METAL: A FORMALISM TO SPECIFY FORMALISMS

G. KAHN, B. LANG, B. MÉLÈSE and E. MORCOS
I.N.R.I.A., Domaine de Voluceau-Rocquencourt, 78153 Le Chesnay Cedex, France

Communicated by G. Berry
Received April 1983

Introduction

Mentor is a general system for the manipulation of structured information. Its main application is the construction of interactive programming environments. In such an environment, a programmer may design, implement, document, debug, test, analyze, validate, maintain and transport his programs. An environment created with Mentor is realistic enough to handle large software developments, involving many programmers and many program modules. It provides a programming team with tools for specifying a design, enforcing a programming methodology and verifying interfaces. Mentor is implemented in Pascal. At the present time, it is being developed on a HB 68 computer under the MULTICS operating system. All features described in this paper are operational in the MULTICS version of Mentor, but for a few that will be singled out. Versions of Mentor exist on other computers/operating systems. However, these versions reflect the state of Mentor at some point in its development, and generally do not offer all the features of the current version. Mentor is in use at INRIA for both research and program development purposes. At several other sites, it is mostly used for research.

The kernel of Mentor is a syntax directed editor in which every object is represented as an operator-operand tree, usually called an abstract syntax tree. For a given language, a tree grammar, called the abstract syntax of the language, defines legal abstract syntax trees in that language. These tree data structures are shared by several processors which, together with the kernel, constitute the core of a programming environment with the following main characteristics:
- It is interactive. The system and the user communicate via an infinite request loop. The user enters commands from a terminal and the system answers directly or possibly in asking questions when more information is needed to complete the command.
- It is programmable. Mentor is driven by a specialized tree manipulation language, Mentol. This language is used both to interact with the system and to write procedures that may then be called in an interactive manner or by other Mentol
procedures. In other words, in the Mentor system, anything that may be done interactively may also be done by programming, and conversely. In fact, giving interactive control to the user is done by a special predefined procedure of Mentol. This procedure is automatically called at the beginning of a session and may also be called from any other Mentol procedure.

- It is language independent. All the components of Mentor are driven by tables that describe the formalisms manipulated. The formalisms are first described in a specification language called Metal. To add a new formalism to the system, a user needs only to compile its Metal definition into a set of tables using the Metal compiler. Metal is described in the first section of this paper. The Metal formalism is able to handle large, realistic programming languages. Ada [1] for example was introduced under Mentor by writing and compiling its Metal definition. At the present time, the available languages under Mentor are: Pascal, Ada, Metal, Flip [8], Asple [5] and Mentol plus a variety of smaller formalisms introduced by users for the purposes of their own activity.

- It is multilingual (i.e. multi-formalism). Several languages may be handled simultaneously and subtrees of the same tree may represent programs in different programming languages. For example, in a Mentor session, it is possible to handle trees in which some subtrees follow the Pascal abstract syntax and other subtrees follow the Ada abstract syntax. However, both abstract syntaxes are not mixed in an anarchic manner and there are special links in a tree through which the language may change.

In the first part of this paper, we describe Metal and show how to write a Metal program to define a given language, once we have a definition of its concrete syntax in a BNF-like format. In the second part, we explain briefly how the virtual tree building machine operates. The virtual tree building machine is the processor of Mentor that builds abstract trees during parsing of programs or program fragments.

1. The language Metal

1.1. Introduction

Metal is the specification language of the Mentor system. To add a new formalism, or programming language, to Mentor, it is necessary to write a Metal program that defines this formalism. Of course, Metal being one of the formalisms that may be manipulated under Mentor, it has been defined in itself.

The user who wants to add a new language to Mentor will naturally do so in the Mentor-Metal programming environment. In a Mentor-Metal session, the compile command creates the tables for the formalism defined by the current Metal program. Then, within the same session, Mentor may be directed to use this new formalism as current language. If the user wishes not to disturb his Metal session of Mentor
while he is testing his new formalism, he may initiate recursively a new session
with his formalism, do his testing, and then switch back to his former Metal session.
This process can be iterated several times, until the user is satisfied with his new
formalism, which can then be used like any other.

Of course, the first implementation of Metal under Mentor has been obtained
by bootstrap, that is hand-compiling the Metal program that defines the Metal
language itself.

Metal and Flip are two examples of languages whose compiler takes as input the
Mentor tree structure rather than the textual representation of programs. Ultimately
we would want this to be the general situation for all languages supported by Mentor.

A Metal program, say mp, that defines a formalism F, contains the following
components:

1. The **concrete syntax** of F. This is a set of rules that make it possible to decide
whether or not a given sentence belongs to F. These rules are written in a BNF-like
format. This collection of rules will be used to create a parser for F.

2. The **abstract syntax** of F. This stands for the syntax of trees that will represent
correct programs in the formalism F.

3. The **tree building functions**. These functions indicate which tree corresponds
to each component of the language. The tree generator of F is then created from
these functions. The interface between the parser and the tree generator is generated
mechanically.

Speaking in operational terms, each concrete syntax production is associated
with a semantic action, called here a tree building function. When parsing a program
in F, whenever a reduction is performed by the parser, the associated semantic
action is executed: a tree is built and pushed onto a stack that will possibly be used
by the following semantic actions.

4. The **unparsing specifications** of F. These specifications specify the converse
connection, between tree form and text form. Roughly speaking, they indicate the
piece of text associated with each elementary abstract tree.

The connection between text form and tree form of the same object is accom-
plished by two processors:

(i) the **universal parser-constructor** that builds a tree from the text form; it is
driven by tables compiled from the concrete syntax and the tree building functions;

(ii) the **unparser** that builds a text from an abstract tree, and is driven by tables
compiled from the unparsing specifications.

Among the four points above, the most crucial and difficult one for the user is
the definition of the abstract syntax. Once the abstract syntax is defined, points (3)
and (4) are derived fairly easily if the syntax is well designed. In addition, a well
designed abstract syntax will make it easier to write procedures to manipulate
abstract trees, i.e. to manipulate programs in F.

In the remainder of this section we describe the main features of Metal. To
accomplish this, we use a small language as an example: the language Asple [5],
introduced in [9] to compare formal definition methods.
A Metal program is organized into chapters and sections. The name of the language defined by the program is given in its header. The abstract syntax of the defined language is given in the *abstract syntax* sections whereas its concrete syntax is given in the *rules* sections. The formalism used in the *rules* sections is an extension of BNF. Each production of concrete syntax is associated with a function indicating how the abstract tree corresponding to the left hand side of the production, is built up from abstract trees corresponding to non-terminals of the right hand side of that production. Terminals (e.g., keywords, punctuation marks, parentheses, . . .) appearing in the right-hand side of a production are not taken into account in these functions because they will not appear in the abstract tree: there is enough information in the nodes of the abstract tree for the unparser to generate these terminals when creating the text form associated to an abstract tree.

The figure below shows the general structure of a Metal program. It is the Metal program that defines Metal itself, unparsed to a level of detail where all subtrees appearing deeper than the fourth level of the tree are replaced by an abbreviation mark. A # sign is used to abbreviate trees whose top operator is a fixed arity operator, while three dots · · · are used to abbreviate trees whose top operator is a list operator. This abbreviation mechanism is a standard feature of the Mentor system.

```plaintext
definition of Metal is
  rules
    # # # # # # #
  abstract syntax
    # # # # # # #
  chapter TRANS_GEN
    . . .
end chapter;
  chapter ABS_SYN_DESCRIPTION
    . . .
end chapter;
  chapter 'TERMINAL OBJECTS and lists of TERMINAL OBJECTS'
    . . .
end chapter;
  chapter ENTRY_POINTS
    . . .
end chapter;
  chapter ERROR_RECOVERY
    . . .
end chapter;
  chapter KEY_WORDS
    . . .
end chapter;
```
abstract syntax
#
end definition

In Appendix 1 we give the complete Metal program defining ASPLE. In Appendix 2 we give the Metal program that defines Metal itself. The reader who is curious about the complete structure of a Metal program is referred to these appendixes.

In this first part we will now discuss the path that the user is encouraged to follow when defining a new language in Metal. We begin by defining the concrete syntax of ASPLE, then we define its abstract syntax. Finally we write the tree building functions associated with each production of that concrete syntax.

1.2. The concrete syntax of ASPLE

The language ASPLE is a very small programming language entirely defined by its denotational semantics in [5].

\[
\begin{align*}
\langle \text{program} \rangle & ::= \text{begin} \langle \text{dcl\_train} \rangle \langle \text{stm\_train} \rangle \text{ end} \\
\langle \text{dcl\_train} \rangle & ::= \langle \text{declaration} \rangle \\
& \quad \mid \langle \text{dcl\_train} \rangle \langle \text{declaration} \rangle \\
\langle \text{declaration} \rangle & ::= \langle \text{mode} \rangle \langle \text{idlist} \rangle; \\
\langle \text{mode} \rangle & ::= \text{bool} \\
& \quad \mid \text{int} \\
& \quad \mid \text{ref} \langle \text{mode} \rangle \\
\langle \text{idlist} \rangle & ::= \langle \text{id} \rangle \\
& \quad \mid \langle \text{idlist} \rangle, \langle \text{id} \rangle \\
\langle \text{stm\_train} \rangle & ::= \langle \text{statement} \rangle \\
& \quad \mid \langle \text{stm\_train} \rangle; \langle \text{statement} \rangle \\
\langle \text{statement} \rangle & ::= \langle \text{asgt\_stm} \rangle \\
& \quad \mid \langle \text{cond\_stm} \rangle \\
& \quad \mid \langle \text{loop\_stm} \rangle \\
& \quad \mid \langle \text{transput\_stm} \rangle \\
\langle \text{asgt\_stm} \rangle & ::= \langle \text{id} \rangle := \langle \text{exp} \rangle \\
\langle \text{cond\_stm} \rangle & ::= \text{if} \langle \text{exp} \rangle \text{ then } \langle \text{stm\_train} \rangle \text{ fi} \\
& \quad \mid \text{if} \langle \text{exp} \rangle \text{ then } \langle \text{stm\_train} \rangle \text{ else } \langle \text{stm\_train} \rangle \text{ fi} \\
\langle \text{loop\_stm} \rangle & ::= \text{while} \langle \text{exp} \rangle \text{ do } \langle \text{stm\_train} \rangle \text{ end} \\
\langle \text{transput\_stm} \rangle & ::= \text{input} \langle \text{id} \rangle \\
& \quad \mid \text{output} \langle \text{exp} \rangle \\
\langle \text{exp} \rangle & ::= \langle \text{factor} \rangle \\
& \quad \mid \langle \text{exp} \rangle + \langle \text{factor} \rangle \\
\langle \text{factor} \rangle & ::= \langle \text{primary} \rangle \\
& \quad \mid \langle \text{factor} \rangle * \langle \text{primary} \rangle
\end{align*}
\]
\( \langle \text{primary} \rangle ::= \langle \text{id} \rangle \\
| \langle \text{constant} \rangle \\
| \langle \exp \rangle \\
| \langle \text{compare} \rangle \)

\( \langle \text{compare} \rangle ::= \langle \exp \rangle = \langle \exp \rangle \\
| \langle \exp \rangle \neq \langle \exp \rangle \)

\( \langle \text{id} \rangle ::= %\text{id} \)

\( \langle \text{constant} \rangle ::= \langle \text{bool} \rangle \\
| \langle \text{num} \rangle \)

\( \langle \text{bool} \rangle ::= \text{true} \\
| \text{false} \)

\( \langle \text{num} \rangle ::= %\text{NUMBER} \)

Identifiers starting with a percent sign, such as \( %\text{id} \) or \( %\text{NUMBER} \), stand for generic lexical items. They are defined by regular expressions in the scanner of the language \text{ASPLE}. We do not expand here on lexical definitions as they are the standard ones.

