Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J.A. Bergstra, J. Heering, P. Klint

Algebraic definition of a simple programming language

Department of Computer Science Report CS-R8504 February

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

ALGEBRAIC DEFINITION OF A SIMPLE PROGRAMMING LANGUAGE

J.A. Bergstra
J. Heering
P. Klint
Centre for Mathematics and Computer Science
Amsterdam

ABSTRACT: What are the potentials and limitations of algebraic specifications for defining programming languages and their processors? This paper tries to answer this question by developing a specification for the toy programming language PICO. This specification describes in detail all necessary steps from entering a PICO program in its textual form to computing its value. A major part of this specification is devoted to general techniques for defining programming languages and does not depend on specific properties of PICO. The size of this specification (more than 350 axioms) makes it mandatory to use modularization techniques. In the specification formalism used we have experimented with polymorphism, infix operators, conditional equations, rules for import and export and with parameterization. The results of this experiment and their implications for further research are discussed.

KEY WORDS & PHRASES: Software Engineering, Algebraic Specifications, Formal Definition of Programming Languages, Programming Environments, Modularization Techniques, Specification Languages, Executable Specifications, Prototyping.

1983-84 CR Categories: D.2.1 [**Software Engineering**]: Requirements, Specifications; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages - *Algebraic Approaches to Semantics.*

1980 Mathematics Subject Classification: 68Bxx [**Software**].

NOTE: Partial support received from the European Communities under ESPRIT projects no. 348 (Generation of Interactive Programming Environments) and no. 432 (An Integrated Formal Approach to Industrial Software Development - METEOR).

Report CS-R8504
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

TABLE OF CONTENTS

1. INTRODUCTION

What are the potentials and limitations of algebraic specifications for defining programming languages and their processors? We will try to answer this question by developing a specification for the toy programming language PICO. This specification describes in detail all necessary steps from entering a PICO program in its textual form to computing its value. The specification has been made more general than strictly necessary. A major part of it does not depend on any specific properties of PICO but is equally usable for definitions of other programming languages.

1.1. Motivation

Our motivation for carrying out this -- quite substantial -- exercise is as follows:

(1) It will clarify how well-suited algebraic specifications are for defining (programming) languages and their processors such as type-checkers, interpreters, editors, etc. We do not claim any originality in this respect: many researchers have addressed problems in this area. Our main goal is to gain experience in constructing large algebraic specifications.

(2) It will give us an opportunity for experimenting with various extensions of the algebraic formalism, such as multiple return values, polymorphism, infix operators and (positive) conditional equations. We have only used *total* functions in our specifications.

(3) The size of the specification to be developed makes the use of modularization techniques mandatory. This gives a good opportunity for experimenting with operations for module composition and parameterization.

(4) The experiment will give some insight in the tools that are desirable for the processing of formal specifications (such as check, cross reference, maintenance and edit tools) and in the problems associated with their implementation. As a side-effect some basic tools for the processing of algebraic specifications have been developed.

In this case study, we will *not* consider the specification of errors and exceptions, for two reasons:

(1) We want to concentrate first on the basic functionality and the alternatives for modularization of the system to be designed; specifying error situations would obscure the design and would probably double its size.

(2) Specification of errors within the algebraic framework has not yet been solved satisfactorily and requires separate research.

1.2. Relations with previous research

This paper uses initial algebra semantics for algebraic specifications with conditional equations. We use modularization mechanisms such as parameterization, imports and exports similar or identical to the ones discussed in [KLA83], [W83], [GAU84] or [LOE84]. The specification uses positive conditional equations. The entire PICO system constitutes a semi-computable algebra in the sense of [BT79].

Many people have carried out similar exercises, for instance [GP81], the work of the CIP project in München has been partly devoted to the topic of algebraic specifications of programming languages. Further, several people have worked on the related topic of algebraic compiler specification, for instance Bothe [BO81], Ganzinger [GAN82] and Gaudel [GAU80]. For a survey of algebraic specifications and initial algebra semantics in particular we refer to Goguen and Meseguer [GM82]. A quite complete bibliography on algebraic techniques is [KL83].

1.3. Verification and validation

It is a major problem to get insight in the correctness of a given formal specification. The algebraic specification method provides a relatively simple formalism with unambiguous semantics, but constructing proofs of correctness remains as difficult as ever. We have the following opinion on this matter:

(1) We consider algebraic specifications as the highest level of specification available, i.e. there is no "super high level" specification against which the correctness of the algebraic specification can be proved.

(2) Specifications can only be *validated* against informal requirements (see next subsection).

(3) A proof will be required that some program correctly implements a given algebraic specification. This will involve verification of the translation steps between an algebraic specification and its implementation.

1.4. Potentials for prototyping

Prototyping can be achieved by transforming an algebraic specification into an executable program. Some alternatives are:

(1) Transform algebraic specification into a conditional rewrite system [BK81] by using a Knuth-Bendix-like algorithm [RZ84] or by constraining the allowed forms of equations [HOD82]. In both cases a more efficient prototype implementation can be realized by using a reduction machine.

(2) Compile algebraic specification into Prolog [DE84]. In this case, one can profit from efficient Prolog implementations [BBC83] and (concurrent) Prolog machines.

1.5. Conclusions

Our conclusions can be summarized as follows:

(1) The specifications as presented in the body of this paper are in our opinion satisfactory. The techniques developed for specifying various aspects of our toy programming language can also be used in the specifications of other -- more realistic -- languages. We expect that a *programming environment* for a given programming language can also be specified within the framework as presented here.

(2) Polymorphism was found to be convenient -- though not indispensable -- for shortening the specifications and making them more readable. Conditional equations were essential for the modeling of partial functions. They also tended to shorten several parts of the specification. The primitive abbreviation scheme used for introducing infix operators was unsatisfactory. The way in which we have to treat integer and string constants is also clumsy. It will be essential to have an elaborate mechanism for introducing arbitrary syntactic extensions and even graphical notations.

(3) The algebraic specification techniques have been of considerable heuristic value in understanding how the specification should (could) be modularized. However, the various modularization techniques (such as import and parameterization) are not orthogonal. It will be important to develop sound heuristics about which technique is to be used where.

(4) Structure diagrams (a high-level graphical notation described in section 2.4) are a considerable aid in finding the proper modularization of a specification.

(5) In view of the size of the specification it was necessary to implement some simple tools for consistency checking. We have implemented a checker for the syntax and type correctness of specifications and generators for structure diagrams and cross reference tables. For the development of larger specifications it will be necessary to have more sophisticated editing facilities, such as a syntax-directed editor with incremental type checking. The question will have to be addressed how arbitrary syntactic extensions and graphical notations can be handled by such an editor.

1.6. Perspectives for further research

During this exercise we have identified the following areas that need further clarification:

(1) Treatment of errors and exceptions.

(2) Multiple export signatures per module.

(3) More flexible export rules with which the number of exported names can be minimized.

(4) Parameterization of modules and formulation of constraints on parameters.

(5) More explicit specification of inherited parameters.

(6) Heuristic rules for proper modularization.

(7) Further development of structure diagrams.

(8) Mechanisms for introducing syntactic extensions.

(9) Techniques and tools for creating, modifying, maintaining and incremental checking of algebraic specifications.

(10) Techniques and tools for transforming algebraic specifications into executable prototypes.

1.7. Acknowledgements

Niek van Diepen, Ed Kuijpers and Ard Verhoog made useful comments on various drafts of this document.

2. THE SPECIFICATION FORMALISM

In this section we give a brief and informal description of the specification formalism. The formalism is based on *signatures* consisting of a set of *sorts* and a set of *functions* over these sorts. A signature combined with a set of equations over that signature and a set of variables occurring in the equations forms a specification (see, for instance, [KLA83]). We will always use the initial algebra semantics of these specifications.

2.1. Syntax of the specification formalism

```
<specification> ::= <module>+ .
<module>        ::= 'module'  <ident>
                    'begin'
                        <parameters>
                        <exports>
                        <imports>
                        <sorts>
                        <functions>
                        <variables>
                        <equations>
                    'end'  <ident> .
<parameters>    ::= [ 'parameters' {<parameter-module>  ','}+ ].
<parameter-module> ::=
                <ident> ['begin'
                                <sorts>
                                <functions>
                            'end' <ident>  ] .
<exports>       ::= [ 'exports' 'begin'
                        <sorts> <functions> 'end'] .
<imports>       ::= [ 'imports' { <module-expression>  ','}+ ] .
<module-expression>
                ::= <ident>
                    ['{'
                        [ 'renamed' 'by' <renames>]
                        ( <ident> 'bound' 'by'  <renames>
                         'to'  <ident> )*
                     '}'] .
<renames>       ::= '[' { <rename>  ',' }* ']' .
<rename>        ::= <fun-ident> [ '->' <fun-ident> ].
<sorts>         ::= [ 'sorts' <ident-list> ] .
<ident-list>    ::= {  <ident>  ',' }+.
<fun-ident-list>::= { <fun-ident>  ',' }+.
<functions>     ::= [ 'functions' <function-list> ] .
<function-list> ::= ( <fun-ident-list>  ':' <fun-type> )+.
<fun-ident>     ::= <ident>  | '_' <operator>  '_' | <operator>  '_' .
<fun-type>      ::= [ <type> ] '->' <out-type> .
<type>          ::= { <type-ident>  '#' }+ .
<out-type>      ::= <type-ident>  | '('  <type> ')' .
<type-ident>    ::= <ident>  | ('*' )+ .
<variables>     ::= [ 'variables' <variable-list> ] .
```

```
<variable-list> ::= (  <ident-list> ':' '->' <out-type>  )+.
<equations>     ::= [ 'equations' <cond-equation>+ ] .
<cond-equation> ::= <tag> <equation> [ 'when' <equation-list>].
<tag>           ::= ['[' <ident> ']' ].
<equation-list> ::= { <equation>  ',' }+.
<equation>      ::= <term> '=' <term>.
<term>          ::= <operator>  <term>  |
                    <primary> [<operator>  <term> ] .
<primary>       ::=  <ident> ['('  <term-list> ')']  |
                    <tuple>  | <string>  | '(' <term>  ')' .
<term-list>     ::= {  <term>  ',' }+.
<tuple>         ::= '<'  <term-list> '>' .
```

2.2. Lexical conventions

The lexical conventions of the specification language are as follows:

1) Identifiers (i.e. `<ident>` in the grammar in the previous section) consist of a non-empty sequence of letters and/or digits with embedded hyphens. For example, `a`, `Z16`, `Very-Long-Identifier` and `6` are legal identifiers, but `-a`, `-` or `a-` are illegal.
2) Strings (i.e. `<string>`) begin and end with a single quote (') and may contain letters, digits and the punctuation marks: (space) " (double quote) ( ) * + , - . / : ; | = .
3) Operators (i.e. `<operator>`) are denoted by a sequences of one or more of the following characters: ! @, $, %, ^, &, + ', |, \, ;, ', . ? / .
4) Comments begin with two hyphens and end with either the end of the line or another pair of hyphens.

2.3. Various aspects of the specification formalism

Our formalism extends the basic algebraic specification formalism based on signatures and sets of equations in several ways. These extensions are discussed in the following subsections.

2.3.1. Prefix and infix operators

Monadic or dyadic functions may be denoted by respectively prefix or infix operators. Operators are denoted by operator-symbols consisting of one or more of the characters specified in the previous paragraph. In the signature, the position of operands of operators is indicated by the underline character (_). For instance,

```
_ + _ : S1 # S2 -> S3
```

defines the infix operator `+` with argument sorts `S1` and `S2` and output sort `S3`. All infix and prefix operators have the same priority. They are just an abbreviation device and can always be replaced by ordinary functions.

2.3.2. Multiple output values

In the signature tuples are allowed as output sorts, i.e. the function

```
f : S1 # S2 -> (S3 # S4)
```

has `S3 # S4` as output sort, this is an ordered sequence with first component of sort `S3` and second component of sort `S4`. In equations, tuples are written as a sequence of terms enclosed by angle brackets, i.e. `<` and `>`. It is required that the sorts of the constituents of a tuple are equal to the corresponding components of a tupled output sort in the signature. Tuples can be removed from the specification by introducing new sorts and construction/projection functions for each tupled output sort in the signature. The above tupled output sort `(S3 # S4)` can, for instance, be removed by introducing the additional sort `S5` and the functions `make-S5`, `first-S5`, `second-S5`, as follows:

```
f         : S1 # S2 -> S5
make-S5   : S3 # S4 -> S5
first-S5  : S5      -> S3
second-S5 : S5      -> S4
```

2.3.3. Polymorphism

Functions may be polymorphic, i.e. the same function name may be used to denote different functions with different types, e.g. after defining

```
f : S1 # S2 -> S3
f : S2      -> S2
```

each occurrence of the function symbol `f` in a term will have to be disambiguated by considering the number and sorts of its arguments.

Definitions of functions may also contain *wild card* sorts, denoted by one or more asterisk characters (*). At the position of a wild card sort, a term of any legal sort is allowed. Wild card sorts are identified by the number of asterisks by which they are denoted. In this way, one can specify the multiple occurrence of the same, but arbitrary, sort. For instance,

```
g : * # S3 # ** # * -> *
```

specifies a function `g` with first and fourth argument of equal, but arbitrary sort, second argument of sort `S3` and third argument of another arbitrary sort which may differ from the sort of the first and fourth argument. The output sort of `g` is the same as the sort of the first and fourth argument.

We impose some restrictions on polymorphic types which allow us to eliminate all polymorphism from the specification by means of simple textual transformations. It is required that all wild card sorts appearing in the output sorts of a function also appear among its input sorts. This restriction excludes, for instance, polymorphic constants. We also impose the restriction that the sets of input sorts of polymorphic functions are pairwise disjoint. This excludes, for instance,

```
f : * # S2 -> S3
f : S1 # *  -> S3
```

since there is a unifying type `S1 # S2 -> S3` in this case.

2.3.4. Module expressions

Module expressions serve the purpose to rename sorts and functions of an existing module or to bind parameters of a module to actual values. The module described by the module expression may then be imported by another module. These three aspects of module expressions are now described in more detail:

- **Exported names**: Each module may contain an `exports` clause giving a list of all names of sorts and functions which are exported from the module, i.e. which remain visible when the module is combined with other modules (see below). External names of a module can be renamed by means of the `renamed by` construct. Currently, all exported names are inherited, i.e. they are also exported by the modules that (directly or indirectly) use the module from which the names were originally exported. This simple scheme has the undesirable property that the number of exported names cannot be controlled. In future versions of the specification formalism, a better mechanism offering more refined control over exported names will be introduced.
- **Parameterization**: In order to make modules more generally usable in different contexts, a form of parameterization is available in the specification language. Parameterization is described by adding one or more `parameters` clauses to a module. Each (formal) parameter is a (possibly incomplete) submodule and contains one or more names of sorts and functions. All these names are formal names which -- in a later stage -- have to be bound to actual ones. This is achieved by the `bound by` construct. Not all parameters of a module have to be bound before it can be imported in another module. Such unbound parameters are *inherited* by the importing module and are indistinguishable from parameters that are specified in the importing module itself.
- **Import of modules**: Import of a module in another module is the fundamental composition operation for modules. It is described by the `imports` clause. The import of module `B` in module `A` is equivalent to constructing a new module `A'` that consists of the unions of the signatures and equations of `A` and `B`. Note that only the exported names of `B` are used for the construction of this union. In the specifications that follow we will -- for reasons of clarity -- frequently import more modules than is strictly necessary.
- **Name identification**: When modules are combined the problem arises how multiple declarations of names should be interpreted. For identification of names we therefore adopt the *origin principle*:

  1) names with identical spelling and type, originating from the same module are equal,

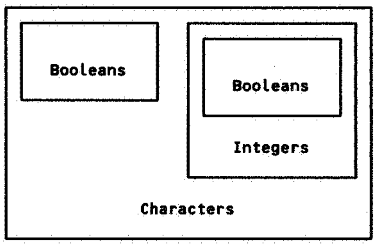
  2) names with identical spelling and type but different origin are forbidden.

  This scheme allows the multiple inclusion of the same module (via different routes), but forbids collisions of names with identical spelling and type, originating from different modules.

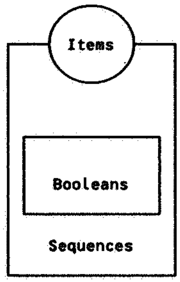
2.4. Structure diagrams

The overall modular structure of specifications will be illustrated by *structure diagrams*. Each module is represented by a rectangular box. The name of each module is shown at the bottom of its box. For example, module `Booleans` does not import any other modules and is represented by:

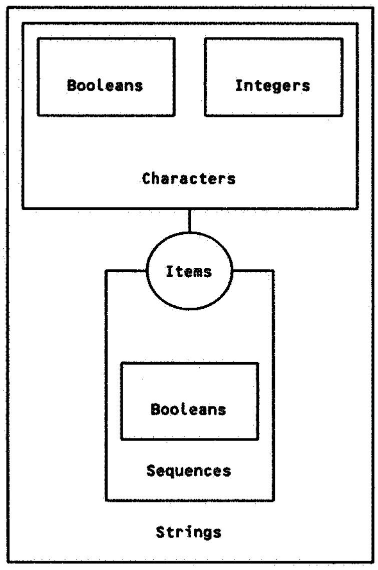
All modules imported by a module M are represented by structure diagrams inside the box representing M. For nested structure diagrams levels of detail may be suppressed to gain space. For example, `Characters` imports `Booleans` and `Integers` (which in its turn also imports `Booleans`) and is represented by:



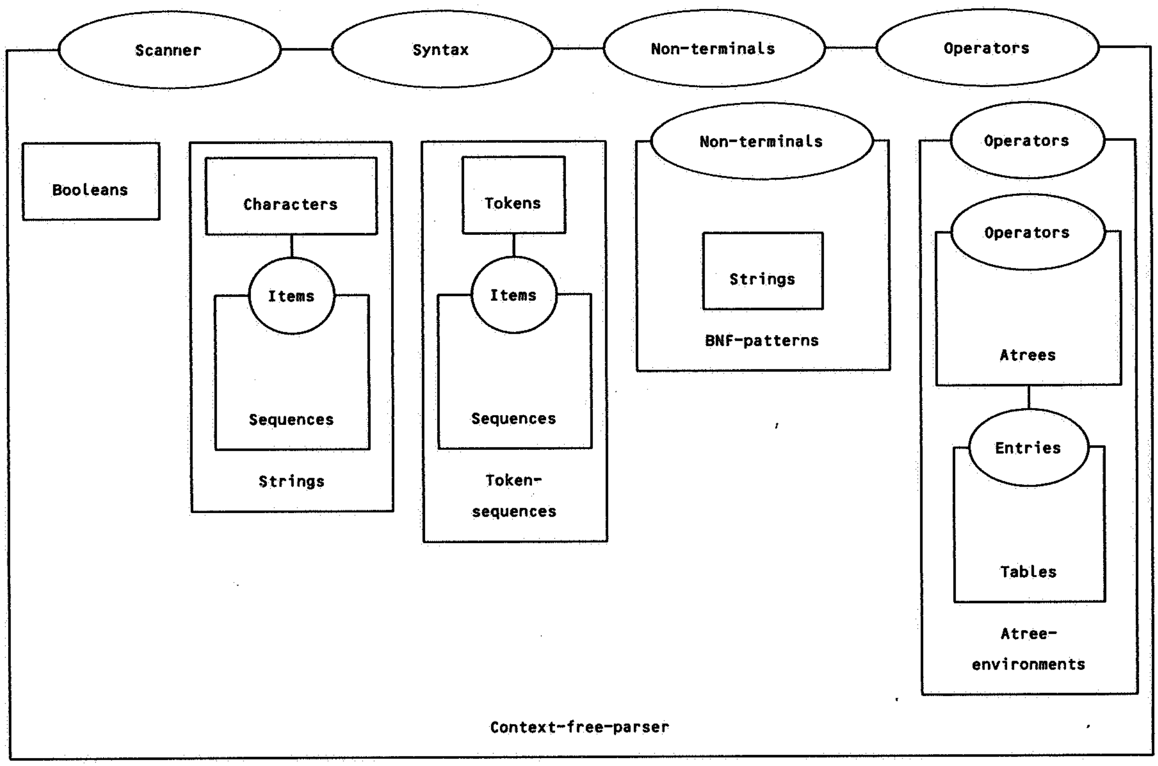
All parameters of a module are represented by ellipses carrying the name of the parameter. For example, `Sequences`, which has parameter `Items` and imports `Booleans`, is represented by:



The binding of a formal parameter is represented by a line joining the formal parameter and the module to which it is bound. For example, `Strings` are defined by binding the parameter `Items` of `Sequences` to `Characters`. The corresponding structure diagram is:



Unbound, inherited parameters are -- not yet very satisfactorily -- represented in structure diagrams by repeating the inherited parameter as a parameter of the module that inherits it. For example, `Context-free-parser` has formal parameters `Scanner` and `Syntax` and imports, among others, `BNF-patterns` with unbound parameter `Non-terminals` and `Atree-environments` with unbound parameter `Operators`. This is represented by the following diagram:



All structure diagrams appearing in this paper have been generated automatically; they were derived from the *text* of the specification.

