stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IW 240/83 NOVEMBER
(DEPARTMENT OF COMPUTER SCIENCE)

P. KLINT

A SURVEY OF THREE LANGUAGE-INDEPENDENT PROGRAMMING ENVIRONMENTS

Preprint

kruislaan 413 1098 SJ amsterdam

INRIA Rapport de Recherche No. 257

Printed at the Mathematical Centre, Kruislaan 413, Amsterdam, The Netherlands.

The Mathematical Centre, founded 11 February 1946, is a non-profit institution for the promotion of pure and applied mathematics and computer science. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

1980 Mathematics subject classification: 68B20, 68F05, 68F20, 68F25

1982 CR. Categories: D.2.2, D.2.3, D.2.5, D.2.6, D.3.1, D.3.4

A survey of three language-independent programming environments†

by

Paul Klint

ABSTRACT

The creation and maintenance of software is becoming increasingly expensive. To improve upon this situation several *software tools* and *language-specific programming environments* have come into existence. The substantial design and implementation effort to build a programming environment for each specific language can, however, be reduced by developing *language-independent* programming environments, which can be tailored towards a particular language by supplying them with the corresponding language definition.

This paper surveys three existing, but still experimental, language-independent programming environments: Mentor, the Synthesizer Generator and CEYX. The similarities between the three systems and their individual goals and characteristics are outlined. Using each system a programming environment for a toy language has been constructed in order to establish some basis for comparison. The results of this experiment are discussed.

KEY WORDS & PHRASES: Software Engineering, Language-Independent Programming Environments, Syntax-Directed Editing, Semantics-Directed Evaluation.

†This paper is not for review; it is intended for publication elsewhere.

1. INTRODUCTION

The creation and maintenance of software is becoming increasingly expensive. To improve upon this situation several *software tools* and *language-specific programming environments* have been proposed, implemented, and come into use, some of them with considerable success. The implementation of a programming environment dedicated to a particular language (e.g., Pascal, Ada, Chill) requires a very substantial design and implementation effort. This is exemplified by the current efforts to build Ada Programming Support Environments (APSEs). It is, however, *not* yet generally recognized that all these language-specific programming environments have many traits in common. These can, in principle, be factored out by developing *language-independent* programming environments, which can be tailored towards a particular language by entering a definition of that language into the system. Another, even more important, argument in favor of language-independent programming environments is that they provide a uniform user interface: programmers using different languages can still work with similar if not identical programming environments for the various languages.

The purpose of this paper is to gain some insight in how well three existing, but still experimental, systems achieve this goal. The systems surveyed are:

- Mentor [2,3],
- The Synthesizer Generator [13,14], and
- CEYX [5].

All three systems are still under development. Mentor, being the oldest of the three, has reached a state of stability and has already been used for several applications in industry. The selection of these systems does not imply any prejudice against other, similar, systems but was largely determined by the possibility to gain access to each system and to consult its designers. The descriptions of the systems are necessarily brief and incomplete and reflect their state as of july 1983. Most notably missing is a discussion of the Gandalf [10] system.

The remaining sections of the paper are organized as follows. In section 2 the similarities between the systems are presented. Section 3 contains a brief outline of each system. In section 4 the methods for language specification, the properties of the resulting environment and various implementational issues are compared. Section 5 contains conclusions and points out some areas for further research. The comparison is based on implementations of the toy language PICO using all three systems. The complete listings of the implementation of PICO under each system are given in the appendices.

2. PROPERTIES COMMON TO ALL THREE SYSTEMS

The three systems have several properties in common. They are all language-independent and use similar notions to structure the definitions of new languages. The use of the word "language" is somewhat misleading and restrictive here: these systems are, in fact, all dedicated to the manipulation of hierarchically structured information in general. Programs in a programming language form just one example of such information. Other examples we shall encounter in the sequel are: systems for document preparation, for VLSI design, and for proof checking. All three systems use *trees* as primary datastructure to represent the objects that are being manipulated.

The global structure of a language-independent programming environment is shown in figure 1. A definition for a new language can globally be subdivided in definitions for:

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

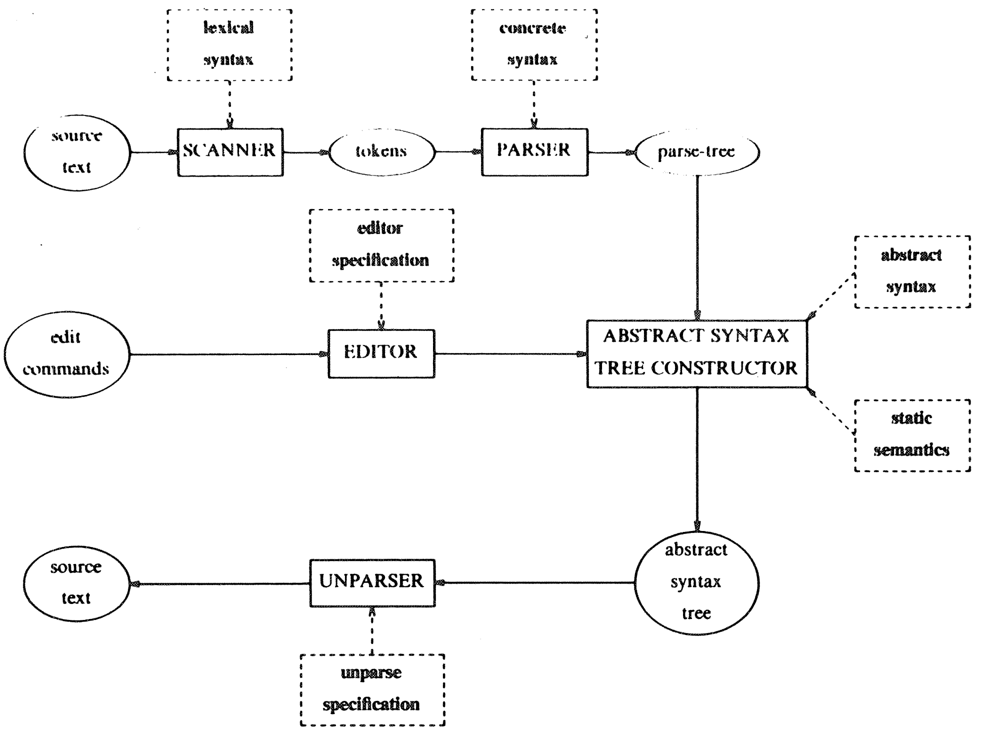

Figure 1. Global organization of a language-independent programming environment.

concrete syntax (also: 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 and the mapping from parse-tree to an abstract syntax tree.

unparsing (also: pretty printing):
: which defines the mapping of a program from its abstract syntactic form onto its written representation.

static semantics:
: which defines certain constraints on programs that can be verified without executing them, 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.

dynamic semantics:
: which defines the meaning of a program, i.e. the relation between its input and output data.

This list is not exhaustive; one could also include documentation, correctness proofs, etc.

Starting from a language definition, the three systems process programs in the defined language in similar ways. The definition of lexical and concrete syntax contains sufficient information to create a parser for the newly defined language and to build a parse-tree. A parse-tree typically contains non-terminals (e.g., `<expression>`) as nodes and terminals (e.g., keywords, `<plus-operator>`) as leaves. Subsequently, this parse-tree is transformed into an abstract syntax tree. An abstract syntax tree typically contains semantic notions (e.g., `while-statement`, `plus-operator`) as nodes and only constants and identifiers as leaves. The abstract syntax tree can be built directly and independently of the parse-tree, when a program is created during syntax-directed editing. If desired, the inverse operation can be carried out: the abstract syntax tree can be transformed into source text by means of unparsing (pretty printing). As can be seen in figure 1 "unparsing" is, in fact, a misnomer since the operations of both the scanner, the parser, and the abstract syntax tree constructor have to be inverted in order to transform an abstract syntax tree back into source text.