1.3. A sample \text{ASPLE} program

We show below an \text{ASPLE} program that reads an integer and computes its factorial:

\begin{verbatim}
begin
  int X, Y, Z;
  input X;
  Y := 1;
  Z := 1;
  if (X \neq 0) then
    while (Z \neq X) do
      Z := Z + 1;
      Y := Y * Z
    end
  fi;
  output Y
end
\end{verbatim}

1.4. The abstract syntax of \text{ASPLE}

The abstract syntax of a language is made of \textit{operators} and \textit{phyla}:

The \textit{operators} label the nodes of the abstract trees. Operators come in two kinds: fixed arity and list operators. The arity of fixed arity operators is arbitrarily limited to 3. Operators with arity zero are the leaves of abstract trees and represent atoms of the language. Operators of fixed, non null, arity can have offsprings of different kinds, whereas all offsprings of list operators must be of the same kind. The meaning of ‘kind of offspring’ is formalized below by the concept of \textit{phylum}.
The *phyla* are non empty sets of operators. To each offspring position of an operator is associated a phylum. This phylum contains precisely those operators that are allowed as head-operator of a subtree in this offspring position.

Thus each occurrence in an abstract syntax tree is associated with a phylum that indicates what operators are allowed at this location. This phylum depends only on (the operator of) the father of this occurrence, and on the rank of this occurrence as a son of its father. Mentor uses these relations between operators and phyla to maintain syntactically correct trees; i.e. trees that give syntactically correct programs when unparsed in the concrete language. Whenever a command that modifies a tree is executed, Mentor checks that this command is legal, i.e. whether the resulting abstract tree is correct according to operators-phyla relations. If the resulting abstract tree is not correct, an error occurs and the tree is not modified. The concept of phyla provides an easy and convenient way to precisely specify the set of trees that can be legally built from a given set of operators.

To define an abstract syntax one must describe both operators and phyla. For operators, this involves defining their arity and the phyla associated with their sons. For phyla, this means defining which operators they include. Definitions of operators and phyla can be interleaved and given in any order convenient for the reader, as can be done for rules of *BNF* definitions of concrete syntaxes.

Ideally the abstract syntax of a language should be defined before its concrete syntax, since its structure better emphasizes the concepts of the language, without diversions from lexical or syntactic analysis constraints. Unfortunately, for most already existing programming languages, only the concrete syntax has yet been standardized (more or less). Hence, for these languages, an abstract syntax must be defined. There is no really systematic way to do it, but we think that the following criteria should be kept in mind:

- Subtrees should always correspond to semantically significant concepts of the language.
- The abstract structures should be close enough to the concrete syntax so that the user may understand and remember them easily. For example, a preorder traversal of an abstract syntax tree should yield objects in the same order as in the concrete syntax of the language.
- Ordinary manipulations in the given language should be easy to do on the abstract tree. In most cases this is a consequence of the first point.
- Of course, the abstract syntax should also be as simple as possible. This last point will often be in conflict with the first and second points above, because semantically meaningless peculiarities often subsist in the concrete syntax of programming languages.

The abstract syntax of *ASPLE* is given below in Metal format: operators are in lower case, phyla are in upper case. To define an operator, one gives the phyla of its sons, and to define a phylum one enumerates the operators that belong to it. The arity of an operator is equal to the number of phyla on the right-hand side of the rule defining the operator. Symbols + ⋯ indicate that the operator is a list
operator, for a list that cannot be empty, while symbols ∗ · · · denote a list that can be empty. Note that a single phylum is given for all sons of a list operator. Let us look for instance at the abstract syntax of ASPLE.

- The operator *program* is binary, its first son is an operator belonging to the phylum DECLS and its second son belongs to the phylum STMS.
- The operator *stmts* is a non-empty list operator, the sons of which belong to the phylum STATEMENT.
- The operators *bool*, *int*, and *id* are nullary operators and they will be leaves of abstract trees.

```
program  → DECLS STMS;
decls    → DECLARATION + · · ·;
declaration → MODE IDLIST;
idlist   → ID + · · ·;
bool     → ;
int      → ;
ref      → MODE;
stmts    → STATEMENT + · · ·;
assign   → ID EXP;
ifthen   → EXP STMS;
ifthenelse → EXP STMS STMS;
while    → EXP STMS;
input    → ID;
output   → EXP;
plus     → EXP EXP;
times    → EXP EXP;
equal    → EXP EXP;
different → EXP EXP;
boolean  → ;
number   → ;
id       → ;
comment  → ;
comment_s → COMMENT * · · ·;
meta     → ;
deref    → EXP;
and      → EXP EXP;
or       → EXP EXP;
```

