then b.env := ENV fi
    od;

    instance := a_composite_instance(procdef, ENV);
    ENV.bind('self', instance);
    [v, sig] := eval(varinit);
    varinit := varinit1;
    if sig ~= N
    then
      ENV := ENV1;
      case sig
      of F:   ERROR,
         FR:  return([v, F]),
         NR:  return([v, N])
      esac
    fi;
    if {{ initexpr == INIT ':' expr:<block> }}
    then
```

```
        ENV.new_proc_scope;
        [v, sig] := eval(expr);
      else
        [v, sig] := [a_undefined, N]
      fi;
      ENV := ENV1;
      case sig
      of F: FR:  return([v, F]),
         N: NR:  return([instance, N])
      esac
    fi
  );
```

The definition of *eval_standard_procedure* is:

```
  proc eval_standard_procedure(procname, actuals)
  ( case procname
    of 'string':
         if actuals.size ~= 1 | ~is_string(actuals[0]) then ERROR fi;
         return(a_string(actuals[0].stringval)),
       'integer':
         if actuals.size ~= 1 | ~is_integer(actuals[0]) then ERROR fi;
         return(a_integer(actuals[0].intval)),
       'undefined':
         if actuals.size ~= 0 then ERROR fi;
         return(a_undefined),
       'type':
         if actuals.size ~= 1 then ERROR fi;
         if is_string(actuals[0])
         then
           return(a_string('string'))
         elif is_integer(actuals[0])
         then
           return(a_string('integer'))
         elif is_undefined(actuals[0])
         then
           return(a_string('undefined'))
         else
           # is_composite_instance(actuals[0]) #
           return(a_string(actuals[0].class_decl.name))
         fi
       esac
  );
```

Notes

1) The function *dereference* is described in {9.3.1}.

2) The above definition realizes a call-by-sharing parameter passing mechanism (described previously).

3) The binding of *self* affects the environment component of the newly created composite-instance.

4) Regarding the treatment of signals other than *N*, <variable-initialization>s are treated differently when they occur in a <summer-program> {9.1.1}, in an <identifier-or-call> related to a class creation procedure (see above), or in a <block> {9.2.11}.

5) The treatment of <formals> of procedures and classes differs: the formals of a procedure exist only temporarily during the call of the procedure; the formals of a class are made part of the environment component of the new composite-instance which survives the call of the class creation procedure.

6) The definitions of *integer* and *string* given above are slightly simplified versions of the definitions given in Sections 10.2 and 10.4 respectively.

9.2.2.d. Examples

1) *x*

{Has as value either the value of the (local or global) constant or variable *x*, the value of a call to the procedure (either a class creation procedure or an ordinary procedure) with name *x*, or, if none of the previous cases applies, the value of the field selection *subject.x* {9.2.9, 9.2.14}.}

2) *fac*(5)

{Calls procedure *fac* with actual parameter 5.}

3) *integer*('123')

{Calls the class creation procedure *integer*, which returns an instance of class *integer* {10.2}.}

9.2.3. Return expressions

9.2.3.a. Syntax

<return-expression> ::= FRETURN | RETURN ['(' <expression> ')'] .

9.2.3.b. Pragmatics

Return expressions terminate the evaluation of the current procedure and give control back to the caller of the current procedure. An '*freturn*' causes the call of the current procedure to fail. A '*return*' not followed by an <expression>, transfers control to the caller without returning a value. A '*return*' followed by an <expression> first evaluates that <expression> and then transmits its value to the caller.

9.2.3.c. Semantics

The semantics of a *return* not followed by an <expression> is:

```
if {{ e == RETURN }}
then
  return([a_undefined, NR])
fi;
```

The semantics of a *return* followed by an <expression> is:

```
if {{ e == RETURN '(' expr:<expression> ')' }}
then
  var v, sig;

  [v, sig] := eval(expr);
  v := dereference(v);
  case sig
  of N: NR:  return([v, NR]),
     F: FR:  return([v, FR])
  esac
fi;
```

The semantics of an *freturn* is:

```
if {{ e == FRETURN }}
then
  return([a_undefined, FR])
fi;
```

Notes

1) The environment of the caller is, ultimately, re-established by *eval_call* {9.2.2}.

9.2.3.d. Examples

1) *return*

   {Returns from the current procedure without delivering a value.}

2) *return*(*x*)

   {Returns from the current procedure and delivers the value of *x*.}

3) *return*(*x* > 3)

   {The same as the previous example, but note that the expression '*x* > 3' can fail. In that case the call to the current procedure also fails.}

4) *freturn*

   {The call to the current procedure is completed and the call as a whole fails.}

9.2.4. If expressions

9.2.4.a. Syntax

```
<if-expression> ::=
      IF <test> THEN <block>
      ( ELIF <test> THEN <block> )*
      [ELSE <block>] FI .

<test> ::= <expression> [ FAILS | SUCCEEDS ] .
```

9.2.4.b. Pragmatics

An <if-expression> corresponds to the if-then-else statement found in most programming languages. First, the <test> part of the <if-expression> is evaluated. A <test> is either an <expression> or an <expression> followed by **'fails'** or **'succeeds'**. The suffix operator **'fails'** transforms failure into success and vice versa. The suffix operator **'succeeds'** has no affect whatsoever, but exists for reasons of symmetry.

If evaluation of the <test> part of the <if-expression> terminates successfully, the <block> following **'then'** is evaluated. Otherwise, the <test>s of successive **'elif'**s are evaluated until one of them terminates successfully (in which case the corresponding <block> is evaluated) or the list is exhausted. In the latter case, the <block> following **'else'** (if any) is evaluated.

9.2.4.c. Semantics

The semantics of a <test> is:

```
if {{ e == expr:<expression> sf: (SUCCEEDS | FAILS) }}
then
  if {{ sf == SUCCEEDS }}
  then
    return(eval(expr))
  else
    var v, sig;

    [v, sig] := eval(expr);
    case sig
    of F:       return([v, N]),
       N:       return([v, F]),
       FR: NR:return([v, sig])
    esac
  fi
fi;
```

The semantics of an <if-expression> is:

```
if {{ e == IF t:<test> THEN b:<block>
          elifpart: (ELIF <test> THEN <block>)*
          elsepart: [ELSE <block>] FI }}
then
  var v, sig;
  [v, sig] := eval(t);
  if sig = N
  then
    return(eval(b))
  elif sig = NR | sig = FR
  then
    return([v, sig])
  else
    var oneelif;
    for oneelif in elifpart
    do {{ oneelif == ELIF t:<test> THEN b:<block> }} ;
       [v, sig] := eval(t);
       if sig = N
       then
         return(eval(b))
       elif sig = NR | sig = FR
       then
         return([v, sig])
       fi
    od;
    if {{ elsepart == ELSE b:<block> }}
    then
      return(eval(b))
    else
      return([a_undefined, N])
    fi
  fi
fi;
```

Notes

1) The case that a <test> contains neither '**succeeds**' nor '**fails**' is already covered by the evaluation of <expression>s {9.2.16, 9.2.17}.

2) A <block> may be empty, see {9.2.11}.

9.2.4.d. Examples

1) `if x > 0 then put('x is positive') fi`

2)
```
if x > 0
then
  put('positive')
else
  put('negative or zero')
fi
```

3) **if** $x > 0$
then
put('positive')
elif $x = 0$
then
put('zero')
else
put('negative')
fi

4) $y :=$ **if** $x > 0$ **then** 1 **else** 0 **fi**

9.2.5. Case expressions

9.2.5.a. Syntax

<case-expression> ::=
CASE <expression> OF { <case-entry> ',' }* [DEFAULT ':' <block>] ESAC .

<case-entry> ::= { <constant-expression> ':' }+ <block> .

9.2.5.b. Pragmatics

A <case-expression> provides a multi-way branch. A <case-expression> of the form

case e_0 **of**
$k_{11} : \cdots : k_{1m_1} : e_1$,
$k_{21} : \cdots : k_{2m_2} : e_2$,
...
$k_{n1} : \cdots : k_{nm_n} : e_n$
default : e_{n+1}
esac

is equivalent to:

$\phi := e_0$;
if $\phi = k_{11} \mid \cdots \mid \phi = k_{1m_1}$ **then** e_1
elif $\phi = k_{21} \mid \cdots \mid \phi = k_{2m_2}$ **then** e_2
...
elif $\phi = k_{n1} \mid \cdots \mid \phi = k_{nm_n}$ **then** e_n
else e_{n+1}
fi

where ϕ is assumed to be a name, not occurring elsewhere in the program. All <case-entry>s must be constant expressions.

9.2.5.c. Semantics

```
if {{ e == CASE expr:  <expression> OF
       caselist:       {<case-entry> ','}*
       def:            [DEFAULT ':' <block>] ESAC }}
then
  var b, onecase, v1, v2, sig;

  [v1, sig] := eval(expr);
  if sig ~= N then return([v1, sig]) fi;
  v1 := dereference(v1);
  for onecase in caselist
  do var onekey, keylist;

     {{ onecase == keylist:{<constant-expression> ':'}+ b:<block> }};
     for onekey in keylist
     do require_constant_expression(onekey);
        [v2, sig] := eval(onekey);
        if sig ~= N then ERROR fi;
        if equal(v1, v2) then return(eval(b)) fi
     od
  od;
  if {{ def == DEFAULT ':' b:<block> }}
  then
    return(eval(b))
  else
    ERROR
  fi
fi;
```

Notes

1) The function *equal* is described in {9.3.2}.

2) The function *dereference* is described in {9.3.1}.

3) The function *require_constant_expression* is described in {9.2.18}.

4) It is not required that the <constant-expression>s in all <case-entry>s are distinct. However, no nondeterminism is introduced by the multiple occurrence of the same <case-entry>.

9.2.5.d. Examples

1)
```
case x of
'pi':    put('name of pi as string'),
'3.14':  put('value of pi as string'),
3.14:    put('value of pi as real')
default: put('x not recognized')
esac
```

{Note that each <case-entry> may be of a different type.}

2)
```
y := 7;
x := case y of
       1; 7: 13; spec_val
       default: general_val
       esac
```

{Note how a <case-expression> delivers the value of the selected case as value. Here, the value of *spec_val* is assigned to *x*.}

3)
```
case type(x) of
'string': put('x has string as value'),
'integer': put('x has integer as value'),
'array': put('x has array as value')
default: put('type of value of x not recognized')
esac
```

{Note how the combination of the procedure *type* {10.11.5} and <case-expression>s can be used for run-time type determination.}

9.2.6. While expressions

9.2.6.a. Syntax

<while-expression> ::= WHILE <test> DO <block> OD .

9.2.6.b. Pragmatics

Apart from the use of recursion to express iteration, <while-expression>s are, in principle, the only available iteration construct. (In the next paragraph we will see that a <for-expression> is only a syntactic shorthand for a special form of <while-expression>.) As long as the <test> part of a <while-expression> succeeds, its <block> will be evaluated.

9.2.6.c. Semantics

```
if {{ e == WHILE t:<test> DO b:<block> OD }}
then
  var v, sig;
  [v, sig] := eval(t);
  case sig
  of N:          # do nothing #,
     F:          return([v, N]),
     NR: FR:     return([v, sig])
  esac;
  [v, sig] := eval(b);
  if sig ~= N then return([v, sig]) fi;
  return(eval(e))
fi;
```

Notes

1) The <block> of a <while-expression> may not fail. If it does fail, then evaluation of the <while-expression> is terminated.

9.2.6.d. Examples

1) *n* := 0;
while *n* < 10 **do** *put*(*n*, '\n'); *n* := *n* + 1 **od**

{Prints the integers from 0 to 9 on consecutive lines.}

2) **while** *line* := *input.get* **do** *process*(*line*) **od**

{Processes all lines of the file to which the variable *input* refers. The operation *input.get* {10.9} fails on end-of-file.}

9.2.7. For expressions

9.2.7.a. Syntax

<for-expression> ::= FOR <identifier> IN <expression> DO <block> OD .

9.2.7.b. Pragmatics

A <for-expression> provides a shorthand for a particular kind of <while-expression> and serves the purpose of iterating over class instances on which the operation *next* is defined. By convention[2], this operation has one parameter *state*, which keeps accounts of the progress of the iteration. At the start of each <for-expression>, *state* is initialized to *'undefined'* At each step of the iteration *next* is called and delivers either an array of two elements ([*v*, *s*], where *v* is the value to be used in the current step and *s* is the state to be used in the next step), or fails, to indicate that the <for-expression> should terminate. Given this behavior of *next* one can rewrite a <for-expression> of the form

for *x* **in** *y* **do** *z* **od**

as

gen := *y*;
state := *undefined*;
while [*x*,*state*] := *gen.next*(*state*) **do** *z* **od**;

where *gen* and *state* are assumed to be names not occurring elsewhere in the program.

9.2.7.c. Semantics

if {{*e* == FOR *id*:<identifier> IN *expr*:<expression> DO *b*:<block> OD }}
then
var *gen*, *v*, *sig*, *state*;
if ~ *is_loc*(*ENV.binding*(*id*)) **then** *ERROR* **fi**;
[*gen*, *sig*] := *eval*(*expr*);
if *sig* ~= *N* **then** *return*([*gen*, *sig*]) **fi**;

2) All *next* operations defined on built-in classes have this form, and a run-time error will result if a user-defined *next* operation does not conform to it.