3. INFORMAL DEFINITION OF THE LANGUAGE PICO

The language PICO is extremely simple. It is essentially the language of while-programs. A program consists of declarations followed by statements. All variables occurring in the statements have to be declared to be either of type integer or of type string. Statements may be assignment statements, if-statements and while-statements. Expressions may be a single identifier, integer addition or string concatenation.

At the lexical level, PICO programs consist of a sequence of lexical items separated by layout. Lexical items are keywords, identifiers, integer and string constants and punctuation marks. The lexical grammar for PICO is:

```
<lexical-stream>      ::= <lexical-item> <lexical-stream> |
                          <lexical-item> .
<lexical-item>        ::= <optional-layout>
                          (<keyword-or-id> |
                           <integer-constant> |
                           <string-constant>  |
                           <literal>) .
<optional-layout>     ::= <layout> | <empty>.
<keyword-or-id>       ::= 'begin' | 'end' | 'declare' | 'integer' |
                          'string' | 'if' | 'then' | 'else' | 'fi' |
                          'while' | 'do' | 'od' |
                          <id> .
<id>                  ::= <letter> <id-chars> .
<id-chars>            ::= <id-char> <id-chars> | <empty> .
<id-char>             ::= <letter> | <digit> .

<integer-constant>    ::= <digit> <digits>
<digits>              ::= <digit> <digits> | <empty> .

<string-constant>     ::= <quote> <string-tail> .
<string-tail>         ::= <any-char-but-quote> <string-tail> | <quote> .

<quote>               ::= '"' .

<any-char-but-quote> ::= <letter> | <digit> | <literal> | <layout> .
<literal>             ::= '(' | ')' | '+' | '-' | ';' | ',' |
                          '||' | ':' | ':=' .
<letter>              ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' |
                          'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' |
                          'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' |
                          'v' | 'w' | 'x' | 'y' | 'z' |
                          'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' |
                          'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' |
                          'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' |
                          'V' | 'W' | 'X' | 'Y' | 'Z' .
<digit>               ::= '0' | '1' | '2' | '3' | '4' |
                          '5' | '6' | '7' | '8' | '9' .
<layout>              ::= ' '  | <newline> | <tab> .
```

Here, <newline> and <tab> are assumed to be primitive notions corresponding to, respectively, the newline character and the tabulation.

The concrete syntax of PICO is:

```
<pico-program>  ::= 'begin' <decls> <series> 'end'  .
<decls>         ::= 'declare' <id-type-list> ';' .
<id-type-list>  ::= <id> ':' <type> (<empty> | ',' <id-type-list>)
<type>          ::= 'integer' | 'string' .
<series>        ::= <empty> | <stat> (<empty> | ';' <series>)
<stat>          ::=  <assign> | <if> | <while> .
<assign>        ::= <id> ':=' <exp> .
<if>            ::= 'if' <exp> 'then' <series>
                               'else' <series> 'fi'
<while>         ::= 'while' <exp> 'do' <series> 'od' .
<exp>           ::= <id> | <integer-constant> | <string-constant> |
                    <plus> | <conc> | '(' <exp> ')' .
<plus>          ::= <exp> '+' <exp> .
<conc>          ::= <exp> '||' <exp> .
<empty>         ::= '' .
```

The non-terminals <id>, <integer-constant> and <string-constant> are defined in the lexical grammar given above and represent identifiers, integer constants and string constants respectively.

There are two overall static semantic constraints on programs:

1) All identifiers occurring in a program should have been declared and their use should be compatible with their declaration. More precisely, this means that all <id>s occurring in an <assign> or an <exp> should have been declared, i.e. should occur in some <id-type> in the <id-type-list> of the <decls>-part of the PICO-program, and that the type of <id>s should be compatible with the expressions in which they occur.

2) The <exp> occurring in an <if>- or <while>-statement should be of type integer.

A type can be given to <exp>s depending on their syntactic form:

- if an <exp> consists of an <id>, that <id> should have been declared and the type of the <exp> is the same as the type of the <id> in its declaration;
- an <exp> consisting of an <integer-constant> has type integer;
- an <exp> consisting of a <string-constant> has type string;
- an <exp> consisting of a <plus> has type integer;
- an <exp> consisting of a <conc> has type string.

Given this notion of types of <exp>s, the static semantic constraints can be formulated in more detail:

- The <exp>s occurring in a <plus> should be of type integer;
- The <exp>s occurring in a <conc> should be of type string;
- The <id> and <exp> that occur in an <assign> should have the same type.
- The <exp>s that occur in <if> and <while> should have type integer.

The dynamic semantics of PICO are straightforward except that

1) integer variables are initialized with value 0,
2) string variables are initialized with "" (empty string),
3) the `<exp>` in an `<if>` or `<while>` is assumed to be true if its value is unequal to 0.

4. ELEMENTARY DATA TYPES

As a prerequisite for the PICO specification some elementary data types are defined in this chapter, specifications are given for:

- **Booleans** (4.1): truth values `true` and `false` with functions `and`, `or`, `not` and the polymorphic function `if`.
- **Integers** (4.2): natural numbers with constants `0`, `1` and `10` and functions `succ` (successor), `add` (addition), `mul` (multiplication), `eq` (equality of integers), `less` (less than), `lesseq` (less than or equal), `greater` (greater than) and `greatereq` (greater than or equal).
- **Characters** (4.3): the alphabet consists of constants for letters, digits, and punctuation marks. The functions `eq` (equality of characters), `ord` (ordinal number of character in the alphabet), `is-letter` (is character a letter?), `is-upper` (is character an upper case letter?), `is-lower` (is character a lower case letter?) and `is-digit` (is character a digit?) are defined on them.
- **Sequences** (4.4): linear lists of items. Sequences are parameterized with the data type of the items. The only constant is `null`, the empty sequence. The following functions are defined: `eq` (equality of sequences), `seq` (combine item with sequence), `conc` (concatenate two sequences) and `conv-to-seq` (convert an item to a sequence).
- **Strings** (4.5): sequences of characters, i.e. sequences with items bound by characters. The only constant is `null-string`, the empty string. The following functions are defined: `eq` (equality of strings), `seq` (combine character with a string), `conc` (concatenate two strings), `string` (convert a character to a string) and `str-to-int` (convert a string to an integer).
- **Tables** (4.6): mapping from strings to entries, where entries are a parameter. The only constant is `null-table`, the empty table. The following functions are defined: `table` (add new entry to table), `lookup` (searches for an entry in a table), `delete` (deletes an entry from a table) and `eq` (equality of tables).

4.1. Booleans

4.1.a. Gobal description

Booleans are truth values `true` and `false` with functions `and`, `or`, `not` and the polymorphic function `if` (see section 2 for a discussion of polymorphism).

Apart from the `if`-function, this is the simplest initial algebra specification of the Booleans. It contains only closed equations. Note that, for instance, the equation

```
not(not(x)) = x
```

is not derivable by equational logic from the axioms given, although it is valid in the initial model. Adding this equation to `Booleans`, does not affect the initial model, but only causes an increase in the power of the specification in the sense that more of the (open) equations valid in the initial model can be derived from the specification by equational logic. See [HEE85] for a discussion of this subject.

4.1.b. Structure diagram

4.1.c. Specification

```
module Booleans
begin
   exports
        begin
           sorts        BOOL
           functions
                true    :                      -> BOOL
                false   :                      -> BOOL
                or      : BOOL # BOOL          -> BOOL
                and     : BOOL # BOOL          -> BOOL
                not     : BOOL                 -> BOOL
                if      : BOOL # * # *         -> *
        end

   variables
        x, y    : -> *

   equations

   [1]   or(true, true)        = true
   [2]   or(true, false)       = true
   [3]   or(false, true)       = true
   [4]   or(false, false)      = false
```

```
  [5]   and(true, true)        = true
  [6]   and(true, false)       = false
  [7]   and(false, true)       = false
  [8]   and(false, false)      = false

  [9]   not(true)              = false
  [10]  not(false)             = true

  [11]  if(true, x, y)         = x
  [12]  if(false, x, y)        = y

end Booleans
```

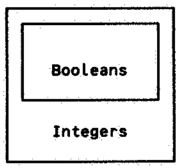
4.2. Integers

4.2.a. Global description

Integers as defined here are in fact natural numbers with constants `0`, `1` and `10` and functions `succ` (successor), `add` (addition), `mul` (multiplication), `eq` (equality), `less` (less than), `lesseq` (less than or equal), `greater` (greater than) and `greatereq` (greater than or equal).

The equations for the constants `1` and `10` are not very satisfactory. Clearly, a mechanism is needed for defining a shorthand notation for *all* integer constants. In section 4.5.a this subject is discussed in connection with string constants.

4.2.b. Structure diagram



4.2.c. Specification

```
module Integers
begin

   exports
       begin
          sorts        INTEGER

          functions
              0         :                        -> INTEGER
              1         :                        -> INTEGER
              10        :                        -> INTEGER
              succ      : INTEGER                -> INTEGER
              add       : INTEGER # INTEGER      -> INTEGER
              mul       : INTEGER # INTEGER      -> INTEGER
              eq        : INTEGER # INTEGER      -> BOOL
              less      : INTEGER # INTEGER      -> BOOL
              lesseq    : INTEGER # INTEGER      -> BOOL
              greater   : INTEGER # INTEGER      -> BOOL
              greatereq: INTEGER # INTEGER       -> BOOL
       end

   imports Booleans

   variables
       x, y, z  : -> INTEGER

   equations
```

```
[13]   1                       = succ(0)
[14]   10                      = succ(succ(succ(succ(succ(succ(
                                     succ(succ(succ(succ(0))))))))))

[15]   add(x, 0)               = x
[16]   add(x, succ(y))         = succ(add(x, y))

[17]   mul(x, 0)               = 0
[18]   mul(x, succ(y))         = add(x, mul(x, y))

[19]   eq(x, x)                = true
[20]   eq(x, y)                = eq(y, x)
[21]   eq(succ(x), succ(y))    = eq(x, y)
[22]   eq(0, succ(x))          = false

[23]   less(x, 0)              = false
[24]   less(0, succ(x))        = true
[25]   less(succ(x), succ(y))= less(x, y)

[26]   lesseq(x, y)            = or(less(x, y), eq(x, y))

[27]   greater(x, y)           = not(lesseq(x, y))

[28]   greatereq(x, y)         = or(greater(x, y), eq(x, y))

end Integers
```

4.3. Characters

4.3.a. Global description

The alphabet of characters consists of constants for letters, digits, and punctuation marks. The functions `eq` (equality), `ord` (ordinal number of character in the alphabet), `is-letter` (is character a letter?), `is-upper` (is character an upper case letter?), `is-lower` (is character a lower case letter?) and `is-digit` (is character a digit?) are defined on them.

Two observations can be made about this specification. First, one may notice that the absence of integer constants forces us two write equations of the form

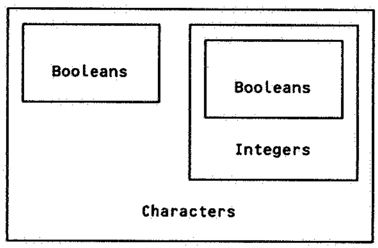
```
ord(char-3)       = succ(ord(char-2))
```

instead of the more natural form

```
ord(char-3)       = 3.
```

Secondly, it is clear that some abbreviation mechanism is needed for specifications that contain many constants as is the case here. At the expense of additional complexity of the specification, this could have been achieved by defining characters in two stages: first, a basic alphabet is defined which consists only of lower case letters and a hyphen; next, this basic alphabet is used to generate all constants for the full alphabet. Names of constants are then only allowed to contain symbols from the basic alphabet, i.e. `char-upper-case-a` instead of `char-A`.

4.3.b. Structure diagram



4.3.c. Specification

```
module Characters
begin

   exports
       begin
          sorts          CHAR
          functions
               eq                 : CHAR # CHAR    -> BOOL
               is-upper           : CHAR           -> BOOL
               is-lower           : CHAR           -> BOOL
               is-letter          : CHAR           -> BOOL
```

```
is-digit        : CHAR      -> BOOL
ord             : CHAR      -> INTEGER

char-0          :           -> CHAR
char-1          :           -> CHAR
char-2          :           -> CHAR
char-3          :           -> CHAR
char-4          :           -> CHAR
char-5          :           -> CHAR
char-6          :           -> CHAR
char-7          :           -> CHAR
char-8          :           -> CHAR
char-9          :           -> CHAR

char-ht         :           -> CHAR        -- tab --
char-nl         :           -> CHAR        -- new line --
char-space      :           -> CHAR        -- space --
char-quote      :           -> CHAR        -- " --
char-lpar       :           -> CHAR        -- ( --
char-rpar       :           -> CHAR        -- ) --
char-times      :           -> CHAR        -- * --
char-plus       :           -> CHAR        -- + --
char-comma      :           -> CHAR        -- , --
char-minus      :           -> CHAR        -- - --
char-point      :           -> CHAR        -- . --
char-slash      :           -> CHAR        -- / --
char-bar        :           -> CHAR        -- | --
char-equal      :           -> CHAR        -- = --
char-colon      :           -> CHAR        -- : --
char-semi       :           -> CHAR        -- ; --

char-A          :           -> CHAR
char-B          :           -> CHAR
char-C          :           -> CHAR
char-D          :           -> CHAR
char-E          :           -> CHAR
char-F          :           -> CHAR
char-G          :           -> CHAR
char-H          :           -> CHAR
char-I          :           -> CHAR
char-J          :           -> CHAR
char-K          :           -> CHAR
char-L          :           -> CHAR
char-M          :           -> CHAR
char-N          :           -> CHAR
char-O          :           -> CHAR
char-P          :           -> CHAR
char-Q          :           -> CHAR
char-R          :           -> CHAR
char-S          :           -> CHAR
```

```
            char-T        :              -> CHAR
            char-U        :              -> CHAR
            char-V        :              -> CHAR
            char-W        :              -> CHAR
            char-X        :              -> CHAR
            char-Y        :              -> CHAR
            char-Z        :              -> CHAR

            char-a        :              -> CHAR
            char-b        :              -> CHAR
            char-c        :              -> CHAR
            char-d        :              -> CHAR
            char-e        :              -> CHAR
            char-f        :              -> CHAR
            char-g        :              -> CHAR
            char-h        :              -> CHAR
            char-i        :              -> CHAR
            char-j        :              -> CHAR
            char-k        :              -> CHAR
            char-l        :              -> CHAR
            char-m        :              -> CHAR
            char-n        :              -> CHAR
            char-o        :              -> CHAR
            char-p        :              -> CHAR
            char-q        :              -> CHAR
            char-r        :              -> CHAR
            char-s        :              -> CHAR
            char-t        :              -> CHAR
            char-u        :              -> CHAR
            char-v        :              -> CHAR
            char-w        :              -> CHAR
            char-x        :              -> CHAR
            char-y        :              -> CHAR
            char-z        :              -> CHAR

    end

imports Booleans, Integers

variables
     c, c1, c2      :              -> CHAR

equations

[29]   ord(char-0)         = 0
[30]   ord(char-1)         = succ(ord(char-0))
[31]   ord(char-2)         = succ(ord(char-1))
[32]   ord(char-3)         = succ(ord(char-2))
[33]   ord(char-4)         = succ(ord(char-3))
[34]   ord(char-5)         = succ(ord(char-4))
```

```
[35]  ord(char-6)          = succ(ord(char-5))
[36]  ord(char-7)          = succ(ord(char-6))
[37]  ord(char-8)          = succ(ord(char-7))
[38]  ord(char-9)          = succ(ord(char-8))

[39]  ord(char-ht)         = succ(ord(char-9))
[40]  ord(char-nl)         = succ(ord(char-ht))
[41]  ord(char-space)      = succ(ord(char-nl))
[42]  ord(char-quote)      = succ(ord(char-space))
[43]  ord(char-lpar)       = succ(ord(char-quote))
[44]  ord(char-rpar)       = succ(ord(char-lpar))
[45]  ord(char-times)      = succ(ord(char-rpar))
[46]  ord(char-plus)       = succ(ord(char-times))
[47]  ord(char-comma)      = succ(ord(char-plus))
[48]  ord(char-minus)      = succ(ord(char-comma))
[49]  ord(char-point)      = succ(ord(char-minus))
[50]  ord(char-slash)      = succ(ord(char-point))
[51]  ord(char-bar)        = succ(ord(char-slash))
[52]  ord(char-equal)      = succ(ord(char-bar))
[53]  ord(char-colon)      = succ(ord(char-equal))
[54]  ord(char-semi)       = succ(ord(char-colon))

[55]  ord(char-A)          = succ(ord(char-semi))
[56]  ord(char-B)          = succ(ord(char-A))
[57]  ord(char-C)          = succ(ord(char-B))
[58]  ord(char-D)          = succ(ord(char-C))
[59]  ord(char-E)          = succ(ord(char-D))
[60]  ord(char-F)          = succ(ord(char-E))
[61]  ord(char-G)          = succ(ord(char-F))
[62]  ord(char-H)          = succ(ord(char-G))
[63]  ord(char-I)          = succ(ord(char-H))
[64]  ord(char-J)          = succ(ord(char-I))
[65]  ord(char-K)          = succ(ord(char-J))
[66]  ord(char-L)          = succ(ord(char-K))
[67]  ord(char-M)          = succ(ord(char-L))
[68]  ord(char-N)          = succ(ord(char-M))
[69]  ord(char-O)          = succ(ord(char-N))
[70]  ord(char-P)          = succ(ord(char-O))
[71]  ord(char-Q)          = succ(ord(char-P))
[72]  ord(char-R)          = succ(ord(char-Q))
[73]  ord(char-S)          = succ(ord(char-R))
[74]  ord(char-T)          = succ(ord(char-S))
[75]  ord(char-U)          = succ(ord(char-T))
[76]  ord(char-V)          = succ(ord(char-U))
[77]  ord(char-W)          = succ(ord(char-V))
[78]  ord(char-X)          = succ(ord(char-W))
[79]  ord(char-Y)          = succ(ord(char-X))
[80]  ord(char-Z)          = succ(ord(char-Y))

[81]  ord(char-a)          = succ(ord(char-Z))
```

```
[82]   ord(char-b)          = succ(ord(char-a))
[83]   ord(char-c)          = succ(ord(char-b))
[84]   ord(char-d)          = succ(ord(char-c))
[85]   ord(char-e)          = succ(ord(char-d))
[86]   ord(char-f)          = succ(ord(char-e))
[87]   ord(char-g)          = succ(ord(char-f))
[88]   ord(char-h)          = succ(ord(char-g))
[89]   ord(char-i)          = succ(ord(char-h))
[90]   ord(char-j)          = succ(ord(char-i))
[91]   ord(char-k)          = succ(ord(char-j))
[92]   ord(char-l)          = succ(ord(char-k))
[93]   ord(char-m)          = succ(ord(char-l))
[94]   ord(char-n)          = succ(ord(char-m))
[95]   ord(char-o)          = succ(ord(char-n))
[96]   ord(char-p)          = succ(ord(char-o))
[97]   ord(char-q)          = succ(ord(char-p))
[98]   ord(char-r)          = succ(ord(char-q))
[99]  ord(char-s)           = succ(ord(char-r))
[100]  ord(char-t)          = succ(ord(char-s))
[101]  ord(char-u)          = succ(ord(char-t))
[102]  ord(char-v)          = succ(ord(char-u))
[103]  ord(char-w)          = succ(ord(char-v))
[104]  ord(char-x)          = succ(ord(char-w))
[105]  ord(char-y)          = succ(ord(char-x))
[106]  ord(char-z)          = succ(ord(char-y))

[107]  eq(c1, c2)           = eq(ord(c1), ord(c2))
[108]  is-upper(c)          = and(greatereq(ord(c), ord(char-A)),
                                  lesseq(ord(c), ord(char-Z)))
[109]  is-lower(c)          = and(greatereq(ord(c), ord(char-a)),
                                  lesseq(ord(c), ord(char-z)))
[110]  is-digit(c)          = and(greatereq(ord(c), ord(char-0)),
                                  lesseq(ord(c), ord(char-9)))
[111]  is-letter(c)         = or(is-upper(c), is-lower(c))

end Characters
```

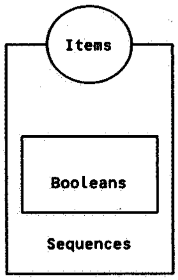
4.4. Sequences

4.4.a. Global description

Sequences are linear lists of items; they are parameterized with the data type of the items. The only constant is `null`, the empty sequence. The following functions are defined: `eq` (equality), `seq` (combine item with sequence), `conc` (concatenate two sequences) and `conv-to-seq` (convert an item to a sequence).