DECLS ::= decls;
STMS ::= stmts;
DECLARATION ::= declaration;
MODE ::= bool int ref;
IDLIST ::= idlist;

```
STATEMENT ::= assign ifthen ifthenelse while input output;
EXP ::= plus times ID equal different CONSTANT AUX;
CONSTANT ::= number boolean;
ID ::= id;
COMMENT ::= comment;
AUX ::= deref and or;

1.5. Defining ASPLE in Metal

Once we have the concrete syntax and the abstract syntax, we can write the Metal program that defines Mentor-ASPLE. An essential component of a Metal program is a list of rules that specify a concrete grammar together with a translation of concrete text representation to abstract tree representation. Each rule is a production of the concrete syntax together with a tree generating function. We explain in this paragraph how to write these rules. The Metal program that defines Mentor-ASPLE will be called ASPLE.Metal.

1.5.1. Atomic trees

Atomic trees are built with the atom constructor. This constructor takes the name of an atomic operator as a left operand and the value of the atom as a right operand. This value may be immediate or it may be obtained from the value of some lexical token. In the first case, it is a character string enclosed within simple quotes. For instance, let us consider the following production:

(id) ::= %ID

When the parser makes a reduction using this production, it means that a token, accepted by the regular expression that defines %ID in the scanner, has been met in the text. The generated tree must then be an atomic tree with the operator id of the abstract syntax of ASPLE, and its value must be the string representation of that token. The function that builds such a tree is as follows:

$id$-atom($%ID$)

Thus, in the Metal program we have the rule:

(id) ::= %ID;
$id$-atom($%ID$)

In addition, this rule means that, once built, the tree will be associated with the left-hand side non terminal (id) to be transmitted to the rest of the parsing.

There are four rules in the program ASPLE.Metal that build ASPLE atomic trees:

(id) ::= %ID;
$id$-atom($%ID$)
(num) ::= %NUMBER;
$number$-atom($%NUMBER$)
The last two rules are examples of atomic trees built up from immediate values.

1.5.2. Fixed arity operators

Consider now the following production, taken from the concrete syntax of ASPLE:

\[
\langle \text{mode} \rangle ::= \text{ref}(\text{mode});
\]

The tree generating function associated with this production will be

\[
\text{ref}((\text{mode}))
\]

This means that the corresponding tree will consist of the unary operator \text{ref} with, as a son, the tree associated with the non-terminal \langle \text{mode} \rangle. As before, this tree is associated with the left-hand side non-terminal to be transmitted to the rest of the parsing-generating process.

All fixed arity operators are built up in that manner, with the number of arguments being the arity of the operator. As an example of a binary operator generation, consider the following production:

\[
\langle \text{factor} \rangle ::= (\text{factor}) \# * (\text{primary});
\]

In this production the sharp sign, \#, that precedes the symbol * is the Metal forcing character. It has to precede every terminal which is also a Metal keyword or punctuation mark. Here the tree generating function will be

\[
times((\text{factor}), (\text{primary}))
\]

This means that a tree is built up with the binary operator \text{times} and, as first son, the tree associated with the non-terminal \langle \text{factor} \rangle and, as second son, the tree associated with the non-terminal \langle \text{primary} \rangle.

Here are the rules in the program ASPLE.Metal that build trees with fixed arity operators:

\[
\langle \text{program} \rangle ::= \text{begin}(\text{dcl_train})(\text{stm_train}) \# \text{end};
\]

\[
\text{program}((\text{dcl_train}), (\text{stm_train}))
\]

\[
\langle \text{declaration} \rangle ::= (\text{mode})(\text{idlist}) \# ; ;
\]

\[
\text{declaration}((\text{mode}), (\text{idlist}))
\]

\[
\langle \text{mode} \rangle ::= \text{ref}(\text{mode});
\]

\[
\text{ref}((\text{mode}))
\]

\[
\langle \text{asgt_stm} \rangle ::= (\text{id}) \# := (\text{exp});
\]

\[
\text{assign}(\text{id}, (\text{exp}))
\]

\[
\langle \text{cond_stm} \rangle ::= \text{if} (\text{exp}) \text{ then } (\text{stm_train}) \text{ fi};
\]

\[
\text{ifthen}(\text{exp}, (\text{stm_train}))
\]
\[\text{cond.stm} ::= \text{if } (\text{exp}) \text{ then } (\text{stm.train}) \text{ else } (\text{stm.train}) \text{ fi};\]
\[\text{ifthenelse} ((\text{exp}), (\text{stm.train}).1, (\text{stm.train}).2)\]
\[\text{loop.stm} ::= \text{while } (\text{exp}) \text{ do } (\text{stm.train}) \neq \text{end};\]
\[\text{while} ((\text{exp}), (\text{stm.train}))\]
\[\text{transput.stm} ::= \text{input} (\text{id});\]
\[\text{input} ((\text{id}))\]
\[\text{transput.stm} ::= \text{output} (\text{exp});\]
\[\text{output} ((\text{exp}))\]
\[\text{exp} ::= (\text{exp}) \# + (\text{factor});\]
\[\text{plus} ((\text{exp}), (\text{factor}))\]
\[\text{factor} ::= (\text{factor}) \# * (\text{primary});\]
\[\text{times} ((\text{factor}), (\text{primary}))\]
\[\text{compare} ::= (\text{exp}) \# = (\text{exp});\]
\[\text{equal} ((\text{exp}).1, (\text{exp}).2)\]
\[\text{compare} ::= (\text{exp}) \# \neq (\text{exp});\]
\[\text{different} ((\text{exp}).1, (\text{exp}).2)\]

In some rules a dot notation for non-terminals appears. This notation is used whenever there are several non-terminals with the same name in the right-hand side of a production. The number at the right of the dot is the ordinal of the non-terminal among non-terminals of the same name in the right-hand side of the production. Using this dot notation, one can refer to non-terminals in the tree-generating functions without any ambiguity.

1.5.3. List operators

We explain now how to build list operators. Consider the following two productions that define a statement train in the concrete syntax of ASPLE:

\[\text{stm.train} ::= (\text{statement});\]
\[\text{stm.train} ::= (\text{stm.train}) \# ; (\text{statement});\]

When the parser makes a reduction using the first of these, one must build a statement list containing only one statement. Each following reduction that uses the second production above recognizes a statement that must be inserted in the list. The tree generating function associated with the first production is then

\[\text{stms-list} (((\text{statement})))\]

The list constructor builds a list operator with a fixed number of items in the list. The name of the operator is given as the left parameter of the list constructor while the list of items to be put as sons of the list operator, is given as the right argument of the constructor. In the function above the list to be built has only one item; the tree associated with the non-terminal (statement). The function associated with the second production is

\[\text{stms-post} ((\text{stm.train}), (\text{statement}))\]
Like the \textit{list} constructor, the \textit{post} constructor takes a list operator as left operand. Its first right argument must denote a list with the same operator and, its second right argument is an item that will be added at the end of this list. In the function above, the non-terminal \((\mathrm{stm\_train})\) denotes the already built list of statements, and \((\mathrm{statement})\) denotes the new element to be appended to this list. Finally, to build statements lists in \texttt{ASPLE} we have the two following rules:

\[
\begin{align*}
(\mathrm{stm\_train}) &::= (\mathrm{statement}); \\
\text{stms-list}((\mathrm{stm\_train})) &::= (\mathrm{stm\_train}) \\
(\mathrm{stm\_train}) &::= (\mathrm{stm\_train}) \#; (\mathrm{statement}); \\
\text{stms-post}((\mathrm{stm\_train}), (\mathrm{statement})) &::= (\mathrm{stm\_train}) \#; (\mathrm{statement}); \\
\end{align*}
\]

There exists a \textit{pre} constructor that adds items at the beginning of the lists. This constructor has the same syntax as the \textit{post} constructor, but its right arguments are in the reverse order.

\section{Pattern matching primitives}

The techniques that we have seen so far are not sufficient, in general, to handle every situation that may arise when defining a new formalism in Metal. The Metal \texttt{let} and \texttt{case} expressions will allow us to test the shape of already built trees, to split them and to use their components in building other trees. These Metal expressions are based on \textit{tree pattern matching}.

\subsection{The case expression}

Consider the following productions in the concrete syntax of \texttt{ASPLE}:

\[
\begin{align*}
(\mathrm{cond\_stm}) &::= \text{if } (\mathrm{exp}) \text{ then } (\mathrm{stm\_train}) \text{ fi}; \\
(\mathrm{cond\_stm}) &::= \text{if } (\mathrm{exp}) \text{ then } (\mathrm{stm\_train}) \\
&\text{ else } (\mathrm{stm\_train}) \text{ fi}; \\
\end{align*}
\]

and let us rewrite them in another way, syntactically equivalent:

\[
\begin{align*}
(\mathrm{cond\_stm}) &::= \text{if } (\mathrm{exp}) \text{ then } (\mathrm{cond\_stm\_end}); \\
(\mathrm{cond\_stm\_end}) &::= (\mathrm{stm\_train}) \text{ fi}; \\
(\mathrm{cond\_stm\_end}) &::= (\mathrm{stm\_train}) \text{ else } (\mathrm{stm\_train}) \text{ fi}; \\
\end{align*}
\]

If the rules had been written that way, to obtain the same trees as in Section 1.5.2, the tree generating functions should be written as follows:

\[
\begin{align*}
(\mathrm{cond\_stm\_end}) &::= (\mathrm{stm\_train}) \text{ fi}; \\
\text{ifthen} (\text{null\_tree}, (\mathrm{stm\_train})) &::= (\mathrm{stm\_train}) \text{ else } (\mathrm{stm\_train}) \text{ fi}; \\
\text{ifthenelse} (\text{null\_tree}, (\mathrm{stm\_train}).1, (\mathrm{stm\_train}).2) &::= (\mathrm{stm\_train}) \text{ else } (\mathrm{stm\_train}) \text{ fi}; \\
(\mathrm{cond\_stm}) &::= \text{if } (\mathrm{exp}) \text{ then } (\mathrm{cond\_stm\_end}); \\
\text{case}(\mathrm{cond\_stm\_end}) &::= (\mathrm{stm\_train}) \text{ fi}; \\
\end{align*}
\]
when ifthen(X, Y)
    ⇒ ifthen((exp), Y)
when ifthenelse(X, Y, Z)
    ⇒ ifthenelse((exp), Y, Z)
end case

In the first two rules, operators ifthen and ifthenelse are built without knowing their first son. This first son is denoted by the operator null_tree to mark the fact that it is not meaningful. In the third rule, the tree associated with the nonterminal (cond_stm_end) is used as a selector in the case expression. It will be matched with all schemas, one after another, until a successful match occurs. The expression found after the arrow is then evaluated to produce a tree that is then returned by the case expression.

In that example the case comprises two alternatives. Each alternative has the form

when schema
    ⇒ function

We call schema here an expression denoting a constant tree. A schema may contain metavariables (we note them with upper case identifiers) that will be instantiated during pattern matching. In the tree generating function following the arrow, a metavariable of the schema refers to the tree to which it has been bound. Thus, one retrieves components of the tree that was split by matching.

In the case expression above, the tree associated with the non-terminal (cond_stm_end) is first matched with the schema ifthen(X, Y). If the matching succeeds, meta-variables X and Y are respectively bound to the first and second son of this tree. Then the function ifthen((exp), Y) is used. The tree it denotes is then the tree denoted by the whole case expression. If this matching fails, the tree associated with the non-terminal (cond_stm_end) is matched by the schema of the second alternative: ifthenelse(X, Y, Z). If this matching fails again, the case expression fails and returns a special error tree called ERRONEOUS-TREE. If it succeeds, meta-variables X, Y, and Z are respectively bound to the first, second and third sons of the tree, the function ifthenelse((exp), Y, Z) is used, etc . . .