The distinction between concrete and abstract syntax is motivated by the fact that the objects dealt with in these systems are most suitably represented by trees, not strings. This most suitable form is an abstract syntax tree: the parse-tree is only an intermediate notion in understanding a string representation of the abstract syntax tree. Therefore, the concrete syntax of any string representation of an abstract syntax tree (there may be many such syntaxes) is regarded as unimportant. In addition to this several other observations can be made:

- Many programming languages have been designed with emphasis on the concrete syntax of the language. The abstract syntactic form is, however, in many cases more suited for semantic processing than the parse-tree representation itself. It can also contain more information than the textual representation of the program. In any case, the textual form can always be recreated from the abstract syntax tree.
- Non-terminals of the grammar do not have to generate nodes in the abstract syntax tree. An identifier may, for instance, occur directly as an `<expression>`, the intermediate levels in the parse-tree such as `<simple-expression>`, `<factor>` and `<term>` being collapsed.
- The systems presented here, all use parsing techniques suitable for the recognition of LALR(1) grammars. In many cases the "natural" grammar for a language does not conform to the LALR(1) restriction and has to be transformed into another grammar that does conform to it. The resulting distortions of the original syntax are, of course, not present in the abstract syntax. In addition to this, certain parts of the concrete syntax, such as lists of items described by (left or right) recursive syntax rules, are more naturally described in the abstract syntax tree by one n-ary node with all items as sons than by a binary tree of items. Note that both problems could be solved by using more general parsing techniques and more elaborate formalisms for the description of the syntax.

A final similarity between the systems is that they all create a syntax-directed editor for each new language and some standard user interface.

3. A BRIEF OUTLINE OF EACH SYSTEM

3.1. MENTOR

The initial design of Mentor started in 1974 at the *Institut National de Recherche en Informatique et Automatique* (INRIA) in France. Implementation of the kernel system was completed in 1977. Since that time the system has been used for its own maintenance. In 1980 a complete programming environment for Pascal was built using Mentor. Later, several other environments were constructed (for Ada, document preparation, etc.). The Mentor system itself is still evolving: efforts are now directed towards improving the user interface, adding specifications of static semantic properties to language definitions and exploring multi-lingual systems.

The original goal of the Mentor system was to create a language-independent system for program manipulation and transformation in which syntactic operations on programs were completely dealt with by the system. Later, emphasis shifted to syntax-directed editors and tools for the development of large software systems. Several of the other ideas on which Mentor is based are:

- The system is multi-lingual: it is possible to manipulate objects belonging to different languages simultaneously.
- The interactive manipulation of objects, i.e. trees, can be programmed: this amounts to extending the user interface by adding new commands to the system.
- The system is open: programs external to the system can access all of its facilities through a standard interface.

Mentor presents a hierarchically structured view of programs to its users, and adheres to the principle that programs are allowed to be incorrect during editing: semantic checks of a program are only performed upon explicit request by the user. It encourages the creation of tools to support particular methodologies (e.g., top-down, bottom-up) for the structured development of programs.

Two sublanguages of the Mentor system deserve special attention. The language *Metal* is used to specify the concrete and abstract syntax of a new language. The language *Mentol* is used both to interact with the Mentor system as well as to add (interactive) tree-manipulation commands. A programming environment for a new language is derived from a definition of that language in Metal. The resulting environment allows syntax-directed editing of programs in the new language by means of a set of standard commands for tree-traversal, editing and searching. Optionally, one can add commands (written in Mentol) to perform more language-specific operations, such as high-level motions in the program (e.g., move to the next procedure-declaration), checks that all variables have been declared, renaming of variables, construction of a graph of procedure calls, etc.

3.2. The Synthesizer Generator

The Synthesizer Generator is still under development. It is based on experience gained since 1978/1979 with the Cornell Program Synthesizer [16], a programming environment dedicated to PL/CS.

The Synthesizer Generator uses the paradigm of top down, hierarchically structured program development. But, contrary to the Mentor approach, programs are immediately checked for their static semantic correctness during editing, and all errors are immediately reported to the user. The current major goal of the project is to provide and improve facilities for incremental checking of static semantic constraints of new languages. These constraints are specified by means of an attribute grammar. A recently developed incremental evaluator for attribute grammars [13] is used to perform a minimal recomputation of attribute values when a modification is made in a program under construction. Ultimately the system will also include methods for defining the run-time semantics of new languages. Tools for source level debugging and for the detection of anomalies in a program by means of flow analysis will then be derived from the semantic definitions. It is envisaged that these versions of the system will also include a database with information on the program being edited. Such a database may be consulted interactively and may, for instance, contain a record of the current error points in the program, of the procedure call dependencies, or of the use-definition relations for the variables in the program.

To date, several systems have been built using the Synthesizer Generator. An experimental environment for Pascal, and several small systems: several toy languages, a desk calculator, a code generator for expressions and a proof checker.

Specifications are presented to the system in the Synthesizer Specification Language (SSL), which allows the definition of lexical, concrete, and abstract syntax, and of rules for static semantics and for unparsing. The resulting environment allows syntax-directed editing (including incremental checking of static semantic constraints) of programs in the new language using a fixed set of commands for tree-traversal and editing; the latter also include commands derived from the language definition to create constructs in the new language.

3.3. CEYX

CEYX is a system for VLSI design being developed at INRIA. At first sight, it may come as a surprise that such a system is included in this survey. Upon closer inspection, it turns out that the process of designing VLSI circuits, as for instance discussed in [9], entails describing various aspects (such as geometrical or electrical properties) of the same, hierarchically structured, abstract object. All these descriptions are expressed in different languages. In an early stage, it was recognized that one needs a very general and flexible system for expressing, editing, and manipulating such multilingual descriptions. Against this background, the CEYX systems attempts to offer:

- simultaneous manipulation of objects in different languages,
- specialized, cooperating editors for different languages, and
- great flexibility in choosing different representations for objects (this is particularly important for VLSI design, since huge design files have to be manipulated).

CEYX draws heavily upon the earlier experience with Mentor. It provides an operators/phyla model (see section 4.1.3) for the definition of abstract syntax comparable to the one used in Mentor. The system is object-oriented and has borrowed several ideas from Smalltalk [4] and other object-oriented languages. All nodes in the abstract syntax tree are represented by objects with associated, user-defined operations for editing, unparsing and the like. In this way one can also attach rules for dynamic semantics to each object. All these user-defined operations have to be programmed in Lisp. In addition to this, abbreviations (*keys*) can be introduced for each semantic operation in a similar way as is done in EMACS [15].

The current implementation of the system does not include facilities to define lexical or concrete syntax.

4. A COMPARISON

As explained earlier, the systems have different goals and use different specification and implementation techniques, which makes a comparison very hard if not impossible. In order to establish some basis for comparison, I have developed under each of the three systems a programming environment for the extremely small programming language PICO. This gives at least some insight in the various specification methods used and in the quality of the resulting environments. The following subsections discuss the methods used for language definitions in each system and the properties of the resulting programming environment. The definition of PICO is used as running example. Here is, for completeness, the syntax of PICO:

```
<pico-program> ::= 'program' <decls> <series> 'end' .
<decls>        ::= 'declare' <id-list> ';' .
<id-list>      ::= <id-list> ',' <id> | <id> .
<series>       ::= <series> ';' <statement> | <statement> .
<statement>    ::= <asg-stat> | <if-stat> | <while-stat> .
<asg-stat>     ::= <id> ':=' <exp> .
<if-stat>      ::= 'if' <exp> 'then' <series>
                   'else' <series> 'fi' .
<while-stat>   ::= 'while' <exp> 'do' <series> 'od' .
<exp>          ::= <simple-exp> '+' <simple-exp> |
                   <simple-exp> '*' <simple-exp> |
                   <simple-exp> .
<simple-exp>   ::= <id> | <number> | '(' <exp> ')' .
```

The non-terminals `<id>` and `<number>` respectively represent identifiers and integer constants. The only static semantic property we will be interested in is the requirement that all variables occurring in an `<asg-stat>` or an `<exp>` should have been declared, i.e. should occur in the `<id-list>` of the `<decls>`-part of the PICO-program.