Note that the function `eq` in the above specification is polymorphic.

4.4.b. Structure diagram



4.4.c. Specification

```
module Sequences
begin

   parameters Items
        begin
           sorts        ITEM
           functions
                eq : ITEM # ITEM -> BOOL

        end Items

   exports
        begin
           sorts        SEQ
           functions
                null          :                -> SEQ
                seq           : ITEM # SEQ     -> SEQ
                conc          : SEQ # SEQ      -> SEQ
                eq            : SEQ # SEQ      -> BOOL
                conv-to-seq   : ITEM           -> SEQ
        end

   imports Booleans
```

```
   variables
        s, s1, s2          : -> SEQ
        it, it1, it2       : -> ITEM

   equations

   [112]  conc(s, null)                 = s
   [113]  conc(null, s)                 = s
   [114]  conc(seq(it, s1), s2)         = seq(it, conc(s1, s2))

   [115]  eq(s, s)                      = true
   [116]  eq(s1, s2)                    = eq(s2, s1)
   [117]  eq(null, seq(it, s))          = false
   [118]  eq(seq(it1,s1), seq(it2,s2))  = and(eq(it1,it2), eq(s1,s2))

   [119]  conv-to-seq(it)               = seq(it, null)

end Sequences
```

4.5. Strings

4.5.a. Global description

Strings are sequences of characters, i.e. `Sequences` with `Items` bound to `Characters`. The only constant is `null-string`, the empty string. The following functions are defined: `eq` (equality), `seq` (combine character with a string), `conc` (concatenate two strings), `string` (convert a character to a string) and `str-to-int` (convert a string to an integer).

In the case of the data type string there is an urgent need for a short hand notation for string constants. The PICO specification would become unreadable without it. We will therefore use an, *ad hoc*, convenient notation for string constants to denote the terms generated by the module `Strings`, e.g. the term

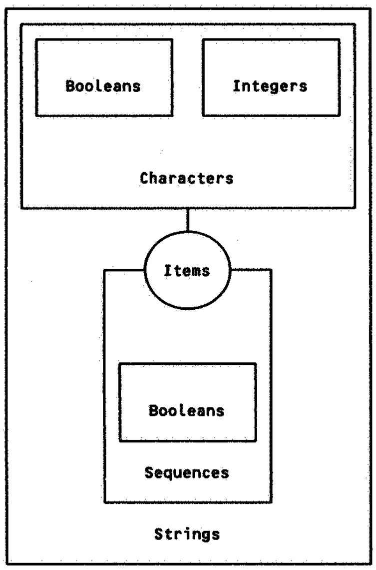
```
seq(char-a, seq(char-b, null-string))
```

will be written as

```
"ab".
```

The empty string, i.e. the constant `null-string`, will be written as "". In the future, a *general* abbreviation scheme will be indispensable for obtaining readable specifications containing integer and string constants, sets, lists, etc.

4.5.b. Structure diagram



4.5.c. Specification

```
module Strings
begin

   exports
      begin
         functions
```

```
        str-to-int : STRING -> INTEGER
    end

  imports Sequences
             { renamed by
                    [ SEQ -> STRING,
                      null -> null-string,
                      conv-to-seq -> string]
               Items bound by
                    [ ITEM -> CHAR,
                      eq -> eq]
                    to Characters
             }

  variables
      c        :-> CHAR
      str      :-> STRING

  equations

  [120] str-to-int(seq(c, str)) = if(eq(str, null-string),
                                     ord(c),
                                     add(mul(ord(c), 10), str-to-int(str)))
  [121] str-to-int(null-string) = 0

end Strings
```

4.6. Tables

4.6.a. Global description

Tables are mappings from strings to entries, where entries are a parameter. The only constant is `null-table`, the empty table. The following functions are defined: `table` (add new entry to table), `lookup` (searches for an entry in a table), `delete` (deletes an entry from a table) and `eq` (equality of tables).

Note that adding a pair (`name, error-entry`) to a table has the somewhat strange, but harmless, effect that

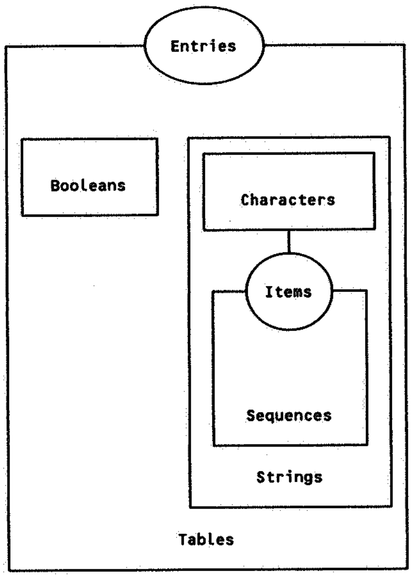
```
lookup(name, table(name, error-entry, tbl1)) = <true, error-entry>
```

and that

```
lookup(name, null-table) = <false, error-entry>.
```

Only in the first case `name` occurs in the table, but except for the `true/false` flag, the same value is delivered.

4.6.b. Structure diagram



4.6.c. Specification

```
module Tables
begin

   parameters Entries
        begin
```

```
          sorts         ENTRY
          functions
              error-entry :                  -> ENTRY
              eq          : ENTRY # ENTRY -> BOOL
      end Entries

exports
      begin
          sorts TABLE
          functions

              null-table      :                          -> TABLE
              table           : STRING # ENTRY # TABLE -> TABLE
              lookup          : STRING # TABLE           -> (BOOL # ENTRY)
              delete          : STRING # TABLE           -> TABLE
              eq              : TABLE # TABLE            -> BOOL
      end

imports Booleans, Strings

variables
      name, name1, name2      : -> STRING
      e, e1, e2               : -> ENTRY
      tbl, tbl1, tbl2         : -> TABLE
      found                   : -> BOOL

equations

[122]  table(name1, e1, table(name2, e2, tbl))
                               = if(eq(name1,name2),
                                     table(name1, e1, tbl),
                                     table(name2, e2, table(name1, e1, tbl)))

[123]  lookup(name, null-table)
                               = <false, error-entry>

[124]  lookup(name1, table(name2, e, tbl))
                               = if(eq(name1, name2),
                                     <true, e>,
                                     lookup(name1, tbl))
[125]  delete(name, null-table)
                               = null-table
[126]  delete(name1, table(name2, e, tbl))
                               = if(eq(name1, name2),
                                     delete(name1, tbl),
                                     table(name2, e, delete(name1, tbl)))


[127]  eq(tbl1, tbl2)          = eq(tbl2, tbl1)
```

```
[128]  eq(null-table, null-table)
                             = true

[129]  eq(null-table, table(name, e, tbl))
                             = false

[130]  eq(table(name, e1, tbl1), tbl2)
                             = if(and(found, eq(e1,e2)),
                                  eq(delete(name, tbl1), delete(name, tbl2)),
                                  false)
                               when <found, e2> = lookup(name, tbl2)
end Tables
```

5. CONTEXT-FREE PARSING

In this chapter the problem will be addressed how a context-free grammar can be specified within the algebraic framework and how the parsing process is to be described. A syntactic definition of a language can globally be subdivided in definitions for:

lexical syntax:
which defines the tokens of the language, i.e., keywords, identifiers, punctuation marks, etc.

context-free syntax:
which defines the concrete form of programs, i.e. the sequences of tokens that constitute a legal program.

abstract syntax:
which defines the abstract tree structure underlying the concrete (textual) form of programs. All further operations on programs may be defined as operations on the abstract syntax tree (see 6).

In this chapter, we will define a parser (`Context-free-parser`, see 5.4) which is parameterized with a lexical scanner and a grammar describing the concrete syntax and the construction rules for abstract syntax trees. The parsing problem is decomposed as follows:

1) Lexical analysis is delegated to a `Scanner` (a parameter of `Context-free-parser`), which transforms an input string into a sequence of lexical tokens (5.1). A token is a pair of strings: the first describes the lexical category of the token, the second gives the string value of the token, e.g. `token("identifier", "xyz")` or `token("integer-constant", "35")`.

2) Abstract syntax trees are represented by the data type `Atrees`. Rules for the construction of abstract syntax trees are part of the grammar for a given language. The essential function is `build`, which specifies for each non-terminal how certain (named) components of the syntax rule have to be combined into an abstract syntax tree (5.2).

3) BNF patterns (5.3) are introduced to allow the description of arbitrary context-free grammars. The main functions and operators introduced are `t` (indicates a terminal in the grammar), `n` (indicates a non-terminal), `+` (sequential composition of components of a grammar rule), and `|` (alternation). A grammar constructed by means of these operators can later be bound to the parameter `Syntax` of `Context-free-parser`.

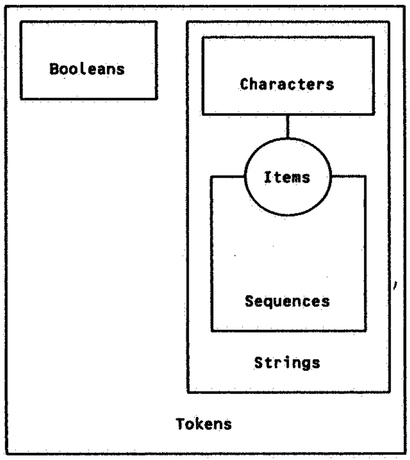
4) Actual parsing is described in `Context-free-parser` (5.4). This module has four parameters of which two are inherited from imported modules. The parameters `Scanner` and `Syntax` define the interface with the lexical scanner and with the concrete syntax and abstract syntax. `Context-free-parser` imports `BNF-patterns` (inheriting the unbound parameter `Non-terminals`) and `Atree-environments` (inheriting the unbound parameter `Operators`). `Context-free-parser` describes a parser which is driven by the BNF operators occurring in `Syntax`. Currently, we require that `Syntax` satisfies the LL(1) restrictions.

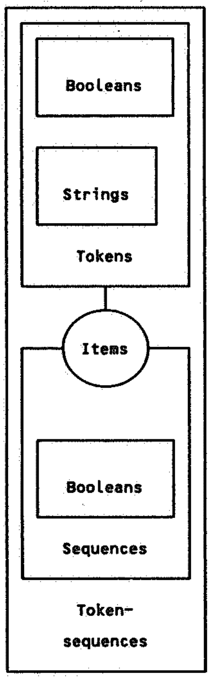
5.1. Interface with lexical scanner

5.1.a. Global description

Lexical analysis transforms an input string into a sequence of lexical tokens. A token is a pair of strings: the first describes the lexical category of the token, the second gives the string value of the token, e.g. `token("identifier", "xyz")` or `token("integer-constant", "35")`. In this section, the data types `Tokens` and `Token-sequences` are defined.

5.1.b. Structure diagrams





5.1.c. Specification

```
module Tokens
begin
   exports
        begin
           sorts TOKEN
           functions
                token    : STRING # STRING        -> TOKEN
                eq       : TOKEN # TOKEN          -> BOOL
        end

   imports Booleans, Strings

   variables
        s1, s2, s3, s4 : -> STRING

   equations

   [131]  eq(token(s1, s2), token(s3, s4))      = and(eq(s1, s3), eq(s2, s4))

end Tokens

module Token-sequences
begin
   imports Sequences
                 { renamed by
                         [ SEQ -> TOKEN-SEQUENCE,
                           null -> null-token-sequence ]
                   Items bound by
                         [ ITEM -> TOKEN,
                           eq -> eq ]
                         to Tokens
                 }

end Token-sequences
```

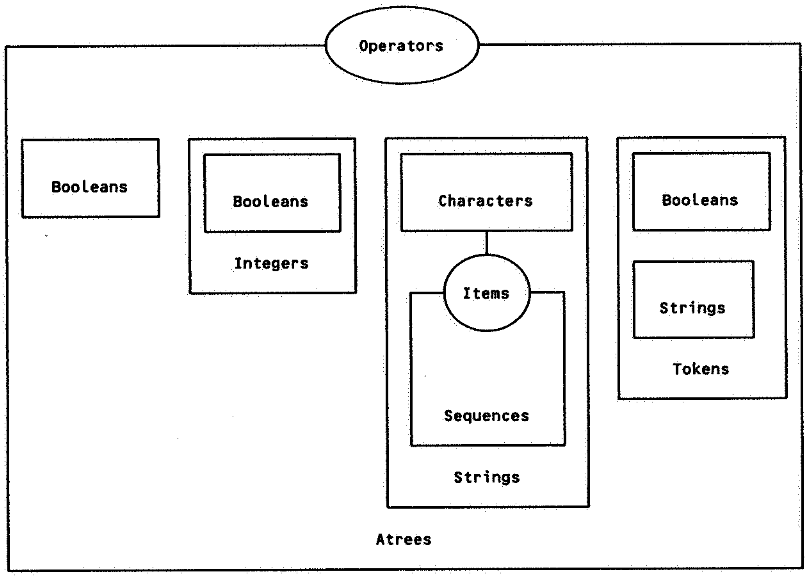
5.2. Interface with abstract syntax tree constructor

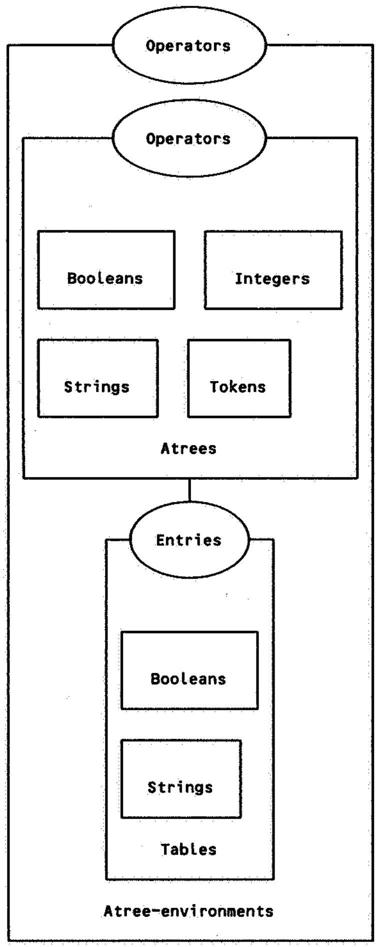
5.2.a. Global description

Abstract syntax trees are defined by the data type `Atrees`. Abstract syntax trees are essentially labelled trees whose nodes consist of an operator, indicating the construction operator of the node, and zero or more abstract syntax trees as sons. `Atrees` has one parameter `Operators`, which defines the interface to the set of operators for constructing abstract syntax trees. Conversion functions are defined for the common cases that the leaves of the abstract syntax tree consist of `Strings`, `Integers` or `Tokens`.

The construction process for abstract syntax trees as described in 5.4 uses the notion of environments of abstract syntax trees, i.e. tables which map strings onto abstract syntax trees. This notion is realized by the data type `Atree-environments`. Note that the parameter `Operators` of `Atrees` is inherited by `Atree-environments`.