Of course, a correct Metal program should never cause a case expression to fail and return ERRONEOUS-TREE when the input text contains no error. However, syntactic error recovery on erroneous input can lead, directly or through failure of a case or let expression, to the production of ERRONEOUS-TREES.

1.6.2. The let expression

Consider again the same productions and rewrite them in a third manner, which is again syntactically equivalent:

\[(\text{cond}\_\text{stm}\_\text{head}) := \text{if} (\exp) \text{then} (\text{stm}\_\text{train});\]

\[(\text{cond}\_\text{stm}) := (\text{cond}\_\text{stm}\_\text{head}) \text{fi};\]

\[(\text{cond}\_\text{stm}) := (\text{cond}\_\text{stm}\_\text{head}) \text{else} (\text{stm}\_\text{train}) \text{fi};\]
To obtain the same trees as above, one should write the tree generating functions as follows:

\[
\begin{align*}
\langle \text{cond\_stm\_head} \rangle &::= \text{if}\ (\exp) \ \text{then}\ \langle \text{stm\_train} \rangle; \\
& \quad \text{ifthen}\ (\exp),\ (\text{stm\_train}) \\
\langle \text{cond\_stm} \rangle &::= \langle \text{cond\_stm\_head} \rangle \ \text{fi}; \\
\langle \text{cond\_stm\_head} \rangle &::= \langle \text{cond\_stm\_head} \rangle \ \text{else}\ (\text{stm\_train}) \ \text{fi}; \\
\langle \text{cond\_stm} \rangle &::= \langle \text{cond\_stm\_head} \rangle \ \text{else}\ (\text{stm\_train}) \ \text{fi}; \\
\end{align*}
\]

let \text{ifthen}(X, Y) = \langle \text{cond\_stm\_head} \rangle \ \text{in} \\
\text{ifthenelse}(X, Y, (\text{stm\_train}))

In the first rule, we build a tree with the operator \text{ifthen} in all cases although we do not know yet whether the right operator will be \text{ifthen} or \text{ifthenelse}. We could also build a tree with top operator \text{ifthenelse} and a meaningless third son as was the case with \text{null\_tree} in the previous section, but it would have made the function of the second rule more complicated. Here, in the second rule, we just return the tree associated with non-terminal \langle \text{cond\_stm\_head} \rangle. In the third rule, the tree associated with non-terminal \langle \text{cond\_stm\_head} \rangle is matched by the schema \text{ifthen}(X, Y). If this matching fails, the let expression fails and returns the tree \text{ERRONEOUS\_TREE}. If the matching succeeds meta-variables \(X\) and \(Y\) are respectively bound to the first and second sons of the tree denoted by \langle \text{cond\_stm\_head} \rangle and the function \text{ifthenelse}(X, Y, (\text{stm\_train})) is then used.

We may note that a let expression is equivalent to a case expression with only one alternative.

From the preceding discussion it might appear that let and case expressions are only useful for rectifying poorly written concrete syntax productions. This may in fact be true, but in some cases let and case cannot be avoided because the concrete syntax was designed without considering how it could be used to build abstract trees in a Mentor-like system. Furthermore, one may want to write twisted concrete syntax productions so that they may be used to generate a parser using a \textsc{LALR} general parser. In any case, the examples given above show that, to simplify the tree generating functions, one should try to write productions in such a way that, for each one of them, a actual subtree could be built.

1.7. Unparsing

Unparsing functions are not yet implemented in Metal and thus we shall describe them only very briefly to give the flavour of our approach to this problem.

Unparsing—or pretty-printing—consists in producing a textual representation for a tree. In spite of an apparent simplicity, the problem requires careful treatment in Mentor because:

1. The output of the unparsers is almost constantly on the screen, in front of a finicky user.

2. Unparsing takes place very often, especially when a video interface is being used, so that it has to be efficient.
In keeping with the rest of the system, it must be table-driven. In Metal, an unparser is specified by a collection of rules of the form:

\[ \text{Pattern} \rightarrow \text{Format} \]

The left-hand side of each rule is a pattern built with distinct metavariables and the exact same constructors that are used in the tree building functions. The right-hand side is a format that contains instructions to print specific strings, various diacritical formatting symbols, and metavariables introduced in the left-hand side that denote the result of unparsing the corresponding subtrees. For example, a very simple rule to unparse assignments would be:

\[ \text{assign}(\text{NAME}, \text{EXP}) \rightarrow \text{NAME} := \text{EXP} \]

Roughly speaking, the meaning of such a rule is that, to unparse a tree whose top node is the operator assign, one should unparse its first son, print ": =" and unparse its second son. In a collection of rules of this type, we do not require that the patterns on the left-hand side be simple patterns with only one operator. On the contrary, we often wish to specify special cases. For example:

\[
\begin{align*}
\text{if}(\text{EXP}, \text{STAT1}, \text{STAT2}) & \rightarrow \text{format1} \\
\text{if}(\text{EXP}, \text{STAT1}, \text{void}) & \rightarrow \text{format2}
\end{align*}
\]

Naturally, the most specific pattern should dictate which rule to apply. To make sure that we can always select a rule unambiguously, we insist that the collection of left-hand sides be closed under unification, a fact which is easy to check. Unparsing of a tree \( t \) proceeds as follows: \( t \) is matched against the left-hand sides of the rules. The unique rule corresponding to the most specific match is selected, and the metavariables of its left-hand side are bound to the appropriate subtrees. Unparsing is called recursively on these subtrees following the prescriptions of the format on the right-hand side. An automaton (decision tree) is compiled from the collection of left-hand sides to perform a simultaneous matching efficiently.

In fact, the formalism used is somewhat more complex than appears in this sketchy description for several reasons. First, an abbreviation mechanism allows to specify as a single rule several rules that have almost the same shape. Second, the formats contain diacritical symbols to specify potentially missing subtrees, where to preferably cut in case of line overflow, where to add special blanks etc ... A simple mechanism gives control over the holophrasting level, which permits a flexible method for ellipsis. It is also possible to have several collections of rules corresponding to unparsing in different contexts.

Processing of formats is dealt by a unique formatting virtual machine [15]. The symmetry of unparsing with parsing is striking: in both cases, a general pattern-matching/tree-traversal automaton decides what action should be taken and performs some sort of binding. The action itself is executed by an abstract machine driven by a microprogram-like machine code. These actions are purely applicative in that they do not modify any global state. Furthermore, since a tree may be
annotated with other trees, that may have to be unparsed, it is necessary to be able to switch unparsers in midstream. This is easily achieved with the design above, but the specification of interface between texts is not always easy.

In the current state of the Mentor system, the unparser for a new language must be written in Pascal. A standard unparser skeleton is available so that the user has fairly simple code to fill in, before obtaining a reasonable unparser. Mentor has a generic display processor. Thus when writing an unparser the user is not concerned with the display interface between Mentor and his terminal. To allow debugging of a new language prior to writing its unparser, Mentor has a generic unparser capable of displaying any tree in a standard format.

1.8. Parsers and scanners

Mentor includes a scanner and a parser generator that are used to build scanning and parsing tables for the new languages defined in Metal. Since we are using table driven scanners and parsers, the user does not have to worry about connecting new scanners and parsers to the Mentor system.

On the MULTICS system at I.N.R.I.A., the general scanner and parser we use is the SYNTAX system. This general purpose system has been designed and implemented by the "Languages and Translators" project at I.N.R.I.A. However, what Mentor needs from a general parser is only

1. To identify the last reduction performed by the parser and
2. To obtain token values from the scanner.

Thus Mentor should be able to run with any reasonable universal parser. However, as we shall see below, Mentor also needs to keep program comments, and thus requires scanners to return them in some fashion. Unfortunately, a large number of available scanner generators do not meet this need.

1.9. Comments and annotations in trees

Although not generally used during execution, comments are an essential constituent of programs and a system like Mentor must be able to handle these informations in a manner consistent with the abstract syntax view of programs. In Mentor, comments are not necessarily restricted to a textual form, but may be any information written in its own language and, then, represented by an abstract syntax tree in the abstract syntax of this language. For instance, one may want to comment some piece of code by first order logic assertions, by a corresponding piece of code in another programming language, or simply, by an equivalent more readable piece of code in the same programming language.

To cope with such a general situation, we have designed and implemented, in Mentor, a general mechanism: any node of any abstract syntax tree may receive an arbitrary number of annotations of different kinds. Each kind of annotation is characterized by a name and the language (formalism) in which it is written.
This approach has two main consequences:
- annotations may have annotations recursively,
- annotations may be handled by Mentol commands in exactly the same manner as other trees.

The main difference between the operands (i.e. sons) of a node and its annotations is that the existence and nature of operands is fixed once for all by the abstract syntax of the language. This is not the case for annotations: new kinds of annotations may be dynamically created, and annotations may be freely and dynamically attached or detached from any node. Another difference is that operands are seen by all processors, but annotations of a given kind may be made visible or invisible to any processor. For example, the pattern matching processor may check also the matching of some annotations if their kind has been specified visible for pattern matching. The unparsers itself is no exception and it unparses only annotations that are specified unparsable. Attributes may naturally be interpreted by some Mentor processors that modify their processing accordingly. Typically, a program may contain pretty-printing annotations which are not generally printed, but nevertheless modify the result of an unparsing.

In all languages existing under Mentor, two annotations are automatically reserved for ordinary comments, the so-called prefix and postfix comments. These annotations are both expressed in terms of a degenerated abstract syntax containing only two operators:
- comment, which is atomic and whose value is a comment line, and
- comment_s (list of comments) which is a list of comment lines.

A fundamental problem of annotations is the determination of the fragment of text that is being annotated. Take for example the following fragment in the Pascal language:

```pascal
while x <> 0 do x := x - a (*assert: x <= 0*)
```

There is no syntactic clue to the part of the program that is actually annotated. It could be as well the variable "a", the expression "x - a", the assignment or the whole loop.

Since Mentor objects are always kept and stored in tree form, annotations can be attached unambiguously to any subtree, i.e. to any meaningful fragment of program, either as prefix or as postfix annotations. However, since programs are usually entered in Mentor by parsing textual input, the system has to resolve ambiguities of annotated text during parsing and program construction.

In the current release of the system, hooking comments in the tree while parsing is done in a standard, language independent, way. This means that the user doesn't have to worry about comments when designing a language. To determine during parsing whether a given comment must be prefix or postfix of a node, and of what node, the system uses heuristics that take in account the state of the parser stack. Of course, comments that have been unsatisfactorily attached by these heuristics
can be moved later to the desired node in the tree by standard Mentor manipulation commands.

In the next version of Metal, we plan to add two constructors for annotations, called prefix and postfix, to allow the user to specify precisely in its Metal program where comments occurring in the input text must be attached into the tree in exactly the same style he specifies how to build the tree. These constructors are infix.

The prefix constructor takes a comment as left operand and a tree as right operand. It returns a tree which is its right operand with its left operand tied as a prefix comment of the top operator of that tree.

The postfix constructor takes a tree as left operand and a comment as right operand. It returns a tree which is its left operand with its right operand tied as a postfix comment of the top operator of that tree.

Like the other tree constructors, annotation constructors can be used in tree pattern matching to undo annotations and reorganize them differently. We show below examples of tree building functions using annotated tree pattern matching to move comments into a tree.

Example 1. Consider the following rule from the Metal program defining ASPLE:

\[
\text{(exp)}::=\text{(exp)} \# + \text{(factor)}; \\
\text{plus}(\text{(exp)}, \text{(factor)})
\]

and suppose that the user wants to express the fact that each time a comment comes in prefix of the expression he wants to move it up to be prefix of the plus operator. Then, he has to rewrite this rule in the following manner:

\[
\text{(exp)}::=\text{(exp)} \# + \text{(factor)}; \\
\text{let } C \text{ prefix } X = \text{(exp)} \\
\text{in } C \text{ prefix } \text{plus}(X, \text{(factor)})
\]

In this rule, \( C \) is an annotation metavariable and \( X \) is a tree metavariable. The pattern matching of the tree \( \text{(exp)} \) against the annotated schema \( C \text{ prefix } X \) associates the annotation metavariable \( C \) with the prefix comment of the tree denoted by the non-terminal \( \text{(exp)} \) if any, and the tree metavariable \( X \) to the tree denoted by \( \text{(exp)} \) without its prefix comment. Then, the tree building function \( C \text{ prefix } \text{plus}(X, \text{(factor)}) \) returns a tree which is the same tree as in the original rule but with the prefix comment of the first son being tied in prefix to the plus operator.

Example 2. Consider now the following rule, again from the Metal definition of ASPLE:

\[
\text{(loop}_\text{stm})::=\text{while } \text{\textbf{exp}} \text{ do } (\text{stm}_\text{train}) \# \text{ end}; \\
\text{while}(\text{\textbf{exp}}, \text{(stm}_\text{train}))
\]

We assume now that the user wants to move the comments appearing in the list of statements denoted by the non-terminal \( (\text{stm}_\text{train}) \), if any, in the following manner:
- if the first statement of the list has a prefix comment, he wants to move it as a
  prefix comment of the whole list and,
- if the last statement of the list has a postfix comment, he wants to move it as a
  postfix comment of the while statement.

To do that, the above rule must be rewritten as follows:

\[
\text{loop}_{stn} ::= \text{while } (\exp) \text{ do } (\text{stm}\_\text{train}) \# \text{end};
\]

\[
\text{let stms-pre}(C1 \text{ prefix } X, Y) = (\text{stm}\_\text{train}) \text{ in}
\]

\[
\text{let stms-post}(Z, C2 \text{ postfix } T) = (\text{stm}\_\text{train}) \text{ in}
\]

\[
\text{while } ((\exp), \text{ C1 prefix } (\text{stm}\_\text{train})) \text{ postfix } C2
\]

In this rule, \(C1\) and \(C2\) are annotation metavariables and \(X, Y, Z, T\) are tree
metavariables. Among these four tree metavariables, \(Y\) and \(Z\) are metavariables
of list. The different kinds of metavariables are identified by the place where they
appear in the schema.

2. The virtual tree processor

We call Virtual Tree Processor the Mentor processor that builds abstract trees
as a program fragment is being parsed. When the parser calls this processor, it
executes code that has been created by the Metal compiler from the tree generating
functions associated to every production in the Metal rules. This processor works
like a machine whose elementary instructions are tree building instructions and
whose memory is organized to handle tree structures in a natural way. It uses a
stack where previously built trees are stored temporarily. Ideally, this processor
should be realized by hardware.

We describe below a few basic instructions of the virtual tree processor, to give
a general idea of its operation. A complete documentation for this processor will
be given in a subsequent paper.

2.1. Tree building commands

These instructions take the operator name as parameter, find on the stack the
subtrees to be used as sons of the tree to be built. Then the newly constructed tree
is pushed onto the stack. A typical example of such an instruction is the following:

- \textbf{MK2 op}: Builds a binary tree with the operator op. Takes the top of the stack
  as first son and pop the stack. Takes the new top as second son and
  pop the stack again. The resulting tree is pushed onto the tack.

On the same pattern as \textbf{MK2}, there are of course instructions to build nullary,
unary, and ternary trees (\textbf{MK0}, \textbf{MK1}, \textbf{MK3}) and to build lists (\textbf{MKLIST}). To build
lists, there are two more instructions, \textbf{POST} and \textbf{PRE} that add elements at the end
(resp. at the beginning) of an existing list. The list and the elements are all taken
on the stack. A numeric parameter indicates the number of elements from the
stack to be added to the list.
2.2. Tree matching commands

In addition to code for the tree building functions, the Metal compiler generates a coded form of the schemas used in the tree matching primitives (let and case), in which, in particular, the names of meta-variables have been replaced by positive integers. These coded forms of the schemas are stored in a table used by the MATCH command of the virtual tree processor.

**MATCH n:** Matches a tree taken at the top of the stack with the schema number $n$. This instruction returns true or false in the logical register depending on success or failure of the matching. If matching succeeds and if schema number $n$ contains meta-variables, an environment is built where each meta-variable is associated (bound) with the subtree it matched.

**GMETA n:** Get-Meta-variable. Pushes onto the stack the tree bound to the meta-variable number $n$ in the current environment.

2.3. Stack commands

**PU:** Pop the stack

**PD n:** Pushes contents of register number $n$ onto the stack. At the present time, the virtual tree processor works with 9 registers.

**INI n:** Loads in reverse order the $n$ first registers ($N \leq 9$) with the $n$ top elements of the stack and pop these elements.

2.4. Miscellaneous commands

The virtual tree processor has other instructions like STOP, to stop execution, JPT to jump to an address, and JPF to jump to an address depending on the value of the logical register. In the examples given below, the arguments of jumps are addresses relative to the address of the corresponding jump instruction. When this code is loaded to be executed, these relative addresses are translated into absolute addresses. Note for reading examples that a command and its parameter take one address position each.

Instructions that often appear as the last instruction of a fragment of code (i.e. is just before a STOP instruction) have an equivalent instruction that in addition stops execution. We shall see examples of some of these in the following section (MK1STOP, MK2STOP, POSTSTOP, · · · ).

2.5. Examples of virtual tree processor code

We show below the code generated by the Metal compiler for some of the tree building functions we have seen before in this paper. On each line, the code used by the virtual tree processor is on the left of the colon. On the right the same code is disassembled.
Metal: A formalism to specify formalisms

\[
\langle \text{program} \rangle ::= \textbf{begin} \langle \text{dcl}\_\text{train} \rangle \langle \text{stm}\_\text{train} \rangle \neq \textbf{end};
\]

\[
\text{program}(\langle \text{dcl}\_\text{train} \rangle, \langle \text{stm}\_\text{train} \rangle)
\]

38 19 : MK2STOP PROGRAM

\[
\langle \text{dcl}\_\text{train} \rangle := \langle \text{declaration} \rangle;
\]

\[
decls\_\text{list}(\langle \langle \text{declaration} \rangle \rangle)
\]

40 32 : MKLSTOP DECLS

\[
\langle \text{dcl}\_\text{train} \rangle := \langle \text{dcl}\_\text{train} \rangle \langle \text{declaration} \rangle;