4.1. Method of language specification

4.1.1. Lexical definitions

Mentor and the Synthesizer Generator both use regular expressions for the definition of lexical syntax and both use an external scanner generator to actually create a lexical scanner (see also section 4.3). CEYX does not yet provide facilities for defining lexical syntax.

4.1.2. Concrete syntax

Mentor and the Synthesizer Generator use both a form of BNF notation to specify the concrete syntax and both depend on an external parser generator for the construction of LALR(1) parsers. A typical rule in a Mentor definition is:

```
<while_stat> ::= while <exp> do <series> od ;
       while(<exp>, <series>)
```

Non-terminals are between angle brackets and terminals are just written as they are. Characters in terminals that conflict with the syntax notation have to be escaped (using a sharp sign). The formalism allows neither alternation nor repetition in syntax rules. The part of the rule following the semicolon specifies the abstract syntax tree to be built for the construct, in the above example a tree labeled with the operator "`while`" with the abstract trees corresponding to `<exp>` and `<series>` as sons.

All constructs of a language that may be created during editing have to be added explicitly as alternatives of the start symbol of the grammar so that appropriate entry points can be created in the generated parser.

The Synthesizer Generator uses a syntax notation akin to the Yacc [6] notation. A typical example is:

```
While_stat ::= WHILE Exp DO Series OD
       { While_stat.abs = MkWhile_stat(Exp.abs, Series.abs);}
```

Here terminals are written in uppercase and have to be declared separately in the lexical syntax. The formalism allows alternation but does not allow repetition in syntax rules. The part of the rule between braces specifies the building of the abstract syntax tree. It is interesting to note that the standard attribute grammar mechanism is used to build this tree: in the above example via the synthesized attribute `abs`. This permits arbitrary computations during construction of the abstract syntax tree and allows, for instance, inclusion of parts of the parse-tree in the abstract tree (e.g., the constituent characters of a string constant). In a similar way, it is possible to include new subtrees that contain values of inherited attributes of the subtree they are replacing (e.g., substitution of constants by their values).

As stated earlier, CEYX does not yet provide facilities for defining concrete syntax.

4.1.3. Abstract syntax

In Mentor, the abstract syntax of a language is defined in terms of *operators* and *phyla*:

- The operators label the nodes of the abstract tree. Operators with fixed arity are allowed to have children of different kinds. Operators with arity zero are the leaves of the abstract tree and represent the atoms of the language. Operators with non-fixed arity (also called *lists*) are only allowed to have children of the same kind.
- The notion "kind of children" is formalized by the concept of *phylum*: this is a non-empty set of operators. A phylum is associated with each child position of an operator and indicates precisely which operators are allowed as labels of subtrees at each child position.

The specification of operators and phyla for a new language contains just enough information to maintain the correct relationships between operators and phyla when abstract syntax trees are modified. This ensures that the abstract trees remain syntactically correct.

A typical definition of an operator is:

```
while  -> EXP SERIES;
```

This defines the operator `while`; nodes labeled with this operator have two children which belong to the phyla `EXP` and `SERIES` respectively. The formalism allows the definition of list operators:

```
series -> STATEMENT + ... ;
```

This defines the operator `series`; nodes labeled with this operator may have one or more children all belonging to the phylum `STATEMENT`. The definition of the related phyla could be:

```
STATEMENT ::= assign if while;
EXP       ::= plus times number;
```

(assuming that the operators `assign`, `if`, `plus`, `times` and `number` have been appropriately defined).

The names of the phyla play an important role in the resulting programming environment: when, during editing, a new construct is about to be entered, Mentor either derives the phylum to which the new construct must belong and uses the phylum name as a prompt, or the phylum can not be derived automatically in which case the system asks the user to enter the name of the required phylum in advance.

Mentor allows large language definitions to be subdivided in *chapters*. This feature has not been used in the definition of PICO.

The Synthesizer Generator uses a slightly more restrictive method for the definition of the abstract syntax than Mentor. In terms of the operators-phyla model, the Synthesizer Generator allows an operator to occur only once in some phylum. A typical definition of an abstract syntax rule is:

```
statement: "wh" => MkWhile_stat(exp series);
```

which states that a statement may have the form of a subtree with as operator `MkWhile_stat` and as children the abstract trees corresponding to exp and series. The (optional) string "wh" defines the command to be added to the user interface to create while-statements. Note that this mechanism can be used in two ways: to define commands for the creation of new nodes in the abstract syntax tree and for the inspection (or modification) of inherited (or synthesized) attribute values at a certain point in the tree. Actual definitions are slightly more complicated since they also contain unparsing information; this is treated further in section 4.1.6. below.

The specification method for the abstract syntax is also used to specify the domains used in the definition of the static semantics.

CEYX uses the notions of "constructor" and "universe" to define the abstract syntax in a way comparable to the model used in Mentor. To a first approximation, a constructor is comparable to an operator and a universe to a phylum. Looking more closely, one observes that a universe has all the properties of a complete language definition: it describes the well-formed abstract trees for that language. As opposed to the approach in Mentor, universes (i.e. formalisms) may be defined in a nested fashion and as a consequence, switching of formalisms can be accomplished in a natural way by "crossing a universe" in the abstract tree, i.e. by encountering a universe node in the tree. Mentor achieves this same effect by associating "annotations" with nodes in the tree which describe the transition to another language. CEYX has in this way unified the notions of phylum and formalism. The advantage of this approach is extreme flexibility. A disadvantage may be that one loses the possibility to structure large definitions by distinguishing different languages.

A fragment from the PICO definition will clarify the hierarchy of types that can be declared in CEYX:

```
(defuniverse pico)

(defcons program~pico (decls series))
(defcons decls~pico id)
(defcons series~pico statement)

(defuniverse statement~pico)
 ...
(defcons while-stat~statement (exp statement))

(defuniverse exp~pico)
(defcons plus~exp (exp exp))
 ...
```

First, the new universe pico is defined. Next, the three constructors program, decls and series are defined, which all three belong to the pico-universe. Note that the notation used indicates that a program has *two* children of type decls and series respectively, and that decls has a *list* of children, all of type id. Also shown are the subuniverses statement and exp.

The system for type definitions is richer than shown here. It is, for instance, possible to specify children in the declaration of a universe: these children are inherited by all constructors that are derived from this universe. Also provided are functions to access the components of universes and constructors, to test their type, etc.

4.1.4. Static semantics

Currently, Mentor does not provide means to define static semantics. This implies that all static semantic checks have to be programmed explicitly in Mentol and that these checks have to be invoked explicitly by the user after he has completed a modification of his program. The Pascal environment built with Mentor contains, for instance, commands for

- adding variables to the variable declaration part associated with the current editing cursor in the program;
- (minimal) renaming of variables so that distinct objects have distinct names;
- detection of unused variables;
- construction and inspection of a procedure call graph.

The recently developed language Typol [1] for the specification of types and declaration checking will probably be incorporated into the Metal specification language in the near future.

