**Centrum voor Wiskunde en Informatica**
Centre for Mathematics and Computer Science

M.H. Logger

# An integrated text and syntax-directed editor

Computer Science/Department of Software Technology Report CS-R8820 May

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

# An Integrated Text and Syntax-Directed Editor

Monique Logger
*B.S.O. / Automation Technology*
*P.O. Box 8112, 3503 RC Utrecht, The Netherlands*

Smooth integration of free text editing and pure structure editing is a major problem in current syntax-directed editing systems. This paper describes an experimental editor developed in the GIPE project which aims at achieving such an integration. An overview is given of existing syntax-directed editing systems as well as the main principles on which the new editor is based.

*Keywords and phrases:* syntax-directed editor, integration of text and structure editing, editing models, fragment parsing, programming environment, implementation techniques.

*1987 CR Categories:* D.2.2 **[Software Engineering]:** Tools and techniques - *User interfaces*; D.2.3 **[Software Engineering]:** Coding - *Program editors;* D.2.6 **[Software Engineering]:** Programming environments - *Pretty printers*; **[Programming Languages]:** Processors - *Parsing.*

*1980 Mathematics Subject Classification:* 68N20 **[Software]:***Compilers and generators.*

*Note:* Partial support received from the European Communities under ESPRIT project 348 (Generation of Interactive Programming Environments - GIPE)

## 1. INTRODUCTION

### 1.1. Why syntax-directed editing?

Conventional editors are text oriented, i.e., they look at text as a collection of characters, words and lines. However, many texts, such as computer programs, have more structure (block structure for example) than can be perceived by a text editor. This structure is expressed in the syntax of the programming language which the user has to know.

A syntax-directed editor, i.e., an editor that is capable of recognizing the required structure, facilitates creation and modification of texts. Combined with functions for error checking (by parsing and type-checking) and for automatic layout generation (by prettyprinting), a syntax-directed editor can assist in:

- preventing certain classes of errors;
- detecting (and eventually correcting) other errors in an early stage;
- presenting a uniform layout;
- manipulating text at a higher abstraction level;
- providing help facilities and suggestions for the novice user.

Both novice and experienced user alike may use a syntax-directed editor. Beginners, unfamiliar with the language syntax, are assisted with the syntactical constructs and may learn the syntax during editing. Experienced users profit by being able to manipulate text at a higher abstraction level.

In practice, however, there is a considerable distance between the expected benefits of syntax-directed editing and the actual benefits obtained from existing editing systems. This paper proposes several ideas for closing the gap between the concept of syntax-directed editing and its realization.

Report CS-R8820
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

## 1.2. What is syntax-directed editing?

Syntax-directed editing is based on editing the *constructs* of a language. The user interacts directly with language constructs and avoids remembering the details of the syntax. Edit commands manipulate the language constructs. One can distuinguish *navigation* and *modification* actions. Among navigation actions are *moving* between the constructs and *search* actions. Modification actions are the *insertion* and *deletion* of constructs. Language constructs can be inserted as *templates* in which the keywords of the construct are already given and the parts which have to be filled in are indicated. The main part of a program can be built using templates.

### 1.2.1. Language constructs

In a programming language most language constructs are defined in terms of others. For example, a program body may consist of a list of statements. Each statement of this list may consist of other constructs. So a hierarchical relation exists between all the constructs in a language. Most languages have a single *start symbol* in the grammar, from which all constructs can be derived.
Language constructs can be optional or obligatory. For example, in a Pascal program, a declaration part in a program is optional. The condition-part and the then-part of an if-statement are obligatory.
A grammar rule describes the required keywords and components of a construct. A construct is either

- a structure with a fixed number of components, for example, the while-statement has two components (namely the condition and the body);
- a structure with a variable number of components, called *list constructs*, for example, a program consists of a list of statements;
- a terminal symbol, which has to be filled in with text, for example, an identifier;
- an *optional* construct, this is a construct which is not obligatory to use; for example, in most programming languages the else-part of an if-statement is an optional construct.