5.2.b. Structure diagrams





5.2.c. Specification

```
module Atrees
begin

   parameters
      Operators
        begin
           sorts OPERATOR

           functions
                eq: OPERATOR # OPERATOR -> BOOL
        end Operators

   exports
        begin
           sorts ATREE
```

```
    functions
         error-atree    :                                 -> ATREE
         null-atree     :                                 -> ATREE
         atree          : OPERATOR                        -> ATREE
         atree          : OPERATOR # ATREE                -> ATREE
         atree          : OPERATOR # ATREE # ATREE        -> ATREE
         atree          : OPERATOR # ATREE # ATREE # ATREE -> ATREE
         string-atree   : STRING                          -> ATREE
         integer-atree  : INTEGER                         -> ATREE
         lexical-atree  : TOKEN                           -> ATREE
         eq             : ATREE # ATREE                   -> BOOL
   end

imports Booleans, Integers, Strings, Tokens

variables
     c, c1, c2 :-> OPERATOR
     a, a1, a2, a3, a4 :-> ATREE
     b1, b2, b3, b4 :-> ATREE
     s, s1, s2 :-> STRING
     n, n1, n2 :-> INTEGER
     t, t1, t2 :-> TOKEN

equations

[132]  eq(a1, a2)                            = eq(a2, a1)

[133]  eq(null-atree, null-atree)            = true
[134]  eq(null-atree, error-atree)           = false
[135]  eq(null-atree, atree(c))              = false
[136]  eq(null-atree, atree(c, a))           = false
[137]  eq(null-atree, atree(c, a1, a2))      = false
[138]  eq(null-atree, atree(c, a1, a2, a3))  = false
[139]  eq(null-atree, string-atree(s))       = false
[140]  eq(null-atree, integer-atree(n))      = false
[141]  eq(null-atree, lexical-atree(t))      = false

[142]  eq(error-atree, error-atree)           = true
[143]  eq(error-atree, atree(c))              = false
[144]  eq(error-atree, atree(c, a))           = false
[145]  eq(error-atree, atree(c, a1, a2))      = false
[146]  eq(error-atree, atree(c, a1, a2, a3)) = false
[147]  eq(error-atree, string-atree(s))       = false
[148]  eq(error-atree, integer-atree(n))      = false
[149]  eq(error-atree, lexical-atree(t))      = false

[150]  eq(atree(c1), atree(c2))              = eq(c1, c2)
[151]  eq(atree(c1), atree(c2, a1))          = false
[152]  eq(atree(c1), atree(c2, a1, a2))      = false
[153]  eq(atree(c1), atree(c2, a1, a2, a3))  = false
```

```
[154]  eq(atree(c), string-atree(s))         = false
[155]  eq(atree(c), integer-atree(n))        = false
[156]  eq(atree(c), lexical-atree(t))        = false

[157]  eq(atree(c1, a1), atree(c2, b1))      = and(eq(c1, c2), eq(a1, b1))
[158]  eq(atree(c1, a1), atree(c2, b1, b2))  = false
[159]  eq(atree(c1, a1), atree(c2, b1, b2, b3))
                                             = false
[160]  eq(atree(c1, a1), string-atree(s))    = false
[161]  eq(atree(c1, a1), integer-atree(n))   = false
[162]  eq(atree(c1, a1), lexical-atree(t))   = false


[163]  eq(atree(c1, a1, a2), atree(c2, b1, b2))
                                             = and(eq(c1, c2),
                                                   and(eq(a1, b1),
                                                       eq(a2, b2)))
[164]  eq(atree(c1, a1, a2), atree(c2, b1, b2, b3))
                                             = false
[165]  eq(atree(c1, a1, a2), string-atree(s))= false
[166]  eq(atree(c1, a1, a2), integer-atree(n))
                                             = false
[167]  eq(atree(c1, a1, a2), lexical-atree(t))
                                             = false


[168]  eq(atree(c1, a1, a2, a3), atree(c2, b1, b2, b3))
                                             = and(eq(c1, c2),
                                                   and(eq(a1, b1),
                                                       and(eq(a2, b2),
                                                           eq(a3, b3))))
[169]  eq(atree(c1, a1, a2, a3), string-atree(s))
                                             = false
[170]  eq(atree(c1, a1, a2, a3), integer-atree(n))
                                             = false
[171]  eq(atree(c1, a1, a2, a3), lexical-atree(t))
                                             = false



[172]  eq(string-atree(s1), string-atree(s2))= eq(s1, s2)
[173]  eq(string-atree(s), integer-atree(n)) = false
[174]  eq(string-atree(s), lexical-atree(t)) = false

[175]  eq(integer-atree(n1), integer-atree(n2))
                                             = eq(n1, n2)
[176]  eq(integer-atree(n), lexical-atree(t))= false

[177]  eq(lexical-atree(t1), lexical-atree(t2))
                                             = eq(t1, t2)
```

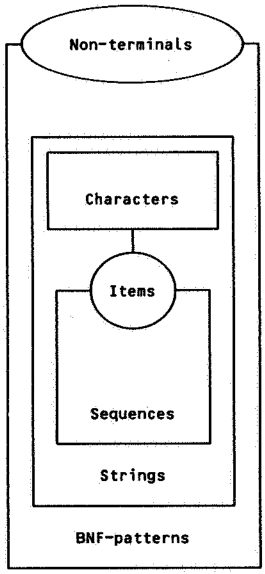
```
end Atrees

module Atree-environments
begin
   exports
        begin
           functions
                  _^_ : STRING # ATREE-ENV -> ATREE
        end

   imports Tables
        { renamed by
                [ TABLE -> ATREE-ENV,
                  null-table -> null-atree-env]
          Entries bound by
                [ ENTRY -> ATREE,
                  eq -> eq,
                  error-entry -> error-atree]
                to Atrees
        }
   variables
        s :-> STRING
        e :-> ATREE-ENV
        f :-> BOOL
        v :-> ATREE

   equations

   [178]  s ^ e           = v
                            when <f, v> = lookup(s, e)

end Atree-environments
```

5.3. BNF patterns

5.3.a. Global description

BNF patterns are introduced to allow the description of arbitrary context-free grammars. The main functions and operators introduced are `t` (indicates a terminal in the grammar), `n` (indicates a non-terminal), `lexical` (indicates a lexical item), `+` (sequential composition of components of a grammar rule), and `|` (alternation). The functions `t`, `n` and `lexical` have two variants: the variant with one argument indicates respectively a terminal, non-terminal or lexical item; the variant with two arguments also associates a name with the syntaxctic notion. These names can later be used to refer to the abstract syntax tree which is the result of parsing the given syntactic notion. An actual grammar constructed with these operators can be bound to the parameter `Syntax` of `Context-free-parser`. Examples of grammars using this notation are the lexical syntax (6.2.2) and concrete syntax (6.3.2) of PICO.

5.3.b. Structure diagram



5.3.c. Specification

```
module BNF-patterns
begin

   parameters
      Non-terminals
        begin
           sorts NON-TERMINAL
        end Non-terminals

   exports
```

```
    begin
       sorts PATTERN

       functions

            _ + _     : PATTERN # PATTERN      -> PATTERN
            _ | _     : PATTERN # PATTERN      -> PATTERN
            t         : STRING                 -> PATTERN
            t         : STRING # STRING        -> PATTERN
            n         : NON-TERMINAL           -> PATTERN
            n         : NON-TERMINAL # STRING  -> PATTERN
            lexical   : STRING                 -> PATTERN
            lexical   : STRING # STRING        -> PATTERN
            null-pattern :                     -> PATTERN

    end

  imports Strings

end BNF-patterns
```

5.4. Context-free parser

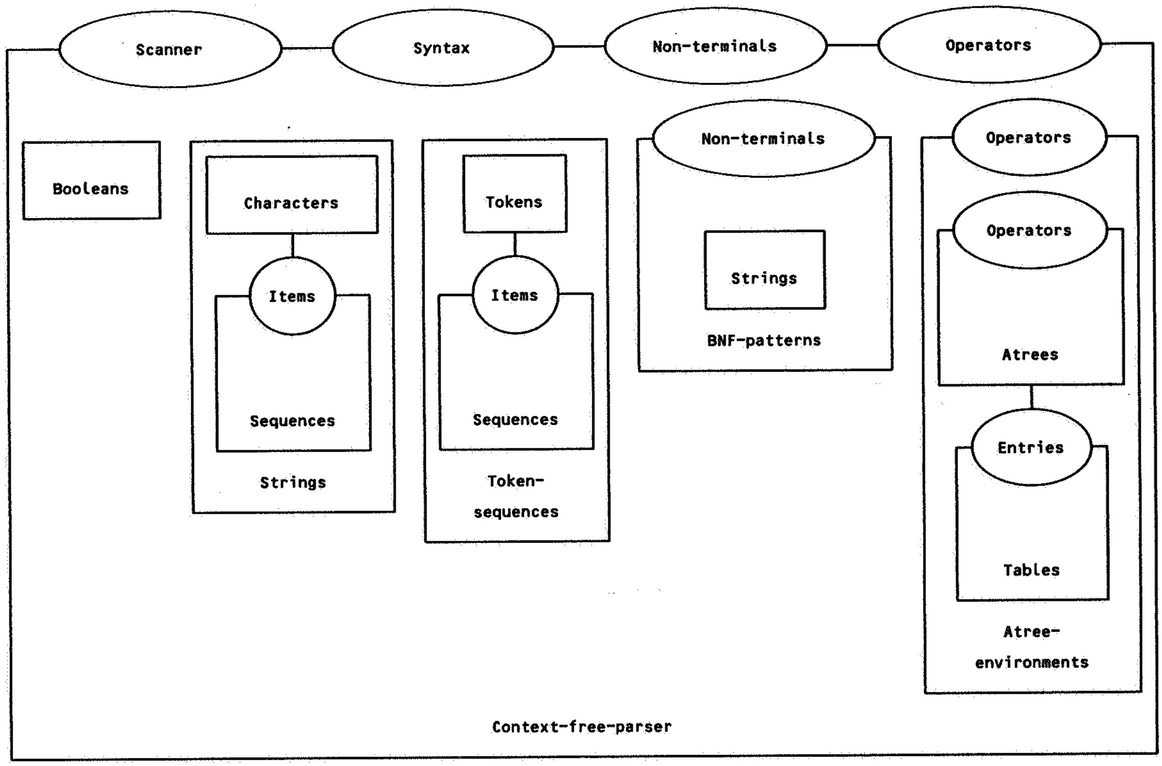
5.4.a. Global description

`Context-free-parser` describes the actual parsing process. It has four parameters of which two are inherited from imported modules. Parameter `Scanner` defines the interface with the lexical scanner, i.e. the function `scan` which converts input strings to `Token-sequences`. Parameter `Syntax` defines the interface with the rules of the syntax (function `rule`) and with the rules for constructing abstract syntax trees (function `build`). `Context-free-parser` imports `BNF-patterns` (inheriting the unbound parameter `Non-terminals`, which defines the interface with the set of non-terminals of the syntax) and `Atree-environments` (inheriting the unbound parameter `Operators`, which defines the interface with the set of construction operators for the abstract syntax).

`Context-free-parser` describes a parser for the language described by the syntax rules. The equations in `Context-free-parser` describe for each type of BNF operator the conditions under which (a part of) the input `Token-sequence` is acceptable. The BNF operator `n` (non-terminal) uses the function `rule` from parameter `Syntax` to associate a pattern with a non-terminal. Acceptance of a part of the input is expressed by constructing an `Atree-environment` consisting of named `Atrees`. Acceptance of a non-terminal is expressed by the function `build` from `Syntax` for that non-terminal .

Currently, we require that the syntax satisfies the LL(1) restrictions. This simplifies the definition of `Context-free-parser` considerably: in the definition given below only *one* abstract syntax tree has to be constructed instead of a *set* of abstract syntax trees as would be necessary in the case of an ambiguous input string if the grammar were not LL(1).

5.4.b. Structure diagram



5.4.c. Specification

```
module Context-free-parser
begin

   parameters
      Scanner
        begin
           functions
                scan : STRING -> TOKEN-SEQUENCE
        end Scanner,

      Syntax
        begin

           functions

                rule    : NON-TERMINAL                  -> PATTERN
                build   : NON-TERMINAL # ATREE-ENV      -> ATREE

        end Syntax

   exports
        begin
```

```
          functions
               parse    : NON-TERMINAL # STRING          -> ATREE
     end

imports Booleans, Strings, Token-sequences, BNF-patterns, Atree-environments

functions

     parse-rule: NON-TERMINAL # TOKEN-SEQUENCE
                              -> (BOOL # ATREE # TOKEN-SEQUENCE)
     parse-pat : PATTERN # TOKEN-SEQUENCE # ATREE-ENV
                              -> (BOOL # ATREE-ENV # TOKEN-SEQUENCE)

variables
     x                        : -> NON-TERMINAL
     p, p1, p2                : -> PATTERN
     env, env1, env2          : -> ATREE-ENV
     atree, atree1, atree2    : -> ATREE
     s, tail, tail1, tail2    : -> TOKEN-SEQUENCE
     id, val, str, lextype    : -> STRING
     r, r1, r2                : -> BOOL

equations

[179]   parse(x, str)         = if(and(r, eq(tail, null-token-sequence)),
                                   atree,
                                   error-atree)
                                when  <r, atree, tail> = parse-rule(x, scan(str))

[180]   parse-rule(x, s)      = if(r, < true, build(x, env), tail >,
                                      < false, error-atree, tail >)
                                when  <r, env, tail> =
                                      parse-pat(rule(x), s, null-atree-env)

[181]   parse-pat(null-pattern, s, env)
                              = <true, env, s>

[182]   parse-pat(p1 + p2, s, env1)
                              = if(r, parse-pat(p2, tail, env2),
                                      < false, env2, tail >)
                                when <r, env2, tail> = parse-pat(p1, s, env1)

[183]   parse-pat(p1 | p2, s, env)
                              = if(not(r1),
                                   < r2, env2, tail2 >,
                                   if(not(r2),
                                      < r1, env1, tail1 >,
                                      < false, env, s >))
                                when  <r1, env1, tail1> = parse-pat(p1, s, env),
                                      <r2, env2, tail2> = parse-pat(p2, s, env)
```

```
[184]  parse-pat(n(x), s, env)
                              = <r, env, tail>
                                when <r, atree, tail> = parse-rule(x, s)

[185]  parse-pat(n(x,id), s, env)
                              = if(r, < true, table(id, atree, env), tail >,
                                      < false, env, tail >)
                                when <r, atree, tail> = parse-rule(x, s)

[186]  parse-pat(t(str), seq(token(lextype, val), s), env)
                              = if(and(eq(str, val),
                                       or(eq(lextype, "keyword"),
                                          eq(lextype, "literal"))),
                                   < true, env, s>,
                                   < false, env, s> )

[187]  parse-pat(t(str), null-token-sequence, env)
                              = if(eq(str, null-string),
                                   <true, env, null-token-sequence>,
                                   <false, env, null-token-sequence>)

[188]  parse-pat(t(str, id), seq(token(lextype, val), s), env)
                              = if(and(eq(str, val),
                                       or(eq(lextype, "keyword"),
                                          eq(lextype, "literal"))),
                                   < true,
                                     table(id,
                                           lexical-atree(token(lextype,str)),
                                           env),
                                     s>,
                                   < false, env, s> )

[189]  parse-pat(t(str, id), null-token-sequence, env)
                              = if(eq(str, null-string),
                                   < true,
                                     table(id,
                                           lexical-atree(token("literal","")),
                                           env),
                                     null-token-sequence>,
                                   <false, env, null-token-sequence>)

[190]  parse-pat(lexical(str), seq(token(lextype, val), s), env)
                              = if(eq(lextype, str),
                                   < true, env, s >,
                                   < false, env, s> )

[191]  parse-pat(lexical(str), null-token-sequence, env)
                              = <false, env, null-token-sequence>

[192]  parse-pat(lexical(str,id), seq(token(lextype, val), s), env)
```

```
                              = if(eq(lextype, str),
                                  < true,
                                    table(id,
                                          lexical-atree(token(lextype,val)),
                                          env),
                                    s >,
                                  < false, env, s> )

   [193]  parse-pat(lexical(str,id), null-token-sequence, env)
                              = <false, env, null-token-sequence>

end Context-free-parser
```

6. ALGEBRAIC SPECIFICATION OF PICO

After the preparations in the previous chapters, the following steps are still needed to obtain a complete specification of PICO:

1) The notions of *types* and *values* in PICO programs have to be formalized (6.1).
2) The lexical syntax of PICO has to be specified. This is done by constructing a lexical scanner on the basis of `Context-free-parser` as defined in the previous chapter (6.2).
3) The concrete syntax of PICO and the rules for the construction of abstract syntax trees have to be specified. This is accomplished by a *second* application of `Context-free-parser` (6.3).
4) The static semantics of PICO has to be specified, defining certain constraints on programs, i.e. constraints that do not depend on input data. For instance, in a "legal" program all variables should have been declared, all expressions should be type consistent, etc. This is described in 6.4.
5) Dynamic semantics of PICO has to be specified, defining the meaning of a program, i.e. the relation between its input and output data (6.5).
6) All the above components of the PICO specification have to be combined into one *PICO system* (6.6).

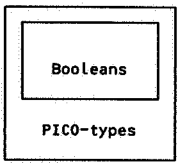
6.1. Types and values

6.1.1. Types

6.1.1.a. Global description

The data type `PICO-types` defines the allowed types in PICO programs, i.e. integers and strings. An additional `error-type` is introduced for describing typing errors.

6.1.1.b. Structure diagram



6.1.1.c. Specification

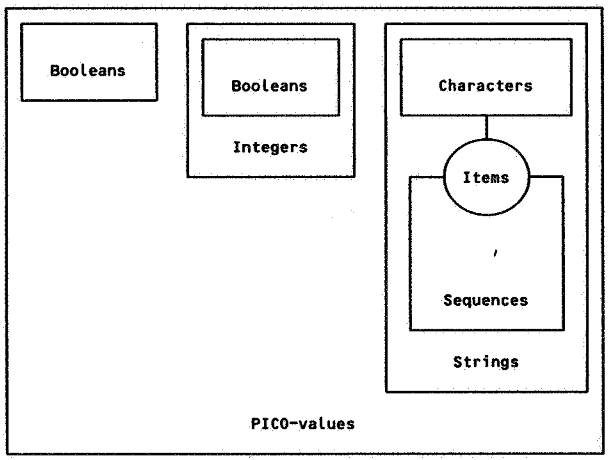
```
module PICO-types
begin
   exports
        begin

           sorts PICO-TYPE

           functions
                integer-type    :                           -> PICO-TYPE
                string-type     :                           -> PICO-TYPE
                error-type      :                           -> PICO-TYPE
                eq              : PICO-TYPE # PICO-TYPE -> BOOL
        end

   imports Booleans

   variables
        x, y            : -> PICO-TYPE


   equations

   [194]  eq(x, x)                          = true
   [195]  eq(x, y)                          = eq(y, x)
   [196]  eq(integer-type, string-type) = false
   [197]  eq(integer-type, error-type)  = false
   [198]  eq(string-type, error-type)   = false

end PICO-types
```

6.1.2. Values

6.1.2.a. Global description

The data type `PICO-values` defines the allowed values as they may occur during the execution of PICO programs, i.e. integers and strings. An additional `error-value` is introduced for describing values that are the result of evaluating erroneous programs. Note that there is no integer or string corresponding to `error-value`. Two conversion functions are defined for converting `Integers` and `Strings` into `PICO-values`.

6.1.2.b. Structure diagram



6.1.2.c. Specification

```
module PICO-values
begin

   exports
        begin
           sorts PICO-VALUE

           functions
                error-value    :                              -> PICO-VALUE
                pico-value     : INTEGER                      -> PICO-VALUE
                pico-value     : STRING                       -> PICO-VALUE
                eq             : PICO-VALUE # PICO-VALUE      -> BOOL
        end

   imports Booleans, Integers, Strings

   variables
        x, y            : -> PICO-VALUE
        int, int1, int2 : -> INTEGER
        str, str1, str2 : -> STRING
```

6.2. Lexical syntax

The lexical syntax describes the lexical tokens that may occur in a PICO program. We construct a lexical scanner for PICO by means of `Context-free-parser`:

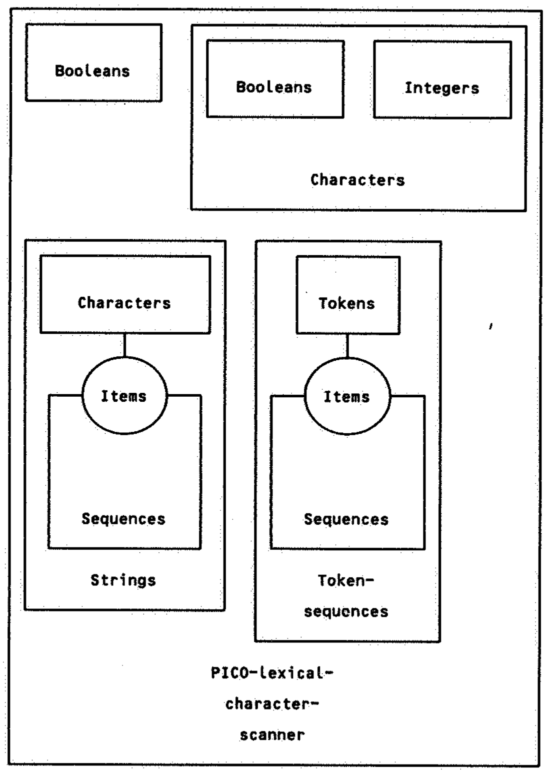
1) A character-level scanner is defined (6.2.1). This character-level scanner distinguishes characters according to their character types, i.e. letter, digit, layout, etc.
2) The lexical syntax for PICO and the construction rules for lexical tokens are defined (6.2.2). This amounts to defining the syntactic form of identifiers, strings, etc. and to defining the result for each case, e.g. parsing the non-terminal `integer-constant` of the lexical syntax will have as result `token("integer-constant", x)`, where `x` is the string representation of the integer constant.
3) A lexical scanner for PICO is obtained by combining the results of the previous two steps with `Context-free-parser`. (6.2.3).