```
    gen := dereference(gen);
    state := a_undefined;
    while [state, sig] := eval_field_selection('fetch', gen, 'next', [state]) & sig = N
    do
      [v, sig] := eval_field_selection('fetch', state, 'retrieve', [a_integer(0)]);
      if sig ~= N then ERROR fi;
      STATE.modify(ENV.binding(id), v);
      [state, sig] := eval_field_selection('fetch', state, 'retrieve', [a_integer(1)]);
      if sig ~= N then ERROR fi;
      [v, sig] := eval(b);
      if sig ~= N then return([v, sig]) fi;
    od;
    return([a_undefined, N]);
  fi;
```

Notes

1) This semantic definition is equivalent to the textual rewriting of <for-expression>s, as shown in the previous section, but it avoids explicit rewriting because this is alien to the style of definition used here.

9.2.7.d. Examples

1)
```
class interval(from, to, by)
begin fetch next;

      proc next(state)
      ( var cur;
        if state = undefined
        then
          cur := from
        else
          cur := state + by
        fi;
        if (from <= to & cur <= to) | (from > to & cur >= to)
        then
          return([cur, cur])
        else
          freturn
        fi
      );
  init: if (from <= to & by < 0) | (from > to & by > 0) | by = 0
        then
          freturn
        fi
end interval;
```

{This is an example of a user-defined *next* operation. The argument *state* is used to administer the progress of the iteration. Initially, *state* has the value *undefined*. An example of the use of this class is given below. See also 10.6.}

2) **for** *n* **in** *interval*(0, 10, 1) **do** *put*(*n*, '\n') **od**

{Prints the values 0 to 10 on consecutive lines.}

3) **for** *c* **in** *'abcd'* **do** *put*(*c*, '\n') **od**

{Prints the characters '*a*', '*b*', '*c*' and '*d*' on consecutive lines. See also 10.4.}

9.2.8. Try expressions

9.2.8.a. Syntax

<try-expression> ::= TRY { <expression> ',' }+ [UNTIL <block>] YRT .

9.2.8.b. Pragmatics

A <try-expression> provides a facility for eliminating the side-effects of the evaluation of a failing expression. A *partial_state_copy* {8.3.3} is made before each <expression> is evaluated. If the evaluation of the <expression> is successful, the <block>, if any, is evaluated. If that evaluation is also successful, the <try-expression> as a whole succeeds and all side-effects are made 'permanent' (see below). If the evaluation of either the <expression> or the <block> fails, then the original environment is restored (except for the local variables of the current procedure, which are not restored to their previous values, (see below)) and the next <expression> is evaluated. This process is repeated until success is achieved or the list of <expression>s is exhausted. In the latter case, the original environment is restored (again the values of local variables are excluded) and the <try-expression> fails.

The values of local variables of the procedure in which the <try-expression> occurs are **never** restored and can hence be used to pass information from an attempt that has failed to the next one. Since <try-expression>s can be nested dynamically, the side-effects of an innermost <try-expression> may be undone by an enclosing <try-expression>. In this sense, only side-effects of an outermost <try-expression> are 'permanent'.

9.2.8.c. Semantics

```
if {{ e == TRY exprlist:{<expression> ','}+ u:[UNTIL <block>] YRT }}
then
  var b, ENV1, expr, v, sig;
  ENV1 := ENV;
  for expr in exprlist
  do ENV := ENV.partial_state_copy;
     [v, sig] := eval(expr);
     case sig
     of F:  ENV := ENV1,
        FR: ENV := ENV1; return([v, FR]),
        NR: return([v, NR]),
        N:  if {{ u == UNTIL b:<block> }}
            then
              [v, sig] := eval(b);
              case sig
              of F:   ENV := ENV1,
                 FR:  ENV := ENV1; return([v, FR]),
```

```
                NR:  return([v, NR]),
                N:   return([v, N])
              esac
            else
              return([v, N])
            fi
        esac
    od;
    return([a_undefined, F])
  fi;
```

Notes

1) In the above definition, it is essential that objects can be **shared**. Consider, for instance, the first assignment *ENV*1 := *ENV*. Here, a reference to the value of *ENV* (an environment) is assigned to *ENV*1, instead of **copying** the whole environment and assigning that value to *ENV*1. This has as consequence that, for instance, all locations which occur in the environment are shared by *ENV*1 and *ENV*. This is precisely what is needed for communicating modifications of values in the environment to the evaluations of enclosing constructs after successful completion of the <try-expression>. The complementary situation (all locations are copied except the locations of the local variables of the current procedure) is realized by the *partial_state_copy* operation.

2) The operation *partial_state_copy* has been defined in such a way that the locations associated with local variables of the current procedure are shared by the original and copied environments. This achieves the effect that the values of these local variables are never restored.

3) Evaluation of a <return-expression> {9.2.3} contained in a <try-expression> leads to premature completion of the evaluation of that <try-expression>. For a normal return ('*return*') the environment is not restored; for a failure return (either an '*freturn*' or a '*return*' of a failing expression) the environment is restored.

4) Side-effects on files {10.9} are also recovered.

9.2.8.d. Examples

1)
```
x := g := 0;
try x := g := −1, x := g := 1 until positive(x) yrt
```

{Assume that x is a local variable and g is a global variable. After the evaluation of $x := g := -1$, the test fails and the previous value of g (i.e. 0) is restored. x, being a local variable, keeps its current value (i.e. -1). Next, $x := g := 1$ is evaluated and the test succeeds. The evaluation of the <try-expression> is complete and g keeps its value (i.e. 1).}

2)
```
if try index := input_contains(search_key) yrt
then
  process(index)
else
  put('key not found')
fi
```

{Suppose that *input_contains*(*search_key*) is a user-defined procedure that reads some input from a file and succeeds or fails depending on whether that input contains the *search_key* as literal substring. If this procedure succeeds then the input is irrevocably consumed. However, if it fails then all read operations are reversed and all other side-effects are undone. The net effect of expressions of the form '**if try** . . . **yrt then** . . . ' is to **probe** without disturbing the program state (apart from local modifications) if the probe fails.}

9.2.9. Scan expressions

9.2.9.a. Syntax

<scan-expression> ::= SCAN <expression> FOR <block> ROF .

9.2.9.b. Pragmatics

A <scan-expression> serves the purpose of introducing a new 'focus of activity' which need not be mentioned explicitly during the evaluation of the associated <block>. This is particularly useful in (but not restricted to) string scanning to avoid having to mention the subject string for each scanning operation. First, the <expression> is evaluated. If the result is of type *string*, it is converted to type *scan_string* {10.8}. The value obtained in this way is bound to the identifier *subject* in the current environment, after saving the previous binding of *subject*. Next the <block> is evaluated, and finally the previous value of *subject* is restored.

Note that the rules for identifier identification {9.2.2} are in accordance with the semantics of <scan-expression>s: identifiers which are not bound to some *DENOTABLE-VALUE* {8.3.3} are interpreted as fields to be selected from the current value of *subject*.

9.2.9.c. Semantics

```
if {{ e == SCAN expr:<expression> FOR b:<block> ROF }}
then
  var ENV1, v, sig;
  [v, sig] := eval(expr);
  if sig ~= N then return([v, sig]) fi;
  v := dereference(v);
  if is_string(v)
  then
    [v, sig] := eval_call('scan_string', [v])
  fi;
  ENV1 := ENV;
  ENV := ENV.name_copy;
  ENV.new_inner_scope;
  ENV.bind('subject', v);
  [v, sig] := eval(b);
  ENV := ENV1;
  return([v, sig])
fi;
```

9.2.9.d. Examples

1)
```
scan 'aardvark,able,baker,clerk,'
for
   while x := break(',')
   do put(x, '\n'); move(1) od
rof
```

{*break* {10.8} assigns the string values *'aardvark'* through *'clerk'* to the variable *x*. These values of *x* are printed on consecutive lines.}

2)
```
name := scan l
        for
              any(letter) || (span(letgit) | empty)
        rof
```

{Assumes that the value of *l* starts with an identifier-like string and assigns that string to *name*.}

9.2.10. Assert expressions

9.2.10.a. Syntax

<assert-expression> ::= ASSERT <expression> .

9.2.10.b. Pragmatics

An <assert-expression> provides a means of specifying an assertion, which should hold at the point where the assertion occurs. The <expression> part of the <assert-expression> is evaluated. It is an error if that <expression> cannot be evaluated successfully.

9.2.10.c. Semantics

```
if {{ e == ASSERT expr:<expression> }}
then
  var v, sig;

  [v, sig] := eval(expr);
  if sig ~= N then ERROR fi;
  return([v, sig])
fi;
```

9.2.10.d. Examples

1)
```
assert pi > 0
```

{The execution of the program is terminated if the condition *pi* > 0 fails.}

9.2.11. Parenthesized expressions and blocks

9.2.11.a. Syntax

<parenthesized-expression> ::= '(' <block> ')' .

<block> ::=
(<variable-declaration>|<constant-declaration>)* {[<expression>] ';'}* .

9.2.11.b. Pragmatics

<parenthesized-expression>s can be used to indicate explicitly the desired grouping in an expression or to introduce a new <block> during the evaluation of an <expression>. A <block> introduces a new scope for the variables and constants declared in it.

9.2.11.c. Semantics

The semantics of a <parenthesized-expression> is:

```
if {{ e == '(' b:<block> ')' }}
then
  return(eval(b))
fi;
```

The semantics of a <block> is:

```
if {{ e == decllist: (<variable-declaration>|<constant-declaration>)*
          exprlist: {[<expression>] ';'}* }}
then
  var decl, expr, ENV1, v, varinit1, sig;

  ENV1 := ENV;
  ENV  := ENV.name_copy;
  ENV.new_inner_scope;
  varinit1 := varinit;
  varinit := '' ;
  for decl in decllist do [v, sig] := eval(decl) od;
  [v, sig] := eval(varinit);
  varinit := varinit1;
  if sig ~= N
  then
    ENV := ENV1;
    if sig = F then ERROR else return([v, sig]) fi
  fi;
  if exprlist.size = 0
  then
    ENV := ENV1;
    return([u_undefined, N])
  else
    var i;

    for i in exprlist.index
    do [v, sig] := eval(exprlist[i]);
       case sig
```

```
            of N:       # do nothing #,
               F:       if i ~= exprlist.size - 1 then ERROR fi,
               NR: FR: ENV := ENV1; return([v, sig])
            esac
         od;
         ENV := ENV1;
         return([v, sig])
      fi
   fi;
```

Notes

1) The evaluation of a <variable-declaration> or <constant-declaration> can never fail, i.e. it always returns the signal value *N*.

2) Regarding the treatment of signals other than *N*, <variable-initialization>s are treated differently when they occur in a <summer-program> {9.1.1}, in an <identifier-or-call> related to a class creation procedure {9.2.2}, or in a <block> (see above).

9.2.11.d. Examples

1) (**var** *t*; *t* := *a*; *a* := *b*; *b* := *t*)

   {A new local variable *t* is declared. The above expression could also be used as a subexpression. This is the case in

   x := (**var** *t*; *t* := *a*; *a* := *b*; *b* := *t*)

   which assigns the result of *b* := *t* —in this case the old value of *a*— to *x*.}

2) **if** *a* > 0 **then var** *x* := *a* * *a*; *return*((*x*−1) * (*x*+1)) **fi**

   {Note that *x* is declared as a local variable in the **then**-part of the <if-expression>.}

9.2.12. Array expressions

9.2.12.a. Syntax

<array-expression> ::=
    ARRAY <size-and-default> [INIT <array-initialization>] |
    <array-initialization> .

<size-and-default> ::= '(' <expression> ',' <expression> ')' .

<array-initialization> ::= '[' { <expression> ',' }* ']' .

9.2.12.b. Pragmatics

Class *array* provides a storage facility that associates integer indices with arbitrary values. The definition of this class is given in the next chapter {10.5}. Here, only <array-expression>s are defined. These introduce an abbreviated notation for the creation of arrays (that is, instances of class *array*). The various forms of <array-expression>s can be rewritten as follows. An <array-expression> of the (first) form

array(*e*1, *e*2) **init** [*i*0, . . . , *in*−1]

is equivalent to

ϕ := *array*(e_1, e_2);
ϕ [0] := i_0;
...
ϕ [$n-1$] := i_{n-1};

where ϕ is assumed to be a name, not occurring elsewhere in the program. An <array-expression> of the (second) form

[i_0, ..., i_{n-1}]

is equivalent to

array(n, *undefined*) **init** [i_0, ..., i_{n-1}]

which can be rewritten according to the previous rule.

The first <expression> in the <size-and-default> defines the length of the array. The second <expression> defines the default initialization value for the array elements.

9.2.12.c. Semantics

The semantics of an <array-expression> of the first form is:

```
if {{ e == ARRAY '(' expr1:<expression> ',' expr2:<expression> ')'
        initexpr: [INIT <array-initialization> ] }}
then
  var def, sz, sig;

  [sz, sig] := eval(expr1);
  if sig ~== N then return([sz, sig]) fi;
  sz := dereference(sz);
  [def, sig] := eval(expr2);
  if sig ~= N then return([def, sig]) fi;
  def := dereference(def);
  return(eval_array_init(sz, def, initexpr));
fi;
```

The semantics of an <array-expression> of the second form is:

```
if {{ e == '[' exprlist:{<expression> ','}* ']' }}
then
  return(eval_array_init(a_integer(exprlist.size), a_undefined, 'init' || e));
fi;
```

The actual creation and initialization of instances of the class *array* is performed by *eval_array_init*, defined below:

```
proc eval_array_init(sz, def, initexpr)
( var ar, exprlist, i, v, v1, sig;

  [ar, sig] := eval_call('array', [sz, def]);
  if sig ~== N then return([ar, sig]) fi;
  {{ initexpr == INIT '[' exprlist:{<expression> ',' }* ']' }};
  for i in exprlist.index
  do [v,sig] := eval(exprlist[i]);
```

```
        if sig ~== N then return([v, sig]) fi;
        v := dereference(v);
        [v1, sig] := eval_field_selection('fetch', ar, 'update', [a_integer(i), v]);
        if sig ~== N then return([v1, sig]) fi;
      od;
      return([ar, N])
    );
```

Notes

1) The size of an array should be greater than or equal to zero.

2) An array denotation, i.e. an expression of the form of an <array-initialization>, may also appear on the left hand side of an assignment operator and then denotes a multiple assignment {9.2.17}.

3) Array-elements are initialized by means of *update* operations {9.2.14, 9.2.17 and 10.5}.

4) The formal definition relies on the existence of a <class-declaration> for *array* outside the kernel {10.5}. See Section 8.4 for a more extensive explanation of this topic.

5) The above definition is intentionally general. Before the initialization loop is performed, the number of initial values could be compared with the size of the new array to ensure that its bounds would not be exceeded during initialization. The call of *eval_field_selection* would never fail in those circumstances. We have opted here for a more general formulation anticipating additions to the language that allow initializations for other classes than *array* and *table*.

9.2.12.d. Examples

1) *array*(5, 0)

   {Creates an instance of the class *array*, with size 5 and all elements initialized to 0.}

2) ```
   n := 5;
   i := 30;
   yellow := 'YELLOW ';
   array(3*n, 0) init [10, 20, i+5, yellow || 'bird']
   ```

   {Creates an instance of class *array* of size 15. The elements with indices 0 to 3 are initialized to 10, 20, 35 and *'YELLOW bird'* respectively. All other elements are initialized to 0.}

3) [3.5, *'sunshine'*, *array*(*n*, *undefined*)]