Because of the hierarchical relation between constructs, a program is often represented by a *tree*. A *parser* is used to recognize the structure in a text and to construct the trees, conversely, a *prettyprinter* can be used to display the tree as formatted text. A prettyprinter knows the layout information of each construct and displays the program in a fixed style.

### 1.2.2. Templates and placeholders

For each construct in a language a *template* can be derived from the corresponding grammar rule and layout information. A template consists of the keywords and the components of the construct placed in a certain layout. The components are represented by *placeholders* (also called *meta variables* or *holes*). When a syntax-directed editor inserts a language construct, it uses the template of the construct.

Example: A template representing an if-then-else-statement may look like:

```
if <cond>
   then <stat>
   else <stat>
```

<cond> and <stat> are the placeholders.

A placeholder indicates the type of the component by which it can be replaced. Placeholders must be filled in later, either by inserting a template (expanding the placeholder with a template) or by typing text. For example, the placeholder <stat> may be replaced by any kind of statement, such as an assignment-statement, a while-statement, an if-statement, etc.

## 1.3. Integration of text and structure editing

In syntax-directed editors, both structure oriented editing and text editing are desirable. Structure oriented editing only, is too restrictive. Building a program by only inserting and expanding templates is quite frustating when inserting expressions, for example, since more editing commands are needed for the expansions than for typing the text directly. Next, tructure oriented editing only, limits the flexibility for changing the existing program structure. One way to increase the flexibility of syntax-directed editors is to provide free text editing.

Several types of syntax-directed editors have been implemented. They differ in the way text editing and structure editing are combined. Existing editor models are discussed in chapter 2. Few syntax-directed editors are being used on a large scale, however. As already indicated, there is a gap between the supposed benefits of syntax-directed editing and the real benefits obtained from existing editing systems. One reason may be the complexity of this kind of editor, since both text and structure oriented editing must be supported.

Here a syntax-directed editor, called EDML, will be discussed which provides good facilities both for structure and text editing. A program can be constructed by insertion and expansion of templates, but, when desired, parts of the program can also be typed in the conventional way. The language construct which is currently processed by structure oriented edit commands is indicated by a *focus*. The focus is used to select a construct, but also to delimit an area for text editing. In both cases the focus can be seen as a window on a structure within the program text. The text in the focus is marked by highlighting or shading. Within the focus area free text editing is possible, but the changes are not restricted to the text in the focus: the program structure within and surrounding the focus may change by editing the text in the focus. The syntax of the edited text in the focus will be automatically checked when the focus is moved by a structure oriented edit action. So alternating structure oriented edit actions and free text editing is possible.

Chapter 3 describes the environment and the aims of the editor EDML. In chapter 4 the functionality of EDML is illustrated by means of an example, and the details of various edit commands are described. Chapter 5 contains a overview of the implementation and in chapter 6 the conclusions are summarised.

## 2. EXISTING SYNTAX-DIRECTED EDITORS

### 2.1. Editor models

Syntax-directed editors differ in the way facilities of conventional text editors are incorporated. Some syntax-directed editors are purely syntax-directed and do not allow text editing, others delimit an area for text editing and others permit text editing in the whole text. The following subdivision can be made, ranging from pure structure oriented editors, with no free text edit facilities, to pure text editors [Kli87]:

- template model
- hybrid model
- token model
- character model

The models and their variants are described below. At the end of this chapter the properties of some existing syntax-directed editors are summarised using this subdivision.

### 2.2. Template model

The major concept of the template model ([Zel84], [FPS84], [TR81], [MMF81]) is a *template* containing placeholders. Writing a text, starts with the placeholder for the start symbol of the language. This placeholder can be expanded by means of templates; this is often supported by menus to choose a permitted construct. Text must be inserted if the placeholder can no longer be expanded: the only allowed construct is a terminal symbol.

The text remains syntactically correct, since the type of the permitted construct is determined by the type of the placeholder. A type-checker is often connected to perform additional checks.

Editors based on the template model are not so practical, when writing a program using the expansion technique requires more work than typing the text directly. Variants therefore exist where low-level constructs (e.g., expressions) can be built in text mode.