The Synthesizer Generator is, so to speak, completely geared towards the specification of static semantics. As stated earlier, attribute grammars are used for this purpose. With each operator from the abstract syntax, one or more attribute equations can be associated. Continuing the example from the previous section, a while-statement might have an inherited attribute `env` (modeling the variable declaration environment) from its ancestor `statement` and this attribute has to be passed on to its children `exp` and `series`:

```
MkWhile_stat :: { exp.env = statement.env;
                  series.env = statement.env;}
```

The Synthesizer Generator also permits references to the values of non-local attributes. The implementation of PICO given in the appendices uses this feature to eliminate most of the inherited attributes that would otherwise have to be passed to the rules for statements and expressions. Instead, the synthesized attribute `env` of the declaration part is made available to the rule for `pico_program` and that value is referred to directly from within the other rules.

During editing of a program changes in attribute values are propagated incrementally. This makes the checking of static semantic constraints immediate and automatic, i.e. it does not have to be explicitly invoked by the user.

CEYX does not support a notion of static semantics. All checks for static semantic constraints have to be programmed explicitly in Lisp.

4.1.5. Dynamic semantics

Only CEYX allows the definition of dynamic semantics. The other two systems do not include facilities for this.

Mentor relies on an external language processor for the execution of programs. In the Mentor/Pascal environment, for instance, programs are edited using the syntax-directed facilities of Mentor. When a program has to be executed, it is written onto an external file, compiled by means of the standard Pascal compiler of the host operating system, and the resulting object program is then executed. It would, in principle, be possible to write an interpreter or compiler for, say Pascal, in Mentol. As it was never designed for this purpose, however, Mentol lacks many of the essential primitives and nobody has ever tried to use it for this purpose. Another, more realistic, approach would be to use the standard interface between Mentor and Pascal and to write a Pascal interpreter in Pascal that operates on the Mentor abstract tree representation of a program.

The Synthesizer Generator allows a similar approach: in principle, procedures written in C could be combined with the generated programming environment in order to interpret the abstract syntax tree. It is, however, also possible to associate expressions in the Synthesizer Specification Language (SSL) with nodes in the tree. This provides a limited, but clean, way to add dynamic semantics.

CEYX explicitly supports the notion of associating dynamic semantics with each node in the tree. The general way to do this is by associating a semantic function under a certain name with a construct or universe in the tree. For instance, to associate the semantic property `pretty` with the construct `while-stat` one would write:

```
(defsem (while-stat pretty) object body)
```

Here, `object` represents the construct with which the semantic property `pretty` is to be associated and `body` is the Lisp code to be executed when this semantic property is invoked.

4.1.6. Unparsing

The three systems use surprisingly simple, even primitive, methods for the specification of unparsing. All systems use a recursive descent, preorder traversal of the abstract syntax tree for the creation of a printed image. During this traversal an indentation level is maintained which can be incremented/decremented at the beginning/end of the unparsing of certain constructs. Two major problems must be solved by the unparser. First, how to choose the correct format for the unparsing of certain constructs. If, for instance, an if-statement with an empty else-part occurs inside another if-statement, the unparser has to create an explicit empty else-part for the innermost statement in order to avoid the dangling else problem. Secondly, the unparser must decide what to do when the unparsing of a certain construct overflows the current line. In principle, there are two solutions:

- associate multiple unparsing formats with each construct and choose one depending on information in the abstract syntax tree and on the amount of available space.
- associate only one format with each construct, but give variable interpretations to this format, depending on the space available on the output medium.

All three systems use a variant of the second method for unparsing. Only CEYX provides a formalized notion for defining formats with multiple interpretations. This allows formatting of the output depending on the space available on the output medium, e.g. an if-statement may be placed on one line, but is spread over several lines if it is too long. In the Mentor system this can also be done but in an ad-hoc way. The Synthesizer Generator allows the definition of a single format per construct with a fixed interpretation.

Mentor does not provide a specification method for unparsing. A rather flexible scheme has been designed but is not yet implemented. The method currently used is to add Pascal procedures for unparsing to the environment. A prototype unparser is available which defines the interface between the unparser and the Mentor system. In addition to this, Mentor generates a file with Pascal constant declarations for all operators and symbols in the new language. The definition of a new unparser amounts to writing a procedure for the unparsing of all operators of the abstract syntax of the new language.

The Synthesizer Generator allows associating one unparsing template with each abstract syntax rule. Continuing the example of the while-statement in the previous paragraphs, the abstract syntax rule, now including unparsing information, assumes the form:

```
statement: "wh" => MkWhile_stat ( "while " exp "do"
                                  "\t\n" series
                                  "\b\n" "od" );
```

The strings appearing in the argument list of `MkWhile_stat` constitute the unparsing template: they will be displayed in proper combination with the strings resulting from the unparsing of `exp` and `series`. The escapes in these strings indicate the beginning of a newline ("`\n`") and increase ("`\t`") or decrease ("`\b`") of the indentation.

The Synthesizer Generator also provides a rudimentary facility for printing the values of local attributes during unparsing. This feature can be used for flagging erroneous statements by introducing a local error attribute in certain rules: the default value of such an error attribute is the empty string, but an error message may be assigned to it which will be printed as part of the unparsing of the rule if the error has occurred.

The specification method for unparsing as used in CEYX was inspired by previous work of several researchers [12,17]. It uses the notion of *horizontal* and *vertical blocks* of text that have to be kept together during unparsing. In these blocks possible cutpoints have to be declared explicitly. The pieces of text between cutpoints form the *components* of a block. The components of a horizontal block are placed on the current line from left to right. If the current line overflows, a new line is created starting at a given indentation after which the remaining components of the block are placed. This process is repeated until all components of the block have been printed. The elements of a vertical block are either placed on the same horizontal line, if there is enough room, or on consecutive lines with the same amount of indentation. The basic functions for unparsing are:

`vprinch`	prints one character
`vpatom`	prints one atom
`vterpri`	prints a new line
`hblock`	defines a horizontal block
`vblock`	defines a vertical block
`cutpoint`	defines a cutpoint
`vdispatch`	recursively unparses subtrees

There are also functions available to explicitly set the required indentation for horizontal and vertical blocks. The above unparsing rules are associated with a construct in the abstract tree by defining semantics with the name `pretty` for it. For instance, the unparsing rules for the while-statement are:

```
(defsem (while-stat pretty) (x)
        (vblock-with-indent 0
           (hblock
              (vpatom "while") (cutpoint)
              (vdispatch (get-son x 1)))    ; test-part
           (cutpoint)
           (vblock
              (vpatom "do") (cutpoint)
              (vdispatch (get-son x 2)))    ; do-part
           (cutpoint)
           (vpatom "od")))
```

4.2. Properties of the resulting environment

4.2.1. User interface and basic primitives

The Synthesizer generator and CEYX both provide the user with a screen-oriented interface. Currently, Mentor only provides a teletype-oriented one, but a screen-oriented version is expected to be released near the end of 1983. All systems use the notion of a "current tree" and of a pointer (cursor) in the current tree.

In Mentor one is, in fact, always communicating with the Mentol command interpreter. The current tree can be traversed using:

- language-independent commands for editing (i.e., "up", "down", "left", "right", "n-th son", "delete", "insert", etc.) and for tree pattern matching (i.e. "find next if-statement", where the operator "if" is a parameter of a language independent command.).
- language-dependent commands (i.e. Mentol procedures for finding, for instance, the next procedure declaration respecting the scope rules of the language under consideration, etc.).

If desired, the current tree can be printed with a certain, optionally specifiable, level of detail. In fact, an arbitrary number of trees can be manipulated simultaneously. Since Mentol allows the definition of procedures, the user interface can be extended in an arbitrary way.