\]

\[
decls\_\text{list}(\langle \langle \text{declaration} \rangle \rangle)
\]

46 1 : POSTSTOP 1

Samples of rules using \textit{let} and \textit{case} expressions

\[
\langle \text{cond}\_\text{stm} \rangle ::= \textbf{if} \langle \text{exp} \rangle \textbf{then} \langle \text{cond}\_\text{stm}\_\text{end} \rangle;
\]

\[
\text{case} \langle \text{cond}\_\text{stm}\_\text{end} \rangle
\]

\[
\text{when} \textit{ifthen} (\langle X, Y \rangle)
\]

\[
\Rightarrow \textit{ifthen}(\langle \text{exp}, Y \rangle)
\]

\[
\text{when} \textit{ifthenelse} (\langle X, Y, Z \rangle)
\]

\[
\Rightarrow \textit{ifthenelse}(\langle \text{exp}, Y, Z \rangle)
\]

\textit{end case}

When execution of the corresponding code below begins, the stack contains at its top the tree built for \langle \text{cond}\_\text{stm}\_\text{end} \rangle, and just below the tree built for \langle \text{exp} \rangle. The first command INI loads these trees in registers 1 and 2, \langle \text{exp} \rangle being loaded in 1.

\[
\begin{align*}
30 & 2: \text{INI} 2 \\
20 & 2: \text{PD} 2 \\
33 & 1: \text{MATCH} 1 \\
27 & 7: \text{JPF} 7 \\
18 & 2: \text{GMETA} Y \\
20 & 1: \text{PD} 1 \\
38 & 22: \text{MK2STOP IFTHEN} \\
33 & 2: \text{MATCH} 2 \\
27 & 9: \text{JPF} 9 \\
18 & 3: \text{GMETA} Z \\
18 & 2: \text{GMETA} Y \\
20 & 1: \text{PD} 1 \\
39 & 40: \text{MK3STOP IFTHENELSE} \\
32 & 1: \text{ERROR} 1 \\
4 & : \text{STOP}
\end{align*}
\]

\[
\langle \text{cond}\_\text{stm}\_\text{head} \rangle ::= \textbf{if} \langle \text{exp} \rangle \textbf{then} \langle \text{stm}\_\text{train} \rangle;
\]

\[
\text{ifthen}(\langle \text{exp}, \langle \text{stm}\_\text{train} \rangle \rangle)
\]

38 22 : MK2STOP IFTHEN

\[
\langle \text{cond}\_\text{stm} \rangle ::= \langle \text{cond}\_\text{stm}\_\text{head} \rangle \textbf{fi};
\]

\[
\langle \text{cond}\_\text{stm}\_\text{head} \rangle
\]

4 : STOP
Let \( \text{ifthen} \) \((X, Y) = (\text{cond_stm_head}) \) in
\[ \text{ifthenelse} (X, Y, (\text{stm_train})) \]

For the sake of readability, the above examples are slightly hand-modified versions of the code presently generated by the Metal compiler. As currently generated, the code is fairly crude and needs to be improved in several ways. A crucial factor is the size of that code since both its loading time and its running time depend directly on that factor. With the existing code generator, the complete code needed for the generation of Ada trees is about 1500 integers in the range 0.255, plus about 400 integers in the same range for the representation of schemas. The size of data to be loaded for the generation of Ada abstract syntax trees is about 2000 bytes. We think that this size can be reduced by 20 or 25 per cent by code improvements.

The major improvement done on the code in the current version of the system is the sharing of the some pieces of code between several rules. Currently the sharing is still limited to trivial cases since two rules share a piece of code only when the whole code is the same for both of them. The next step in this direction is to generalize the sharing to smaller pieces of code even if the code for a given rule has to be broken into separate fragments.

With this trivial sharing the reduction of code size is already considerable. For example, the Metal program defining Ada had initially about 550 rules to which 200 were added to implement the entry points in the parser (see the chapters ‘ENTRY POINTS’ in the Metal programs of Appendixes 1 and 2). After the sharing between rules was done only 250 rules remained, that is to say that only 250 different pieces of code appeared among the 750 initially generated.

Appendix 1. The program ASPLE.Metal

In this Appendix, we give the complete Metal program that defines Mentor-ASPLE. This program is structured in chapters, with rules and abstract syntax zones. The goal of such a structure is to make the program more readable and easier to handle in Mentor-Metal. For instance, chapter names serve to introduce the various
components of an ASPLE program. Likewise, the breaking of rules and abstract syntax items in different rules and abstract syntax zones promotes a logical organization of the Metal program. The user may group together related rules, operators and phyla. Notice that the ordering of rules in rules zones and of operators and phyla in abstract syntax zones has no influence on the meaning of the Metal program (hence on the language defined by this program). However the language name given in the program heading is meaningful: it will be the actual name of the newly defined language under Mentor.

The last chapter in the Metal program given below defines the entry points in the ASPLE parser. Remember that, in Mentor every program fragment entered from a file or from a user terminal is parsed and converted into its abstract tree form. Thus, the parser must be able to parse each subexpression of the language. The names of entry points are names of phyla. They are enclosed in brackets at the beginning of the right hand side of productions.

In some abstract syntax zones, one can see implementation directives, on the right-hand side of the definitions of nullary operators. (Expressions like implemented as SOMETHING). These expressions may be omitted. When they appear, they give a hint to Mentor in choosing the good way to implement the corresponding atomic operator.

It is possible to comment a Metal program (although there is no comment in the following example): every line beginning with a star will be understood by Metal processors as a comment.

The following program contains a phylum COMMENT and operators comment and comment_s. Together they define the trees which will be built for comments in ASPLE.

The sharp sign # that precedes certain terminals on the right-hand side of productions is the Metal forcing character. It has to precede every terminal which is also a Metal keyword or punctuation mark.

Definition of ASPLE is

chapter 'Programs in ASPLE'
rules
⟨axiom⟩ ::= ⟨program⟩;
⟨program⟩
⟨program⟩ ::= begin (dcl_train)(stm_train) # end;
program ((dcl_train),(stm_train))
abstract syntax

program → DECLS STMS;
PROGRAM ::= program ;
DECLS ::= decls ;
STMS ::= stms ;
end chapter;
chapter 'Declarations'
rules
(dcl_train) ::= (declaration);
decls-list(((decl_train)))
(dcl_train) ::= (dcl_train)(declaration);
decls-post((dcl_train),(declaration))
(declaration) ::= (mode)(idlist) # ;
declaration ((mode),(idlist))
(mode) ::= bool;
bool( )
(mode) ::= int;
int( )
(mode) ::= ref(mode);
ref((mode))
(idlist) ::= (id);
idlist-list(((id)))
(idlist) ::= (idlist) # , (id);
idlist-post((idlist),(id))
abstract syntax
decls → DECLARATION + ···;
declaration → MODE IDLIST;
idlist → ID + ···;
bool → implemented as SINGLETON;
int → implemented as SINGLETON;
ref → MODE;

DECLARATION ::= declaration ;
MODE ::= bool int ref ;
idLIST ::= idlist ;
end chapter;

chapter 'Statements'
rules
(stm_train) ::= (statement);
stmts-list(((stm_train)))
(stm_train) ::= (stm_train) # ; (statement);
stmts-post((stm_train),(statement))
(statement) ::= (asgt_stm);
(asgt_stm)
(statement) ::= (cond_stm);
(cond_stm)
(statement) ::= (loop_stm);
(loop_stm)
Metal: A formalism to specify formalisms

\[
\begin{align*}
\text{STATEMENT} &::= \text{assign \;} \text{ifthen \;} \text{ifthenelse \;} \text{while} \\
\text{assign} &::= \langle \text{id} \rangle \# \::= \langle \text{exp} \rangle \\
\text{ifthen} &::= \langle \text{exp} \rangle \text{then} \langle \text{stm\_train} \rangle \text{fi} \\
\text{ifthenelse} &::= \langle \text{exp} \rangle \langle \text{stm\_train} \rangle.1,\langle \text{stm\_train} \rangle.2 \\
\text{while} &::= \langle \text{exp} \rangle \text{do} \langle \text{stm\_train} \rangle \# \text{end} \\
\text{input} &::= \langle \text{id} \rangle \\
\text{output} &::= \langle \text{exp} \rangle \\
\end{align*}
\]

abstract syntax

\[
\begin{align*}
\text{stm} &\rightarrow \text{STATEMENT} + \cdots; \\
\text{assign} &\rightarrow \text{id \;} \text{exp}; \\
\text{ifthen} &\rightarrow .\text{exp \;} \text{stm\_s}; \\
\text{ifthenelse} &\rightarrow \text{exp \;} \text{stm\_s \;} \text{stm\_s}; \\
\text{while} &\rightarrow \text{exp \;} \text{stm\_s}; \\
\text{input} &\rightarrow \text{id}; \\
\text{output} &\rightarrow \text{exp}; \\
\text{STATEMENT} &::= \text{assign \;} \text{ifthen \;} \text{ifthenelse \;} \text{while} \\
&\quad \text{input \;} \text{output}; \\
\end{align*}
\]

end chapter;

chapter 'Expressions'

rules

\[
\begin{align*}
\langle \text{exp} \rangle &::= \langle \text{factor} \rangle; \\
\langle \text{factor} \rangle &::= \langle \text{exp} \rangle \# \# \langle \text{factor} \rangle; \\
&\quad \text{plus}(\langle \text{exp} \rangle,\langle \text{factor} \rangle) \\
\langle \text{primary} \rangle &::= \langle \text{factor} \rangle \# \# \langle \text{primary} \rangle; \\
&\quad \text{times}(\langle \text{factor} \rangle,\langle \text{primary} \rangle) \\
\langle \text{id} \rangle &::= \langle \text{id} \rangle; \\
\langle \text{constant} \rangle &::= \langle \text{constant} \rangle; \\
\langle \text{exp} \rangle &::= \#(\langle \text{exp} \rangle \#); \\
\end{align*}
\]
(primary) ::= #((compare) #);
(compare)

(compare) ::= (exp) ≠ (exp);

\( equal((\text{exp}).1,(\text{exp}).2) \)

(compare) ::= (exp) ≠ # (exp);

\( different((\text{exp}),(\text{exp})) \)

abstract syntax

\[
\begin{align*}
\text{plus} & \rightarrow \text{EXP EXP}; \\
\text{times} & \rightarrow \text{EXP EXP}; \\
\text{equal} & \rightarrow \text{EXP EXP}; \\
\text{different} & \rightarrow \text{EXP EXP}; \\
\text{EXP} & ::= \text{plus} \ \text{times} \ \text{ID} \ \text{equal} \ \text{different} \\
& \quad \text{CONSTANT AUX};
\end{align*}