Template model editors are especially useful for writing a new program. With expansion of templates a new program can be constructed quickly and it is guaranteed to be syntactically correct. Changing an existing text is not as efficient since replacing one construct by another causes all information belonging to the old construct to be deleted. A possible solution is to add a special cut and paste window, for storing parts of text for later use [Zel84].

### 2.3. Hybrid model

Hybrid editors ([HM84], [BS85], [Rei85], [PR87]) provide manipulation of language constructs as in the template model as well as a certain degree of free text editing. A *focus* is used to delimit an area for text editing. The focus contains the text of the current language construct, which is also used for the structure oriented actions. Text editing is only possible within the focus area, where the text may be (temporarily) incorrect. Sometimes, a special window is opened which contains the text to be edited. The syntax of the text is checked after moving the focus or closing the edit window. All manipulations on language constructs take place on the focus as a whole.

Using the focus to delimit an area of text improves the possibilities for free text editing but its usefulness is reduced because most systems require that the old and the modified construct are of the same syntactic type, so text editing is limited to the focus barriers, the program *structure* can not be changed.

An alternative to the hybrid model is the PSG editor [BS85], which has different modes to provide both text and structure editing. In one mode (structure mode) the editor is a pure template editor, in the other mode (text mode) the editor acts as conventional text editor on the whole text. In structure mode no parsing is done and all actions are on the abstract syntax of the language. In text mode free text editing is possible. When leaving the text mode a parser is called.

### 2.4. Token model

The token model ([MW80]) does not delimit an area for text editing, but text editing is permitted everywhere in the text. In a token model editor the text is checked continuously during editing. The user is assisted with the syntax of the language in order to let him or her at each moment know whether the typed text is syntactically correct or not.

Because of the text oriented character of the token model, there are generally no possibilities for manipulation of language constructs. This model could, in principle, be extended with such facilities.

Text is perceived as a sequence of tokens in the token model, this means that each word in the text has a lexical type assigned to it.

After each modification in the text (by editing a token or giving some edit command) the text is parsed to check for errors. Because it takes too long to parse all the text an *incremental parser* is desired. An incremental parser minimises the amount of parsing by comparing the old and the modified text. To this end a lot of administration is required such as keeping all parse stacks with all tokens (see [GM80], [Kli87]).

### 2.5. Character model

Character model editors are the conventional text editors. They operate on objects consisting of characters, words and lines. They have no knowledge of the grammar of the used language and no syntax checking is incorporated.

A variant which provides a certain degree of syntax-directed editing is the LSE editor of Digital [Dec85] which uses placeholders, templates and menus. Because no parser is connected, the text may become (and remain) incorrect.

### 2.6. Comparison of different editors

Syntax-directed editors are concerned with structure oriented navigation, edit actions and template insertion. Often a focus is used to determine the current construct. The various editor models differ in the use of templates and in the way facilities for conventional text editing and structure oriented editing are integrated. Error handling varies from error detection to error correction. Sometimes it is permitted to correct an error at a later time. Generally a prettyprinter is used to display formatted text.

Templates can be inserted in two ways: menu oriented and text oriented. In the first case expansion of placeholders is supported by menus from which the allowed templates can be selected. In the second case the editor incorporates an incremental lexical scanner (on a character basis), which automatically expands typed text to templates of language constructs as soon as a part of the first keyword of a construct has been recognized. In most cases, placeholders are represented by the *type* of the placeholder, for example <stat>. The type is the name of a set of language constructs, such as "stat" which stands for statement types. Sometimes optional and list constructs are represented explicitly, for

example [<stat>] is an optional statement and {<stat>}* represents a statement-list. The advantage is that more information about the type of the placeholder is shown to the user. In [ST84] (not in table) placeholders are represented without any type names.