After each command, the Synthesizer Generator automatically unparses the current tree and displays it on the screen with the cursor highlighted. The current tree can be traversed using:

- language-independent commands (i.e., "up", "down", "left", "right", "delete", "insert", etc.),
- language-dependent commands for the creation of new language constructs (this was discussed earlier in section 4.1.3).

The system does not provide facilities for searching or for further extending the user interface. Another noteworthy property of this system is that at any moment during editing there are two alternative ways to enter a new construct:

- by entering the shorthand command (e.g., "`wh`") that is given in the the definition of the abstract syntax, or
- by typing the full text of the construct to be entered; this text is then parsed and converted into an abstract tree.

The shorthand commands that are applicable at each stage are also displayed in a menu.

CEYX provides the notion of dynamically defined *keys* that define the properties of the user-interface. At each moment, during editing, there is a "current" object. As explained earlier, there is an arbitrary number of semantic rules associated with each object. These semantic rules can be invoked via a shorthand notation that associates a key with a semantic rule. During editing, the user types such keys as commands to the system. The interpretation of these keys depends on the key definitions for the current object. The interpretation of keys is completely dynamic: if the current object does not define a certain key, the key definitions of the parent object are searched, and so on, until a definition is found. If no definition is found either a system-provided default is assumed or an error message is given. In this way, standard operations like up, down, insert or delete can be specialized for certain classes of objects. This mechanism also allows the evaluation of arbitrary Lisp expressions during editing and gives access to an EMACS-like text editor which forms an integral part of the system.

4.2.2. Communication with the outside world

Mentor provides three methods for communication with the outside world:

- External text-files can be parsed and converted into an abstract syntax tree and vice versa.
- Abstract syntax trees can be saved on external files and be reloaded into the system.
- External Pascal procedures can be called and the Mentor primitives are also accessible from Pascal.

The Synthesizer Generator only allows calling of external C procedures. Saving and restoring of abstract syntax trees is planned for inclusion in coming versions of the system.

CEYX allows abstract trees to be saved on external files by saving a Lisp expression that will recreate the tree. Moreover, Lisp functions can be called at any time from CEYX so it inherits all facilities for communication with the outside world from the Lisp system.

4.2.3. Multiple languages.

Mentor and CEYX permit the simultaneous manipulation of abstract trees belonging to different languages. This allows, for instance, the manipulation of Pascal programs in which the comments are written in some formal specification language, or the manipulation of documents containing mixtures of ordinary text and formal notation (e.g., a language reference manual, a book on quantum mechanics, this paper).

Both systems are also self-descriptive: their meta-language (i.e. the formalism used for language specifications) can be defined in the meta-language itself. Such a self-referential definition makes it possible to extend the meta-language and to create new versions of the system by means of a bootstrap.

The Synthesizer Generator supports the manipulation of abstract syntax trees in just one language.

4.3. Implementation of the systems

Mentor is implemented in Pascal on a Honeywell-Bull 68 under Multics. For the creation of a lexical scanner and a LALR(1) parser it uses the SYNTAX system developed at INRIA by the "Langages et Traducteurs" team. Currently, the system is being ported to a VAX under Berkeley UNIX. The UNIX implementation uses Lex [8] and Yacc [6] for the creation of a scanner and a parser.

The Synthesizer Generator is implemented in C on a VAX under Berkeley UNIX. It also uses Lex for the creation of a lexical scanner and Yacc for the creation of a LALR(1) parser.

CEYX is completely implemented in Lisp. Versions exist for Maclisp (Multics) and LeLisp (VAX, M68000, SM90). The current system does not include parsers, but an adaptation of Yacc to Lisp has been completed and will be incorporated in newer versions of the system.

5. CONCLUSIONS

An absolute comparison of the three systems, it must be reiterated, is hardly possible, but some general lessons can nevertheless be drawn.

First, it can be concluded that it is realistic to build language-independent programming environments. In the long run they will have great advantages over language-specific systems. Advantages not only in terms of savings in design and implementation effort, but also in terms of affordable generality and sophistication of the resulting systems. This type of programming environment will make it economically feasible to develop environments for very specialized application languages and to experiment with new features while a language is still under development.

Secondly, several particularly important features can be identified in the systems under consideration:

- a screen-oriented user-interface,
- programmability of the user-interface,
- pattern matching,
- parsing/unparsing of external files and storage of programs in their abstract tree form,
- multiple unparsing schemes,
- manipulation of multiple languages, including the possibility to annotate a program in one formalism with comments in another formalism,
- incremental checking of static semantic constraints, and
- self-referential system descriptions.

The definition of both concrete and abstract syntax in these systems leads to much duplication in the language specifications. One can easily envision that these definitions will be simplified in future systems since major parts of the abstract syntax can be derived automatically from the concrete syntax.

Thirdly, the specification of dynamic semantics forms the weakest part of all three systems. Much work remains to be done in order to build language-independent programming environments that cover the complete editing-execution-debugging cycle. Three topics deserving special attention are:

- language-independent debugging,
- language-independent tools for flow analysis, and
- general, language-independent, optimization techniques.

Finally, it should be noted that none of the systems under consideration provides primitives for version control or project management nor does any of them support a particular, explicit, methodology for software development.

ACKNOWLEDGEMENTS

This study was made possible by financial support of *Institut National de Recherche en Informatique et Automatique* in France and *Centrum voor Wiskunde en Informatica* (Centre for Mathematics and Computer Science, formerly: Mathematical Centre) in The Netherlands. Gilles Kahn created the opportunity for my visit to INRIA in july 1983 and he was also extremely helpful in arranging all the details. The designers of the three systems discussed in this paper provided access to their systems and helped me a great deal in running the example programs given in the paper: Bertrand Mélèse (Mentor), Thomas Reps and Tim Teitelbaum (Synthesizer Generator) and Jean Marie Hullot (CEYX). I found the resulting discussions fruitfull and inspiring. Comments made on various drafts of this paper by Véronique Donzeau-Gouge, Jan Heering, Gérard Huet, Jean Marie Hullot, Bernard Lang, Bertrand Mélèse, Thomas Reps, Tim Teitelbaum, and Alan Wills are gratefully acknowledged. Finally, I want to thank all inhabitants of "bâtiment huit" for their hospitality and support.

LITERATURE

1. Despeyroux, Th. & Donzeau-Gouge, V., "Typol, introduction de spécifications sémantiques dans Mentor", INRIA Research Report, 1983.

2. Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, B. & Lévy, J.J., "A structure oriented program editor: a first step toward computer assisted programming", International Computing Symposium, North Holland, 1975.

3. Donzeau-Gouge, Huet, G., Kahn, G. & Lang, B., "Programming environments based on structured editors: the Mentor experience", INRIA Research Report, No. 26, 1980.

4. Goldberg, A. & Robson, D., *Smalltalk-80, The Language and its Implementation*, Addison-Wesley, 1983.

5. Hullot, J.M., "CEYX - a multiformalism programming environment", (to be presented at IFIP 1983)

6. Johnson, S.C., "Yacc - Yet Another Compiler-Compiler", Computer Science Technical Report No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.

7. Kahn, G., Lang, B. & Mélèse, B., "Metal: a formalism to specify formalisms", to appear in Science of Computer Programming, North-Holland, 1983.

8. Lesk, M.E., "Lex - A Lexical Analyzer Generator", Computer Science Technical Report No. 39, Bell Laboratories, Murray Hill, New Jersey, 1975.

9. Mead, C & Conway, L., *Introduction to VLSI Systems*, Addison-Wesley, 1980.

10. Medina-Mora, P. & Feiler, P., "An incremental programming environment", IEEE Transactions on Software Engineering, **SE-7** (1981) 5, 472-482.