\]

end chapter;

chapter 'Constants_and_Identifiers'

rules

\[
\begin{align*}
\langle \text{id} \rangle & ::= \%\text{ID}; \\
\text{id-atom}(\%\text{ID}) & \\
\langle \text{constant} \rangle & ::= \langle \text{bool} \rangle; \\
\langle \text{bool} \rangle & \\
\langle \text{constant} \rangle & ::= \langle \text{num} \rangle; \\
\langle \text{num} \rangle & \\
\langle \text{bool} \rangle & ::= \text{true}; \\
\text{boolean-atom}(\text{true}) & \\
\langle \text{bool} \rangle & ::= \text{false}; \\
\text{boolean-atom}(\text{false}) & \\
\langle \text{num} \rangle & ::= \%\text{NUMBER}; \\
\text{number-atom}(\%\text{NUMBER}) & \\
\end{align*}
\]

abstract syntax

\[
\begin{align*}
\text{boolean} & \rightarrow ; \\
\text{number} & \rightarrow \text{implemented as INTEGER}; \\
\text{id} & \rightarrow \text{implemented as IDENTIFIER}; \\
\text{CONSTANT} & ::= \text{number boolean}; \\
\text{id} & ::= \text{id};
\end{align*}
\]

end chapter;

chapter 'Comments'

abstract syntax

\[
\begin{align*}
\text{comment} & \rightarrow ; \\
\text{comment}\_s & \rightarrow \text{COMMENT}\_s
\end{align*}
\]
COMMENT ::= comment;
end chapter;

chapter 'Entry Points in Analyzer'

rules

(axiom) ::= ![PROGRAM](program);
(program)

(axiom) ::= ![PROGRAM](metavar);
(metavar)

(axiom) ::= ![DECLS](dcl_train);
(dcl_train)

(axiom) ::= ![DECLS](metavar);
(metavar)

(axiom) ::= ![DECLARATION](declaration);
(declaration)

(axiom) ::= ![DECLARATION](metavar);
(metavar)

(axiom) ::= ![MODE](mode);
(mode)

(axiom) ::= ![MODE](metavar);
(metavar)

(axiom) ::= ![IDLST](idlist);
(idlist)

(axiom) ::= ![IDLST](metavar);
(metavar)

(axiom) ::= ![STMS](stm_train);
(stm_train)

(axiom) ::= ![STMS](metavar);
(metavar)

(axiom) ::= ![STATEMENT](statement);
(statement)

(axiom) ::= ![STATEMENT](metavar);
(metavar)

(axiom) ::= ![EXP](exp);
(exp)

(axiom) ::= ![EXP](metavar);
(metavar)

(axiom) ::= ![ID](id);
(id)

(axiom) ::= ![ID](metavar);
(metavar)

(meta-atom) ::= %META

(meta-atom) ::= %META
abstract syntax

\[ meta \] implemented as IDENTIFIER;

end chapter;

chapter 'Extra Operators'

abstract syntax

\[ deref \] \[ \rightarrow \] EXP;

\[ and \] \[ \rightarrow \] EXP EXP;

\[ or \] \[ \rightarrow \] EXP EXP;

\[ AUX \] \[ ::= \] \( deref \) \( and \) \( or \);

end chapter;

end definition

Appendix 2. The program Metal.Metal

definition of METAL is

rules

\[ (language\_def) \] \[ ::= \] \( (language) \);

\[ (language) \] \[ ::= \] \# definition \# of \( (ucid) \) \# is \( (zone\_s) \)
\# end \# definition \( (terminal\_l\_op) \);

\[ language((ucid),(zone\_s),(terminal\_l\_op)) \]

\[ (zone\_s) \] \[ ::= \] \( (zone\_s)(zone) \);

\[ zone\_s\_post((zone\_s),(zone)) \]

\[ (zone\_s) \] \[ ::= \] \( (zone) \);

\[ zone\_s\_list(((zone))) \]

\[ (zone) \] \[ ::= \] \( (chapt) \);

\[ (chapt) \] \[ ::= \] \( (trans\_gen) \);

\[ (trans\_gen) \] \[ ::= \] \( (zone) \);

\[ (zone) \] \[ ::= \] \( (abs\_syn) \);

\[ (abs\_syn) \] \[ ::= \] \# chapter\( (chapt\_name)(zone\_s) \) \# end
\# chapter \# ;

\[ chapt((chapt\_name),(zone\_s)) \]

abstract syntax

\[ language \] \[ \rightarrow \] UCID ZONE\_S TERMINAL\_S\_OP;

\[ zone\_s \] \[ \rightarrow \] ZONE \( + \cdots \);

\[ chapt \] \[ \rightarrow \] CHAPT\_NAME ZONE\_S;

\[ LANGUAGE \] \[ ::= \] \( language \);

\[ ZONE \] \[ ::= \] \( chapt\ rule\_s\ abs\_syn \);
ZONE_S ::= zone_s,
CHAPT_NAME ::=ucid string;

chapter TRANS_GEN
rules
⟨trans_gen⟩ ::= #rules⟨rule_s⟩;
⟨rule_s⟩ ::= ⟨rule_s⟩⟨rule⟩;
rule_s-post(⟨rule_s⟩,⟨rule⟩)
⟨rule_s⟩ ::= ⟨rule⟩;
rule_s-list(⟨⟨rule⟩⟩)
⟨rule⟩ ::= ⟨syntactic_rule⟩ #; ⟨semantic_descr⟩;
rule(⟨syntactic_rule⟩,⟨semantic_descr⟩)

abstract syntax
rule → PRODUCTION SEMANTIC_DESCR;
rule_s → RULE + ···;
RULE ::= rule;

chapter SYNTACTIC_RULE
rules
⟨syntactic_rule⟩ ::= ⟨non_terminal⟩ # ::= ⟨elem_s_op⟩;
production(⟨non_terminal⟩,⟨elem_s_op⟩)
⟨elem_s_op⟩ ::= ;
elem_s-list(⟨⟩)
⟨elem_s_op⟩ ::= ⟨elem_s⟩;
⟨elem_s⟩ ::= ⟨elem⟩;
elem_s-list(⟨⟨elem⟩⟩)
⟨elem_s⟩ ::= ⟨elem_s⟩⟨elem⟩;
elem_s-post(⟨elem_s⟩,⟨elem⟩)
⟨elem⟩ ::= ⟨non_terminal⟩;
⟨non_terminal⟩
⟨elem⟩ ::= ⟨terminal⟩;
⟨terminal⟩
⟨elem⟩ ::= ⟨generic⟩;
⟨generic⟩
abstract syntax
production → NON_TERMINAL ELEM_S;
elem_s → ELEM* ···;
PRODUCTION ::= production;
ELEM_S ::= elem_s;
ELEM ::= terminal non_terminal genericatom;
end chapter;
chapter GENERATION DESCRIPTION
rules
\(<\text{semantic descr}\> :: = \langle\text{letop}\rangle;
\langle\text{letop}\>
\(<\text{semantic descr}\> :: = \langle\text{caseop}\rangle;
\langle\text{caseop}\>
\(<\text{semantic descr}\> :: = \langle\text{factor}\rangle;
\langle\text{factor}\>
\text{abstract syntax}
\text{SEMANTIC_DESCR ::= named_listop named_atomop terminal}
\text{non_terminal node letop caseop}
\text{dot_non_terminal;}

chapter FACTOR
rules
\(<\text{factor}\> :: = \langle\text{simple_factor}\rangle;
\langle\text{simple_factor}\>
\(<\text{factor}\> :: = \langle\text{factor2}\rangle;
\langle\text{factor2}\>
\(<\text{factor}\> :: = \langle\text{factor3}\rangle;
\langle\text{factor3}\>
\(<\text{simple_factor}\> :: = \langle\text{metavar}\rangle;
\langle\text{metavar}\>
\(<\text{simple_factor}\> :: = \langle\text{non_terminal}\rangle;
\langle\text{non_terminal}\>
\(<\text{simple_factor}\> :: = \langle\text{non_terminal}\> \# \langle\text{intconst}\rangle;
\langle\text{non_terminal}\>
\text{dot_non_terminal}((\text{non_terminal}),\langle\text{intconst}\rangle))
\(<\text{factor3}\> :: = \langle\text{named_listop}\rangle;
\langle\text{named_listop}\>
\(<\text{factor2}\> :: = \langle\text{named_atomop}\rangle;
\langle\text{named_atomop}\>
\(<\text{factor2}\> :: = \langle\text{node}\rangle;
\langle\text{node}\>
\(<\text{factor4}\> :: = \langle\text{simple_factor}\rangle;
\langle\text{simple_factor}\>
\(<\text{factor4}\> :: = \langle\text{factor3}\rangle;
\langle\text{factor3}\>
\(<\text{factor_s}\> :: = 
\text{factor_s-list}(( ))
\(<\text{factor_s}\> :: = \langle\text{factor}\rangle;
\text{factor_s-list}(((\text{factor}))
\(<\text{factor_s}\> :: = \langle\text{factor_s}\> \# , \langle\text{factor}\rangle;
\text{factor_s-post}((\text{factor_s}),\langle\text{factor}\rangle)
Metal: A formalism to specify formalisms

abstract syntax

\[ \text{FACTOR ::= named_listop named_atomop} \]
\[ \text{dot_non_terminal ucid non-terminal node;} \]
\[ \text{factor_s \rightarrow FACTOR*;} \]
\[ \text{dot_non_terminal \rightarrow NON_TERMINAL INT_CONST;} \]
\[ \text{INT_CONST ::= int_const;} \]

end chapter;

chapter LISTOP_ATOMOP

rules

\[ \langle \text{node} \rangle ::= \langle \text{lcid} \rangle; \]
\[ \text{node}(\langle \text{lcid},\text{factor_s-list}(\ )) \rangle; \]
\[ \langle \text{node} \rangle ::= \langle \text{lcid} \rangle \# (\langle \text{factor_s} \rangle \#); \]
\[ \text{node}(\langle \text{lcid},\langle \text{factor_s} \rangle \rangle) \]
\[ \langle \text{named_atomop} \rangle ::= \langle \text{lcid} \rangle \# -\langle \text{atomop} \rangle; \]
\[ \text{named_atomop}(\langle \text{lcid},\langle \text{atomop} \rangle \rangle) \]
\[ \langle \text{atomop} \rangle ::= \# \text{atom} \# (\langle \text{atomarg} \rangle \#); \]
\[ \text{atomop}(\langle \text{atomarg} \rangle) \]
\[ \langle \text{atomarg} \rangle ::= (\langle \text{generic} \rangle); \]
\[ \langle \text{generic} \rangle \]
\[ \langle \text{atomarg} \rangle ::= (\langle \text{string} \rangle); \]
\[ \langle \text{string} \rangle \]
\[ \langle \text{atomarg} \rangle ::= (\langle \text{metavar} \rangle); \]
\[ \langle \text{metavar} \rangle \]
\[ \langle \text{named_listop} \rangle ::= \langle \text{lcid} \rangle \# -\langle \text{listop} \rangle; \]
\[ \text{named_listop}(\langle \text{lcid},\langle \text{listop} \rangle \rangle) \]
\[ \langle \text{listop} \rangle ::= \# \text{list} \# (\langle \text{simple_factor} \rangle \#); \]
\[ \text{listop}(\langle \text{simple_factor} \rangle) \]
\[ \langle \text{listop} \rangle ::= \# \text{list} \# (\# (\langle \text{factor_s} \rangle \# \#); \]
\[ \text{listop}(\langle \text{factor_s} \rangle) \]
\[ \langle \text{listop} \rangle ::= \# \text{list} \# (\# (\langle \text{factor_s} \rangle \# \#); \]
\[ \text{postop}(\langle \text{factor4},\langle \text{factor} \rangle \rangle) \]
\[ \langle \text{listop} \rangle ::= \# \text{post} \# (\langle \text{factor4} \rangle \#,\langle \text{factor} \rangle \#); \]
\[ \text{preop}(\langle \text{factor},\langle \text{factor4} \rangle \rangle) \]

abstract syntax

\[ \text{node} \rightarrow \text{LCID FACTOR_S;} \]
\[ \text{named_listop} \rightarrow \text{LCID LST_OPS;} \]
\[ \text{named_atomop} \rightarrow \text{LCID ATOMOP;} \]
\[ \text{atomop} \rightarrow \text{ATOMARG;} \]
\[ \text{listop} \rightarrow \text{SF_S;} \]
\[ \text{postop} \rightarrow \text{POSTFREE_ARG FACTOR;} \]
\[ \text{preop} \rightarrow \text{FACTOR POSTPRE_ARG;} \]
LST_OPS ::= postop preop listop;
SIMPLE_FACTOR ::= acid non_terminal dot_non_terminal;
SF_S ::= factor_s SIMPLE_FACTOR;
POSTPRE_ARG ::= SIMPLE_FACTOR LST_OPS;
ATOMARG ::= acid genericatom string;
FACTOR_S ::= factor_s;
ATOMOP ::= atomop;
end chapter;

chapter MATCHING_OPS
rules

⟨letop⟩ ::= # let ⟨factor⟩ ≠ ⟨simple_factor⟩
# in ⟨semantic_descr⟩;
letop (bind((factor),(simple_factor)),(semantic_descr))
⟨caseop⟩ ::= # case ⟨simple_factor⟩
(alternative_s) ≠ end ≠ case;
caseop ((simple_factor),(alternative_s))
⟨alternative_s⟩ ::= ⟨alternative⟩;
alternative_s-list(((alternative)))
⟨alternative_s⟩ ::= ⟨alternative_s⟩⟨alternative⟩;
alternative_s-post((alternative_s),(alternative))
⟨alternative⟩ ::= # when ⟨choice_s⟩ ≠ ⟩
⟨semantic_descr⟩;
alternative ((choice_s),(semantic_descr))
⟨choice_s⟩ ::= ⟨choice⟩;
choice_s-list(((choice)))
⟨choice_s⟩ ::= ⟨choice_s⟩|⟨choice⟩;
choice_s-post((choice_s),(choice))
⟨choice⟩ ::= ⟨factor⟩;
⟨factor⟩
⟨choice⟩ ::= # others;
othersop ( )

abstract syntax