The table below summarizes characteristics of some existing editing systems.

| Name of editor | editor model | templates | errors | placeholder | pretty-printer |
|---|---|---|---|---|---|
| SUPPORT [Zel84] | P | M | C | T | yes |
| Gandalf [MMF81] | P | M | D | T | yes |
| Cornell Pr Syn [TR81] | P | M | D | TA | yes |
| POE [FPS84] | P | T | C | TA | yes |
| Syned [HM84] | H | M | D | T | yes |
| PECAN [Rei84] | H | M | D | T | yes |
| PSG [BS85] | H | M | D | T | yes |
| Centaur [PR87] | H | M | D | T | yes |
| PDE1L | T | - | D | T | yes |
| LSE [Dec85] | C | MT | - | TA | no |
| EDML | H | M | D | T | yes |

The following topics are tabulated:

- *editor models*
  This column contains the editing model to which an editor is closely related:
  P. template model
  H. hybrid model
  T. token model
  C. character oriented
- *templates*
  This column indicates the way the syntax-directed editor inserts templates.
  M. menu oriented
  T. text oriented
  MT. both menu and text oriented
- *error handling*
  Tabulated is the fact whether the editor is either
  D. error detecting or
  C. error correcting
- *representation of placeholders*
  The code in this column means:
  T. type information only
  TA. type information and additional information concerning lists and optional structures
- *prettyprinter*
  The last column indicates whether a prettyprinter is connected.

## 3. A NEW APPROACH TO SYNTAX-DIRECTED EDITING

### 3.1. The editor environment

The editor to be described is designed as part of the Gipe project. This project aims at developing a language independent programming environment [CHI87]. From a language definition, system components are generated which together form the basic elements of a programming environment. The Gipe system consists of both the generated programming environment and the environment generator itself. The elements of the generated programming environment are:

- a syntax-directed editor
- a parser
- a prettyprinter
- a type-checker
- a debugger
- an interpreter

A type-checker, debugger and interpreter are not yet connected to the editor described here.

The Gipe system uses a multi-windowing system. During one session several applications may run, each in its own window. These applications may influence one another: in one window a new language is defined and a programming environment for this language is generated from the language definition, while in another window the newly generated environment can be used. This requires incremental generation of all elements of the programming environment.

The internal representation of the system is based on *abstract syntax trees*. The syntax-directed editor builds and modifies abstract syntax trees by structure oriented edit actions. The prettyprinter displays the tree in formatted text form. After some text oriented edit actions the editor calls a parser to check the text and build trees from the text. The type-checker and the interpreter work on the tree representation of the program text to check the static semantics and to interpret the program.
Abstract syntax trees are implemented by means of the VTP (Virtual Tree Processor [Lan86]). The VTP provides the data structure and functions to store and manipulate abstract syntax trees.

### 3.2. The editor model

As indicated before, a major aim of this editor is to provide good facilities both for structure oriented and for text editing. With this goal in mind, the four models of chapter 2 (template, hybrid, token and character model) are now reconsidered.

A character based model is insufficient since it does not check the text for syntactic correctness. A template model provides good facilities for structure oriented editing but is too restrictive for text editing.

A token model provides text editing everywhere in the text and can be extended with structure oriented actions. Only a few token model editors have been implemented, probably because of the difficulty of implementing incremental parsing. When the user is editing text, an incremental parser checks continuously what is typed and how this fits into the tree representation of the rest of the program. This approach is interesting but due to the anticipated technical difficulties we will not consider it further here.

The remaining alternative is the hybrid model. This model provides structure oriented editing and an area for free text editing, but in existing systems text editing is restricted because the type of the construct in the focus cannot be changed. EDML is an implementation of the hybrid model where *no* restriction is imposed on text editing, and where the overhead for switching between text and structure edit actions is minimised.

### 3.3. An overview of EDML

EDML is a language independent syntax-directed editor which provides both structure and text oriented editing. The editor is parameterised with language dependencies, which are derived from the syntax of each desired language.

The following principles guided the design of EDML:

a. the editor can be used by both novice and experienced users;
b. using structure oriented actions, a program can be built and modified with minimal knowledge of the language involved;
c. combining structure oriented actions with text edit actions must be easy. The system should allow a smooth transition between text and structure oriented editing.
d. the syntax of the text must be checked incrementally. However, text can be temporarily syntactically incorrect.
e. no restriction is imposed on text editing: the program structure outside the focus may change during text editing inside the focus.