11. Mélèse, B., "Mentor: l'environnement Pascal", INRIA Technical Report, No. 5, 1981.

12. Oppen, D.C., "Prettyprinting", ACM Transactions on Programming Languages and Systems, **2** (1980) 4, 465-483.

13. Reps, T., "Generating language-based environments", Technical Report 82-514, Cornell University, Ithaca, 1982.

14. Reps, T., Teitelbaum, T. & Demers, A., "Incremental context-dependent analysis for language-based editors", ACM Transactions on Programming Languages and Systems, **5** (1983) 3, 449-477.

15. Stallman, R.M., "EMACS, the extensible, customizable self-documenting display editor", Proceedings of the ACM SIGPLAN, SIGCOA Symposium on Text Manipulation, Portland, Oregon, june 8-10, 1981, SIGPLAN Notices, **16** (1981) 6, 147-156.

16. Teitelbaum, T. & Reps, T., "The Cornell program synthesizer: a syntax directed programming environment", Communications of the ACM **24** (1981) 9, 563-573.

17. Waters, R.C., "GPRINT: A Lisp pretty printer providing extensive user format-control mechanisms", MIT AI Memo No. 611, 1981.

Appendix I. PICO implemented with Mentor

I.a. Lexical definitions

```
SIMPLE CLASSES

        UCLET    = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" ;
        LCLET    = "abcdefghijklmnopqrstuvwxyz" ;
        CHIF     = "0123456789" ;
        SP       = " "
                 , #011 ; -- HT
        EOL      = #012 ;
        CLAM     ="!";
        UNDERSCORE= "_" ;
        OTHERS   = "&{}^\~'?/%|.<>@""'-" ;

COMPOUND CLASSES
        LETTER   = UCLET + LCLET ;
        LETCHIF  = LETTER + CHIF ;
        ANY_EOL  = ANY-EOL;

ABBREVIATIONS


        IDENT    = LETTER { [UNDERSCORE] LETCHIF  } @1;
        UCIDENT  = UCLET { [UNDERSCORE] UCLET  } ;
        COMMENT  = -SP&1{-SP}|-SP|-EOL|(-SP&1{-SP}-CLAM|-CLAM&1){ANY}EOL . ;
                 -- &1 : true if clam lay at colomn 1

COMMENTS

        COMMENT {COMMENT} ;

TOKENS


        PHYLUM          = "["&2 UCLET { [UNDERSCORE] UCLET } "]"  ;
        %ID             =  IDENT ; CONTEXT ALL BUT %ID, %NUMBER ;
        %NUMBER         = CHIF { CHIF } ;
        %METAVAR        = - "$" UCIDENT  ;
```

I.b. Abstract and concrete syntax

```
definition of PICO is

    rules

        <root>              ::= <pico_program> ;
            <pico_program>
        <pico_program>      ::= program <decls> #; <series> #end ;
            pico_program(<decls>,<series>)
        <decls>             ::= declare <id_list> ;
            <id_list>
        <id_list>           ::= <id> ;
            decls-list((<id>))
        <id_list>           ::= <id_list> #, <id> ;
            decls-post(<id_list>,<id>)
        <series>            ::= <statement> ;
            series-list((<statement>))
        <series>            ::= <series> #; <statement> ;
            series-post(<series>,<statement>)
        <statement>         ::= <asg_stat> ;
            <asg_stat>
        <statement>         ::= <if_stat> ;
            <if_stat>
        <statement>         ::= <while_stat> ;
            <while_stat>
        <asg_stat>          ::= <id> #:= <exp> ;
            assign(<id>,<exp>)
        <if_stat>           ::= if <exp> then <series> else <series> fi ;
            if(<exp>,<series>.1,<series>.2)
        <while_stat>        ::= while <exp> do <series> od ;
            while(<exp>,<series>)
        <exp>               ::= <plus> ;
            <plus>
        <exp>               ::= <times> ;
            <times>
        <exp>               ::= <simple_exp> ;
            <simple_exp>
        <simple_exp>        ::= <id> ;
            <id>
        <simple_exp>        ::= <number> ;
            <number>
        <simple_exp>        ::= #( <exp> #) ;
            <exp>
        <plus>              ::= <simple_exp> #+ <simple_exp> ;
            plus(<simple_exp>.1,<simple_exp>.2)
        <times>             ::= <simple_exp> #* <simple_exp> ;
            times(<simple_exp>.1,<simple_exp>.2)
        <id>                ::= %ID ;
        id-atom(%ID)
    <number>            ::= %NUMBER ;
        number-atom(%NUMBER)

    <root>              ::= #[PICO_PROGRAM] <pico_program> ;
        <pico_program>
    <root>              ::= #[PICO_PROGRAM] <metavar> ;
        <metavar>
    <root>              ::= #[DECLS] <decls> ;
        <decls>
    <root>              ::= #[DECLS] <metavar> ;
        <metavar>
    <root>              ::= #[SERIES] <series> ;
        <series>
    <root>              ::= #[SERIES] <metavar> ;
        <metavar>
    <root>              ::= #[IF] <if_stat> ;
        <if_stat>
    <root>              ::= #[IF] <metavar> ;
        <metavar>
    <root>              ::= #[WHILE] <while_stat> ;
        <while_stat>
    <root>              ::= #[WHILE] <metavar> ;
        <metavar>
    <root>              ::= #[EXP] <exp> ;
        <exp>
    <root>              ::= #[EXP] <metavar> ;
        <metavar>
    <root>              ::= #[ID] <id> ;
        <id>
    <root>              ::= #[ID] <metavar> ;
        <metavar>
    <metavar>           ::= %METAVAR ;
        meta-atom(%METAVAR)


abstract syntax


    pico_program        -> DECLS SERIES;
    decls               -> ID + ... ;
    series              -> STATEMENT + ... ;
    assign              -> ID EXP;
    if                  -> EXP SERIES SERIES;
    while               -> EXP SERIES;
    plus                -> EXP EXP;
    times               -> EXP EXP;
    comment_s           -> COMMENT + ... ;
    id                  -> implemented as IDENTIFIER;
    comment             -> implemented as STRING;
```

```
        number              -> implemented as INTEGER;
        meta                -> implemented as IDENTIFIER;


        PICO_PROGRAM        ::= pico_program;
        DECLS               ::= decls;
        SERIES              ::= series;
        STATEMENT           ::= assign if while;
        EXP                 ::= plus times ID number;
        ID                  ::= id;
        COMMENT             ::= comment;

end definition
```

I.c. Checking of static constraints

```
% Mentol procedures for checking variable declarations
.rec
mentol
:&                     % begin of a function definition
% The following procedure checks that all identifiers in a
% PICO program are actually declared.
.def <.all_declared ,
(    pd;               % save current cursor
     @d:u* s1;         % u* goes to the root of the tree
                       % d becomes the first son of the root
                       % (i.e. the declaration part)
     @k:u* s2;         % set current pointer to series part
     .foreach<@id,     % for each occurence of an <id> in series
                       % check that it occurs in decls.
              .occurs<@k, @d>
             >;
     pu;               % restore previous cursor
) >

ss2;.lredef;           % end of a function definition
:&
% The following function ensures that a given name occurs
% in a list of identifiers
.def <.occurs<@name, @list>,
(    @list f @name;    % find name in  list
        ? ,            % if found do nothing

        ( .pnnl<@name>; % otherwise give errormessage
          @mess s1 p
        )
) >

ss2;.lredef
% here follows one string constant
@mess:&
comment_s]
is not defined
```