   {Creates an array of size 3, with elements initialized to the values 3.5, *'sunshine'* and *array*(*n*, *undefined*). In this way arbitrarily shaped multi-dimensional arrays may be created.}
   ```

9.2.13. Table expressions

9.2.13.a. Syntax

<table-expression> ::=
TABLE <size-and-default> [INIT <table-initialization>] |
<table-initialization> .

<size-and-default> ::= '(' <expression> ',' <expression> ')' .

<table-initialization> ::= '[' { <table-element> ',' }* ']' .

<table-element> ::= {<expression> ':'}+ ':' <expression> .

9.2.13.b. Pragmatics

Class *table* provides an associative memory that maps keys of arbitrary type onto values. The definition of this class is given in the next chapter (10.7). Here, only <table-expression>s are defined, which introduce an abbreviated notation for the creation of tables (that is, instances of class *table*). The various forms of <table-expression>s can be rewritten as follows. A <table-expression> of the (first) form

$table(e_1, e_2)$ **init**
$[\ k_{00} : \cdots : k_{0m_0} : i_0\ ,$
$\cdots$
$k_{n0} : \cdots : k_{nm_n} : i_n\ ]$

is equivalent to

$\phi := table(e_1, e_2);$
$\phi\ [k_{00}] := \cdots := \phi\ [k_{0m_0}] := i_0;$
$\cdots$
$\phi\ [k_{n0}] := \cdots := \phi\ [k_{nm_n}] := i_n;$

where ϕ is assumed to be a name not occurring elsewhere in the program. A <table-expression> of the (second) form

$[\ k_{00} : \cdots : k_{0m_0} : i_0\ ,$
$\cdots$
$k_{n0} : \cdots : k_{nm_n} : i_n\ ]$

is equivalent to

$\phi := table(s, undefined);$
$\phi\ [k_{00}] := \cdots := \phi\ [k_{0m_0}] := i_0;$
$\cdots$
$\phi\ [k_{n0}] := \cdots := \phi\ [k_{nm_n}] := i_n$

where s is equal to $\sum_{i=0}^{n} m_i$.

The first <expression> in the <size-and-default> defines the estimated size of the table; the total number of entries in the table may become larger than this estimate. The second <expression> defines the default value to be returned when a value is looked up which has not been previously entered in the table.

9.2.13.c. Semantics

The semantics of a <table-expression> of the first form is:

```
if {{ e == TABLE '(' expr1:<expression> ',' expr2:<expression> ')'
        initexpr: [INIT <table-initialization>] }}
then
  var def, sig, sz;

  [sz, sig] := eval(expr1);
  if sig ~= N then return([sz, sig]) fi;
  sz := dereference(sz);
  [def, sig] := eval(expr2);
  if sig ~= N then return([def, sig]) fi;
  def := dereference(def);
  return(eval_table_init(sz, def, initexpr));
fi;
```

The semantics of a <table-expression> of the second form is:

```
if {{ e == '[' exprlist:{<table-element> ','}* ']' }}
then
  return(eval_table_init(a_integer(exprlist.size), a_undefined, 'init' || e))
fi;
```

The actual creation and initialization of tables is performed by *eval_table_init*:

```
proc eval_table_init(sz, def, initexpr)
( var tb, sig, v, expr, exprlist;

  [tb, sig] := eval_call('table', [sz, def]);
  if sig ~= N then return([tb, sig]) fi;
  {{ initexpr == INIT '[' exprlist:{<table-element> ',' }* ']' }};
  for expr in exprlist
  do var ar, k, key, keylist, expr1, v1;

     {{ expr == keylist:{<expression> ':'}+ ':' expr1:<expression> }};
     ar := array(keylist.size, undefined);
     for k in keylist.index
     do [v, sig] := eval(keylist[k]);
        if sig ~= N then return([v, sig]) fi;
        ar[k] := dereference(v);
     od;
     [v1, sig] := eval(expr1);
     if sig ~= N then return([v1, sig]) fi;
     v1 := dereference(v1);
     for key in ar
     do [v, sig] := eval_field_selection('fetch', tb, 'update', [key, v1]);
        if sig ~= N then return([v, sig]) fi;
     od
  od;
  return([tb, N])
);
```

Notes

1) Table-elements are initialized by means of *update* operations {9.2.14, 9.2.17 and 10.7}.

2) The formal definition relies on the existence of a <class-declaration> for *table* outside the kernel {10.7}. See Section 8.4 for a more extensive explanation of this topic.

9.2.13.d. Examples

1) *table*(300, *undefined*)

{Creates a new instance of class *table*, with estimated size 300 and default value *undefined*.}

2) *table*(*i* + 50, *'symbol−not−in−table'*)

{Creates a new instance of class *table*, with as estimated size the value of the expression *i* + 50 and as default value the string *'symbol−not−in−table'*.}

3) *t* := *table*(100, 0) **init** [*'a'*:*'b'*: 3, *'p'*:5]

{Assigns to *t* a new instance of class *table* with estimated size 100 and default value 0. It performs the initializations

t['a'] := *t*['b'] := 3;
t['p'] := 5.

The values of *t*['a'], *t*['p'] and *t*['z'] are now 3, 5, and 0 respectively.}

9.2.14. Field selection

9.2.14.a. Syntax

<primary> ::= <unit> (<subscript> | <select>)* .

<select> ::= '.' <identifier> [<actuals>] .

9.2.14.b. Pragmatics

Accessing a field of a class instance is referred to as **field selection** {9.1.5}. We will only describe the basic mechanism of field selection here. A complete description is postponed until {9.2.17}.

The following items are required for a field selection:

1. The *accesstype* to distinguish an assignment ('store') to the field from any other operation ('fetch').
2. The *instance* on which the field selection is to be performed.
3. The *field* to be selected.
4. Optional actual parameters.

To carry out a field selection, the environment is switched to the environment contained in the left operand of the selection (i.e. the instance obtained by evaluating the <unit>). Depending on the access type, either the <fetch-associations> or the <store-associations> of the corresponding <class-declaration> are searched for a <field-identifier> equal to *field*. Depending on the form of the association in which

field occurs, the value of the field is either simply modified and returned as value, or the procedure associated with the field is called. In the latter case the value returned by the procedure call is the result of the field selection. Finally, the original environment is restored.

9.2.14.c. Semantics

The semantics of a <unit> followed by a <select> is:

```
if {{ e == expr1:<unit> '.' field:<identifier> act: [<actuals>] }}
then
  var actual_vals, v1, sig;

  [v1, sig] := eval(expr1);
  if sig ~= N then return([v1, sig]) fi;
  v1 := dereference(v1);
  if {{ actuals == '(' exprlist: {<expression> ','}* ')' }}
  then
    var i, exprlist, v;

    actual_vals := array(exprlist.size, undefined);
    for i in exprlist.index
    do [v, sig] := eval(exprlist[i]);
       if sig ~= N then return([v, sig]) fi;
       actual_vals[i] := dereference(v)
    od
  else
    actual_vals := array(0, undefined)
  fi;
  return(eval_field_selection('fetch', v1, field, actual_vals));
fi;
```

The actual field selection is performed by *eval_field_selection* described below.

```
proc eval_field_selection(access_type, instance, field, actuals)
( var flist, slist, list, v, sig, f, ENV1, p, a;

  if is_basic_instance(instance)
  then
    return(instance.operation( field, actuals))
  fi;
  if ~ is_composite_instance(instance) then ERROR fi;
  {{ instance.class_decl.text ==
        CLASS <identifier> <formals>
        BEGIN FETCH flist:{<association> ','}+ ';'
              STORE slist:{<association> ','}+ ';'
              ( <variable-declaration> | <constant-declaration> |
                <procedure-declaration> | <operator-declaration>
              )*
              [INIT ';' <block>]
        END <identifier> ';'
  }};
  ENV1 := ENV;
  ENV := instance.env;
```

```
    list := if access_type = 'fetch' then flist else slist fi;
    for a in list
    do if {{ a == f:<field-identifier> }} & string_equal( field, f )
       then
         if is_loc(ENV.binding( f ))
         then
           if access_type = 'fetch'
           then
             if actuals.size ~= 0 then ERROR fi;
             v := dereference(ENV.binding( f ));
             ENV := ENV1;
             return([v, N])
           else
             if actuals.size ~= 1 then ERROR fi;
             v := actuals[0];
             STATE.modify(ENV.binding( f ), v);
             ENV := ENV1;
             return([v, N])
           fi
         else
           if access_type = 'fetch'
           then
             [v, sig] := eval_call( f, actuals);
             ENV := ENV1;
             return([v, sig])
           else
             ERROR
           fi
         fi;
       elif {{ a == f:<field-identifier> ':' p:<identifier> }} &
            string_equal( field, f )
       then
           [v, sig] := eval_call(p, actuals);
           ENV := ENV1;
           return([v, sig])
       fi
    od;
    ERROR
);
```

The utility procedure *has_field* succeeds or fails depending on whether a given field selection can or cannot be performed. It is used for operator identification {9.2.16 and 9.2.17}.

```
proc has_field(access_type, instance, field)
( var flist, slist, list, f, p, a;

  if is_basic_instance(instance)
  then
    return(instance.has( field))
  fi;
```

```
if ~ is_composite_instance(instance) then freturn fi;
{{ instance.class_decl.text ==
      CLASS <identifier> <formals>
      BEGIN FETCH flist:{<association> ','}+ ';'
            STORE slist:{<association> ','}+ ';'
            ( <variable-declaration> | <constant-declaration> |
              <procedure-declaration> | <operator-declaration>
            )*
            [INIT ':' <block>]
      END <identifier> ';'
}};
list := if access_type = 'fetch' then flist else slist fi;
for a in list
do if {{ a == f:<field-identifier> [':' <identifier> ] }} &
     string_equal( field, f )
   then
     return
   fi
od;
freturn
);
```

Notes

1) In the formal definition it is assumed that all <expression>s are fully parenthesized. This implies that repeated field selections of the form

$$e \cdot f_1 \cdot f_2 \cdot f_3 \cdots$$

have been replaced by

$$(\cdots((e \cdot f_1) \cdot f_2) \cdot f_3 \cdots)$$

Hence, the formal definition only has to cover the case of a single field selection.

2) The *class_decl.text* part of each instance under consideration is required to have an empty <subclass-declaration>, since this has been removed by *expand_super_class* {9.1.5}.

9.2.14.d. Examples

1)
```
n := 5;
s := stack(7 * n)
```

{Assigns a new instance of class *stack* with actual parameter 35 to variable *s*. A declaration for class *stack* is given as an example in {9.1.5}.}

```
s.push(4)
```

{Next, the integer 4 is pushed onto that stack by performing the field selection *push* with actual parameter 4.}

2) *x* := *s.pop*

{Similar to the above, but now an element is popped from the stack by performing the field selection *pop*.}

9.2.15. Subscription

9.2.15.a. Syntax

<primary> ::= <unit> (<subscript> | <select>)* .

<subscript> ::= '[' <expression> ']' .

9.2.15.b. Pragmatics

Subscription is an abbreviated notation for selection of the fields *retrieve* and *update* respectively. These fields are, amongst others, defined for the classes *array* {10.5}, *bits* {10.10}, *string* (*retrieve* only) {10.4} and *table* {10.7}, but each class which defines these operations can use the subscript notation. The operation *update* is used when subscription occurs on the left hand side of an assignment operator and is described in {9.2.17}; The operation *retrieve* is used in all other cases and is described here.

Since subscription is reduced to field selection, its semantics is defined by defining the reduction from

e1[*e2*]

to

e1.*retrieve*(*e2*) .

9.2.15.c. Semantics

if {{ *e* == *expr1*:<unit> '[' *expr2*:<expression> ']' }}
then
  var *v1*, *v2*, *sig*;

  [*v1*, *sig*] := *eval*(*expr1*);
  if *sig* ~= *N* **then** *return*([*v1*, *sig*]) **fi**;
  v1 := *dereference*(*v1*);
  [*v2*, *sig*] := *eval*(*expr2*);
  if *sig* ~= *N* **then** *return*([*v2*, *sig*]) **fi**;
  v2 := *dereference*(*v2*);
  return(*eval_field_selection*('fetch', *v1*, 'retrieve', [*v2*]))
fi;

Notes

1) In the formal definition it is assumed that all <expression>s are fully parenthesized. This implies that repeated subscriptions of the form

$$e[e_1]\,[e_2]\,[e_3]\,\cdots$$

have been replaced by

$$(\,\cdots\,((e[e_1])[e_2])[e_3]\,\cdots\,)$$

9.2.15.d. Examples

1) *a* := *array*(10, 7);

{Assigns to *a* an instance of class *array*, containing 10 elements which are all initialized to 7.}

x := *a*[5]

{Evaluates, in the above context, the expression *x* := *a.retrieve*(5), which assigns the integer 7 to variable *x*.}

9.2.16. Monadic expressions

9.2.16.a. Syntax

<monadic-expression> ::= (<monadic-operator>)* <primary> .

<monadic-operator> ::= '~' | '−' | <operator-symbol> .

9.2.16.b. Pragmatics

All monadic operators have the same priority, which is higher than the priority of dyadic operators. In the formal definition it is assumed that all <monadic-expression>s are fully parenthesized hence only <monadic-expression>s containing just one <monadic-operator> will be described.

The evaluation of a <monadic-expression> proceeds in the following steps:

1. Evaluate the primary. Unless the <monadic-operator> is '~', the evaluation of the <monadic-expression> fails if the evaluation of the primary fails.
2. If the <monadic-operator> is **not** ('~'), turn success into failure and vice versa.
3. Otherwise, if a field with the name of the <monadic-operator> is defined on the value obtained in step 1, then perform a field selection of that field.
4. Otherwise, there must be a global <operator-declaration> for <monadic-operator>, i.e. an <operator-declaration> immediately contained in the <summer-program>. In this case perform a call to the procedure declared by that <operator-declaration>.

9.2.16.c. Semantics

```
if {{ e == mop:<monadic-operator> expr:<primary> }}
then
  var v, sig;

  [v, sig] := eval(expr);
  if string_equal(mop, '~')
  then
    case sig
    of N:       return([a_undefined, F]),
       F:       return([a_undefined, N]),
       FR: NR: return([v, sig])
    esac
  elif sig ~= N
  then
```