The editor gives support to the user by indicating which constructs are permitted when expanding placeholders and by allowing deletion only if the construct is not an obligatory part of the environment (a, b).

Structure oriented actions as well as text oriented actions are supported (b). All operations (on text and structure) are performed relative to a *focus*, which contains a (possibly incomplete or incorrect) language construct. General text editing is possible within the focus. Structure oriented editing is performed on the language construct in the focus.

The focus delimits an area for free text editing (d). Moving the focus causes the text in the focus to be parsed. If an error occurs, first the error must be corrected before the structure action can be performed. The used parsing technique is called *fragment* parsing. The parser determines the applicable non-terminal(s) of the parsed text. This is an extension of a parsing technique based on *multiple entry points* where the parser is called with an expected non-terminal as argument. The parsing overhead is thus limited by use of a focus and a fragment parser.

Structure and text edit actions are combined by using the focus both for the structure and for the text edit actions (c). During structure editing the focus determines the current language construct on which the structure actions are performed. During text editing the focus delimits an area where free text editing is possible. Because structure actions require a correct tree representation, the edited text in the focus wil be reparsed when a structure action is invoked. Between two structure oriented actions, the text may be syntactically incorrect which makes it possible to edit in a conventional way.

Through unrestricted text editing the program structure of the text both within and surrounding the focus may change (e). For example one statement may be replaced by a list of statements or two constructs may be combined to one construct. So it may be necessary to parse text outside the focus.

The parser returns all possible parse trees associated with the (ambiguous) text or an error in case no parse can be made. The editor in combination with the parser must decide (as far as possible) how to use the results of the parser. The following results are possible:

- an error has occured, the text cannot be parsed:
  - a syntax error has occured;
  - the text could be parsed correctly by enlarging the focus;
- the text is parsed correctly:
  - the text belongs to a construct which is not acceptable at the place of insertion;
  - due to priority conflicts the focus has to be enlarged to rearrange the subtrees;
  - only one alternative is acceptable, it is selected;
  - more alternatives are acceptable: the user must select one.

Example, using the programming language C:

*Before text editing:*

```
if (x>1) x=y;
if (x>2) x=z;
   else x=w;
```

*After:*

```
if (x>1) x=y;
   else x=z;
```

In the example two constructs, two statements in a statement-list, are combined to one construct. If the C-parser doesnot recognise the string "else x=z;" as a language construct an error is returned. Enlarging the focus allows the text to be parsed correctly. Otherwise, if the C-parser knows a construct "else-part", which is an optional construct of an if-statement, the string is recognised, so no error is returned. But the recognised construct doesnot fit on the place of a statement in a statement-list. Now the focus is enlarged, which allows the text to be parsed correctly.

## 4. FUNCTIONALITY OF THE EDITOR

### 4.1. Overview

The editor consists of functions for structure oriented editing, text editing and for file handling. Most structure oriented actions cause the focus to be moved and, if the text has been modified, cause the text in the focus to be

parsed. During text editing the cursor is always on the screen. File handling manages reading and writing (parts of) files. The last action can always be *undone*.

The implemented structure oriented actions are:
- *expand* parts of the current construct
- *navigate* through the constructs
- *insert* placeholders
- modify: *cut*, *copy*, *paste* and *replace* a construct
- *search* for a construct or for the next placeholder

Text editing is possible inside the focus and is done relative to a cursor position:
- *navigate* through the text
- *insert* characters and lines
- *delete* characters and lines
- *search* for text

First, an example of an edit session is given. Next the edit commands are described in more detail.

### 4.2. An edit session

In this section an example is given how the various commands of the editor can be used for writing programs.
Starting the editor, the first input must be the language name and the filename to be used. The editor reads the grammar and looks for the named file. The parser and scanner of this language are incrementally built during the edit session (see [HKR87], [HKR88]).

Consider, for instance, the program which consists of the keywords "program" and "end" and a placeholder representing a statement-list.

*Initial situation:*

```
program
    <stat-list>
end
```

The focus is represented by the shaded area.

With the *expand* command language constructs can be inserted.