6.2.1. Lexical character scanner

6.2.1.a. Global description

`PICO-lexical-character-scanner` defines the character-level scanner `char-scan` which distinguishes characters according to their character types, i.e. letter, digit, layout and literal, and converts the input string into a `Token-sequnece`.

6.2.1.b. Structure diagram



6.2.1.c. Specification

```
module PICO-lexical-character-scanner
begin
   exports
        begin
           functions
                char-scan : STRING -> TOKEN-SEQUENCE
        end

   imports Booleans, Characters, Strings, Token-sequences

   functions
        char-scan1 : CHAR -> TOKEN
```

```
    is-layout  : CHAR -> BOOL

variables
    c          : -> CHAR
    str        : -> STRING

equations

[206]  char-scan(seq(c, str))       = seq(char-scan1(c), char-scan(str))

[207]  char-scan("")          = null-token-sequence

[208]  char-scan1(c)          = if(is-layout(c), token("layout", string(c)),
                                if(is-letter(c),token("letter", string(c)),
                                if(is-digit(c),token("digit", string(c)),
                                   token("literal", string(c)))))

[209]  is-layout(c)           = or(eq(c, char-space),
                                   or(eq(c, char-ht),
                                      eq(c, char-nl)))

end PICO-lexical-character-scanner
```

6.2.2. Lexical syntax and rules for token construction

6.2.2.a. Global description

The lexical syntax for PICO and the construction rules for lexical tokens are defined in this section. This amounts to defining the syntactic form of identifiers, strings, etc. and to defining the result for each case, e.g. parsing the non-terminal `integer-constant` of the lexical syntax will have as result `token("integer-constant", x)`, where `x` is the string representation of the integer constant.

The following data types are defined here:

`PICO-non-terminals-of-lexical-syntax`: defines the sort `LEX-NON-TERMINAL` and all non-terminals of the lexical syntax.

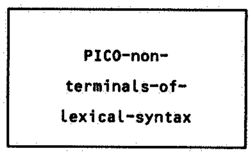
`PICO-lex-BNF-patterns`: defines a version of `BNF-patterns` with parameter `Non-terminals` bound to `PICO-non-terminals-of-lexical-syntax`.

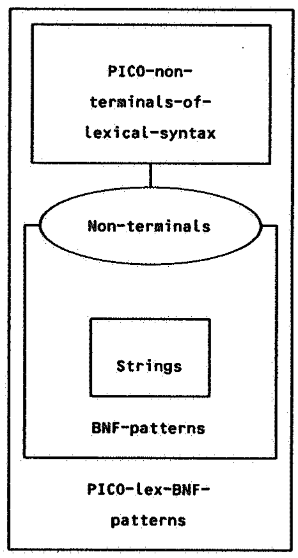
`PICO-atree-operators-of-lexical-syntax`: defines the sort `LEX-OPERATOR` and the operators for constructing abstract syntax trees for the lexical syntax.

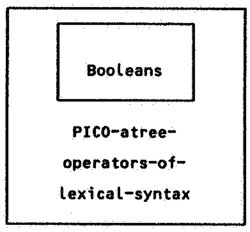
`PICO-lex-atree-environments`: defines a version of `Atree-environments` with parameter `Operators` bound to `PICO-atree-operators-of-lexical-syntax`.

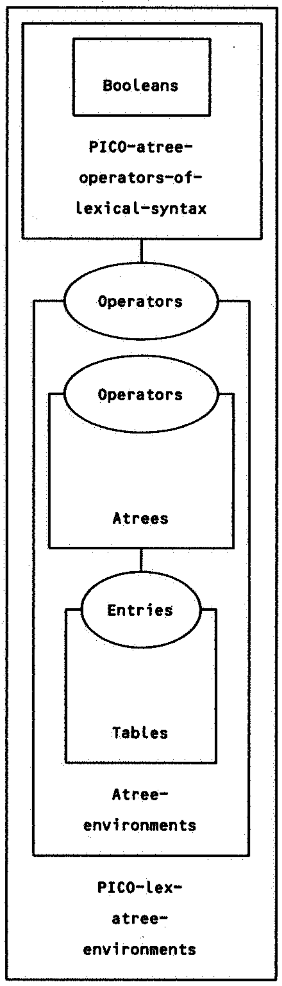
`PICO-lexical-syntax`: defines the lexical syntax for PICO and the rules for token construction. Essentially the grammar contains for each non-terminal pairs of equations for the functions `rule` (i.e. the actual syntax rule) and `build` (i.e. the construction procedure for abstract syntax trees). Note that all syntax rules with names starting with `non-empty` do not appear in the original grammar. These rules are artefacts made necessary by limitations in the descriptive power of `BNF-patterns`; most notably, it is impossible to associate different `build` functions with the alternatives in one `rule`.

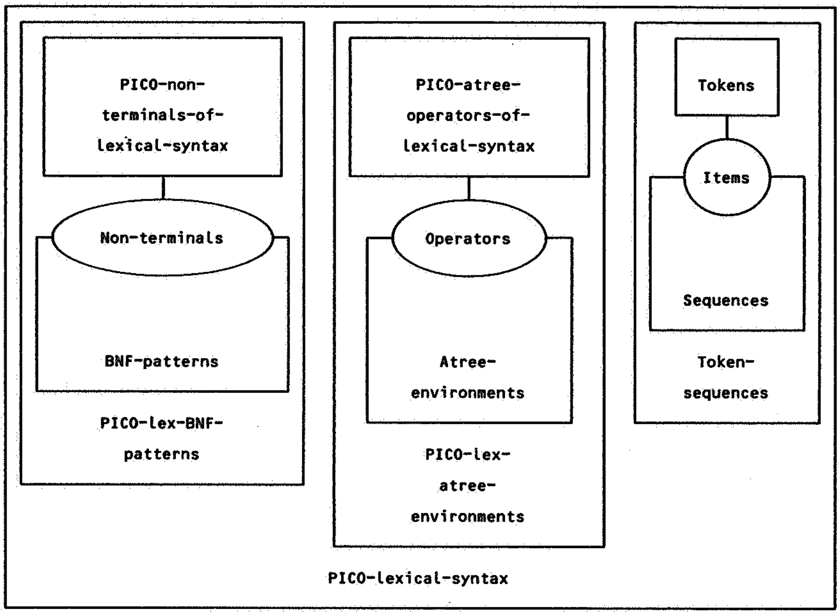
6.2.2.b. Structure diagrams




PICO-non-terminals-of-lexical-syntax
Non-terminals
Strings
BNF-patterns
PICO-lex-BNF-patterns


Booleans
PICO-atree-operators-of-lexical-syntax


Booleans
PICO-atree-
operators-of-
lexical-syntax
Operators
Operators
Atrees
Entries
Tables
Atree-
environments
PICO-lex-
atree-
environments



6.2.2.c. Specification

```
module PICO-non-terminals-of-lexical-syntax
begin
   exports
        begin
           sorts LEX-NON-TERMINAL

           functions
                lexical-stream           :          -> LEX-NON-TERMINAL
                non-empty-lexical-stream:           -> LEX-NON-TERMINAL
                empty-lexical-stream     :          -> LEX-NON-TERMINAL
                lexical-item             :          -> LEX-NON-TERMINAL
                optional-layout          :          -> LEX-NON-TERMINAL
                keyword-or-ident         :          -> LEX-NON-TERMINAL
                ident                    :          -> LEX-NON-TERMINAL
                ident-chars              :          -> LEX-NON-TERMINAL
                non-empty-ident-chars    :          -> LEX-NON-TERMINAL
                ident-char               :          -> LEX-NON-TERMINAL
                integer-const            :          -> LEX-NON-TERMINAL
                digits                   :          -> LEX-NON-TERMINAL
                non-empty-digits         :          -> LEX-NON-TERMINAL
                digit                    :          -> LEX-NON-TERMINAL
                string-const             :          -> LEX-NON-TERMINAL
                string-tail              :          -> LEX-NON-TERMINAL
                non-empty-string-tail    :          -> LEX-NON-TERMINAL
                quote                    :          -> LEX-NON-TERMINAL
                any-char-but-quote       :          -> LEX-NON-TERMINAL
                letter                   :          -> LEX-NON-TERMINAL
```

```
                layout            :                -> LEX-NON-TERMINAL
                literal           :                -> LEX-NON-TERMINAL
                concat            :                -> LEX-NON-TERMINAL
                assign-or-colon   :                -> LEX-NON-TERMINAL
                empty             :                -> LEX-NON-TERMINAL
        end

end PICO-non-terminals-of-lexical-syntax

module PICO-lex-BNF-patterns
begin
   imports BNF-patterns
        { renamed by
                [ PATTERN -> LEX-PATTERN,
                  t -> lt ]
          Non-terminals bound by
                [ NON-TERMINAL -> LEX-NON-TERMINAL ]
                to PICO-non-terminals-of-lexical-syntax
        }

end PICO-lex-BNF-patterns

module PICO-atree-operators-of-lexical-syntax
begin
   exports
      begin
         sorts LEX-OPERATOR

         functions
           op-lex-item  :                               -> LEX-OPERATOR
           op-lex-stream:                               -> LEX-OPERATOR
           eq           : LEX-OPERATOR # LEX-OPERATOR -> BOOL

      end

   imports Booleans

   variables
        o1, o2 :-> LEX-OPERATOR

   equations

   [210]  eq(o1, o2)                          = eq(o2, o1)
   [211]  eq(op-lex-item, op-lex-item)        = true
   [212]  eq(op-lex-item, op-lex-stream)      = false
   [213]  eq(op-lex-stream, op-lex-stream)    = true

end PICO-atree-operators-of-lexical-syntax
```

```
module PICO-lex-atree-environments
begin

   imports Atree-environments
        { renamed by
                [ ATREE -> LEX-ATREE,
                  atree -> lex-atree,
                  null-atree -> null-lex-atree,
                  error-atree -> error-lex-atree,
                  lexical-atree -> lexical-lex-atree,
                  ATREE-ENV -> LEX-ATREE-ENV,
                  null-atree-env -> null-lex-atree-env,
                  ATREE -> LEX-ATREE,
                  error-atree -> error-lex-atree ]
          Operators bound by
                [ OPERATOR -> LEX-OPERATOR,
                  eq -> eq ]
                to PICO-atree-operators-of-lexical-syntax
        }

end  PICO-lex-atree-environments

module PICO-lexical-syntax
begin

   exports
        begin
           functions
                rule       : LEX-NON-TERMINAL                    -> LEX-PATTERN
                build      : LEX-NON-TERMINAL # LEX-ATREE-ENV -> LEX-ATREE
                lex-stream : TOKEN-SEQUENCE                      -> LEX-ATREE
                lex-item   : TOKEN                               -> LEX-ATREE
        end

   imports PICO-lex-BNF-patterns, PICO-lex-atree-environments, Token-sequences


   variables
        env        :-> LEX-ATREE-ENV
        l, l1, l2 :-> TOKEN-SEQUENCE
        t, t1, t2 :-> TOKEN
        s, s1, s2 :-> STRING
        d, d1, d2 :-> STRING

   equations

   [214]  rule(lexical-stream)  = n(non-empty-lexical-stream,"ls") |
                                  n(empty-lexical-stream,"ls")
   [215]  build(lexical-stream, env)
                                = "ls" ^ env
```

```
[216]  rule(non-empty-lexical-stream)
                           = n(lexical-item,"t") + n(lexical-stream,"l")
[217]  build(non-empty-lexical-stream, env)
                           = lex-atree(op-lex-stream, lex-stream(seq(t, l)))
                             when  lex-atree(op-lex-item, lex-item(t))
                                          = "t" ^ env,
                                   lex-atree(op-lex-stream, lex-stream(l))
                                          = "l" ^ env
[218]  rule(empty-lexical-stream)
                           = n(empty)
[219]  build(empty-lexical-stream, env)
                           = lex-atree(op-lex-stream,
                                   lex-stream(null-token-sequence))

[220]  rule(lexical-item)  = n(optional-layout) +
                                  ( n(keyword-or-ident,"i") |
                                    n(integer-const,"i") |
                                    n(string-const,"i") |
                                    n(literal,"i")
                                  )
[221]  build(lexical-item, env)
                           = "i" ^ env

[222]  rule(optional-layout) = n(layout) | n(empty)
[223]  build(optional-layout, env)
                           = null-lex-atree

[224]  rule(keyword-or-ident)= n(ident,"i")
[225]  build(keyword-or-ident, env)
                           = if(or(eq(s, "begin"),
                                or(eq(s, "end"),
                                or(eq(s, "declare"),
                                or(eq(s, "integer"),
                                or(eq(s, "string"),
                                or(eq(s, "if"),
                                or(eq(s, "then"),
                                or(eq(s, "else"),
                                or(eq(s, "fi"),
                                or(eq(s, "while"),
                                or(eq(s, "do"),
                                   eq(s, "od"))))))))))))),
                                   lex-atree(op-lex-item,
                                         lex-item(token("keyword", s))),
                                   lex-atree(op-lex-item,
                                         lex-item(token("id", s))))

                             when lexical-lex-atree(token("id",s)) = "i" ^ env


[226]  rule(ident)         = n(letter,"s1") + n(ident-chars,"s2")
```

```
[227]  build(ident, env)      = lexical-lex-atree(token("id", conc(s1, s2)))
                                when  string-atree(s1) = "s1" ^ env,
                                      string-atree(s2) = "s2" ^ env

[228]  rule(ident-chars)      = n(non-empty-ident-chars,"s") | n(empty,"s")
[229]  build(ident-chars, env)
                              = "s" ^ env
[230]  rule(non-empty-ident-chars)
                              = n(ident-char,"s1") + n(ident-chars,"s2")
[231]  build(non-empty-ident-chars, env)
                              = string-atree(conc(s1, s2))
                                when  string-atree(s1) = "s1" ^ env,
                                      string-atree(s2) = "s2" ^ env
[232]  rule(ident-char)       = n(letter,"x") | n(digit,"x")
[233]  build(ident-char, env)= "x" ^ env

[234]  rule(integer-const)    = n(digit,"d1") + n(digits,"d2")
[235]  build(integer-const, env)
                              = lex-atree(op-lex-item,
                                      lex-item(token("integer-constant",
                                                     conc(d1, d2))))
                                when  string-atree(d1) = "d1" ^ env,
                                      string-atree(d2) = "d2" ^ env

[236]  rule(digits)           = n(non-empty-digits,"d") | n(empty,"d")
[237]  build(digits, env)     = "d" ^ env

[238]  rule(non-empty-digits)= n(digit,"d1") + n(digits,"d2")
[239]  build(non-empty-digits, env)
                              = string-atree(conc(d1, d2))
                                when  string-atree(d1) = "d1" ^ env,
                                      string-atree(d2) = "d2" ^ env

[240]  rule(string-const)     = n(quote) + n(string-tail,"s")
[241]  build(string-const, env)
                              = lex-atree(op-lex-item,
                                      lex-item(token("string-constant", s)))
                                when  string-atree(s) = "s" ^ env

[242]  rule(string-tail)      = n(non-empty-string-tail,"s") | n(quote,"s")
[243]  build(string-tail, env)
                              = "s" ^ env
[244]  rule(non-empty-string-tail)
                              = n(any-char-but-quote,"s1") + n(string-tail,"s2")
[245]  build(non-empty-string-tail, env)
                              = string-atree(conc(s1, s2))
                                when  string-atree(s1) = "s1" ^ env,
                                      string-atree(s2) = "s2" ^ env

[246]  rule(quote)            = lt(string(char-quote))
```

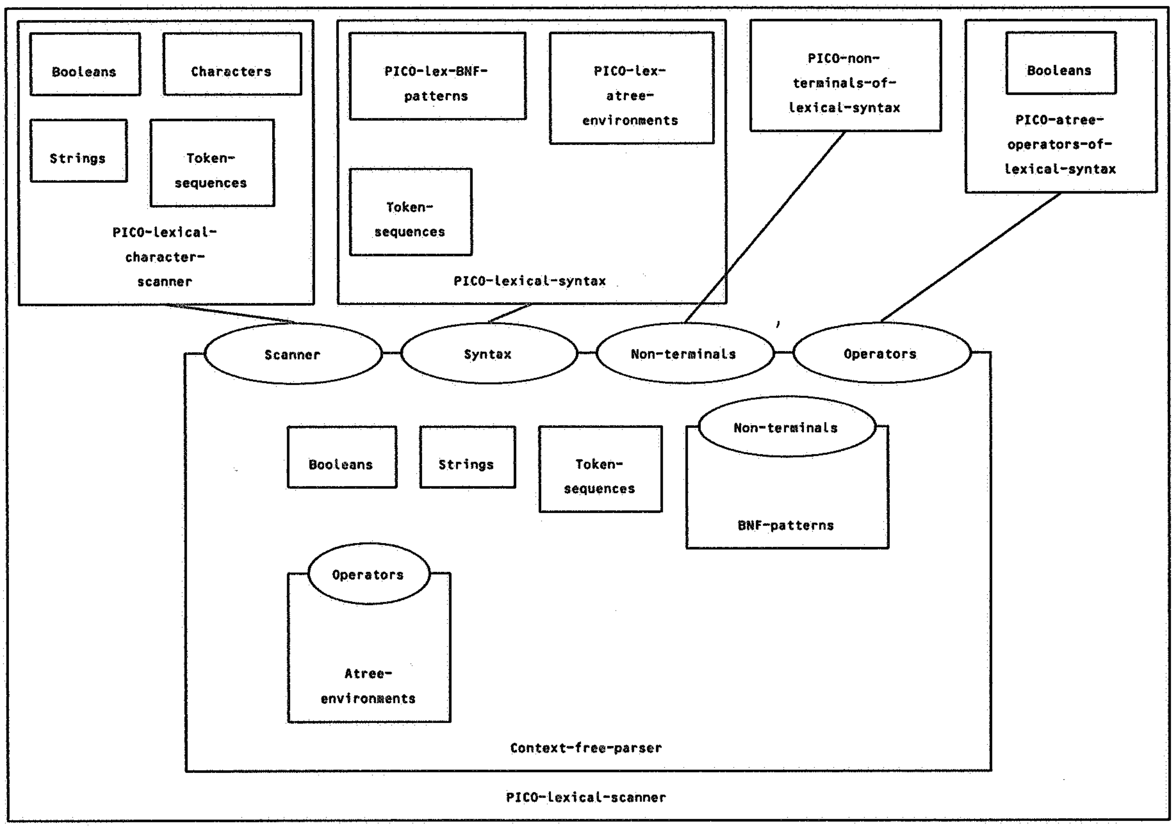
```
[247]  build(quote, env)      = string-atree("")

[248]  rule(any-char-but-quote)
                              = n(letter,"c") |
                                n(digit,"c") |
                                n(literal,"c") |
                                n(layout,"c")
[249]  build(any-char-but-quote, env)
                              = "c" ^ env

[250]  rule(letter)           = lexical("letter","s")
[251]  build(letter, env)     = string-atree(s)
                                when lexical-lex-atree(token("letter",s)) = "s" ^ env

[252]  rule(digit)            = lexical("digit","d")
[253]  build(digit, env)      = string-atree(d)
                                when lexical-lex-atree(token("digit", d)) = "d" ^ env
[254]  rule(layout)           = lexical("layout","s")
[255]  build(layout, env)     = string-atree(s)
                                when lexical-lex-atree(token("layout", s)) = "s" ^ env
[256]  rule(literal)          = lt("(","s") |
                                lt(")","s") |
                                lt("+","s") |
                                lt("-","s") |
                                lt(";","s") |
                                lt(",","s") |
                                n(concat,"s") |
                                n(assign-or-colon,"s")
[257]  build(literal, env)    = "s" ^ env

[258]  rule(concat)           = lt("|") + lt("|")
[259]  build(concat, env)     = string-atree("||")

[260]  rule(assign-or-colon) = lt(":") + (lt("=","s") | n(empty,"s"))
[261]  build(assign-or-colon, env)
                              = if(eq(s, "="),
                                   string-atree(":="),
                                   string-atree(":"))
                                when string-atree(s) = "s" ^ env

[262]  rule(empty)            = lt("")
[263]  build(empty, env)      = string-atree("")


end PICO-lexical-syntax
```

6.2.3. Lexical scanner

6.2.3.a. Global description

In this section a lexical scanner for PICO is obtained by combining `PICO-lexical-character-scanner`, `PICO-lexical-syntax`, `PICO-non-terminals-of-lexical-syntax` and `PICO-atree-operators-of-lexical-syntax` with `Context-free-parser`.