I.d. Unparsing

```
(* The unparser for PICO is obtained by inserting certain language-
   dependent parts in a prototype unparser written in Pascal.
   The resulting program consists of about 500 lines and is not
   reproduced here; only the language dependent parts are shown.
   Procedures and types not defined here are part of the unparser
   interface with Mentor *)

const
     (* node types in the abstract syntax tree *)
     IDNODE  =1;
     COMMENTNODE=2;
     NUMBERNODE=3;
     METANODE=4;
     PICOPROGRAMNODE=13;
     ASSIGNNODE=14;
     WHILENODE=15;
     PLUSNODE=16;
     TIMESNODE=17;
     DECLSNODE=22;
     SERIESNODE=23;
     COMMENTSNODE=24;
     IFNODE  =29;
     (* Keywords and special symbols *)
     TKPROGRAM=1;
     TKEND   =2;
     TKDECLARE=3;
     TKIF    =4;
     TKTHEN  =5;
     TKELSE  =6;
     TKFI    =7;
     TKWHILE =8;
     TKDO    =9;
     TKOD    =10;
     (* special characters -- skip constant 11 *)
     TKSEMI  =12;
     TKCOMMA =13;
     TKASG   =14;
     TKLPAR  =15;
     TKRPAR  =16;
     TKPLUS  =17;
     TKTIMES =18;

(* DECTABLE is the unparser *);

procedure DECTABLE(P:TREES;HOLO:INTEGER);

     (* Unparsing of comments. *)
```

```
procedure DECCOM(WHERE:TREES;K:KINDS);
     var  COM:TREES;
     begin ... end;

(* Unparsing of list nodes *)
procedure DECLIST(P:TREES);
     begin ...  end;

(* Unparsing of fixed arity nodes. *)
procedure DECEXP(P:TREES;HOLO:INTEGER);
     begin
     case OPER(P) of
          IDNODE,NUMBERNODE: PRINTATOM(P);
          METANODE: PRINTMETA(P);
          PLUSNODE,TIMESNODE:
               begin
               KEYPRINT(TKLPAR);
               DECTABLE(P,HOLO);
               KEYPRINT(TKRPAR);
               end
     end
     end (* DECEXP *);

procedure DECTA1(var P:TREES);
     begin
     case OPER(P) of
          NUMBERNODE,IDNODE: PRINTATOM(P);
          COMMENTNODE:
               begin
               SPPRINT(3);
               PRINTATOM(P)
               end;
          METANODE: PRINTMETA(P);
          PICOPROGRAMNODE:
               begin
               KEYPRINT(TKPROGRAM);
               WLINE;
               WTAB;
               DECTABLE(CHILD(1,P),HOLO-1);
               WLINE;
               DECTABLE(CHILD(2,P),HOLO-1);
               BACKTAB;
               WLINE;
               KEYPRINT(TKEND)
               end;
          ASSIGNNODE:
               begin
               DECEXP(CHILD(1,P),HOLO-1);
               KEYPRINT(TKASG);
               DECEXP(CHILD(2,P),HOLO-1)
```

```
                    end;
               IFNODE:
                    begin
                    KEYPRINT(TKIF);
                    DECEXP(CHILD(1,P),HOLO-1);
                    KEYPRINT(TKTHEN);
                    WTAB;
                    DECTABLE(CHILD(2,P),HOLO-1);
                    BACKTAB;
                    WLINE;
                    KEYPRINT(TKELSE);
                    WTAB;
                    DECTABLE(CHILD(3,P),HOLO-1);
                    BACKTAB;
                    WLINE;
                    KEYPRINT(TKFI)
                    end;
               WHILENODE:
                    begin
                    KEYPRINT(TKWHILE);
                    DECEXP(CHILD(1,P),HOLO-1);
                    KEYPRINT(TKDO);
                    WTAB;
                    DECTABLE(CHILD(2,P),HOLO-1);
                    BACKTAB;
                    WLINE;
                    KEYPRINT(TKOD)
                    end;
               PLUSNODE:
                    begin
                    DECEXP(CHILD(1,P),1000);
                    KEYPRINT(TKPLUS);
                    DECEXP(CHILD(2,P),1000);
                    end;
               TIMESNODE:
                    begin
                    DECEXP(CHILD(1,P),1000);
                    KEYPRINT(TKTIMES);
                    DECEXP(CHILD(2,P),1000);
                    end
          end
          end (* DECTA1 *);

     (* BODY OF DECTABLE *)
     begin ...
     end (* DECTABLE *);
```

Appendix II: PICO implemented with the Synthesizer Generator

```
/* Definition of PICO for the Synthesizer Generator */

/* PART I -- lexical syntax */

NUMBER:        < [0-9]+ >;
WHITESPACE:    < [\ \t\n]* >;
PROGRAM:       < "program" >;
END:           < "end" >;
DECLARE:       < "declare" >;
IF:            < "if" >;
THEN:          < "then" >;
ELSE:          < "else" >;
FI:            < "fi" >;
WHILE:         < "while" >;
DO:            < "do" >;
OD:            < "od" >;
ID:            < [a-zA-Z][a-zA-Z0-9]*|[?] >;
ASSIGN:        < ":=" >;

/* PART II -- abstract syntax */

root pico_program;

pico_program:                MkProgramBot( "<pico_program>" )
       |       "p"     => MkProgram ( "program\t\n"
                                      decls "\n"
                                      series "\b\n"
                                      "end\n" )
        ;
decls   :                    MkNullDecls ( "<decls>" )
        |      "d"     => MkDecls     ( "declare " id_list ";\n" )
        ;

id_list :                    MkId_list1 ( id )
        |      "2"     => MkId_list2 ( id ", " id_list )
        ;
series  :                    MkSeries1( statement )
        |      "2"     => MkSeries2( statement ";\n" series)
        ;
statement:                   "<statement>"
        |      ":="    => MkAsg_stat( /* error attribute: */ "\003\001"
                                    id " := " exp )
        |      "if"    => MkIf_stat( "if " exp " then" "\t\n"
                                   series "\b\n"
                                   "else" "\t\n"
                                   series "\b\n"
                                   "fi" )
        |      "wh"    => MkWhile_stat( "while " exp "\n"
                                      "do" "\t\n"
```

```
                                         series "\b\n"
                                         "od")
        ;
exp     :                     "<exp>"
        |          "+"     => MkPlus( "(" exp " + " exp ")" )
        |          "*"     => MkTimes( "(" exp " * " exp ")" )
        |                     MkUse ( ID /* error attribute: */ "\003\001" )
        |                     MkNumber( NUMBER )
        ;
id      :                     MkNullId("<name>")
        |                     MkId( ID )
        ;

/* PART III -- concrete syntax
   All productions of the concrete syntax produce an abstract
   syntax tree via the synthesized attribute "abs".
*/

Pico_program    {synthesized pico_program abs;};
Decls           {synthesized decls abs;};
Id_list         {synthesized id_list abs;};
Series          {synthesized series abs;};
Statement       {synthesized statement abs;};
Exp             {synthesized exp abs;};
Id              {synthesized ID abs;};

/* Declare priority and associativity of '+' and '*' */

left '+';
left '*';
Pico_program ::=
          PROGRAM Decls Series END     { Pico_program.abs =
                                         MkProgram(Decls.abs, Series.abs); }
        ;
Decls   ::=
          DECLARE Id_list ';'          { Decls.abs = MkDecls(Id_list.abs); }
        ;
Id_list ::=
          Id                           { Id_list.abs = MkId_list1(Id.abs); }
        | Id ',' Id_list               { Id_list$1.abs =
                                         MkId_list2( Id.abs, Id_list$2.abs); }
        ;
Series  ::=
          Statement                    { Series.abs = MkSeries1(Statement.abs); }
        | Statement ';' Series         { Series$1.abs =
                                         MkSeries2(Statement.abs, Series$2.abs);}
        ;
Statement ::=
          Id ASSIGN Exp                { Statement.abs =
                                         MkAsg_stat(Id, Exp.abs); }
        | IF Exp THEN Series ELSE Series FI
                                        { Statement.abs =
                                          MkIf_stat(Exp.abs, Series$1.abs,
                                                    Series$2.abs);}
        | WHILE Exp DO Series OD        { Statement.abs =
                                          MkWhile_stat(Exp.abs, Series.abs);}
        ;
Exp     ::=
          Exp '+' Exp                   { Exp$1.abs =
                                          MkPlus( Exp$2.abs, Exp$3.abs);}
        | Exp '*' Exp                   { Exp$1.abs =
                                          MkTimes(Exp$2.abs, Exp$3.abs); }
        | '(' Exp ')'                   { Exp$1.abs = Exp$2.abs; }
        | ID                            { Exp.abs = MkUse (ID); }
        | NUMBER                        { Exp.abs = MkNumber(NUMBER); }
        ;

Id      ::=
          ID                            { Id.abs = MkId(ID); }
        ;

/* Now correlate concrete and abstract rules with each other */

pico_program    ~ Pico_program.abs;
decls           ~ Decls.abs;
id_list         ~ Id_list.abs;
series          ~ Series.abs;
statement       ~ Statement.abs;
exp             ~ Exp.abs;
id              ~ Id.abs;

/* PART IV -- static semantic rules */

/* Definition of the environment datatype */

ENV     :       NullEnv( )
        |       EnvConcat( ID ENV)
        ;

ID lookup(id, env)
        ID id; ENV env;
        { return( with (env) (
                 NullEnv():      "?",
                 EnvConcat(i, e): i == id ? id : lookup(id, e)));
        };

/* Inherited and synthesized attributes in the abstract syntax tree */

pico_program {synthesized ENV env;};
decls        {synthesized ENV env;};
```

```
id_list       {synthesized ENV env;};
id            {synthesized ID name;};

/* definition of static semantic rules */

MkProgramBot    :: {};
MkProgram       :: { pico_program.env = decls.env; };
MkNullDecls     :: { decls.env = NullEnv(); };
MkDecls         :: { decls.env = id_list.env; };
MkId_list1      :: { id_list.env = EnvConcat(id.name, NullEnv()); };
MkId_list2      :: { id_list$1.env = EnvConcat(id.name, id_list$2.env); } ;
MkAsg_stat      :: { local STR error;
                     error = lookup(id.name, {pico_program.env}) == "?"
                             ? "UNDECLARED --> "
                             : "";
                   };
MkNullId        :: { id.name = "?"; };
MkId            :: { id.name = ID; };
MkUse           :: { local STR error;
                     error = lookup(ID, {pico_program.env}) == "?"
                             ? " <-- UNDECLARED"
                             : "" ;
                   };
```