*Expand <stat-list> with while-construct:*

```
program
    while <cond>
        do <stat>
end
```

During the *expand* command all constructs which are permitted at the position of a placeholder are shown on a menu on the screen. So the user sees the syntax of the language (learning effect) and only permitted constructs can be selected (preventing errors).
Inserting more statements can be done by the *insert* command.

*Insert after the focus:*

```
program
    while <cond>
        do <stat>;
    <stat>
end
```

Note that the focus has been moved to the newly created placeholder <stat>.

Within the focus free text editing is permitted.

*Text editing:*

```
program
     while <cond>
          do <stat>;
     a := 7;
     c := 3.5
end
```

Moving the focus will cause the text in the focus to be checked for syntax errors. If no errors occur the focus will move, else the errors must be corrected first. A structured *search* action thus first checks the text in the focus for syntactic correctness, and only then moves the focus.

*Search for construct while-statement:*

```
program
     while <cond>
          do <stat>;
     a := 7;
     c := 3.5
end
```

Navigation through the program is done by the commands *zoom out*, *zoom in*, *move to next* or *previous construct*.

*Move to next construct:*

```
program
     while <cond>
          do <stat>;
     a := 7;
     c := 3.5
end
```

The focus now contains one statement in a list of statements. When *deleting* the construct, the focus moves to the next element in the list.

*Delete:*

```
program
     while <cond>
          do <stat>;
     c := 3.5
end
```

### 4.2. The expand command

The editor looks automatically for possible expansions of the construct within the focus: such as placeholders, empty lists and optional constructs, and shows these possibilities to the user. The user selects one position in the construct and all alternative constructs which are permitted at that position are shown. The template of the construct chosen by the user is inserted. If there is only one alternative, this one is inserted directly.

Further expansion is not possible if the placeholder represents a lexical notion. For example, a placeholder representing an identifier can only be replaced by typing text.

If the file is empty, all constructs which are permitted to fill the start symbol of the grammar are shown.

### 4.3. Deleting the current language construct

When deleting the construct currently in the focus, the following possibilities exist. If the construct in the focus is an obligatory part of another construct, the text in the focus will be replaced by a placeholder indicating the position and type of the deleted construct. If the construct in the focus is an element of a list of constructs or an optional part, the text in the focus will be deleted and the focus moves to the next construct in the text (or if no next construct in the list exists, it moves to the previous construct in the program).

*Before the delete command:*

```
program
     while <cond>
          do <stat>;
      a:= b
end
```

*After:*

```
program
     a:=b
end
```

Two kinds of list-constructions can be distinguished: lists which must contain at least one element and lists which may be empty. In case the list may be empty and the deleted construct was the last construct of the list, the text will be replaced by the placeholder for the list structure. In case the list must contain at least one element the text will be replaced by a placeholder representing one element of the list.

For example, a program consists of a list of statements, but at least one statement must be present:

*Before the delete command:*

```
program
     while <cond>
          do <stat>;
      a:= b
end
```

*After:*

```
program
     <stat>
end
```

If the whole program is within the focus, it will be deleted without being replaced by a placeholder.

### 4.4. Cut, copy, paste and insert

The deleted construct is copied to a buffer. The *copy* command puts the current language construct in the buffer, without deleting it. The buffer contents can be pasted by the commands *paste-before*, *paste-after* or *replace*. The type of the construct in the buffer is checked for compatibility with the point of insertion in the program.

Adding new elements in lists is also done by *inserting* a placeholder before or after the focus. The type of the placeholder is determined by the the type of the list.

### 4.5. Search

The search function looks for the next occurence of a certain construct. The search is circular in the text. A special search, *search next placeholder*, moves the focus to the next placeholder.

### 4.6. Navigation functions

*Previous construct* moves the focus to the construct before the current one.
*Next construct* moves the focus to the construct after the current one.
*Zoom in* moves the focus to the first variable part of the current construct.
*Zoom out* moves the focus to the construct of which is the current one a variable part.