6.2.3.b. Structure diagram



6.2.3.c. Specification

```
module PICO-lexical-scanner
begin
   exports
        begin
           functions
                lex-scan : STRING -> TOKEN-SEQUENCE
        end

   imports Context-free-parser
        { Scanner bound by
```

```
              [ scan -> char-scan ]
              to PICO-lexical-character-scanner
        Syntax bound by
              [ rule -> rule,
                build -> build ]
              to PICO-lexical-syntax
        Non-terminals bound by
              [ NON-TERMINAL -> LEX-NON-TERMINAL ]
              to PICO-non-terminals-of-lexical-syntax
        Operators bound by
              [ OPERATOR -> LEX-OPERATOR,
                eq -> eq ]
              to PICO-atree-operators-of-lexical-syntax
      }

   variables
        l :-> TOKEN-SEQUENCE
        s :-> STRING

   equations

   [264]  lex-scan(s)          = l
                                 when   lex-atree(op-lex-stream, lex-stream(l)) =
                                        parse(lexical-stream, s)

end PICO-lexical-scanner
```

6.3. Abstract and concrete syntax

In this section we specify the abstract and concrete syntax for PICO; this will result in a specification for a parser that transforms PICO-programs from their textual form into abstract syntax trees. We proceed as follows:

1) The abstract syntax for PICO is defined (6.3.1).

2) The concrete syntax and the rules for constructing abstract syntax trees are defined (6.3.2).

3) The lexical scanner (as defined in the previous section), the concrete syntax and the rules for the construction of abstract syntax trees (both defined in this section) are combined with `Context-free-parser`. In this way we obtain a parser that transforms PICO programs into abstract syntax trees (6.3.3).

6.3.1. Abstract syntax

6.3.1.a. Global description

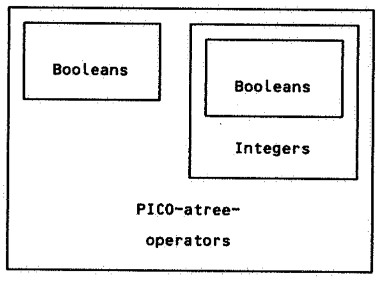
In this section the abstract syntax for PICO is defined. This involves the following data types:

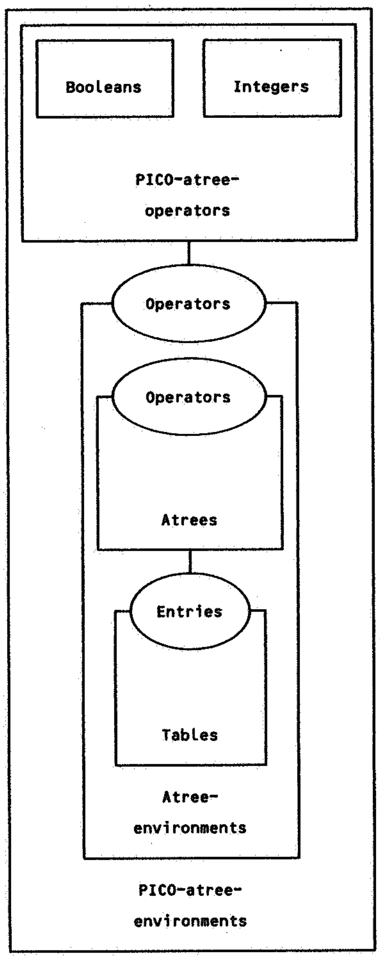
`PICO-atree-operators`: the operators for constructing abstract syntax trees.

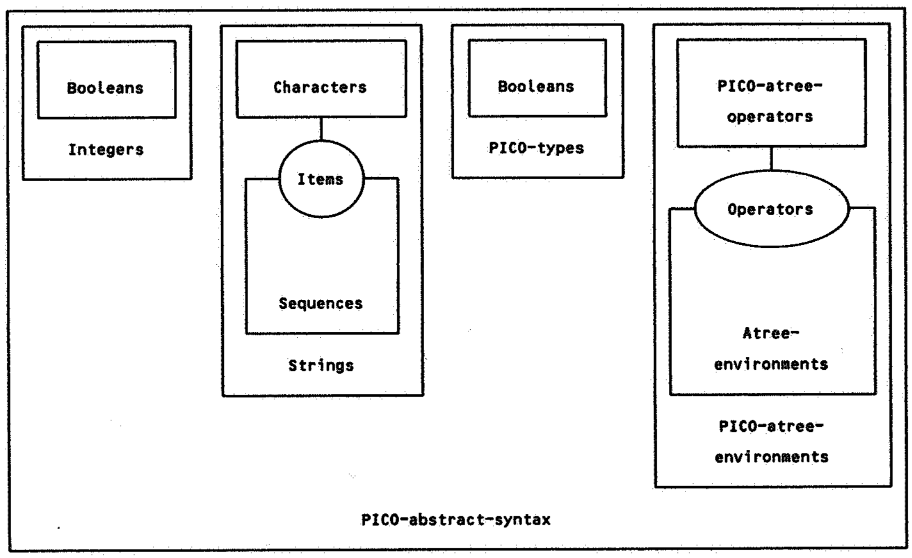
`PICO-atree-environments`: a version of `Atree-environments` with parameter `Operators` bound to `PICO-atree-operators`.

`PICO-abstract-syntax`: defines the actual abstract syntax. Essentially, this module defines higher-level constructor functions (e.g. `abs-if`, `abs-while`, etc.) which allow a natural expression of the PICO abstract syntax tree. These constructor functions are defined in terms of `Atrees`.

6.3.1.b. Structure diagrams




Booleans
Integers
PICO-atree-
operators
Operators
Operators
Atrees
Entries
Tables
Atree-
environments
PICO-atree-
environments


Booleans
Integers
Characters
Items
Sequences
Strings
Booleans
PICO-types
PICO-atree-
operators
Operators
Atree-
environments
PICO-atree-
environments
PICO-abstract-syntax

6.3.1.c. Specification

```
module PICO-atree-operators
begin
   exports
        begin
           sorts PICO-OPERATOR

           functions

                op-pico-program        :               -> PICO-OPERATOR
                op-decls               :               -> PICO-OPERATOR
                op-empty-decls         :               -> PICO-OPERATOR
                op-series              :               -> PICO-OPERATOR
                op-empty-series        :               -> PICO-OPERATOR
                op-assign              :               -> PICO-OPERATOR
                op-if                  :               -> PICO-OPERATOR
                op-while               :               -> PICO-OPERATOR
                op-plus                :               -> PICO-OPERATOR
                op-conc                :               -> PICO-OPERATOR
                op-var                 :               -> PICO-OPERATOR
                op-integer-constant    :               -> PICO-OPERATOR
                op-string-constant     :               -> PICO-OPERATOR
                op-id                  :               -> PICO-OPERATOR
                op-integer-type        :               -> PICO-OPERATOR
                op-string-type         :               -> PICO-OPERATOR

                ord             : PICO-OPERATOR                -> INTEGER
                eq              : PICO-OPERATOR # PICO-OPERATOR -> BOOL
        end

   imports Booleans, Integers

   variables

        c1, c2 :-> PICO-OPERATOR

   equations

   [265]  ord(op-pico-program)           = 0
   [266]  ord(op-decls)                  = succ(ord(op-pico-program))
   [267]  ord(op-empty-decls)            = succ(ord(op-decls))
   [268]  ord(op-series)                 = succ(ord(op-empty-decls))
   [269]  ord(op-empty-series)           = succ(ord(op-series))
   [270]  ord(op-assign)                 = succ(ord(op-empty-series))
   [271]  ord(op-if)                     = succ(ord(op-assign))
   [272]  ord(op-while)                  = succ(ord(op-if))
   [273]  ord(op-plus)                   = succ(ord(op-while))
   [274]  ord(op-conc)                   = succ(ord(op-plus))
```

```
   [275]  ord(op-var)                    = succ(ord(op-conc))
   [276]  ord(op-integer-constant)       = succ(ord(op-var))
   [277]  ord(op-string-constant)        = succ(ord(op-integer-constant))
   [278]  ord(op-id)                     = succ(ord(op-string-constant))
   [279]  ord(op-integer-type)           = succ(ord(op-id))
   [280]  ord(op-string-type)            = succ(ord(op-integer-type))

   [281]  eq(c1, c2)                     = eq(ord(c1), ord(c2))

end PICO-atree-operators

module PICO-atree-environments
begin

   imports Atree-environments
        { renamed by
                [ ATREE -> PICO-ATREE,
                  atree -> pico-atree,
                  null-atree -> null-pico-atree,
                  error-atree -> error-pico-atree,
                  string-atree -> string-pico-atree,
                  integer-atree -> integer-pico-atree,
                  lexical-atree -> lexical-pico-atree,
                  ATREE-ENV -> PICO-ATREE-ENV,
                  null-atree-env -> null-pico-atree-env]
          Operators bound by
                [ OPERATOR -> PICO-OPERATOR,
                  eq -> eq]
                to PICO-atree-operators
        }

   variables
        s :-> STRING
        e :-> PICO-ATREE-ENV
        f :-> BOOL
        v :-> PICO-ATREE

   equations

   [282]  s ^ e         = v
                          when <f, v> = lookup(s, e)


end PICO-atree-environments

module PICO-abstract-syntax
begin
   exports
        begin
           sorts  PICO-PROGRAM, DECLS, EXP, ID, SERIES, STATEMENT
```

```
    functions

        abs-pico-program       : DECLS # SERIES             -> PICO-PROGRAM
        abs-decls              : ID # PICO-TYPE # DECLS     -> DECLS
        abs-empty-decls        :                            -> DECLS
        abs-series             : STATEMENT # SERIES         -> SERIES
        abs-empty-series       :                            -> SERIES
        abs-assign             : ID  # EXP                  -> STATEMENT
        abs-if                 : EXP # SERIES # SERIES      -> STATEMENT
        abs-while              : EXP # SERIES               -> STATEMENT
        abs-plus               : EXP # EXP                  -> EXP
        abs-conc               : EXP # EXP                  -> EXP
        abs-var                : ID                         -> EXP
        abs-integer-constant   : INTEGER                    -> EXP
        abs-string-constant    : STRING                     -> EXP
        abs-id                 : STRING                     -> ID

        pico-program           : PICO-ATREE                 -> PICO-PROGRAM
        decls                  : PICO-ATREE                 -> DECLS
        series                 : PICO-ATREE                 -> SERIES
        statement              : PICO-ATREE                 -> STATEMENT
        exp                    : PICO-ATREE                 -> EXP
        id                     : PICO-ATREE                 -> ID

        pico-type-atree        : PICO-TYPE                  -> PICO-ATREE

        append-statement       : SERIES # STATEMENT         -> SERIES
    end

imports Integers, Strings, PICO-types, PICO-atree-environments

variables

    ds                 :-> PICO-ATREE
    sr, sr1, sr2       :-> PICO-ATREE
    st                 :-> PICO-ATREE
    i                  :-> PICO-ATREE
    t                  :-> PICO-TYPE
    x, x1, x2          :-> PICO-ATREE
    str                :-> STRING
    n                  :-> INTEGER
    stat, stat1, stat2 :-> STATEMENT
    ser                :-> SERIES

equations

[283]  abs-pico-program(decls(ds), series(sr))
                           = pico-program(pico-atree(op-pico-program, ds, sr))
[284]  abs-decls(id(i), t, decls(ds))
                           = decls(pico-atree(op-decls, i, pico-type-atree(t), ds))
```

```
[285]  abs-empty-decls        = decls(pico-atree(op-empty-decls))
[286]  abs-series(statement(st), series(sr))
                              = series(pico-atree(op-series, st, sr))
[287]  abs-empty-series       = series(pico-atree(op-empty-series))
[288]  abs-assign(id(i), exp(x))
                              = statement(pico-atree(op-assign, i, x))
[289]  abs-if(exp(x), series(sr1), series(sr2))
                              = statement(pico-atree(op-if, x, sr1, sr2))
[290]  abs-while(exp(x), series(sr))
                              = statement(pico-atree(op-while, x, sr))
[291]  abs-plus(exp(x1), exp(x2))
                              = exp(pico-atree(op-plus, x1, x2))
[292]  abs-conc(exp(x1), exp(x2))
                              = exp(pico-atree(op-conc, x1, x2))
[293]  abs-var(id(i))         = exp(pico-atree(op-var, i))
[294]  abs-integer-constant(n)
                              = exp(pico-atree(op-integer-constant,
                                               integer-pico-atree(n)))
[295]  abs-string-constant(str)
                              = exp(pico-atree(op-string-constant,
                                               string-pico-atree(str)))
[296]  abs-id(str)            = id(pico-atree(op-id, string-pico-atree(str)))


[297]  append-statement(abs-empty-series, stat)
                              = abs-series(stat, abs-empty-series)
[298]  append-statement(abs-series(stat1, ser), stat2)
                              = abs-series(stat1, append-statement(ser, stat2))

end PICO-abstract-syntax
```

6.3.2. Concrete syntax and rules for abstract syntax tree construction

6.3.2.a. Global description

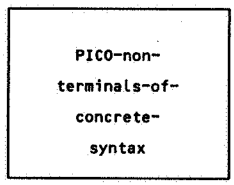
In this section the concrete syntax and the rules for abstract syntax tree construction for PICO are defined. This involves the following modules:

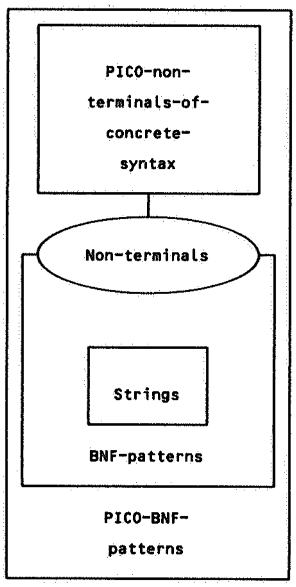
`PICO-non-terminals-of-concrete-synyax`: defines the sort `PICO-NON-TERMINAL` and all non-terminals of the concrete syntax.

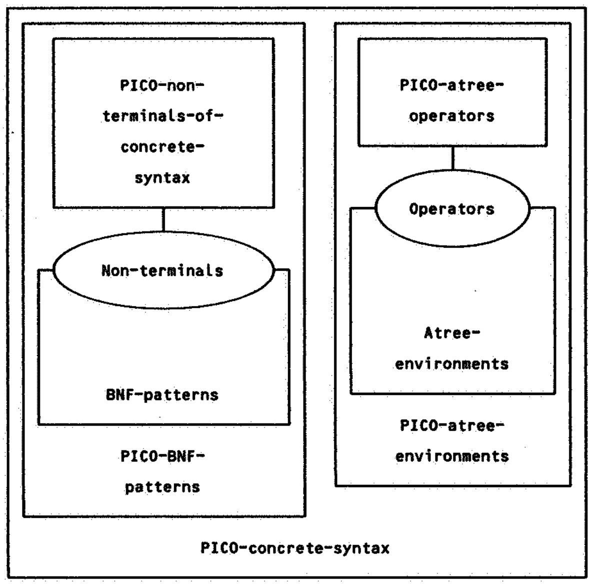
`PICO-BNF-patterns`: defines a version of `BNF-patterns` with parameter `Non-terminals` bound to `PICO-non-terminals-of-concrete-syntax`.

`PICO-concrete-syntax`: defines the concrete syntax for PICO and the rules for abstract syntax tree construction. Essentially the grammar contains for each non-terminal in the concrete syntax pairs of equations for the functions `rule` (i.e. the actual syntax rule) and `build` (i.e. the construction procedure for abstract syntax trees).