Appendix III: PICO implemented with CEYX

```
; Definitions for the lexical items id and number

(deftype id 'is-id)
(de is-id (x) (and (atomp x) (not (numberp x))))

(deftype integer 'is-integer)
(de is-integer (x) (numberp x))

; Definitions for the abstract syntax trees

(defuniverse pico)
(defcons program~pico (decls series))
(defcons decls~pico id)
(defcons series~pico statement)

(defuniverse statement~pico)
(defcons if-stat~statement (exp statement statement))
(defcons while-stat~statement (exp statement))
(defcons asg-stat~statement (id exp))

(defuniverse exp~statement)
(defcons plus~exp (exp exp))
(defcons times~exp (exp exp))
(defcons use~exp (id))
(defcons number~exp (integer))

; Definitions for the unparsing rules

(defsem (program pretty) (x)
    (vterpri)
       (vpatom "program")
    (vblock-with-indent 4
       (cutpoint)
       (vdispatch (get-son x 1))
       (vpatom ";")
       (cutpoint)
       (vdispatch (get-son x 2)))
    (vterpri)
    (vpatom "end"))

(defsem (decls pretty) (x)
    (setq x (get-sons x))
    (hblock-with-indent 4
       (vpatom "declare ")
       (vdispatch (car x))
       (setq x (cdr x))
       (while x (vpatom ", ")
              (cutpoint)
              (vdispatch (car x))
              (setq x (cdr x)))))

(defsem (series pretty) (x)
     (setq x (get-sons x))
     (vblock-with-indent 0
        (vdispatch (car x))
        (setq x (cdr x))
        (while x (vpatom ";")
              (cutpoint)
              (vdispatch (car x))
              (setq x (cdr x)))))

(defsem (if-stat pretty) (x)
     (vblock-with-indent 0
        (hblock
           (vpatom "if")
           (cutpoint)
           (vdispatch (get-son x 1))) ; test-part
        (cutpoint)
        (vblock
           (vpatom "then")
           (cutpoint)
           (vdispatch (get-son x 2))) ; then-part
        (cutpoint)
        (vblock
           (vpatom "else")
           (cutpoint)
           (vdispatch (get-son x 3))) ; else-part
        (cutpoint)
        (vpatom "fi")))

(defsem (while-stat pretty) (x)
     (vblock-with-indent 0
        (hblock
           (vpatom "while")
           (cutpoint)
           (vdispatch (get-son x 1))) ; test-part
        (cutpoint)
        (vblock
           (vpatom "do")
           (cutpoint)
           (vdispatch (get-son x 2))) ; do-part
        (cutpoint)
        (vpatom "od")))

(defsem (asg-stat pretty) (x)
     (hblock
        (vdispatch (get-son x 1)); id
        (vpatom " := ")
        (vdispatch (get-son x 2))))   ; exp

(defsem (plus pretty) (x)
     (vdispatch (get-son x 1))    ; left operand
     (vpatom " + ")
     (vdispatch (get-son x 2)))   ; right operand

(defsem (times pretty) (x)
     (vdispatch (get-son x 1))    ; left operand
     (vpatom " * ")
     (vdispatch (get-son x 2)))   ; right operand

(defsem (use pretty) (x)
     (vpatom (get-son x 1)))

(defsem (number pretty) (x)
     (vpatom (get-son x 1)))


; Constructors for the editor

(de mk-program () (program (metavar 'decls) (metavar 'series)))
(de mk-decls () (decls (metavar 'id)))
(de mk-series () (series (metavar 'statement)))

(de mk-if-stat () (if-stat (metavar 'exp)
                           (metavar 'statement)
                           (metavar 'statement)))
(de mk-while-stat () (while-stat (metavar 'exp) (metavar 'statement)))
(de mk-asg-state () (asg-stat (metavar 'id) (metavar 'exp)))

(de mk-plus () (plus (metavar 'exp) (metavar 'exp)))
(de mk-times () (times (metavar 'exp) (metavar 'exp)))
(de mk-use () (use (metavar 'id)))
(de mk-number () (number (metavar 'integer)))

; Checking of static semantic constraints

(de all-declared (prog)
   (let ((id-list (get-sons (get-son prog 1)))
      (series (get-son prog 2)))
        (send 'not-declared series id-list)))

;  By default all tree nodes propagate the signal "not-declared"

(defsem (pico not-declared) (x id-list)
   (let ((sons (get-sons x)))
        (while sons
               (send 'not-declared (car sons) id-list)
               (setq sons (cdr sons)))))

; Special treatment of "not-declared" for asg-stat and exp

(defsem (asg-stat not-declared) (x id-list)
   (let ((id (get-son x 1))
         (exp (get-son x 2)))
        (is-declared id id-list)
        (send 'not-declared exp id-list)))

(defsem (use not-declared) (x id-list)
    (is-declared (get-son x 1) id-list))

(defsem (number not-declared) (x id-list))

(de is-declared (id id-list)
   (if (not (member id id-list))
       (print id " not declared")))
```