letop → BIND SEMANTIC_DESCR;
bind → FACTOR SIMPLE_FACTOR;
caseop → SIMPLE_FACTOR ALT_S;
alternative_s → ALTERNATIVE + ···;
alternative → CHOICE_S SEMANTIC_DESCR;
choice_s → CHOICE + ···;

CHOICE ::= othersop FACTOR;
BIND ::= bind;
ALT_S ::= alternative_s no _tree;
Meral: A formalism IO specify formalisms

ALTERNATIVE ::= alternative;
CHOICE :: = choice _s;
end chapter;
end chapter;
end chapter;

chapter ABS_SYN_DESCRIPTION
rules
  \( \langle \text{abs\_syn} \rangle ::= \# \text{abstract} \# \text{syntax} \langle \text{operators\_sorts\_s} \rangle; \)
  \( \langle \text{operators\_sorts\_s} \rangle ::= \langle \text{operators\_sorts\_s} \rangle \langle \text{operators\_sorts} \rangle; \)
  \( \text{abs\_syn\_post}((\langle \text{operators\_sorts\_s} \rangle, \langle \text{operators\_sorts} \rangle)) \)
  \( \langle \text{operators\_sorts\_s} \rangle ::= \langle \text{operators\_sorts} \rangle; \)
  \( \text{ops}\_\text{syn\_list}((\langle \text{operators\_sorts} \rangle)) \)
  \( \langle \text{operators\_sorts} \rangle ::= \langle \text{operators} \rangle; \)
  \( \langle \text{operators} \rangle ::= \langle \text{operators} \rangle; \)
  \( \langle \text{operators\_sorts} \rangle ::= \langle \text{sorts} \rangle; \)
  \( \langle \text{operators} \rangle ::= \langle \text{lcid} \rangle \# \rightarrow \langle \text{sons\_description} \rangle \# ; \)
  \( \text{operator}((\langle \text{lcid} \rangle, \langle \text{sons\_description} \rangle)) \)
  \( \langle \text{sons\_description} \rangle ::= \langle \text{ucid\_s\_op} \rangle; \)
  \( \langle \text{ucid\_s\_op} \rangle \)
  \( \langle \text{sons\_description} \rangle ::= \langle \text{arbitrary} \rangle; \)
  \( \langle \text{arbitrary} \rangle ::= \langle \text{ucid} \rangle \# * \# \cdots ; \)
  \( \text{star\_arbitrary}((\langle \text{ucid} \rangle)) \)
  \( \langle \text{non\_empty\_arbitrary} \rangle ::= \langle \text{ucid} \rangle \# + \# \cdots ; \)
  \( \text{plus\_arbitrary}((\langle \text{ucid} \rangle)) \)
  \( \langle \text{sorts} \rangle ::= \langle \text{ucid} \rangle \# ::= \langle \text{ucid\_s\_op} \rangle \# ; \)
  \( \text{sort}((\langle \text{ucid} \rangle, \langle \text{ucid\_s\_op} \rangle)) \)

abstract syntax
  \( \text{abs}\_\text{syn} \rightarrow \text{op\_sort} \# \cdots ; \)
  \( \text{operator} \rightarrow \text{lcid sons\_description} ; \)
  \( \text{sort} \rightarrow \text{ucid uclid\_s} ; \)
  \( \text{star\_arbitrary} \rightarrow \text{ucid} ; \)
  \( \text{plus\_arbitrary} \rightarrow \text{ucid} ; \)
  \( \text{at\_impl} \rightarrow \text{ucid} ; \)
  \( \text{op\_sort} ::= \text{operator sort} ; \)
Sons_description ::= ucid_s star_arbitrary plus_arbitrary at_impl;
end chapter;

chapter 'TERMINAL OBJECTS and lists of TERMINAL OBJECTS'

rules

(terminal) ::= %TERMINAL;

terminal-atom (%TERMINAL)

(ucid) ::= %UCID;

ucid-atom (%UCID)

(lcid) ::= %LCID;

lcid-atom (%LCID)

(ucldcid) ::= (lcid);

(lcid)

(ucldcid) ::= (ucid);

(ucid)

(generic) ::= %GENERIC;

genericatom-atom (%GENERIC)

(intconst) ::= %INTCONST;

int_const-atom (%INTCONST)

(string) ::= %STRING;

string-atom (%STRING)

(terminal) ::= (lcid);

let lcid-atom(MV) = (lcid) in terminal-atom(MV)

(terminal) ::= (ucid);

let ucid-atom(MV) = (ucid) in terminal-atom(MV)

(non_terminal) ::= #⟨lcid⟩ #;

let lcid-atom(MV) = (lcid) in non_terminal-atom(MV)

(non_terminal) ::= #⟨ucid⟩ #;

let ucid-atom(MV) = (ucid) in non_terminal-atom(MV)

(meetavar)

(ucid)

(chapt_name) ::= (ucid);

(ucid)

(chapt_name) ::= (string);

(string)

chapter LISTES

rules

(ucid_s_op) ::= ;

ucid_s-list( )

(ucid_s_op) ::= (ucid_s);

(ucid_s)
Metal: A formalism to specify formalisms

\[ \text{ucid}_s \] :: = (ucid);

\[ \text{ucid}_s\text{-list}(((ucid))) \]

\[ \text{ucid}_s \] :: = (ucid_s)(ucid);

\[ \text{ucid}_s\text{-post}((ucid_s),(ucid)) \]

\[ \text{ucid}_s\text{\_op} \] :: = ;

\[ \text{ucid}_s\text{-list}(( )) \]

\[ \text{ucid}_s\text{\_op} \] :: = (ucid_s);

\[ \text{ucid}_s \]

\[ \text{ucid}_s \] :: = (ucid);

\[ \text{ucid}_s\text{-list}(((ucid))) \]

\[ \text{ucid}_s \] :: = (ucid_s)(ucid);

\[ \text{ucid}_s\text{-post}((ucid_s),(ucid)) \]

end chapter;

abstract syntax

\[ \text{lcid} \] → ;

\[ \text{ucid} \] → ;

\[ \text{ucldid} \] → ;

\[ \text{lcid}_s \] → LCID * * ;

\[ \text{ucid}_s \] → UCID * * ;

\[ \text{ucldid}_s \] → UCLCID * * ;

\[ \text{non\_terminal} \] → ;

\[ \text{terminal} \] → ;

\[ \text{othersop} \] → implemented as SINGLETON;

\[ \text{int\_const} \] → ;

\[ \text{genericatom} \] → ;

\[ \text{string} \] → implemented as STRING;

\[ \text{LCID} \] :: = lcid;

\[ \text{UCID} \] :: = uclid;

\[ \text{UCLCID} \] :: = lcid uclid;

\[ \text{UCLCID\_S} \] :: = uclcid_s no\_tree ;

\[ \text{NON\_TERMINAL} \] :: = non\_terminal ;

end chapter;

chapter ENTRY\_POINTS

rules

\[ \langle \text{language\_def} \rangle \] :: = \# [\text{LANGUAGE}](language);

\[ \langle \text{language} \rangle \]

\[ \langle \text{language\_def} \rangle \] :: = \# [\text{LCID}](lcid);

\[ \langle \text{lcid} \rangle \]

\[ \langle \text{language\_def} \rangle \] :: = \# [\text{UCID}](ucid);

\[ \langle \text{ucid} \rangle \]
\[
\langle \text{language\_def} \rangle \quad ::= \quad \#]\text{uclcid}(\text{uclcid});
\]

\[
\langle \text{uclcid} \rangle
\]

\[
\langle \text{language\_def} \rangle \quad ::= \quad \#]\text{uclcid\_s}(\text{uclcid\_s});
\]

\[
\langle \text{language\_def} \rangle \quad ::= \quad \#]\text{chapt\_name}(\text{string});
\]

\[
\langle \text{language\_def} \rangle
\]

\[
\langle \text{string} \rangle
\]

\[
\langle \text{language\_def} \rangle
\]

\[
\langle \text{ucid} \rangle
\]

\[
\langle \text{language\_def} \rangle \quad ::= \quad \#]\text{int\_const}(\text{intconst});
\]

\[
\langle \text{language\_def} \rangle
\]

\[
\langle \text{intconst} \rangle
\]

\[
\langle \text{language\_def} \rangle \quad ::= \quad \#]\text{rule}(\text{rule});
\]

\[
\langle \text{language\_def} \rangle
\]

\[
\langle \text{rule} \rangle
\]

\[
\langle \text{language\_def} \rangle \quad ::= \quad \#]\text{zone}(\text{zone});
\]

\[
\langle \text{language\_def} \rangle
\]

\[
\langle \text{zone} \rangle
\]

\[
\langle \text{language\_def} \rangle \quad ::= \quad \#]\text{zone\_s}(\text{zone\_s});
\]

\[
\langle \text{language\_def} \rangle
\]

\[
\langle \text{syntactic\_rule} \rangle
\]

\[
\langle \text{language\_def} \rangle \quad ::= \quad \#]\text{production}(\text{syntactic\_rule});
\]

\[
\langle \text{language\_def} \rangle
\]

\[
\langle \text{elem\_s} \rangle
\]

\[
\langle \text{language\_def} \rangle \quad ::= \quad \#]\text{elem\_s}(\text{elem\_s});
\]

\[
\langle \text{language\_def} \rangle
\]

\[
\langle \text{elem} \rangle
\]

\[
\langle \text{language\_def} \rangle \quad ::= \quad \#]\text{elem}(\text{elem});
\]

\[
\langle \text{language\_def} \rangle
\]

\[
\langle \text{factor} \rangle
\]

\[
\langle \text{language\_def} \rangle \quad ::= \quad \#]\text{factor}(\text{factor});
\]

\[
\langle \text{language\_def} \rangle
\]

\[
\langle \text{choice} \rangle
\]

\[
\langle \text{language\_def} \rangle \quad ::= \quad \#]\text{non\_terminal}(\text{non\_terminal});
\]

\[
\langle \text{language\_def} \rangle
\]

\[
\langle \text{listop} \rangle
\]

\[
\langle \text{language\_def} \rangle \quad ::= \quad \#]\text{choice}(\text{choice});
\]

\[
\langle \text{language\_def} \rangle
\]

\[
\langle \text{alternative} \rangle
\]

\[
\langle \text{language\_def} \rangle \quad ::= \quad \#]\text{semantic\_descr}(\text{semantic\_descr});
\]

\[
\langle \text{language\_def} \rangle
\]

\[
\langle \text{alternative\_s} \rangle
\]

\[
\langle \text{language\_def} \rangle \quad ::= \quad \#]\text{alternative}(\text{alternative});
\]

\[
\langle \text{language\_def} \rangle
\]

\[
\langle \text{alternative\_s} \rangle
\]

\[
\langle \text{language\_def} \rangle \quad ::= \quad \#]\text{alternative\_s}(\text{alternative\_s});
\]

\[
\langle \text{language\_def} \rangle
\]

\[
\langle \text{simple\_factor} \rangle
\]

\[
\langle \text{language\_def} \rangle \quad ::= \quad \#]\text{simple\_factor}(\text{simple\_factor});
\]

\[
\langle \text{language\_def} \rangle
\]

\[
\langle \text{factor\_s} \rangle
\]

\[
\langle \text{language\_def} \rangle \quad ::= \quad \#]\text{factor\_s}(\text{factor\_s});
\]

\[
\langle \text{language\_def} \rangle
\]

\[
\langle \text{factor\_s} \rangle
\]

\[
\langle \text{language\_def} \rangle \quad ::= \quad \#]\text{factor\_s}(\text{factor\_s});
\]

\[
\langle \text{language\_def} \rangle
\]

\[
\langle \text{atomop} \rangle
\]

\[
\langle \text{language\_def} \rangle \quad ::= \quad \#]\text{atomop}(\text{atomop});
\]
Metal: A formalism to specify formalisms

(language_def) ::= # [ATOMARG] (atomarg);
(atomarg)
(language_def) ::= # [CHOICE_S] (choice_s);
(choice_s)
(language_def) ::= # [BIND] (factor) # = (simple_factor);
   bind(factor, simple_factor)
(language_def) ::= # [OP_SORT] (operators_sorts);
(operators_sorts)
(language_def) ::= # [SONS_DESCRIPTION] (sons_description);
(sons_description)
(language_def) ::= # [TERMINAL] (terminal);
(terminal)
end chapter;

chapter ERROR_RECOVERY
rules
(terminal_l_op) ::= ;
   terminal_s_op-list(( ))
(terminal_l_op) ::= #$ (terminal_s) #$;
(terminal_s)
(terminal_s) ::= (terminal);
   terminal_s_op-list((terminal))
(terminal_s) ::= (terminal_s)(terminal);
   terminal_s_op-post((terminal_s), (terminal))
abstract syntax
terminal_s_op   → TERMINAL * · · · ;
TERMINAL_S_OP  ::= terminal_s_op;
TERMINAL       ::= terminal;
end chapter;

abstract syntax
meta            → ;
no_tree         → implemented as SINGLETON;
comment         → implemented as STRING;
comment_s       → COMMENT + · · · ;
COMMENT         ::= comment;
end definition

References