```
    return([v, sig])
  fi;
  v := dereference(v);
  if has_field('fetch', v, mop)
  then
    return(eval_field_selection('fetch', v, mop, []))
  elif ENV.has_binding(mop) & is_proc(ENV.binding(mop))
  then
    return(eval_call(mop, [v]))
  else
    ERROR
  fi
fi;
```

9.2.16.d. Examples

1) $-x$

2) $\sim positive(x)$

3) $-$ **if** $a > b$ **then** $x + 1$ **else** $x - 1$ **fi**

9.2.17. Dyadic expressions

9.2.17.a. Syntax

<dyadic-expression> ::=
    <monadic-expression> (<dyadic-operator> <monadic-expression>)* .

<dyadic-operator> ::=
    '+' | '-' | '*' | '/' | '%' | '||' | '<' | '<=' | '>' | '>=' |
    '=' | '~=' | ':=' | '&' | '|' | <operator-symbol> .

9.2.17.b. Pragmatics

The priorities of <monadic-operator>s and <dyadic-operator>s, are in decreasing order:

All monadic operators	1
||	2
* / %	3
+ −	4
< <= > >= = ~=	5
User defined operators	6
:=	7
&	8
|	9

In the formal definition it is assumed that all <dyadic-expression>s are fully parenthesized in order to establish the relative priorities of the <dyadic-operator>s. In the sequel, only <dyadic-expression>s containing just one <dyadic-operator> will be described.

The evaluation of a <dyadic-expression> proceeds in the following steps:

1. If the <dyadic-operator> is not equal to ':=' (assignment), goto step 3, otherwise evaluate the right-hand operand. If it evaluates to a location, use the contents of that location as value.

2. The <dyadic-operator> is ':='. We distinguish four cases depending on the form of the left-hand operand:

   2a. **Field selection.** The left operand consists of a <unit> followed by a <select>. Evaluate the <unit> and perform a 'storing' field selection {9.2.14} on the value obtained in this way, using the given field name and with the value obtained in step 1 as actual parameter. The value of this field selection is the result of the evaluation of the <dyadic-expression>.

   2b. **Subscription.** The left operand consists of a <unit> followed by a <subscript>. Evaluate the <unit> and <subscript>. Perform a field selection {9.2.14} on the value of the <unit>, using the field name *update*, the value of the <subscript>, and the value obtained in step 1 as actual parameters. The value of this field selection is the result of the evaluation of the <dyadic-expression>.

   2c. **Multiple assignment.** The left operand consists of an <array-initialization>. The expressions e_i in the <array-initialization> are evaluated from left to right. The value of each e_i should be a location. For each e_i the result of evaluating the expression $V[i]$ is placed in the cell corresponding to location e_i, where V is the value obtained in step 1. The value of the last expression in the <array-initialization> is the result of the evaluation of the <dyadic-expression>.

   2d. **Simple assignment.** Neither of the above cases applies. The left operand of the assignment is evaluated. The resulting value should be a location. The value obtained in step 1 is placed in the cell associated with that location and that value is also the result of the evaluation of the <dyadic-expression>.

3. The <dyadic-operator> is not ':='. Evaluate the left operand. If it fails and the <dyadic-operator> is not '|' (Boolean *or*), the evaluation of the <dyadic-expression> as a whole fails. Evaluate the right operand: If it fails, the evaluation of the <dyadic-expression> fails. If the <dyadic-operator> is '|' (Boolean *or*) or '&' (Boolean *and*), the value of the right operand becomes the value of the whole <dyadic-expression> whose evaluation is then complete. Otherwise, the following cases exist:

   3a. The value of the left operand is an instance of some class, say C, and a field with the same name as the <dyadic-operator> is defined in the <class-declaration> for C. Perform a selection of that field; the result of this field selection then becomes the result of the evaluation of the <dyadic-expression>.

   3b. There exists an <operator-declaration> for the <dyadic-operator>. Perform a procedure call to the procedure declared in that <operator-declaration>; the result of this procedure call then becomes the result of the evaluation of the <dyadic-expression>.

3c. The <dyadic-operator> is '=' or '~=='. Perform an (in)equality test on the values of the left and right operands. Depending on the outcome of this test, evaluation of the <dyadic-expression> either fails or delivers the value of the right operand.

9.2.17.c. Semantics

```
if {{ e == expr1: <monadic-expression>
          dop:    <dyadic-operator> expr2: <monadic-expression> }}
then
  var left, right, sig;

  if {{ dop == '~=' }}
  then
    var exprlist, field, sub, u, vu, vs;

    [right, sig] := eval(expr2);
    if sig ~= N then return([right, sig]) fi;
    right := dereference(right);
    if {{ expr1 == u:<unit> '.' field:<identifier> }}
    then    # Case 2a. ( field selection) #
      [vu, sig] := eval(u);
      if sig ~= N then return([vu, sig]) fi;
      vu := dereference(vu);
      return(eval_field_selection('store', vu, field, [right]));
    elif {{ expr1 == u:<unit> '[' sub:<expression> ']' }}
    then    # Case 2b. (subscription) #
      [vu, sig] := eval(u);
      if sig ~= N then return([vu, sig]) fi;
      vu := dereference(vu);
      [vs, sig] := eval(sub);
      if sig ~= N then return([vs, sig]) fi;
      vs := dereference(vs);
      return(eval_field_selection('fetch', vu, 'update', [vs, right]))
    elif {{ expr1 == '[' exprlist:{<expression> ','}+ ']' }}
    then    # Case 2c. (multiple assignment) #
      var i, vi, v;

      for i in exprlist.index
      do [vi, sig] := eval(exprlist[i]);
         if sig ~= N then return([vi, sig]) fi;
         if ~ is_loc(vi) then ERROR fi;
         [v, sig] :=
         eval_field_selection('fetch', right, 'retrieve', [a_integer(i)]);
         if sig ~= N then return([v, sig]) fi;
         STATE.modify(vi, v);
      od;
      return([vi, N])
    else    # Case 2d. (simple assignment) #
      [left, sig] := eval(expr1);
      if sig ~= N then return([left, sig]) fi;
      if ~is_loc(left) then ERROR fi;
```

```
        STATE.modify(left, right);
        return([right, N])
      fi
    else        # Case 3. operator unequal ':=' #
      [left, sig] := eval(expr1);
      if string_equal(dop, '|')
      then
        if sig ~= F then return([left, sig]) fi
      else
        if sig ~= N then return([left, sig]) fi
      fi;
      [right, sig] := eval(expr2);
      if sig ~= N | string_equal(dop, '|') | string_equal(dop, '&')
      then
        return([right, sig])
      fi;
      left := dereference(left);
      right := dereference(right);
      if has_field('fetch', left, dop)
      then      # Case 3a. #
        return(eval_field_selection('fetch', left, dop, [right]))
      elif ENV.has_binding(dop) & is_proc(ENV.binding(dop))
      then      # Case 3b. #
        return(eval_call(dop, [left, right]))
      elif dop = '=' | dop = '~='
      then      # Case 3c. #
        sig := if equal(left, right) then N else F fi;
        if dop = '~='
        then
          sig := if sig = N then F else N fi
        fi;
        return([right, sig])
      else
        ERROR
      fi
    fi
  fi;
```

Notes

1) It is not possible to redefine the operators '|', '&', and ':='.

2) It is not even possible to define operators with a similar effect to '|' and '&', since these operators do not evaluate their arguments before they are applied. This form of parameter transmission is not available in SUMMER. For similar reasons, it is not possible to define 'assignment-like' operators.

3) The operators '=' and '~=' can be redefined. The order of steps 3b and 3c above ensures this.

9.2.17.d. Examples

1) $x := 10;$
$y := x + 1;$

{Assigns 11 to y.}

2) $p := 3;\ q := 23;\ r := 29;$
$s1 := \text{'}A\ Space\ Odyssey\text{'};$
$s2 := string(p * q * r) \parallel s1;$

{Assigns the value *'2001 A Space Odyssey'* to *s2*.}

3) $[x, y, z] := [1, 2, 3]$

{Is equivalent to: $x := 1; y := 2; z := 3$}

4) $[x, y] := [y, x]$

{Is equivalent to: $tmp := x; x := y; y := tmp$}

5) p & q

{Succeeds if both p and q succeed, and fails otherwise.}

6) p & **if** $a > b$ **then** $x > a$ **else** $x > b$ **fi**

9.2.18. Constant expressions

9.2.18.a. Syntax

<constant-expression> ::=
<simple-constant-expression>
(<constant-operator> <constant-expression>)* .

<simple-constant-expression> ::=
<constant> | <identifier> | '(' <constant-expression> ')' |
'–' <simple-constant-expression> .

<constant-operator> ::= '+' | '–' | '*' | '/' | '%' | '||' .

9.2.18.b. Pragmatics

<constant-expression>s occur in <case-entry>s {9.2.5} and <constant-initialization>s {9.1.3} and consist solely of <constant>s, <identifier>s which have been declared in a <constant-declaration>, and a limited set of operators. <constant-expression>s are evaluated by *eval*. The following procedure *require_constant_expression* ensures that all <identifier>s occurring in a <constant-expression> have indeed been declared in a <constant-declaration>.

9.2.18.c. Semantics

```
proc require_constant_expression(e)
( var se1, se2, name, tail, ce, c;
  {{ e == se1: <simple-constant-expression>
             tail: (<constant-operator> <constant-expression>)* }};
  if {{ se1 == <constant> }}
  then
    * ok *
  elif {{ se1 == name:<identifier> }}
  then
    if ~is_integer(ENV.binding(name)) & ~is_string(ENV.binding(name))
    then
      ERROR
    fi
  elif {{ se1 == '-' se2: <simple-constant-expression> }}
  then
    require_constant_expression(se2)
  elif {{ se1 == '(' ce:<constant-expression> ')' }}
  then
    require_constant_expression(ce)
  fi;
  for c in tail
  do {{ c == <constant-operator> ce:<constant-expression> }} ;
     require_constant_expression(ce)
  od
);
```

9.2.18.d. Examples

1) *pi*

{Assume that the <constant-declaration>

const *pi* := 3.14;

has occurred previously.}

2) 2 * *pi*

3) 7 + (2 * *pi*)

9.3. Miscellaneous functions used in the formal definition

This section is devoted to some functions which are used in the formal definition, but have not been described in preceding sections.

9.3.1. The function *dereference*

SUMMER allows very complex left hand sides of assignments. For example,

```
if a > b then
  if c > d then x else y fi
else
  z
fi := 3
```

assigns the value 3 to either x, y or z. To handle such general cases, evaluation always delivers values which are not dereferenced (see below). If a variable is evaluated, the result is the location bound to that variable and not the contents of that location. Locations are not *STORABLE-VALUE*s however, and as a consequence it is necessary to dereference the result of an evaluation before certain operations (use as actual parameter, right hand side of assignment, or operand in expression) can be performed. Dereferencing converts a location to the value it contains. This is precisely what is done by the procedure defined below.

```
proc dereference(val)
( if is_loc(val)
  then
    return(STATE.contents(val))
  else
    return(val)
  fi
);
```

9.3.2. The function *equal*

The procedure *equal* performs the basic equality operation between values and is described below:

```
proc equal(a, b)
( if is_integer(a) & is_integer(b)
  then
    return(a.intval = b.intval)
  elif is_string(a) & is_string(b)
  then
    return(string_equal(a.stringval, b.stringval))
  elif is_composite_instance(a) & is_composite_instance(b) &
      string_equal(a.class_decl.name, b.class_decl.name)
  then
    return(a.same_as(b))
  else
    freturn
  fi
);
```

9.3.3. The functions *substring* and *string_equal*

Two auxiliary operations on strings are used in the definition. The procedure *substring* extracts a substring of given length which begins at a given position from a given string. The procedure *string_equal* succeeds if two strings have the same size and contain the same characters. The definition of these procedures is omitted.

10. THE SUMMER LIBRARY

10.1. Introduction

This chapter describes the SUMMER library of standard classes. The library is made available to each SUMMER program automatically, i.e. the standard classes can be used without any definitions or declarations being required on the part of the programmer.

The types of formal parameters and return values of procedures and operators are indicated explicitly, in a PASCAL-like style.[1] For example,

proc *move*(*n* : *integer*) : *string*

denotes a procedure with *integer* parameter *n* and a return value of type *string*. If more than one type is allowed, the names of the legal types are joined with the infix *-or-* as exemplified by *integer-or-real-or-string*. A value of arbitrary type is denoted by *arbtype*.

If during the execution of a program, some procedure or operator described in this chapter is called with actual parameter values of a type that is not equal to one of the specified types, then an error is signalled. In the sequel the phrase 'an error is signalled' will be understood to mean that a semantic error is detected, that the execution of the program is terminated and that the user is notified of this fact by an appropriate error message.

The reader should be aware that this chapter contains an informal definition of the standard classes; this implies that no formal definitions in metalanguage will be given.

Each of the following paragraphs describes a separate class definition.

10.2. Class integer

Class *integer* defines integer values and their associated operations. The global organization of this class is:

```
class integer(arg : integer-or-real-or-string)
begin fetch +, −, *, /, %, <, <=, =, ~=, =>, >;

    var intval;

    op + (n) ( . . . );
    . . .
    op > (n) ( . . . );

init: intval := convert_to_integer(arg);

end integer;
```

When an instance of class *integer* is created, the actual value of *arg* may be one of several types. The following conversion rules apply:

---

1) This notation is used in this chapter only, and is not available in SUMMER itself.

Integer: No conversion is required.

Real: The real value *arg* is rounded to an integer, i.e. to the value *sign*(*arg*) × *entier*(*abs*(*arg*) + 0.5).

String: If the string *arg* has the form

['+' | '−'] <integer-constant>

then it can be converted to an integer. The creation of this instance of class *integer* fails if *arg* does not have this form.

From now on we assume that the result of the above conversion has been assigned to the instance variable *intval*.

As described in Section 9.2.1, a special denotation (i.e. <integer-constant>s) exists for instances of class *integer*. For example, *integer*('37') may be also be written as '37'.

The operations on integers are now described in more detail.

Operator: + *(n : integer-or-real) : integer-or-real*
− *(n : integer-or-real) : integer-or-real*
* *(n : integer-or-real) : integer-or-real*
/ *(n : integer-or-real) : real*

Action: Performs the arithmetic operation *intval* ⊕ *n*, where ⊕ is one of the above arithmetic operators. If the type of *n* is *integer*, the (integer) value obtained by the arithmetic operation is returned. If the type of *n* is *real*, the value of *real*(*intval*) ⊕ *n* is returned as a (real) value. For division ('/'), *n* is always converted to real.

Examples:

2 + 3	{value: 5}
2 + 3.5	{value: 5.5}
2 − 3.5	{value: −1.5}
integer(2.8) + 2	{value: 5}
8 / 5	{value: 1.6}

Operator: % *(n : integer) : integer*

Action: The (integer) value obtained by integer division of *intval* by *n* is returned as value. The result is positive if the values of *intval* and *n* are either both positive or both negative; the result is negative otherwise.

Examples:

7 % 2	{value: 3}
−7 % 2	{value: −3}
7 % −2	{value: −3}
−7 % −2	{value: 3}

Operator: $<$ *(n : integer-or-real) : integer-or-real*
$<=$ *(n : integer-or-real) : integer-or-real*
$=$ *(n : integer-or-real) : integer-or-real*
$\sim=$ *(n : integer-or-real) : integer-or-real*
$>=$ *(n : integer-or-real) : integer-or-real*
$>$ *(n : integer-or-real) : integer-or-real*

Action: Performs the arithmetic comparison *intval* $\ominus n$, where $\ominus$ is one of the above operators. If the type of n is *real*, the comparison *real(intval)* $\ominus n$ is performed. If the comparison succeeds, the (unconverted) value of n is returned as the value of the operation. Otherwise the operation fails. The correspondence between these operators and their mathematical counterparts is as follows:

Operation	Corresponds to
$m < n$	$m < n$
$m <= n$	$m \leqslant n$
$m = n$	$m = n$
$m \sim= n$	$m \neq n$
$m >= n$	$m \geqslant n$
$m > n$	$m > n$

Examples:

$2 < 3$	{value: 3}
$3 < 2$	{fails}
$3 < 4.5$	{value: 4.5}
$3 < 1 < 5$	{fails}
$3 < 4 < 5$	{value: 5}

10.3. Class real

Class *real* defines floating point numbers and their associated operations. The global organization of this class is:

```
class real(arg : integer-or-real-or-string)
begin fetch +, -, *, /, <, <=, =, ~=, >, >=;

    var realval;

    op + (r) ( . . . );
    . . .
    op >= (r) ( . . . );

init: realval := convert_to_real(arg);

end real;
```

When an instance of *real* is created, the actual value of *arg* may be one of several types. The following conversion rules apply to values of these types:

Integer: The integer value *arg* is converted to the corresponding real number.

Real: No conversion is required.

String: If the string *arg* has the form of a <real-constant>, then it can be converted to a real value. Otherwise, the creation of this instance of class *real* fails.

From now on we assume that the result of the above conversion has been assigned to the instance variable *realval*.

As described in Section 9.2.1, a special denotation (i.e. <real-constant>s) exists for instances of class *real*.

The operations on reals are now described in more detail.

Operator: + *(r : real-or-integer) : real*
− *(r : real-or-integer) : real*
* *(r : real-or-integer) : real*
/ *(r : real-or-integer) : real*

Action: Performs the operation *real* ⊕ *r*, where ⊕ is one of the above arithmetic operators. If the type of *r* is *integer* then *r* is first converted to *real* before the operation is performed.

Examples: 2.0 + 3.0 (value: 5.0)
2.5 + 3 (value: 5.5)

Operator: < *(r : real-or-integer) : real-or-integer*
<= *(r : real-or-integer) : real-or-integer*
= *(r : real-or-integer) : real-or-integer*
~= *(r : real-or-integer) : real-or-integer*
> *(r :real-or-integer) : real-or-integer*
>= *(r : real-or-integer) : real-or-integer*

Action: Performs the arithmetic comparison *realval* ⊖ *r*, where ⊖ is one of the above operators. If the type of *r* is *integer*, *r* is first converted to real before the comparison is performed. If the comparison succeeds, the operation returns the (unconverted) value of *r* as result. Otherwise, the operation fails. The correspondence between these operators and their mathematical counterparts is as follows:

Operation	Corresponds to
$p < q$	$p < q$
$p <= q$	$p \leqslant q$
$p = q$	$p = q$
$p \sim= q$	$p \neq q$
$p >= q$	$p \geqslant q$
$p > q$	$p > q$

(This correspondence is only approximate, since it depends on the precision of the floating point representation used to implement *reals*.)

Examples:	2.0 < 3.0	{value: 3.0}
	2.5 < 2.0	{fails}
	2.5 < 3	{value: 3}

10.4. Class string

Class *string* provides character strings and their associated operations. The global organization of this class is:

```
class string(arg : integer-or-real-or-string)
begin fetch center, index, left, next, repl, replace,
            retrieve, reverse, right, size, substr,
            <, <=, =, ~=, >=, >, || ;

    var stringval;

    proc center(n, s) ( . . . );
    . . .
    op || (s) ( . . . );

init: stringval := convert_to_array_of_char(arg);

end string;
```

When an instance of class *string* is created, the actual value of *arg* may be one of several types. The following conversion rules apply:

Integer: The result of converting the integer value of *arg* to a string is assigned to *stringval*. The conversion is performed in such a way that the equality *arg* = *integer*(*stringval*) holds.

Real: The result of converting the real value of *arg* to a string is assigned to *stringval*. The conversion is performed in such a way that the equality *arg* = *real*(*stringval*) holds.

String: No conversion is required.

As described in Section 9.2.1, a special denotation (i.e. <string-constant>s) exists for instances of class *string*.

Although *character* is not an available data type in SUMMER, we will, for reasons of convenience, consider strings to be equivalent to **arrays of characters**. Characters are defined to be equivalent to strings of length one. In this context, a phrase like 'some character in string *S*' means 'some substring of *S* of length one'. The string *'abc'*, for example, is equivalent to an array of length three, consisting of the characters '*a*', '*b*' and '*c*'.

The operations on strings are now described in detail.

Proc: *center(n : integer, s : string) : string*

Action: Returns a new instance of class *string* obtained by centering *stringval* in a string of length *n*. The remainder of that string is filled with replications of *s*, starting at both the left and right ends. Coming from the left end, *s* is truncated on the right, if necessary. Coming from the right end, *s* is truncated on the left, if necessary. If *stringval* cannot be centered exactly, it is placed one position left of the center. *n* should be non-negative.

Examples:

''.center(0, '.')	{value: ''}
''.center(5, '.')	{value: '. . . . .'}
'a'.center(0, '.')	{value: ''}
'a'.center(1, '.')	{value: 'a'}
'a'.center(5, '.')	{value: '. .a. .'}
'a'.center(4, '.')	{value: '.a. .'}
'a'.center(6, '. ')	{value: '. a . '}
'#'.center(5, *'abcd'*)	{value: *'ab#cd'*}
'ab'.center(5, '.')	{value: '.ab. .'}

Proc: *index () : interval*

Action: Delivers the set of all legal indices in this string, i.e. *interval*(0, *self.size* − 1, 1) {10.6}.

Examples: **for** *k* **in** *'grass'.index* **do** *put*(*k*, '\n') **od**

{Prints the integers 0, 1, 2, 3 and 4.}

Proc: *left(n : integer, s : string) : string*

Action: Returns a new instance of class *string* obtained by positioning *stringval* at the left of a string of length *n*. The remainder of that string is filled with replications of *s*, starting at the right. The last replication of *s* is truncated on the left, if necessary. *stringval* is truncated on the right if its size is greater than *n*. *n* should be non-negative.

Examples:

''.left(0, '.')	{value: ''}
''.left(5, '.')	{value: '. . . . .'}
'a'.left(0, '.')	{value: ''}
'a'.left(1, '.')	{value: *'a'*}
'a'.left(5, '.')	{value: *'a. . . .'*}
'a'.left(5, '. ')	{value: *'a. . '*}
'#'.left(3, *'abc'*)	{value: *'#bc'*}
'ab'.left(5, '.')	{value: *'ab. . .'*}
'ab'.left(5, '. ')	{value: *'ab . '*}

Proc: *next(state : undefined-or-integer) : array*

Action: This procedure produces the consecutive characters of *stringval* as strings of size one. This procedure is most often used in <for-expression>s {9.2.7} that iterate over all characters in a string. Calls to this procedure are generated automatically in <for-expression>s, and usually one need not be aware of the existence of this procedure. *next* proceeds as follows. If *state* has the value *undefined*, then first set *state* to zero. Then, if *state* < *self.size*, the array [*self.retrieve*(*state*), *state* + 1] is returned, otherwise *next* fails.

Examples: **for** *c* **in** *'lawn'* **do** *put*(*c*, '\n') **od**

{Prints the characters *'l'*, *'a'*, *'w'* and *'n'* on consecutive lines. Note that this **for**-expression is equivalent to:

```
gen := 'lawn';
state := undefined;
while [c, state] := gen.next(state)
do put(c, 'n') od;
```

where *gen* and *state* are hidden local variables.}

Proc: *repl(n : integer) : string*

Action: Returns a new instance of class *string* obtained by concatenating *n* copies of *stringval*. *n* should be non-negative.

Examples:

''.repl(2)	{value: *''*}
''.repl(0)	{value: *''*}
'a'.repl(0)	{value: *''*}
'a'.repl(1)	{value: *'a'*}
'a'.repl(4)	{value: *'aaaa'*}
'ab'.repl(0)	{value: *''*}
'ab'.repl(1)	{value: *'ab'*}
'ab'.repl(4)	{value: *'abababab'*}

Proc: *replace(s1, s2 : string) : string*

Action: Returns a new instance of class *string* obtained by replacing in *stringval* all characters that occur in *s*1 by the corresponding character in *s*2. If *s*2 is shorter than *s*1, the characters occurring in the tail of *s*2 are deleted from *stringval*. If *s*1 contains the same character more than once, the value corresponding to the right-most occurrence is used.

Examples:

'abcba'.replace('a', '')*	{value: *'*bcb*'*}
*'abcba'.replace('ac', '**')*	{value: *'*b*b*'*}
'abcba'.replace('aa', '−')*	{value: *'−bcb−'*}
'abcba'.replace('ax', '−')*	{value: *'*bcb*'*}
'abcba'.replace('xy', '−')*	{value: *'abcba'*}
'abcba'.replace('a', '')	{value: *'bcb'*}
'abcba'.replace('b', '')	{value: *'aca'*}
'abcba'.replace('', '')	{value: *'abcba'*}
''.replace('', '')	{value: *''*}

Proc: *retrieve(n : integer) : string*

Action: Returns a new instance of class *string* of size one, which consists of the *n*-th character in *stringval*. The condition $0 \leq n < self.size$ should hold.

Examples: *'apocalypse now'*[6] {value: *'y'*}

Proc: *reverse() : string*

Action: Let *stringval* consist of the characters $c_0, \ldots, c_{n-1}$. The result of *reverse* is a new instance of class *string* obtained by concatenating the characters $c_{n-1}, \ldots, c_0$ (in this order).

Examples:

''.reverse	{value: *''*}
'a'.reverse	{value: *'a'*}
'wolf'.reverse	{value: *'flow'*}
'rever'.reverse	{value: *'rever'*}

Proc: *right(n : integer, s : string) : string*

Action: Returns a new instance of class *string* obtained by positioning *stringval* at the right of a string of length *n*. The remainder of that string is filled with replications of *s*, starting at the left. The last replication of *s* is truncated on the right, if necessary. *stringval* is truncated on the left if its size is greater than *n*. *n* must be non-negative.

Examples:

''.right(0, *'.'*)	{value: *''*}
''.right(5, *'.'*)	{value: *'. . . . .'*}
'ab'.right(0, *'.'*)	{value: *''*}
'ab'.right(1, *'.'*)	{value: *'b'*}
'ab'.right(5, *'.'*)	{value: *'. . .ab'*}
'ab'.right(5, *'. '*)	{value: *'. .ab'*}
'#'.right(5, *'abcdef'*)	{value: *'abcde#'*}

Proc: *size () : integer*

Action: Returns the number of characters in *stringval*.

Examples:

'andromeda'.size	{value: 9}
''.size	{value: 0}

Proc: *substr(offset, length : integer) : string*

Action: Returns a new instance of class *string* obtained by taking a substring from *stringval*. If *length* = 0, the empty string is returned. Otherwise, let N be the number of characters in *stringval* and let M be equal to $min(N - 1, offset + length - 1)$. Then a string consisting of the characters with indices $offset, offset+1, \ldots, offset+M$ is returned. The conditions $0 \leqslant offset < N$ and $length \geqslant 0$ should hold.

Examples:

'abcd'.substr(0, 2)	{value: *'ab'*}
'abcd'.substr(1, 3)	{value: *'bcd'*}
'abcd'.substr(2, 7)	{value: *'cd'*}

Operator:

< (s : string) : string
<= (s : string) : string
= (s : string) : string
~= (s : string) : string
>= (s : string) : string
> (s : string) : string

Action: Let *ascii*(*c*) be a function that maps a character on its ordinal position in the ASCII character set. The lexical comparison of two characters c_1 and c_2 can now be defined by reducing lexical comparison to integer comparison, e.g.:

$$c_1 < c_2 \equiv ascii(c_1) < ascii(c_2)$$

Lexical comparison of two strings S_1 and S_2 depends on the lexical comparison of the characters in both strings and on the size of the two strings. Let $\{c_1, \ldots, c_n\}$ denote the characters in string S_1, let $\{d_1, \ldots, d_m\}$ denote the characters in string S_2, and let $min(n,m)$ be the smallest of the integers n and m. The string S_1 is then defined to be **lexically-less-than** (**llt**) the string S_2 iff **either** there exists an index k, $1 \leqslant k \leqslant min(n,m)$, such that $c_i = d_i$ for all $1 \leqslant i < k$ and $c_k < d_k$, **or** $n < m$.

The string S_1 is defined to be **lexically-equal-to** (**leq**) string S_2 iff $n = m$ and $c_i = d_i$ for all $1 \leqslant i \leqslant n$.

The lexical operators can now be defined as follows. The lexical comparison *stringval* $\ominus s$ is performed, where $\ominus$ is one of the lexical comparison operators given above. If the comparison succeeds, the operation returns s as value. Otherwise, the operation fails. Success or failure of the lexical operators is defined as follows:

Operation	Succeeds if
$P < Q$	P **llt** Q
$P <= Q$	P **llt** Q or P **leq** Q
$P = Q$	P **leq** Q
$P \sim= Q$	$\neg$ (P **leq** Q)
$P >= Q$	$\neg$ (P **llt** Q)
$P > Q$	$\neg$ (P **llt** Q) and $\neg$ (P **leq** Q)

Examples:

'a' < 'b'	{value: *'b'*}
'b' < 'a'	{fails}
'a' < 'a'	{fails}
'abc' < 'abd'	{value: *'abd'*}
'abc' < 'aba'	{fails}
'abc' < 'abcdef'	{value: *'abcdef'*}
'abc' <= 'abc'	{value: *'abc'*}
'abc' <= 'abd'	{value: *'abd'*}

Operator: || *(s : string) : string*

Action: Returns a new instance of class *string* consisting of the characters in *stringval* followed by the characters in *s*.

Examples:

'leg' || 'end'	{value: *'legend'*}
'now' || 'here'	{value: *'nowhere'*}

10.5. Class array

Class *array* provides a facility to create sequences of values. The organization of this class is:

```
class array(nelems : integer, defval : arbtype)
begin fetch append, delete, index, last, next,
            retrieve, size, sort, update;

    proc append(val) ( . . . );
    . . .
    proc update(i, val) ( . . . );

end array;
```

The elements of the array have arbitrary type. Different elements of an array may have different types. All array elements are initialized to the default value *defval*. See Section 9.2.12 for a description of <array-expression>s and in particular for a description of the initialization of arrays.

The operations on arrays are now described in more detail.

Proc: *append (val : arbtype) : arbtype*

Action: Extends the array by adding a new element to it at index position *nelems* with value *val*. The size of the array (i.e. *nelems*) is thus effectively incremented by one.

Examples: *a* := *array*(4, −1);

{Establishes a context for the following examples by creating a new instance of the class *array* and assigning it to *a*.}

a.append(10);
a.append(20);

{These two statements extend array *a* with array elements at index positions 4 and 5 and with values 10 and 20 respectively.}

a[3]	{value: −1}
a[4]	{value: 10}
a[5]	{value: 20}
a[6]	{error}

Proc: *delete () : arbtype*

Action: Removes the last element (i.e. the element at index position *nelems*−1) from the array and returns its value. The size of the array (i.e. *nelems*) is thus effectively decremented by one. The condition *nelems* > 0 should hold prior to the *delete* operation.

Examples: *a* := [10, 20, 30];

{Establishes the context for the following examples.}

a[2]	{value: 30}
a.delete	{value: 30}
a[2]	{error}

Proc: *index () : interval*

Action: Returns *interval*(0, *nelem* − 1, 1) {10.6} as value. This procedure is particularly useful in <for-expression>s {9.2.7} that iterate over all indices of an array.

Examples: a := *array*(6, 0);
for n **in** *a.index* **do** $a[n] := n * n$ **od**

{First an instance of *array* is assigned to variable a and then the square of its index is assigned to each array element.}

Proc: *last() : arbtype*

Action: Returns the value of the array element with index *nelems* − 1. This is useful when the array is treated in a stack-like fashion by means of *append* and *delete* operations.

Examples: a := [10, 20, 30];

a.last	{value: 30}
a.delete	{value: 30}
a.last	{value: 20}

Proc: *next(state : undefined-or-integer) : array*

Action: Returns the value of the next array element (if any) and fails otherwise. This procedure is mostly used in connection with <for-expression>s {9.2.7}. *next* proceeds as follows. If *state* has the value *undefined* and *nelem* > 0 holds, the array [*self.retrieve*(0), 1] is returned. If the type of *state* is *integer* and *state* < *nelem* − 1 holds, the array [*self.retrieve*(*state*), *state* + 1] is returned. Otherwise, *next* fails.

Examples: a := *array*(5, −1); $a[1] := 10$; $a[3] := 20$;
for x **in** a **do** *put*(x, '\n') **od**

{First a new array is created and initialized, next the values −1, 10, −1, 20, and −1 are printed on consecutive lines.}

Proc: *retrieve(i : integer) : arbtype*

Action: Returns the i-th value from the array. The condition $0 \leq i < nelems$ should hold. See also {9.2.15}.

Examples: a := [−2, −2, −2, −2];
$a[1] := 7$;

{Establishes a context for the following examples. First a new array is assigned to variable a. Next, 7 is assigned to the array element with index 1.}

a.retrieve(3);
a.retrieve(1);
a.retrieve(−5);

{The array elements with indices 3 and 1 are retrieved; this gives the respective values −2 and 7. An error will be signalled for the last expression since the index −5 is out of range. Note that these three expressions can be abbreviated to $a[3]$, $a[1]$ and $a[-5]$ respectively.}

Proc: *size () : integer*

Action: Returns the integer value *nelem*.

Examples: *a* := *array*(17, *'def'*) {creates new array}
a.size {returns 17}

Proc: *sort() : array*

Action: Returns a new instance of class *array* obtained by sorting the contents of the array, i.e. *self*. Values are sorted according to their type in the following order:

undefined
integer
real
string
array
table
file
bits
scan_string
interval
(user defined classes in order of declaration in the program)

Integers and reals are put in order of increasing numeric value. Strings are sorted lexicographically. Other values are sorted according to their creation time: older values come before values that were created more recently.

Examples: [4,'z',3,'a'].*sort*

{value: [3, 4, *'a'*, *'z'*]}

Proc: *update(i : integer, val : arbtype) : arbtype*

Action: This procedures updates the value of the *i*-th element from the array in such a way that subsequent retrieval (without an intervening update) of the *i*-th element returns the value *val*. *val* may be of arbitrary type. The condition $0 \leq i < nelems$ should hold. See also 9.2.17.

Examples: *a* := *array*(4, −1);

{Establishes the context for the following examples.}

a.update(2, 5);
a.update(7, *'ab'*);

{The first expression assigns the integer value 5 to the array element with index 2. For the second expression an error will be signalled, since the index 7 is out of range. Note that these two expressions can be abbreviated to *a*[2] := 5 and *a*[7] := *'ab'* respectively.}

10.6. Class interval

Class *interval* defines intervals of integer or real values. Instances of this class are most often used in <for-expression>s (9.2.7). The organization of this class is as follows:

```
class interval( from, to, by : integer-or-real)
begin fetch next;

    proc next(state) ( . . . );

end interval;
```

The formal parameters *from*, *to* and *by* must be of type *integer* or *real*. If one is of type *real*, then the values of the other (integer) parameters are first converted to *real*. Note that a definition of this class has been given as an example in Section 9.2.7.d. The only operation (*next*) on intervals is now described in detail.

Proc: *next(state : undefined-or-integer-or-real) : array*

Action: Returns the next value in the *interval* (if any) and fails otherwise. This procedure is most often used in connection with <for-expression>s. *next* proceeds as follows. If the type of *state* is *undefined* then set V to the value of *from*, otherwise set V to the value of *state* + *by*. If the value of *by* is positive and $V <$ *to* holds, or the value of *by* is negative and $V >=$ *to* holds, then the array $[V, V]$ is returned. In all other cases *next* fails.

Examples: **for** x **in** *interval*(1, 5, 2) **do** *put*(x, '\n') **od**

{Prints the integers 1, 3 and 5}

for x **in** *interval*(5, 0, −2) **do** *put*(x '\n') **od**

{Prints the integers 5, 3 and 1}

for x **in** *interval*(2, 5, 1.3) **do** *put*(x, '\n') **od**

{Prints the reals 2.0, 3.3 and 4.6}

10.7. Class table

Class *table* provides an associative memory that can be indexed with values of arbitrary type. Basically, it is organized as follows:

```
class table(nentries : integer, defval : arbtype)
begin fetch index, next, retrieve, size, update;

    proc index () ( . . . );
    . . .
    proc update(i, val) ( . . . );

end table;
```

A table may contain an arbitrary number of entries, but (as a hint for the implementation) an estimate of the expected number of entries must be given as value of *nentries*. Each entry in the table consists of a (key, value) pair. An update operation either adds a new entry to the table or replaces the value part of an existing entry. A retrieve operation returns the value part of the entry corresponding to a given key. If such an entry does not exist, then the default value *defval* is returned. As a

consequence, no distinction can be made between entries that were never added to the table and entries that contain *defval* as value part. This property of tables can be used to delete table entries. A table entry will be called **live** if its value part is not equal to *defval*. All other entries are called **dead**. Assigning *defval* to the value part of an entry, **kills** that entry. The operations *index*, *next* and *size* operate on the live entries in the table. This distinction between live, dead and nonexistent entries is necessary for the description of the behavior of tables in <for-expression>s.

See Section 9.2.13 for a description of <table-expression>s and in particular for a description of the initialization of tables.

In the examples the table *taste* will be used, It is defined as follows:

```
taste := table(20, 0) init [
        'sweet':        10,
        'bitter':       -8,
        'sour': 'acid': -11,
        'pickled':      +17 ];
```

Operations on tables are now described in more detail.

Proc: *index() : array*

Action: Returns an array with (sorted) table keys, for all live table entries. The keys are sorted as described in 10.5.

Examples: *taste.index*

{value: ['acid', 'bitter', 'pickled', 'sour', 'sweet'] }

for *t* **in** *taste.index* **do** *put*(*t*, '\n') **od**

{Prints the strings *'acid'*, *'bitter'*, *'pickled'*, *'sour'* and *'sweet'* on consecutive lines.}

Proc: *next(state : undefined-or-array) : array*

Action: Enumerates the value parts of the live entries in the table. If *state* is equal to *undefined*, the effect of *next* is the same as the effect of *next*(*self.index*, 0). Otherwise, *state* should have the form [*A* , *N*] and the following steps are performed: As long as $N < A.size$ and *self.retrieve*(*A* [*N*]) is equal to *defval*, *N* is incremented by one. If $N = A.size$, *next* fails. Otherwise, [*self.retrieve*(*A* [*N*]), [*A* , $N + 1$]] is returned.

Examples: **for** *flavor* **in** *taste* **do** *put*(*flavor*, '\n') **od**

{Prints the integers −11, −8, 17, −11 and 10.}

Notes: If we restrict our attention to the use of *next* in <for-expression>s {9.2.7}, the following can be observed: The *index* of the table is computed only once. This gives an array *A* with a fixed number of keys, which cannot be affected by operations on the table inside the body of the <for-expression>. However, the value parts of the entries containing the keys in *A* may be affected by intermediate update operations. The procedure *next* is defined in such a way that table entries that were killed since the creation of the index, are effectively skipped.

Proc: *retrieve(key : arbtype) : arbtype*

Action: If a call of the form *update(key,val)* has occurred previously, *val* is returned. Otherwise the default value *defval* is returned. Note that tables are also initialized by means of *update* operations {9.2.7}.

Examples: *taste.retrieve('sour')* {value: −11}
taste.retrieve('dulce') {value: 0}

{Note that these examples may be abbreviated to *taste['sour']* and *taste['dulce']* respectively.}

Proc: *size () : integer*

Action: Returns the number of live entries in the table.

Examples: *taste.size* {value: 5}

Proc: *update(key, val : arbtype) : arbtype*

Action: Associates the value *val* with *key*. If *val* is equal to the default value *defval* this operation kills the table entry containing *key* (if it exists). The value *val* is returned as result of *update*.

Examples: *taste.update('salted', 17)*

{Adds the (key, value) pair *('salted'*, 17) to the table and returns 17 as value. Note that the above expression may be abbreviated to *taste['salted'] := 17.*}

10.8. Class scan_string

Class *scan_string* provides a scanning facility for strings and files. Basically, it is organized as follows:

```
class scan_string(arg : string-or-file)
begin fetch any, bal, break, cursor, find, lit, move,
            pos, rtab, span, tab, text;
    var text, cursor;

    proc any(s) ( . . . );
    . . .
    proc tab(n) ( . . . );

init: cursor := 0
    text := convert_to_string(arg);
end scan_string;
```

When an instance of class *scan_string* is created, the actual value of *arg* may be either a *string* or a *file*. In the latter case, the file is (at least conceptually) converted to a string. All scanning procedures operate on this string. The (perhaps converted) value of *arg* is assigned to the class variable *text* which is thereafter used by all operations defined for the class. The current cursor position is maintained in the class variable *cursor*. Both *text* and *cursor* may be fetched from outside the class instance. The value of *text* remains invariant under all class operations, but the value of *cursor* may change.

Sometimes it is convenient to look at the subject string (*text*) as being an array of characters. This is done in the way described in Section 10.4.

A few general restrictions apply to the following definitions:

1) Unless stated otherwise, operations fail if *cursor* is equal to *text.size*.

2) If more than one cursor value satisfies some condition, then the smallest of these values is used.

It must be emphasized that the examples given in this section are not typical: they serve to illustrate the working of just one procedure, but do not properly illustrate the use of procedures in more realistic applications. All these procedures will, in general, be used in conjunction with <scan-expression>s {9.2.9}. When used in this way *scan_string* objects need hardly ever be created or mentioned explicitly. See chapters 4 and 11 for more interesting examples.

The meaning of the various procedures is defined below.

Proc: *any(s: string) : string*

Action: If *text*[*cursor*] is equal to one of the characters in *s*, then *text*[*cursor*] is returned and *cursor* is incremented by one. Otherwise *any* fails.

Examples: *p* := *scan_string('quota')*;

{Establishes the context for the following examples.}

p.any('0123456789');

{Tests for the occurrence of digits at the start of the subject string and fails; the cursor is not affected.}

x := *p.any('pqr')*

{This test for the occurrence of one of the characters *'p'*, *'q'* or *'r'* succeeds: the cursor of *p* is moved to the right (i.e. *p.cursor* = 1) and the string *'q'* is returned by *any* and assigned to the variable *x*.}

Proc: *bal(S, P, Q : string) : string*

Action: If there exists a cursor value *c* such that *c* ≥ *cursor*, *text*[*c*] equal to one of the characters in *S*, and *text.substr*(*cursor*, *c* − *cursor*) "balanced" with respect to *P* and *Q*, the balanced substring is returned and *cursor* is set to *c*. If more than one value of *c* satisfies the above condition, then the smallest (i.e. leftmost) one is used. Otherwise *bal* fails.

A string *S* is balanced with respect to two other strings *P* and *Q* iff:

S is a single character not in *P* or *Q*

or

S has the form c_1 || *A* || c_2 where c_1 is a single character occurring in *P* and c_2 is a single character occurring in *Q* and *A* is balanced with respect to *P* and *Q*.

or

S has the form *A* || *B* and both *A* and *B* are balanced with respect to *P* and *Q*.

Examples: *e* := *scan_string('x:=a+b[i]*(y+5); z:=3')*;

{Establishes the context for the next example.}

x := *e.bal(';', '([', ')]')*;

{Assigns *'x:=a+b[i]*(y+5)'* to *x* and moves the cursor to 15, i.e. *e.cursor* = 15.}

e := *scan_string('{xx(x())!xx}')*;

{Establishes the context for the next two examples.}

e.bal('!', '({', '})')

{This expression fails and does not move the cursor.}

x := *e.bal('!', '(', ')')*;

{Assigns *'{xx(x())'* to *x* and moves the cursor to 8.}

Proc: *break(s : string) : string*

Action: If there exists a cursor value *c* such that *c* ≥ *cursor* and *text*[*c*] is equal to one of the characters in *s*, then *text.substr*(*cursor*, *c* − *cursor*) is returned and the cursor is set to *c*. If more than one value of *c* satisfies the above condition, the smallest (i.e. leftmost) one is used. *break* fails otherwise.

Examples: *answer* := *scan_string('yes/no')*;

{Establishes the context for the next two examples.}

answer.break('!?;') {fails}
answer.break('/')

{The last expression succeeds, moves the cursor to 3 and returns the string *'yes'*.}

Proc: *find(s : string) : string*

Action: If there exists a cursor value *c* such that *c* ≥ *cursor* and *s* is a substring of *text* starting at *c*, then *text.substr*(*cursor*, *c* − *cursor*) is returned and the cursor is set to *c*. Otherwise *find* fails.

Examples: *fruit* := *scan_string('apple;pear;lemon;peach')*;

{Establishes the context for the following examples.}

fruit.find('banana') {fails}
fruit.find('apple')

{Returns the value '' (i.e. the empty string) and leaves the cursor at 0}

fruit.find('lemon')

{Succeeds, moves the cursor to 11, and returns the string *'apple;pear;'*.}

Proc: *lit(s : string) : string*

Action: If *s* is a substring of *text* starting at the current cursor position, *s* is returned and *cursor* is incremented by the length of *s*. Otherwise *lit* fails.

Examples: *person* := *scan_string('the man')*;

{Establishes the context for the following examples.}

person.lit('some'); {fails}
person.lit('man'); {fails}
person.lit('the')

{The last expression succeeds, moves *cursor* to 3 and returns the string *'the'*.}

Proc: *move(n : integer) : string*

Action: If $0 \leq cursor + n \leq text.size$, there are two different cases:
1) $n \geq 0$: Let *R* be *text.substr*(*cursor*, *n*).
2) $n < 0$: Let *R* be *text.substr*(*cursor* + *n*, − *n*).
In both cases *cursor* is incremented by *n* and *R* is returned. Otherwise *move* fails.

Examples: *digits* := *scan_string('0123456789')*;

{Establishes the context for the following examples.}

digits.move(15) {fails}
digits.move(5)

{Succeeds, moves the cursor to 5, and returns the string *'01234'*.}

digits.move(−2)

{Succeeds, moves the cursor back to 3, and returns the string *'34'*.}

Proc: *pos(n : integer)*

Action: If *cursor* = *n*, the empty string is returned. Otherwise *pos* fails.

Examples: *digits* := *scan_string('0123456789')*;

{Establishes the context for the following examples.}

digits.pos(0); {value: ''}
digits.move(2);

{Returns the value *'01'* and moves the cursor to 2}

digits.pos(0); {fails}
digits.pos(2); {value: ''}

Proc: *rpos(n : integer)*

Action: If *cursor* = *text.size* − *n*, the empty string is returned as value. Otherwise *rpos* fails.

Examples: *digits* := *scan_string('0123456789')*;

{Establishes the context for the following examples.}

digits.rpos(10) {value: ''}
digits.rpos(4) {fails}
digits.move(10)

{Returns the value *'0123456789'* and moves the cursor to 10}

digits.rpos(0) {value: ''}

Proc: *rtab(n : integer) : string*

Action: Let R be equal to *text.size* $-$ n. If $0 \leq n \leq$ *text.size*, there are two different cases:
1) *cursor* $\leq R$: Set V to *text.substr*(*cursor*,$R -$ *cursor*).
2) *cursor* $> R$: Set V to *text.substr*(R,*cursor* $- R$).
In both cases the cursor is set to R and V is returned. Otherwise *rtab* fails.

Notes: *rtab*(n) has the same effect as *tab*((*text.size*)$-n$).

Proc: *span(s : string) : string*

Action: If there exists a cursor value c such that $c >$ *cursor*, and *text.substr*(*cursor*, $c -$ *cursor*) consists solely of characters in s, this substring is returned and the cursor is set to c. If more than one value of c satisfies this condition, the largest (i.e. rightmost) one is used. Otherwise *span* fails.

Examples: *p* := *scan_string('. . . . finally!')*;

{Establishes the context for the following examples.}

p.span('?;') {fails}

p.span('.')

{Succeeds, moves the cursor to 5, and returns the string '. . . . .'.}

Proc: *tab(n : integer) : string*

Action: If $0 \leq n \leq$ *text.size*, there are two different cases:
1) $n \geq$ *cursor* : Set V to *text.substr*(*cursor*,$n -$ *cursor*).
2) $n <$ *cursor* : Set V to *text.substr*(n,*cursor* $- n$).
In both cases the cursor is set to n and V is returned. Otherwise *tab* fails.

Examples: *h* := *scan_string('History of exact sciences')*;

{Establishes the context for the following examples.}

h.tab(7)

{Returns the value *'History'* and moves the cursor to 7}

h.tab(3)

{Returns the value *'tory'* and moves the cursor to 3}

10.9. Class file

The class *file* provides external character files with associated operations on them. The basic organization of this class is as follows:

```
class file(name, accesstype : string)
begin fetch close, get, put;

    proc close() ( . . . );

    proc get() ( . . . );

    proc put(v1, v2, . . . ) ( . . . );

end file;
```

When a new instance of class file is created, an external file with name *name* is opened or created; *name* must be an acceptable file name for the local operating system. *accesstype* determines the intended use of the file as follows:

'r'	for reading
'w'	for writing

Three predefined file names exist: *'stand_in'*, *'stand_out'* and *'stand_er'* which correspond to the standard input , standard output, and standard error file respectively. Apart from these predefined file names, there are also three predefined global variables *stand_in*, *stand_out* and *stand_er*, which have these three predefined files as initial value.

Proc: *close()*

Action: Close file. After a file has been closed no further i/o operations can be performed on it.

Examples:
```
scratch := file('tmp1', 'w');

{A file with name 'tmp1' is opened for writing and the resulting file
instance is assigned to scratch.}

scratch.close;

{Next, that file is closed.}
```

Notes: All files are closed when the program terminates. Closing a file from within the scope of a <try-expression> {9.2.8} is forbidden.

Proc: *get() : string*

Action: Returns the next line (without trailing newline symbol) from the file and fails on end of file.

Examples:
```
source := file('mytext', 'r');

{The file 'mytext' is opened for reading and the resulting file instance is
assigned to source.}

line := source.get

{Next, one line is read from that file and the resulting string is assigned to
line.}
```

stand_in.get

{Reads one line from standard input.}

Notes: See also global procedure *get* {10.11} which operates on standard input.

Proc: *put(v1, v2, . . . : integer-or-real-or-string)*

Action: Writes the values *v1*, *v2*, . . . on the file, after converting them to strings. No newline character is appended.

Examples: *f* := *file('results', 'w')*;
f.put('The value of "x" is :', x);

{Creates the file *'results'* and writes on it.}

stand_er.put('Fatal error!')

{Writes a string on the standard error stream.}

Notes: See also global procedure *put* {10.11} which operates on standard output.

10.10. Class bits

Class *bits* provides arbitrary length bit-strings and associated operations. A bit-string consists solely of the integer values 0 and 1. This class is organized as follows:

```
class bits(nelem, defval : integer)
begin fetch conj, compl, disj, index, next,
            retrieve, update, size;

    proc conj(b) ( . . . );
    . . .
    proc update(i, val) ( . . . );

end bits;
```

The formal parameter *nelem* must be of type integer and defines the number of elements in the bit-string. *defval* must be either integer 0 or integer 1 and defines the value to which all elements of the bit-string are initialized.

There is no way to initialize bit-strings other than by explicit assignment to individual elements.

Proc: *conj(b : bits) : bits*

Action: Computes the bit-wise **and** with the bits value *b*. If *nelem* and *b.size* are unequal, the smallest bit-string is first padded with ones at the right end, i.e. starting at the highest index.

Proc: *disj(b : bits) : bits*

Action: Computes the bit-wise **or** with the bits value *b*. If *nelem* and *b.size* are unequal, then the smallest bit-string is first padded with zeros at the right end, i.e. starting at the highest index.

Proc: *compl () : bits*

Action: Computes the bit-wise complement.

Proc: *index () : interval*

Action: Returns the index set *interval*(0,*nelems* − 1,1) {10.6}.

Proc: *next (state : undefined-or-integer) : array*

Action: Returns the value of the next bit (if any) and fails otherwise. This procedure is mostly used in conjunction with <for-expression>s {9.2.7}. *next* proceeds as follows. If *state* is *undefined* and *nelem*>0, the array [*self.retrieve*(0), 1] is returned. If the type of *state* is *integer* and *state* < *nelem* − 1, the array [*self.retrieve*(*state*), *state* + 1] is returned. Otherwise *next* fails.

Proc: *retrieve(i : integer) : integer*

Action: Returns the *i*-th element. The condition 0 ≤ *i* < *nelem* should hold.

Proc: *size () : integer*

Action: Returns the integer value *nelem*.

Proc: *update(i, val : integer) : integer*

Action: Assigns the integer value *val* to the *i*-th element. *val* should be 0 or 1, and the condition 0 ≤ *i* < *nelem* should hold.

10.11. Miscellaneous procedures

Proc: *copy(a : arbtype) : arbtype*

Action: Returns a copy of the value *a*. If *a* is composite, all its components are copied recursively.

Proc: *get() : string*

Action: This is an abbreviation for *stand_in.get*() {10.9}. It reads the next line (without trailing newline character) from the standard input file and fails on end of file.

Proc: *put(v1, v2, . . . : integer-or-real-or-string)*

Action: This is an abbreviation for *stand_out.put*(v1, v2, . . .) {10.9}. It writes the values v1, v2, . . . on the standard output file, after converting them to strings.

Proc: *stop(n : integer)*

Action: Stops program execution. If *n* is negative, the names and values of the actual parameters of all currently entered procedures are printed. The absolute value of *n* is returned to the operating system. By convention, values unequal to zero are considered error codes.

Examples: *stop*(0)

Proc: *type(a : arbtype) : string*

Action: Determine the type of *a*. The following string values can be returned depending on the operand type:

'integer'	integer
'real'	real
'string'	string
'file'	file
'undefined'	undefined value
'table'	table
'array'	array
'bits'	bits

For class instances the class name is returned.

Notes: A formal definition of *type* was given in Section 9.2.2.

11. SOME ANNOTATED SUMMER PROGRAMS

11.1. Introduction

This chapter presents some annotated SUMMER programs. The description of each program has the following overall structure:

a) Global description of the task the program performs and a list of language features illustrated by it.

b) Listing of the source text (with added line numbers for ease of reference).

c) Sample input (if any).

d) Sample output.

e) A detailed description.

11.2.1. Word tuples

11.2.1.a. Overall description

This program reads a number of files and for each different pair or triple of consecutive words the frequency is counted. A word is a sequence of upper and lower case letters. The result of the program is an alphabetically ordered frequency table.

The program illustrates: <for-expression>s {9.2.7}, <scan-expression>s {9.2.9}, *scan_string* operations (*break*, *span*) {10.8}, multiple assignment {9.2.17} and *tables* {10.7}.

11.2.1.b. Source text

```
 1 # tuples -- count frequency of word pairs and triples #

 2 const letter := 
 3       'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ',
 4       interesting := 3;

 5 proc combine(a, b)
 6   return(a || '/' || b);

 7 proc word(data)
 8 ( break(letter) & return(span(letter))
 9 );

10 program tuples( file_names)
11 ( var f, tuple, tupletab := table(1000, 0);

12   for f in file_names
13   do var data;
14      if data := file( f,'r') fails then
15        put('Can''t open ', f, '\n')
16      else
17        scan data
18        for var w1 , w2 := word(data), w3 := word(data),
19                pair, triple;
20            while [w1, w2, w3] := [w2, w3, word(data)]
21            do pair := combine(w1, w2);
```

```
22              tupletab[pair] := tupletab[pair] + 1;
23              triple := combine(pair, w3);
24              tupletab[triple] := tupletab[triple] + 1;
25          od;
26          pair := combine(w2, w3);    # the last pair #
27          tupletab[pair] := tupletab[pair] + 1
28      rof;
29      data.close
30    fi
31  od;
32  for tuple in tupletab.index
33  do var freq := tupletab[tuple];
34    if freq >= interesting then
35      put(tuple.left(20,' '),string(freq).right(10,' '),'\n')
36    fi
37  od
38 )
```

11.2.1.c. Input

The source text of the programs 'tuples.sm' (the above program) and 'flex.sm' {11.2.2}.

11.2.1.d. Output

```
f/i                 6
f/size              4
fi/proc             5
flexible/array      4
i/i                 9
i/i/i               4
i/od                4
in/f                4
mem/i               4
mem/size            5
return/mem          4
size/then           6
then/return         4
tupletab/pair       4
w/w                 7
word/data           4
```

11.2.1.e. Detailed description

1 Comment

2-4 Constant declaration for *letter* and *interesting*. The former is used to recognize words in the text (line 8), the latter is used as threshold while printing the frequency table (line 34).

5-6 The procedure *combine* concatenates its arguments separated by the character '/' and is used in lines 21, 23 and 26 to build table keys consisting of two or three words.

7-9 The procedure *word* is the basic procedure for recognizing 'words'. It operates on the current subject (established in line 17) and first locates the beginning of the next word by *break*(*letter*) which moves the cursor to a position just before a letter. The operation *span*(*letter*) spaces over the longest letter sequence in the subject and delivers that sequence as (string) value. This value is ultimately returned. Note that *break* fails if there are no more letters in the subject. In that case the procedure *word* fails and the while loop in line 20 is terminated.

10-38 Main program with argument *file_names*, which contains an array of string values corresponding to the actual parameters of program *tuples*. If, for example, the program was called with the two arguments *tuples.sm* and *flex.sm*, the argument *file_names* has the array [*'tuples.sm'*, *'flex.sm'*] as value. The <for-expression> in line 12 iterates over the values in this array.

11 Variable declarations for *f*, *tuple* and *tupletab*. Note how *tupletab* is initialized to an instance of class *table*, in this case of estimated size 1000 and default value 0. It is used to keep account of the occurrence of pairs and triples. The default value 0 is a convenient initial value for tuples that were not encountered before.

12-31 The consecutive values in the array *file_names* are assigned to *f*.

13 Declaration for local variable *data*.

14 Create an instance of class *file* with the current value of *f* as name and access type *'r'* (for read access). If the file can be opened, an instance of class file is assigned to variable *data*. Otherwise the operation fails and an error message is printed (line 15).

17-28 Scan expression, which establishes the value of *data* (a file) as subject. Operations like *span* and *break* now operate on the value of *data*. In the current context the expression *break*(*letter*) is equivalent to *data.break*(*letter*). The main purpose of scan expressions is that they allow the omission of the current subject (in this case *data*).

18 Declaration of *w1*, *w2*, *w3*, *pair* and *triple* with initialization of *w2* and *w3*. There is an implicit assumption here: since the procedure *word* can fail (see above), the input file should contain at least two 'words'. If not, the run-time error 'Undetected failure of procedure word' will be given and the execution of the program will be terminated.

20-25 <while-expression> terminated by failure of *word*. The expression in the test part of the while expression is a multiple assignment and is equivalent to:

```
w1 := w2;
w2 := w3;
w3 := word(data)
```

21-24 The frequencies of the pair (w1, w2) and the triple (w1, w2, w3) are incremented. Because *tupletab* has a default value of 0 (cf. line 11), the frequency of pairs and triples that did not occur previously, is set to 1 by these statements.

29 Terminate i/o operations on this file.

32-37 <for-expression> that loops over all (sorted) indices in *tupletab*.

33-35 The value corresponding to the current value of *tuple* is assigned to local variable *freq*. If it is higher than the threshold *interesting*, the value of *tuple* (left aligned in a field of width 20) is printed, followed by the value of *freq* (right aligned in a field of width 10).

11.2.2. Flexible arrays

11.2.2.a. Overall description

This program shows the declaration of class *flexible_array*, a rather flexible kind of array on which the following operations are defined:

a) *update* and *retrieve* operations similar to the corresponding operations defined on arrays.

b) *append* and *delete* operations to add an element at the end of the flexible array or to delete a number of elements from the end.

c) A *size* operation, which can both be used to inspect the current size and to reset the size.

d) Operations *next* and *index* similar to those for normal arrays.

This program illustrates <class-declaration>s {9.1.5}, <for-expression>s {9.2.7} (cooperating with user-defined fields *next* and *index*) and <assert-expression>s {9.2.10}.

Note that the built-in arrays in SUMMER are already quite 'flexible': operations similar to *append*, *delete* and *top* are defined on them. The purpose of the following example is to show how such a data type could be defined in a user program.

11.2.2.b. Source text

```
1  # flexible_array #

2  class flexible_array ()
3  begin fetch update, retrieve, append, delete, size, next,
4              index, top;
5        store size : change_size;

6        var mem, size;

7        proc extend(n)
8        ( var i, m1 := array(n, undefined);
9          for i in mem.index do m1[i] := mem[i] od;
10         mem := m1
11       );

12       proc retrieve(i)
13         if 0 <= i < size then return(mem[i]) else stop(-1) fi;

14       proc update(i, v)
15         if 0 <= i < size then return(mem[i] := v) else stop(-1) fi;

16       proc append(v)
```

```
17        ( if size >= mem.size then extend(size + 10) fi;
18          mem[size] := v;
19          size := size + 1;
20          return(v)
21        );
22        proc delete(n)
23          if n >= 0 then return(change_size(size - n)) else stop(-1) fi;
24        proc change_size(n)
25          if n < 0
26          then
27            freturn
28          else
29            if n > mem.size
30            then
31              extend(n)
32            else
33              var i;
34              for i in interval(n, size-1, 1)
35              do mem[i] := undefined od
36            fi;
37            return(size := n)
38          fi;
39        proc next(state)
40        ( if state = undefined then state := 0 fi;
41          if state < size
42          then
43            return([mem[state], state + 1])
44          else
45            freturn
46          fi
47        );
48        proc index()
49          return(interval(0, size - 1, 1));
50        proc top()
51          if size = 0 then freturn else return(mem[size-1]) fi;
52  init: mem := array(10, undefined);
53        size := 0;
54  end flexible_array;
55  proc p4(v) put(string(v).right(4, ' '));
56  program demo_flex()
57  ( var f, i, k;
58    const N := 12;
59    f := flexible_array;
60    put('Initialize f:');
```

```
61  for i in interval(0, N−1, 1)
62  do f.append(i*i); p4(f[i]);
63      assert f.top = i*i & f[i] = i*i & f.size = i+1;
64  od;
65  put('\nIndices in f:');
66  i := 0;
67  for k in f.index
68  do p4(k);
69      assert k = i;
70      i := i + 1
71  od;
72  assert i = N;
73  put('\nValues in f: ');
74  i := 0;
75  for k in f
76  do p4(k);
77      assert k = f[i] & k = i*i;
78      i := i + 1
79  od;
80  assert i = N;
81  f.delete(2);
82  assert f.size = N−2;
83  f.size := 7;
84  assert f.size = 7;
85 )
```

11.2.2.c. Input

None

11.2.2.d. Output

```
Initialize f:  0  1  4  9  16  25  36  49  64  81  100  121
Indices in f:  0  1  2  3  4   5   6   7   8   9  10   11
Values in f:   0  1  4  9  16  25  36  49  64  81  100  121
```

11.2.2.e. Detailed description

2-54, 55, 56-85 Declarations of class *flexible_array*, procedure *p4* and program *demo_flex*.

3-5 Declarations of fields that may be accessed from outside the class. The fields listed in lines 3-4 may be fetched. Assignments to the field *size* are allowed and are carried out by the procedure *change_size*.

7-11 Procedure *extend* is used only inside the class declaration and extends array *mem*: a new array *m1* is created, all values in *mem* are copied to it, and, finally, the new array is assigned to *mem*.

12-13 Proccdure *retrieve* ensures that i lies in the interval $0 \leqslant i < size$ and then returns the i-th element of *mem*. If, for example, f is an instance of class *flexible_array*, then

$x := f[5];$

is equivalent to

$x := f.retrieve(5)$

14-15 Procedure *update* changes the value of the i-th element into v provided that $0 \leqslant i < size$. If, again, f is a *flexible_array*,

$f[5] := 7$

is equivalent to

$f.update(5, 7)$

16-21 Procedure *append* adds the value v at the end of the *flexible_array*. If necessary, *mem* is extended first.

22-23 Procedure *delete* deletes the rightmost n elements by calling *change_size* (see below).

24-38 Procedure *change_size* implements assignments to the field *size* and is called to evaluate expressions like

$f.size := 3$.

The seeming complexity of *change_size* is due to the fact that the invariant

$size \leqslant i < mem.size \rightarrow mem[i] = undefined$

is maintained to ensure that elements 'added' to the array are properly set to *undefined*.

39-47 A user-defined *next* procedure. Its operation is explained in conjunction with the <for-expression> in lines 75-79.

48-49 A user-defined *index* procedure, which delivers an *interval* of the legal indices in this *flexible_array*.

50-51 Procedure *top* delivers the rightmost element of the *flexible_array*.

52-53 Initialization code. Assigns an array value to *mem* and sets *size* to zero.

55 Procedure *p4* prints its argument v right aligned in a field of width 4.

59 An instance of *flexible_array* is assigned to f.

61-64 Fill the *flexible_array* with squares. Note how the <assert-expression> checks that the operations have been performed properly. If the <assert-expression> fails, execution of the whole program is aborted.

67-71 Loop using the *index* operation. The variable i is used to maintain a 'shadow' administration of the index values; this is used by the <assert-expression in line 72.

75-79 Loop using the *next* operation. The expression

for k **in** f **do** . . . **od**

is equivalent to

phi := *f*;
sigma := *undefined*;
while [*k*, *sigma*] := *phi.next*(*sigma*) **do** . . . **od**

As we see in line 40, *next* sets its argument *state* to 0 when its value is *undefined*. On subsequent calls, the value of *state* is compared with the value of *size* (line 41). If the former is less than the latter, an array with value-for-this-iteration and new-state is returned. Note how the multiple assignment

[*k*, *sigma*] := *phi.next*(*sigma*)

assigns the value-for-this-iteration to *k* and the new-state to *sigma*. The next operation fails if there are no more elements in the *flexible_array*.

81-82 Delete the last two elements and make sure that the size of *f* has been reduced accordingly.

83-84 Assign to the *size* field and check resulting size.

12. SUMMARY OF SUMMER SYNTAX

```
<summer-program> ::=
      ( <variable-declaration> | <constant-declaration> |
        <procedure-declaration> | <operator-declaration> |
        <class-declaration> | <operator-symbol-declaration>
      )*
      <program-declaration> .

<program-declaration> ::=
      PROGRAM <identifier> '(' [<identifier>] ')' [<expression>] .

<operator-symbol-declaration> ::=
      (MONADIC | DYADIC) { <operator-symbol> ',' }+ ';' .

<class-declaration> ::=
      CLASS <identifier> <formals>
      BEGIN <subclass-declaration>
            <fetch-associations> <store-associations>
            ( <variable-declaration> | <constant-declaration> |
              <procedure-declaration> | <operator-declaration>
            )*
            [ INIT ':' <block> ]
      END <identifier> ';' .

<subclass-declaration> ::= [ SUBCLASS OF <identifier> ';' ] .

<fetch-associations> ::= [ FETCH <associations> ';' ] .

<store-associations> ::= [ STORE <associations> ';' ] .

<associations> ::= { <association> ',' }+ .

<association> ::= <field-identifier> [ ':' <identifier> ] .

<field-identifier> ::= <identifier> | <operator-symbol> .

<procedure-declaration> ::=
      PROC <identifier> <formals> [ <expression> ] ';' .

<operator-declaration> ::=
      OP <operator-symbol> <formals> [ <expression> ] ';' .

<variable-declaration> ::= VAR { <variable-initialization> ',' }+ ';' .

<variable-initialization> ::= <identifier> [ ':=' <expression> ] .

<constant-declaration> ::= CONST { <constant-initialization> ',' }+ ';' .

<constant-initialization> ::= <identifier> ':=' <constant-expression> .

<formals> ::= '(' { <identifier> ',' }* ')' .

<constant> ::= <string-constant> | <integer-constant> | <real-constant> .
```

```
<constant-operator> ::= '+' | '−' | '*' | '/' | '%' | '||' .

<constant-expression> ::=
        <simple-constant-expression>
        (<constant-operator> <constant-expression>)* .

<simple-constant-expression> ::=
        <constant> | <identifier> | '(' <constant-expression> ')' |
        '−' <simple-constant-expression> .

<expression> ::= <dyadic-expression> .

<monadic-expression> ::= (<monadic-operator>)* <primary> .

<monadic-operator> ::= '~' | '−' | <operator-symbol> .

<dyadic-operator> ::=
        '+' | '−' | '*' | '/' | '%' | '||' | '<' | '<=' | '>' | '>=' |
        '=' | '~=' | ':=' | '&' | '|' | <operator-symbol> .

<dyadic-expression> ::=
        <monadic-expression> (<dyadic-operator> <monadic-expression>)* .

<primary> ::= <unit> ( <subscript> | <select> )* .

<unit> ::=
        <constant> | <identifier-or-call> | <return-expression> |
        <if-expression> | <case-expression> | <while-expression> |
        <for-expression> | <scan-expression> | <try-expression> |
        <assert-expression> | <parenthesized-expression> |
        <array-expression> | <table-expression> .

<identifier-or-call> ::= <identifier> [ <actuals> ] .

<actuals> ::= '(' { <expression> ',' }* ')' .

<subscript> ::= '[' <expression> ']' .

<select> ::= '.' <identifier> [ <actuals> ] .

<return-expression> ::= FRETURN | RETURN [ '(' <expression> ')' ] .

<while-expression> ::= WHILE <test> DO <block> OD .

<try-expression> ::= TRY { <expression> ',' }+ [UNTIL <block>] YRT .

<scan-expression> ::= SCAN <expression> FOR <block> ROF .

<for-expression> ::= FOR <identifier> IN <expression> DO <block> OD .

<case-expression> ::=
        CASE <expression> OF { <case-entry> ',' }* [ DEFAULT ':' <block>] ESAC .

<case-entry> ::= { <constant-expression> ':' }+ <block> .

<assert-expression> ::= ASSERT <expression> .
```

```
<if-expression> ::=
        IF <test> THEN <block>
        ( ELIF <test> THEN <block> )*
        [ELSE <block>] FI .

<test> ::= <expression> [ FAILS | SUCCEEDS ] .

<parenthesized-expression> ::= '(' <block> ')' .

<block> ::=
        (<variable-declaration>|<constant-declaration>)* ([<expression>] ';'}* .

<array-expression> ::=
        ARRAY <size-and-default> [ INIT <array-initialization> ] |
        <array-initialization> .

<size-and-default> ::= '(' <expression> ',' <expression> ')' .

<array-initialization> ::= '[' { <expression> ',' }* ']' .

<table-expression> ::=
        TABLE <size-and-default> [ INIT <table-initialization> ] |
        <table-initialization> .

<table-initialization> ::= '[' { <table-element> ',' }* ']' .

<table-element> ::= {<expression> ':'}+ ':' <expression> .
```

INDEX FOR PART II

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Y

Stellingen behorende bij het proefschrift

From SPRING to SUMMER

Design, Definition and Implementation of Programming Languages for String Manipulation and Pattern Matching

door

Paul Klint

1. Formele taaldefinities vormen een waardevol hulpmiddel bij het ontwerpen van programmeertalen. Het heeft voordelen als dergelijke definities bovendien executeerbaar zijn.

   Hoofdstuk 5 van dit proefschrift.

2. Het verdient aanbeveling om in imperatieve programmeertalen een taalconstructie in te voeren die het mogelijk maakt om de neveneffecten van de uitvoering van bepaalde programmadelen ongedaan te maken. De 'try-expressie' in SUMMER is hiervan een voorbeeld. De door Randell geïntroduceerde 'recovery cache' kan met succes gebruikt worden bij de implementatie van een dergelijke taalconstructie.

   Hoofdstukken 3, 4 en 6 van dit proefschrift.

3. Indien men het 'class' concept gebruikt om polymorfe, polyadische operaties te modelleren, doet zich het, vanuit wiskundig standpunt ongewenste, verschijnsel voor dat de identificatie van dergelijke operaties afhangt van hun eerste argument. Verdergaande toepassing van het class concept wordt hierdoor bemoeilijkt.

4. De methode van Ammann voor herstel van syntactische fouten bij het ontleden van programmateksten functioneert beter naarmate meer syntactische constructies uit de programmeertaal omsloten worden door, te onderscheiden, paren terminale symbolen.

5. Het 'pijp'-mechanisme in het UNIX timesharing systeem biedt de mogelijkheid om de in- en uitvoerstromen van twee of meer processen met elkaar te verbinden maar geeft geen enkele garantie voor een consistente interpretatie van deze stromen door de samenwerkende processen.

6. De toename in rekencapaciteit die het gevolg is van de toepassing van de chiptechnologie dient, althans gedeeltelijk, benut te worden om de interactie met computersystemen te humaniseren.

7. Men zou veel misverstanden kunnen voorkomen door het vakgebied 'Kunstmatige Intelligentie' voortaan met 'Geavanceerde Programmeertechnieken' aan te duiden.

8. In het klassieke 'data flow' model worden conditionele takken van een programma pas geëvalueerd nadat de bijbehorende test geëvalueerd is. In het 'nijvere data flow' model is deze laatste restrictie opgeheven en worden alle beschikbare processors dus maximaal bezet gehouden met (mogelijk overbodige) berekeningen. Het is zinvol om te onderzoeken wat de effectieve snelheidswinst is die met het nijvere data flow model behaald kan worden in vergelijking tot het klassieke geval.

9. De voortdurende verbetering der communicatiemiddelen vermindert geenszins de isolatie van het individu. Het toenemend beroep dat op Telefonische Hulpdiensten gedaan wordt wijst hierop.

10. Bij het onderwijs dient meer aandacht besteed te worden aan het onderricht in het (zowel mondeling als schriftelijk) overdragen van kennis en aan de hulpmiddelen die daarbij ter beschikking staan.

11. Het contact tussen automatiseringsdeskundigen en hun toekomstige slachtoffers behoeft in vele opzichten verbetering.