6.3.2.b. Structure diagrams







6.3.2.c. Specification

```
module PICO-non-terminals-of-concrete-syntax
begin
   exports
        begin
           sorts PICO-NON-TERMINAL

           functions
                pico-program    :                 -> PICO-NON-TERMINAL
                decls           :                 -> PICO-NON-TERMINAL
                empty-decls     :                 -> PICO-NON-TERMINAL
                id-type-list    :                 -> PICO-NON-TERMINAL
                type            :                 -> PICO-NON-TERMINAL
                type-integer    :                 -> PICO-NON-TERMINAL
                type-string     :                 -> PICO-NON-TERMINAL
                series          :                 -> PICO-NON-TERMINAL
                empty-series    :                 -> PICO-NON-TERMINAL
                non-empty-series:                 -> PICO-NON-TERMINAL
                stat            :                 -> PICO-NON-TERMINAL
                assign          :                 -> PICO-NON-TERMINAL
                if              :                 -> PICO-NON-TERMINAL
                while           :                 -> PICO-NON-TERMINAL
                exp             :                 -> PICO-NON-TERMINAL
                plus            :                 -> PICO-NON-TERMINAL
                conc            :                 -> PICO-NON-TERMINAL
                var             :                 -> PICO-NON-TERMINAL
                id              :                 -> PICO-NON-TERMINAL
                integer-constant:                 -> PICO-NON-TERMINAL
```

```
                string-constant :                -> PICO-NON-TERMINAL
        end

end PICO-non-terminals-of-concrete-syntax

module PICO-BNF-patterns
begin
   imports BNF-patterns
        { renamed by
                [ PATTERN -> PICO-PATTERN,
                 t -> pt,
                 lexical -> plexical ]
          Non-terminals bound by
                [ NON-TERMINAL -> PICO-NON-TERMINAL ]
                to PICO-non-terminals-of-concrete-syntax
        }

end PICO-BNF-patterns

module PICO-concrete-syntax
begin
   exports
        begin
           functions
                rule :  PICO-NON-TERMINAL                -> PICO-PATTERN
                build: PICO-NON-TERMINAL # PICO-ATREE-ENV -> PICO-ATREE
        end

   imports PICO-BNF-patterns, PICO-atree-environments

   variables
        env     :-> PICO-ATREE-ENV
        str     :-> STRING


   equations


   [299] rule(pico-program)     = pt("begin") + n(decls,"d")
                                              + n(series,"s") + pt("end")
   [300] build(pico-program, env)
                                = pico-atree(op-pico-program, "d" ^ env, "s" ^ env)

   [301] rule(decls)            = pt("declare") + n(id-type-list,"l") + pt(";")
   [302] build(decls, env)      = "l" ^ env

   [303] rule(empty-decls)      = pt("")
   [304] build(empty-decls, env)= pico-atree(op-empty-decls)

   [305] rule(id-type-list)     = n(id,"i") + pt(":") + n(type,"t") +
```

```
                                  ( n(empty-decls,"l") |
                                    pt(",") + n(id-type-list,"l")
                                  )
[306] build(id-type-list, env)
                              = pico-atree(op-decls,
                                    "i" ^ env,
                                    "t" ^ env,
                                    "l" ^ env)

[307] rule(type)              = n(type-integer,"t") | n(type-string,"t")
[308] build(type, env)        = "t" ^ env

[309] rule(type-integer)      = pt("integer")
[310] build(type-integer, env)
                              = pico-atree(op-integer-type)

[311] rule(type-string)       = pt("string")
[312] build(type-string, env)
                              = pico-atree(op-string-type)

[313] rule(series)            = n(empty-series,"s") | n(non-empty-series,"s")
[314] build(series, env)      = "s" ^ env

[315] rule(empty-series)      = pt("")
[316] build(empty-series, env)
                              = pico-atree(op-empty-series)

[317] rule(non-empty-series) = n(stat,"st") + ( n(empty-series,"s") |
                                                 pt(";") + n(series,"s")
                                               )
[318] build(non-empty-series, env)
                              = pico-atree(op-series, "st" ^ env, "s" ^ env)
[319] rule(stat)              = n(assign,"st") | n(if,"st") | n(while,"st")
[320] build(stat, env)        = "st" ^ env

[321] rule(assign)            = n(id,"i") + pt(":=") + n(exp,"e")
[322] build(assign,env)       = pico-atree(op-assign, "i" ^ env, "e" ^ env)

[323] rule(if)                = pt("if") + n(exp,"e")
                                        + pt("then") + n(series,"s1")
                                        + pt("else") + n(series,"s2") + pt("fi")
[324] build(if, env)          = pico-atree(op-if,
                                      "e" ^ env,
                                      "s1" ^ env,
                                      "s2" ^ env)

[325] rule(while)             = pt("while") + n(exp,"e")
                                           + pt("do") + n(series,"s") + pt("od")
[326] build(while, env)       = pico-atree(op-while, "e" ^ env, "s" ^ env)
```

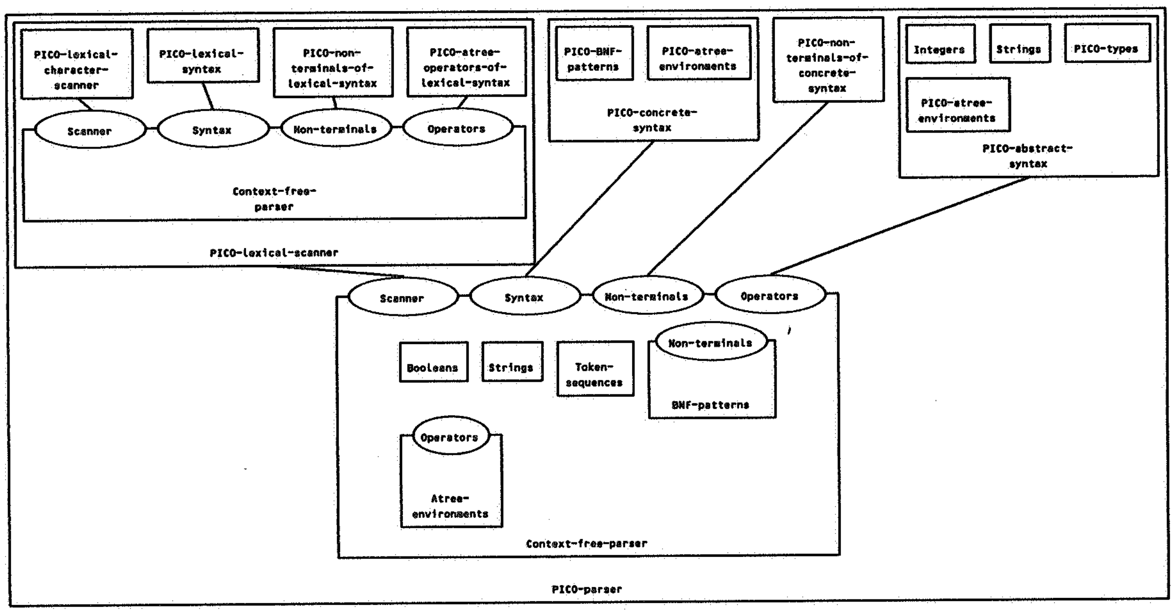
```
[327] rule(exp)              = n(var,"e") |
                               n(integer-constant,"e") |
                               n(string-constant,"e") |
                               n(plus,"e") |
                               n(conc,"e") |
                               ( pt("(") + n(exp,"e") + pt(")") )
[328] build(exp, env)        = "e" ^ env

[329] rule(plus)             = n(exp,"e1") + pt("+") + n(exp,"e2")
[330] build(plus, env)       = pico-atree(op-plus, "e1" ^ env, "e2" ^ env)

[331] rule(conc)             = n(exp,"e1") + pt("||") + n(exp,"e2")
[332] build(conc, env)       = pico-atree(op-conc, "e1" ^ env, "e2" ^ env)
[333] rule(var)              = n(id,"i")
[334] build(var, env)        = pico-atree(op-var, "i" ^ env)

[335] rule(id)               = plexical("id","i")
[336] build(id, env)         = pico-atree(op-id, string-pico-atree(str))
                               when lexical-pico-atree(token("id", str)) = "i" ^ env

[337] rule(integer-constant) = plexical("integer-constant","i")

[338] build(integer-constant, env)
                             = pico-atree(op-integer-constant,
                                          integer-pico-atree(str-to-int(str)))
                               when lexical-pico-atree(token("integer-constant", str))
                                            = "i" ^ env

[339] rule(string-constant)  = plexical("string-constant","s")
[340] build(string-constant, env)
                             = pico-atree(op-string-constant, string-pico-atree(str))
                               when lexical-pico-atree(token("string-constant", str))
                                            = "s" ^ env

end PICO-concrete-syntax
```

6.3.3. Parser

6.3.3.a. Global description

In this section a parser for PICO is obtained by combining `PICO-lexical-scanner`, `PICO-concrete-syntax`, `PICO-non-terminals-of-concrete-syntax` and `PICO-atree-operators-of-concrete-syntax` with `Context-free-parser`.

6.3.3.b. Structure diagram



6.3.3.c. Specification

```
module PICO-parser
begin
   exports
        begin
           functions
                parse-and-construct : STRING -> PICO-ATREE
        end

   imports Context-free-parser
        { Scanner bound by
                [ scan -> lex-scan ]
                to PICO-lexical-scanner
          Syntax bound by
                [ rule -> rule,
                  build -> build ]
                to PICO-concrete-syntax
          Non-terminals bound by
```

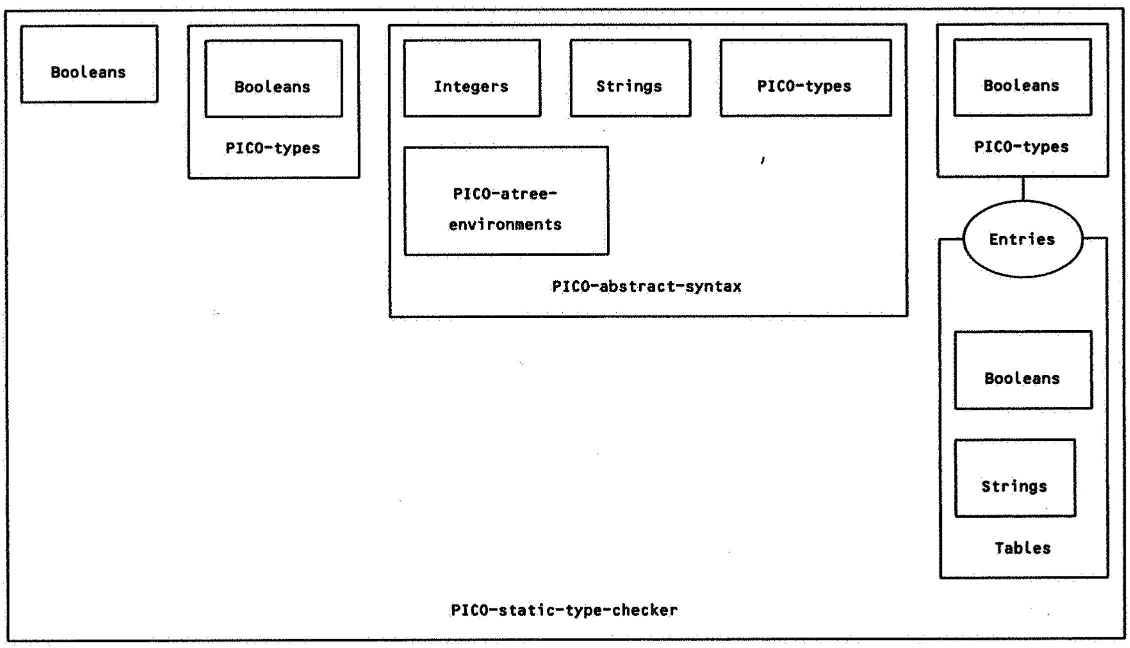
```
                [ NON-TERMINAL -> PICO-NON-TERMINAL ]
                to PICO-non-terminals-of-concrete-syntax
          Operators bound by
                [ OPERATOR -> PICO-OPERATOR,
                  eq -> eq ]
                to PICO-abstract-syntax
        }

   variables
        str :-> STRING

   equations

   [341] parse-and-construct(str) = parse(pico-program, str)

end PICO-parser
```

6.4. Static semantics

6.4.a. Global description

In this section we specify the checking of static semantic constraints on PICO programs as defined informally in section 3. The principal function is `check` which operates on an abstract PICO program and checks whether this program is in accordance with the static semantic constraints. For each construct in the abstract syntax tree these constraints are expressed as transformations on a type-environment. Type-environments are defined as a combination of `Tables` and `PICO-types`. Checking the declaration section of a PICO program amounts to constructing a type-environment, and checking the statement section amounts to checking each statement for conformity with a given type-environment.

6.4.b. Structure diagram



6.4.c. Specification

```
module PICO-static-type-checker
begin
   exports
      begin
         functions
               check: PICO-PROGRAM              -> BOOL
               check: DECLS # TYPE-ENV          -> (BOOL # TYPE-ENV)
               check: SERIES # TYPE-ENV         -> (BOOL # TYPE-ENV)
               check: STATEMENT # TYPE-ENV      -> (BOOL # TYPE-ENV)
      end
```

```
imports Booleans, PICO-types, PICO-abstract-syntax,
        Tables
             { renamed by
                     [ TABLE -> TYPE-ENV,
                       null-table -> null-type-env ]
               Entries bound by
                     [ ENTRY -> PICO-TYPE,
                       eq -> eq,
                       error-entry -> error-type ]
                     to PICO-types
             }
functions
     type-of-exp      : EXP # TYPE-ENV        -> PICO-TYPE

variables
     dec : -> DECLS
     ser, ser1, ser2 : -> SERIES
     stat : -> STATEMENT
     name : -> STRING
     int : -> INTEGER
     typ : -> PICO-TYPE
     str : -> STRING
     x, x1, x2 : -> EXP
     env, env1, env2 : -> TYPE-ENV
     b, b1, b2, found : -> BOOL

equations

[342]  check(abs-pico-program(dec, ser))
                     = and(b1, b2)
                       when  <b1, env1> = check(dec, null-type-env),
                             <b2, env2> = check(ser, env1)

[343]  check(abs-decls(abs-id(name), typ, dec), env)
                     = check(dec, table(name, typ, env))
[344]  check(abs-empty-decls, env)
                     = < true, env >

[345]  check(abs-series(stat, ser), env)
                     =  < and(b1, b2), env2 >
                       when  <b1, env1> = check(stat, env),
                             <b2, env2> = check(ser, env1)
[346]  check(abs-empty-series, env)
                     = < true, env >

[347]  check(abs-assign(abs-id(name), x), env)
                     = < and(found, eq(typ, type-of-exp(x, env))), env >
                       when <found, typ> = lookup(name, env)
[348]  check(abs-if(x, ser1, ser2), env)
                     = < and(eq(type-of-exp(x,env),integer-type), and(b1,b2)),
```

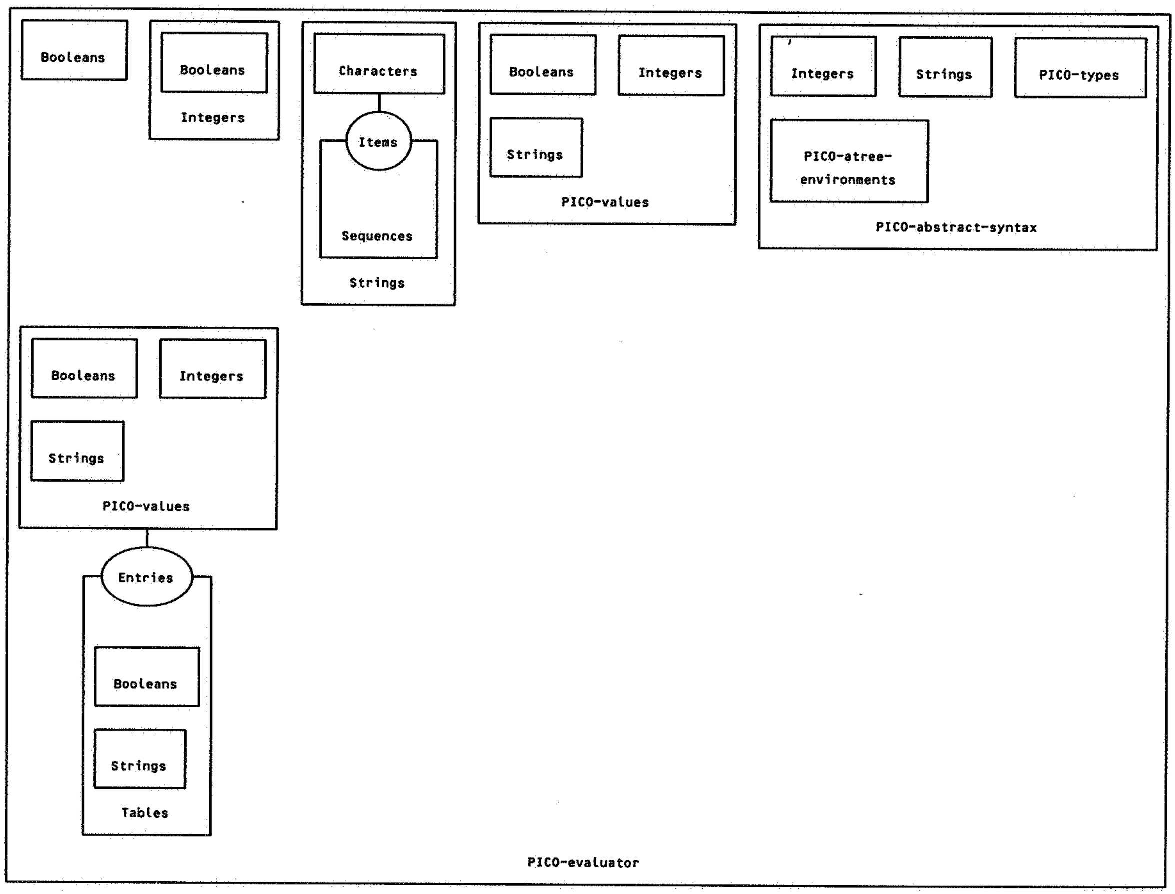
```
                     env2 >
                   when  <b1, env1> = check(ser1, env),
                         <b2, env2> = check(ser2, env1)
[349]  check(abs-while(x, ser), env)
                 = < and(eq(type-of-exp(x, env), integer-type), b),
                     env1 >
                   when <b, env1> = check(ser, env)
[350]  type-of-exp(abs-plus(x1, x2), env)
                 = if(and(eq(type-of-exp(x1, env), integer-type),
                          eq(type-of-exp(x2, env), integer-type)),
                      integer-type,
                      error-type)

[351]  type-of-exp(abs-conc(x1, x2), env)
                 = if(and(eq(type-of-exp(x1, env), string-type),
                          eq(type-of-exp(x2, env), string-type)),
                      string-type,
                      error-type)

[352]  type-of-exp(abs-integer-constant(int), env)
                 = integer-type
[353]  type-of-exp(abs-string-constant(str), env)
                 = string-type
[354]  type-of-exp(abs-var(abs-id(name)), env)
                 = if(found, typ, error-type)
                   when <found, typ> = lookup(name, env)

end PICO-static-type-checker
```

6.5. Dynamic semantics

6.5.a. Global description

In this section the evaluation of PICO programs is defined. To a first approximation, the evaluation of programs is defined by defining the evaluation of each kind of construct that may appear in the abstract syntax tree. Evaluation is expressed as transformation on value-environments which describe the values of the variables in the program. Value-environments are defined as combinations of `Tables` and `PICO-values`. However, since programs need not terminate this would make the evaluation function a partial function. Therefore, we introduce the notion of a `program-state` and define program evaluation as a function from program-states to program-states. This transformation of program-states can be described by a total function. The cases in which programs do not terminate are covered by conditional equations: conditions appearing in the `when`-parts of equations which describe the evaluation of a certain language construct enforce the evaluation of that construct to be only defined if the evaluation of all of its components terminates.

6.5.b. Structure diagram



6.5.c. Specification

```
module PICO-evaluator
begin

   exports
      begin
         sorts       PROGRAM-STATE

         functions
            program-state  : PICO-PROGRAM             -> PROGRAM-STATE
            program-state  : SERIES # VALUE-ENV       -> PROGRAM-STATE
            program-state  : STATEMENT # VALUE-ENV    -> PROGRAM-STATE
            program-state  : EXP # VALUE-ENV          -> PROGRAM-STATE
            program-state  : VALUE-ENV                -> PROGRAM-STATE

            eval           : PROGRAM-STATE            -> PROGRAM-STATE
            eval-decls     : DECLS # VALUE-ENV        -> VALUE-ENV
            eval-exp       : EXP # VALUE-ENV          -> PICO-VALUE
      end

   imports Booleans, Integers, Strings, PICO-values, PICO-abstract-syntax,
         Tables
            { renamed by
                    [ TABLE -> VALUE-ENV,
                      null-table -> null-value-env]
              Entries bound by
                    [ ENTRY -> PICO-VALUE,
                      eq -> eq,
                      error-entry -> error-value]
                    to PICO-values
            }
   variables
      dec : -> DECLS
      ser, ser1, ser2 : -> SERIES
      stm : -> STATEMENT
      name : -> STRING
      int, int1, int2 : -> INTEGER
      val, val1, val2 : -> PICO-VALUE
      str, str1, str2 : -> STRING
      x, x1, x2 : -> EXP
      env, env1, env2 : -> VALUE-ENV
      found : -> BOOL

   equations

   [355]  eval(program-state(abs-pico-program(dec, ser)))
                = eval(program-state(ser, eval-decls(dec, null-value-env)))

   [356]  eval-decls(abs-decls(abs-id(name), integer-type, dec), env)
```

```
              = eval-decls(dec, table(name, pico-value(0), env))
[357]   eval-decls(abs-decls(abs-id(name), string-type, dec), env)
              = eval-decls(dec, table(name, pico-value(null-string), env))
[358]   eval-decls(abs-empty-decls, env)
              = env

[359]   eval(program-state(abs-series(stm, ser), env))
              = eval(program-state(ser, env1))
                when  eval(program-state(stm, env)) = program-state(env1)

[360]   eval(program-state(abs-empty-series, env))
              = program-state(env)

[361]   eval(program-state(abs-assign(abs-id(name), x), env))
              = program-state(table(name, eval-exp(x, env), env))

[362]   eval(program-state(abs-if(x, ser1, ser2), env))
              = if(eq(eval-exp(x,env), pico-value(0)),
                   eval(program-state(ser2, env)),
                   eval(program-state(ser1, env)))

[363]   eval(program-state(abs-while(x, ser), env))
              = if(eq(eval-exp(x,env), pico-value(0)),
                   program-state(env),
                   eval(program-state(append-statement(ser, abs-while(x,ser)),
                                       env)))

[364]   eval-exp(abs-plus(x1, x2), env)
              = pico-value(add(int1,int2))
                when  pico-value(int1) = eval-exp(x1,env),
                      pico-value(int2) = eval-exp(x2,env)

[365]   eval-exp(abs-conc(x1, x2), env)
              = pico-value(conc(str1, str2))
                when  pico-value(str1) = eval-exp(x1, env),
                      pico-value(str2) = eval-exp(x2, env)

[366]   eval-exp(abs-integer-constant(int), env)
              = pico-value(int)

[367]   eval-exp(abs-string-constant(str), env)
              = pico-value(str)

[368]   eval-exp(abs-var(abs-id(name)), env)
              = val
                when <found, val> = lookup(name, env)

end PICO-evaluator
```

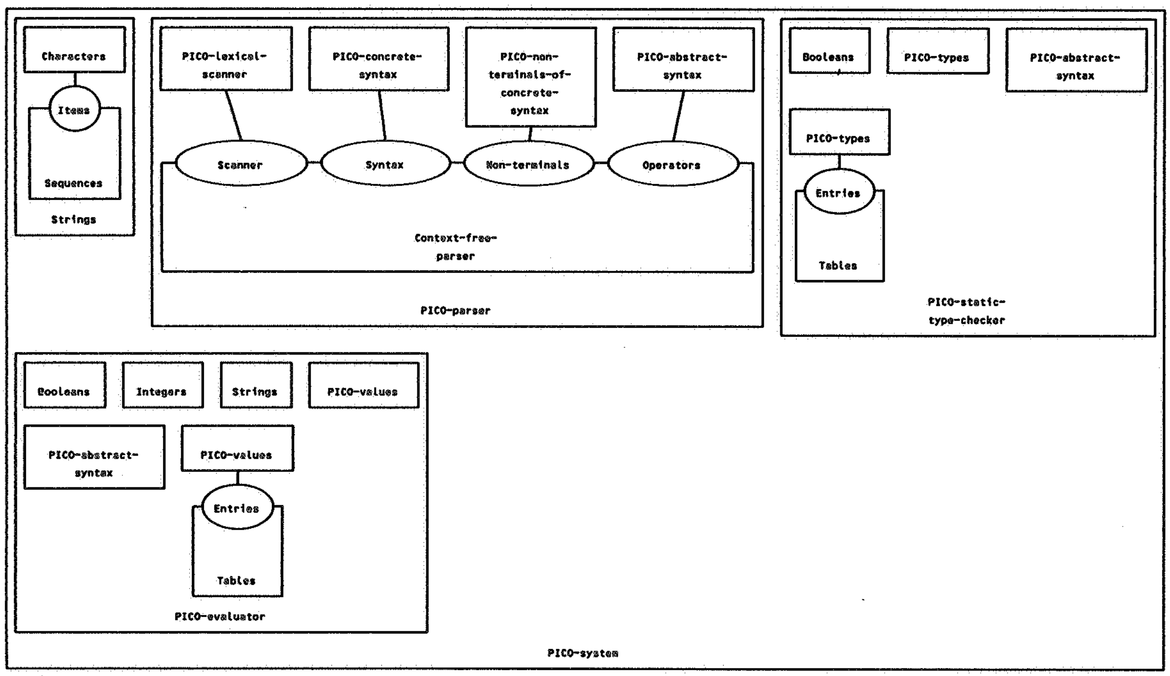
6.6. The PICO system

6.6.a. Global description

In this final section we combine all previously defined modules to form a PICO system. The top level function is `run` which converts, if this is possible, a string into a PICO-value. The following steps are necessary:

1) The input string is parsed and converted into an abstract syntax tree using `parse-and-construct` as defined in `PICO-parser`.
2) The types of the, syntactically correct, program are checked using `check` as defined in `PICO-static-type-checker`.
3) The, statically correct, program is evaluated using `eval` as defined in `PICO-evaluator`. If this evaluation terminates it produces a value-environment. The result of evaluating the original program is the final value of the variable `output` as extracted from this value-environment.

6.6.b. Structure diagram



6.6.c. Specification

```
module PICO-system
begin
   exports
       begin
          functions
               run:    STRING  -> PICO-VALUE
       end
```

```
imports Strings, PICO-parser, PICO-static-type-checker, PICO-evaluator

functions
     run1: PICO-ATREE          -> PICO-VALUE
     run2: PICO-PROGRAM        -> PICO-VALUE

variables
     s        : -> STRING
     p        : -> PICO-ATREE
     abs-prog: -> PICO-PROGRAM
     has-output: -> BOOL
     v        : -> PICO-VALUE
     env      : -> VALUE-ENV

equations

[369]   run(s)                      = run1(parse-and-construct(s))

[370]   run1(error-pico-atree)      = error-value
[371]   run1(p)                     = if(check(pico-program(p)),
                                         run2(pico-program(p)),
                                         error-value)

[372]   run2(abs-prog)              = if(has-output, v, error-value)

                                      when  program-state(env) =
                                                    eval(program-state(abs-prog)),
                                            <has-output, v> =
                                                    lookup("output", env)

end PICO-system
```

7. LITERATURE

[BBC83] Bowen, D.L., Byrd, L.M. & Clocksin, W.F., "A portable PROLOG compiler", Proceedings of the Logic Programming Workshop 1983, Portugal, 74 - 83.

[BHK84] Bergstra, J.A., Heering, J. & Klop, J.W., "Object-oriented algebraic specifications: proposal for a notation and 12 examples", Centre for Mathematics and Computer Science, Report CS-R8411, 1984.

[BIE84] Biebow, B., "Specification of a telephone subscriber connection unit using abstract algebraic data types in the language PLUSS", Laboratoire de Marcoussi, Centre de Recherche de la C.G.E., France, 1984.

[BK81] Bergstra, J.A. & Klop, J.W., "Conditional rewrite rules", Centre for Mathematics and Computer Science, Report IW198/82, 1982.

[BO81] Bothe, K., "Restructuring a compiler by abstract data types — an experiment in using abstractions for software modularization", Humboldt University Berlin, Seminar Bericht Nr. 40, 1981.

[BT79] Bergstra, J.A. & Tucker, J.V., "Algebraic specifications of computable and semi-computable data structures", Centre for Mathematics and Computer Science, Report IW 115/79, 1979.

[DE84] Drosten, K. & Ehrich, H.-D., "Translating algebraic specifications to PROLOG programs", Informatik Bericht Nr. 84-08, Technische Universität Braunschweig, 1984.

[GAU80] Gaudel, M.C., "Specification of compilers as abstract data type representations", Springer Lecture Notes in Computer Science, Volume 94, 1980.

[GAU84] Gaudel, M.C., "Introduction to PLUSS", draft document, Paris, 1984.

[GAN82] Ganzinger, H., "Denotational semantics for languages with modules", Proceedings of IFIP Working Conference *Formal Description of Programming Concepts*, North-Holland, 1982.

[GM82] Goguen, J.A. & Meseguer, J., "An Initiality Primer", in Nivat, M., & Reynolds, J., (eds), *Application of Algebra to Language Definition and Compilation*, North-Holland, 1983.

[GP81] Goguen, J.A. & Parsaye-Ghomi, K., "Algebraic denotational semantics using parameterized abstract modules", in Diaz, J. & Ramos, I. (eds.) *Formalizing Programming Concepts*, Springer Lecture Notes in Computer Science, Volume 107, 1981, 292-309,

[HEE85] Heering, J., "Partial evaluation and ω-completeness of algebraic specifications", Centre for Mathematics and Computer Science, Report CS-R8501, 1985.

[HOD82] Hoffmann, C.M. & O'Donnell, M.J., "Programming with equations", ACM Transactions on Programming Languages and Systems, 4(1982)1, 83-112.

[KLA83] Klaeren, H.A., *Algebraïsche Spezifikationen: Eine Einführung*, Springer-Verlag, 1983.

[KL83] Kutzler, B. & Lichtenberg, F., "Bibliography on abstract data types", Informatik Fachberichte 68, Springer, 1983.

[LOE84] Loeckx, J., "Algorithmic specifications: a constructive method for abstract data types", Report A84/03, Universität des Saarlandes, 1984.

[RZ84] Remy, J.L. & Zhang, H., "Reveur4: a system for validating conditional algebraic specifications of abstract data types", *European Conference on Artificial Intelligence 84*, 563-572, 1984.

[W83] Wirsing, M., "A Specification Language", Dissertation, Münich University, 1983.

APPENDIX A.1. Dependency hierarchy of modules

Module	imports the modules
Atree-environments:	Tables
Atrees:	Booleans, Integers, Strings, Tokens
BNF-patterns:	Strings
Booleans:	-
Characters:	Booleans, Integers
Context-free-parser:	Atree-environments, BNF-patterns, Booleans, Strings, Token-sequences
Integers:	Booleans
PICO-BNF-patterns:	BNF-patterns
PICO-abstract-syntax:	Integers, PICO-atree-environments, PICO-types, Strings
PICO-atree-environments:	Atree-environments
PICO-atree-operators:	Booleans, Integers
PICO-atree-operators-of-lexical-syntax:	Booleans
PICO-concrete-syntax:	PICO-BNF-patterns, PICO-atree-environments
PICO-evaluator:	Booleans, Integers, PICO-abstract-syntax, PICO-values, Strings, Tables
PICO-lex-BNF-patterns:	BNF-patterns
PICO-lex-atree-environments:	Atree-environments
PICO-lexical-character-scanner:	Booleans, Characters, Strings, Token-sequences
PICO-lexical-scanner:	Context-free-parser
PICO-lexical-syntax:	PICO-lex-BNF-patterns, PICO-lex-atree-environments, Token-sequences
PICO-non-terminals-of-concrete-syntax:	-
PICO-non-terminals-of-lexical-syntax:	-
PICO-parser:	Context-free-parser
PICO-static-type-checker:	Booleans, PICO-abstract-syntax, PICO-types, Tables
PICO-system:	PICO-evaluator, PICO-parser, PICO-static-type-checker, Strings
PICO-types:	Booleans
PICO-values:	Booleans, Integers, Strings

Sequences:	Booleans
Strings:	Sequences
Tables:	Booleans, Strings
Token-sequences:	Sequences
Tokens:	Booleans, Strings

APPENDIX A.2. Declaration of sorts per module

Module	declares the sorts
Atree-environments:	-
Atrees:	ATREE, OPERATOR
BNF-patterns:	NON-TERMINAL, PATTERN
Booleans:	BOOL
Characters:	CHAR
Context-free-parser:	-
Integers:	INTEGER
PICO-BNF-patterns:	-
PICO-abstract-syntax:	DECLS, EXP, ID, PICO-PROGRAM, SERIES, STATEMENT
PICO-atree-environments:	-
PICO-atree-operators:	PICO-OPERATOR
PICO-atree-operators-of-lexical-syntax:	LEX-OPERATOR
PICO-concrete-syntax:	-
PICO-evaluator:	PROGRAM-STATE
PICO-lex-BNF-patterns:	-
PICO-lex-atree-environments:	-
PICO-lexical-character-scanner:	-
PICO-lexical-scanner:	-
PICO-lexical-syntax:	-
PICO-non-terminals-of-concrete-syntax:	PICO-NON-TERMINAL
PICO-non-terminals-of-lexical-syntax:	LEX-NON-TERMINAL
PICO-parser:	-
PICO-static-type-checker:	-
PICO-system:	-
PICO-types:	PICO-TYPE
PICO-values:	PICO-VALUE
Sequences:	ITEM, SEQ
Strings:	-
Tables:	ENTRY, TABLE
Token-sequences:	-

Tokens: TOKEN

APPENDIX A.3. Declaration of functions per module

Module	declares the functions
Atree-environments:	_^_
Atrees:	atree, eq, error-atree, integer-atree, lexical-atree, null-atree, string-atree
BNF-patterns:	_+_, _|_, lexical, n, null-pattern, t
Booleans:	and, false, if, not, or, true
Characters:	char-0, char-1, char-2, char-3, char-4, char-5, char-6, char-7, char-8, char-9, char-A, char-B, char-C, char-D, char-E, char-F, char-G, char-H, char-I, char-J, char-K, char-L, char-M, char-N, char-O, char-P, char-Q, char-R, char-S, char-T, char-U, char-V, char-W, char-X, char-Y, char-Z, char-a, char-b, char-bar, char-c, char-colon, char-comma, char-d, char-e, char-equal, char-f, char-g, char-h, char-ht, char-i, char-j, char-k, char-l, char-lpar, char-m, char-minus, char-n, char-nl, char-o, char-p, char-plus, char-point, char-q, char-quote, char-r, char-rpar, char-s, char-semi, char-slash, char-space, char-t, char-times, char-u, char-v, char-w, char-x, char-y, char-z, eq, is-digit, is-letter, is-lower, is-upper, ord
Context-free-parser:	build, parse, parse-pat, parse-rule, rule, scan
Integers:	0, 1, 10, add, eq, greater, greatereq, less, lesseq, mul, succ
PICO-BNF-patterns:	-
PICO-abstract-syntax:	abs-assign, abs-conc, abs-decls, abs-empty-decls, abs-empty-series, abs-id, abs-if, abs-integer-constant, abs-pico-program, abs-plus, abs-series, abs-string-constant, abs-var, abs-while, append-statement, decls, exp, id, pico-program, pico-type-atree, series, statement
PICO-atree-environments:	-
PICO-atree-operators:	eq, op-assign, op-conc, op-decls, op-empty-decls, op-empty-series, op-id, op-if, op-integer-constant, op-integer-type, op-pico-program, op-plus, op-series, op-string-constant, op-string-type, op-var, op-while, ord
PICO-atree-operators-of-lexical-syntax:	eq, op-lex-item, op-lex-stream

PICO-concrete-syntax:	build, rule
PICO-evaluator:	eval, eval-decls, eval-exp, program-state
PICO-lex-BNF-patterns:	-
PICO-lex-atree-environments:	-
PICO-lexical-character-scanner:	char-scan, char-scan1, is-layout
PICO-lexical-scanner:	lex-scan
PICO-lexical-syntax:	build, lex-item, lex-stream, rule
PICO-non-terminals-of-concrete-syntax:	assign, conc, decls, empty-decls, empty-series, exp, id, id-type-list, if, integer-constant, non-empty-series, pico-program, plus, series, stat, string-constant, type, type-integer, type-string, var, while
PICO-non-terminals-of-lexical-syntax:	any-char-but-quote, assign-or-colon, concat, digit, digits, empty, empty-lexical-stream, ident, ident-char, ident-chars, integer-const, keyword-or-ident, layout, letter, lexical-item, lexical-stream, literal, non-empty-digits, non-empty-ident-chars, non-empty-lexical-stream, non-empty-string-tail, optional-layout, quote, string-const, string-tail
PICO-parser:	parse-and-construct
PICO-static-type-checker:	check, type-of-exp
PICO-system:	run, run1, run2
PICO-types:	eq, error-type, integer-type, string-type
PICO-values:	eq, error-value, pico-value
Sequences:	conc, conv-to-seq, eq, null, seq
Strings:	str-to-int
Tables:	delete, eq, error-entry, lookup, null-table, table
Token-sequences:	-
Tokens:	eq, token

APPENDIX A.4. Modules in which each function is declared

<table>
<tr><th>Function</th><th>is declared in module</th></tr>
<tr><td>0:</td><td>Integers</td></tr>
<tr><td>1:</td><td>Integers</td></tr>
<tr><td>10:</td><td>Integers</td></tr>
<tr><td>_+_:</td><td>BNF-patterns</td></tr>
<tr><td>_^_:</td><td>Atree-environments</td></tr>
<tr><td>_|_:</td><td>BNF-patterns</td></tr>
<tr><td>abs-assign:</td><td>PICO-abstract-syntax</td></tr>
<tr><td>abs-conc:</td><td>PICO-abstract-syntax</td></tr>
<tr><td>abs-decls:</td><td>PICO-abstract-syntax</td></tr>
<tr><td>abs-empty-decls:</td><td>PICO-abstract-syntax</td></tr>
<tr><td>abs-empty-series:</td><td>PICO-abstract-syntax</td></tr>
<tr><td>abs-id:</td><td>PICO-abstract-syntax</td></tr>
<tr><td>abs-if:</td><td>PICO-abstract-syntax</td></tr>
<tr><td>abs-integer-constant:</td><td>PICO-abstract-syntax</td></tr>
<tr><td>abs-pico-program:</td><td>PICO-abstract-syntax</td></tr>
<tr><td>abs-plus:</td><td>PICO-abstract-syntax</td></tr>
<tr><td>abs-series:</td><td>PICO-abstract-syntax</td></tr>
<tr><td>abs-string-constant:</td><td>PICO-abstract-syntax</td></tr>
<tr><td>abs-var:</td><td>PICO-abstract-syntax</td></tr>
<tr><td>abs-while:</td><td>PICO-abstract-syntax</td></tr>
<tr><td>add:</td><td>Integers</td></tr>
<tr><td>and:</td><td>Booleans</td></tr>
<tr><td>any-char-but-quote:</td><td>PICO-non-terminals-of-lexical-syntax</td></tr>
<tr><td>append-statement:</td><td>PICO-abstract-syntax</td></tr>
<tr><td>assign:</td><td>PICO-non-terminals-of-concrete-syntax</td></tr>
<tr><td>assign-or-colon:</td><td>PICO-non-terminals-of-lexical-syntax</td></tr>
<tr><td>atree:</td><td>Atrees</td></tr>
<tr><td>build:</td><td>Context-free-parser, PICO-concrete-syntax, PICO-lexical-syntax</td></tr>
<tr><td>char-0:</td><td>Characters</td></tr>
<tr><td>char-1:</td><td>Characters</td></tr>
<tr><td>char-2:</td><td>Characters</td></tr>
</table>

char-c:	Characters
char-colon:	Characters
char-comma:	Characters
char-d:	Characters
char-e:	Characters
char-equal:	Characters
char-f:	Characters
char-g:	Characters
char-h:	Characters
char-ht:	Characters
char-i:	Characters
char-j:	Characters
char-k:	Characters
char-l:	Characters
char-lpar:	Characters
char-m:	Characters
char-minus:	Characters
char-n:	Characters
char-nl:	Characters
char-o:	Characters
char-p:	Characters
char-plus:	Characters
char-point:	Characters
char-q:	Characters
char-quote:	Characters
char-r:	Characters
char-rpar:	Characters
char-s:	Characters
char-scan:	PICO-lexical-character-scanner
char-scan1:	PICO-lexical-character-scanner
char-semi:	Characters
char-slash:	Characters
char-space:	Characters
char-t:	Characters
char-times:	Characters
char-u:	Characters