A mouse can also be used to point at a construct. The editor determines which construct is addressed by the mouse position. Example:

*Mouse selection:*

```
if  a>b
    then c
    else d;
while x<0 do
   a:=10
```

*The focus will be moved to:*

```
if a>b
    then c
    else d;
while x<0 do
   a:=10
```

### 4.8. File handling

The program being edited is saved in a file in two representations: as an abstract syntax tree and as text. The advantage of saving the abstract syntax tree is that reading from file is faster because the programs need not be parsed. The text file has the advantage that it is in a human readable form.
When desired, parts of the program in the editor can be saved in a file. Conversely, the contents of other files can be inserted in the program in the editor.

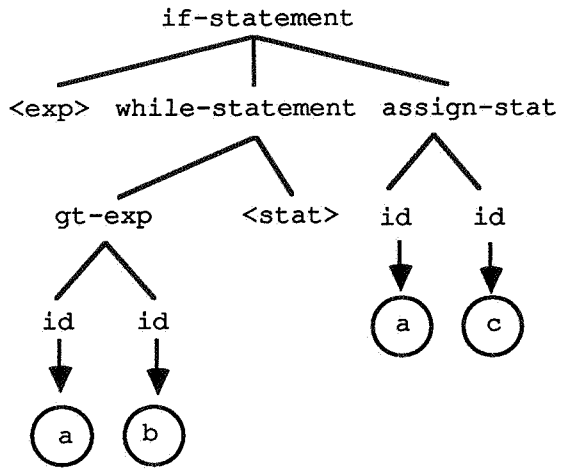
## 5. IMPLEMENTATION OF THE EDITOR

### 5.1. Datastructures

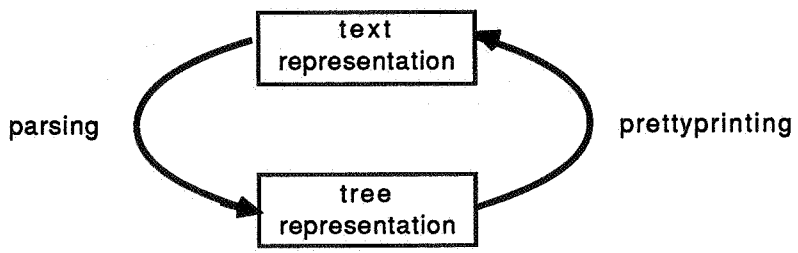
The editor manipulates programs as represented by their abstract syntax trees. For example, the if-statement

```
if <exp> then while a>b do <stat> else a:=c
```

has the following abstract syntax tree:



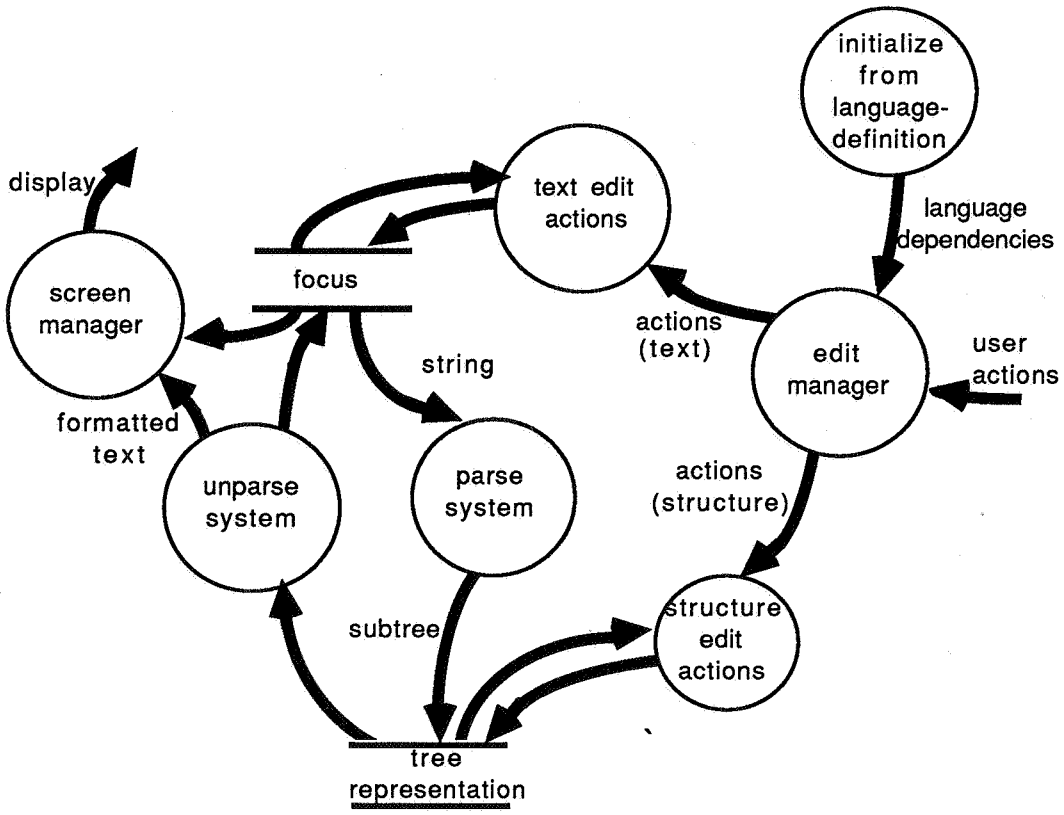

A tree representation is not the best representation for text editing. The area where text editing occurs, the focus, is therefore also stored as text. The text in the focus is converted to trees by a parser and placed in the overall tree representation by the editor. Conversely, the current subtree is converted to text in the focus by a prettyprinter.



In order to implement quick updates of the screen due to scrolling or due to selection of subtrees (with the mouse), information is needed that relates displayed text to tree representation. A third data structure is used to explicitly represent this relation.

### 5.2. Design of the editor

The design of EDML is shown in the figure below. It is represented by a dataflow diagram, where the bubbles represent functions, pairs of horizontal lines represent data-stores and the arrows represent dataflows.




The kernel of the editor is formed by the *Edit Manager*. The edit manager gets input from the user (by command, menu choice or mouse input) and determines the action or sequence of actions to be taken. The edit manager calls the appropriate subsystems, which are *text edit actions*, *structure edit actions*, *parse system*, *unparse system* and *screen manager* and passes data if necessary to those subsystems.

The main cycle of the *edit manager* is:

- get a command and select the appropriate function
- in case of a structure action, parse text in focus (when necessary)
- if additional information is needed, get input from user
- select structure or text edit function to perform the action on the tree or text representation
- redisplay screen.

The Gipe system is a language independent programming environment. So the editor consists of two parts: a language independent part describing the functionality of the editor which is fixed for all environments and a language dependent part which describes the language constructs (*initialise from language definition*). The language dependencies may change during an edit session since an programming environment for a language can be modified concurrently.

The *text edit actions* perform the edit actions (change cursor position, insert and delete of characters and search strings) on the text in the focus.

The *structure edit actions* perform the structure edit actions (expand, navigate, insert, cut, copy, paste and search) on

the tree representation.

The *parse subsystem* passes the text in the focus to the parser, which checks for errors and if no errors are found, converts the text to one or more trees. The parse subsystem decides how to handle the results depending on the environment of the parsed text.

The *unparse subsystem* is used in two ways: it calls the prettyprinter to unparse the current subtree to the text representation in the focus; it also calls the prettyprinter to unparse the tree to be displayed in combination with the text in the focus.

The *screen manager* displays the output via one of the graphical objects. The used graphical objects are:

- edit screen
- menubars and pull-down menus, popup menus
- input screen
- mouse selection
- message screen or display screen

### 5.3. Deriving information from a language definition

Within the Gipe project SDF (Syntax Definition Formalism) was designed to define the concrete and abstract syntax of a language.

When starting an environment for a language, the language definition in SDF is used to generate:

- a scanner for the language (see [HKR87]). A scanner generator constructs incrementally (during the edit session) the lexical scanner from the lexical syntax definition.
- a parser for the language (see [HKR88]). A parser generator constructs incrementally (during the edit session) the parser from the concrete syntax definition.
- an editor. The editor uses the abstract syntax definition and the layout information. Each construct has a name to communicate with the user.

### 5.4. Interface with the VTP

The VTP [Lan86] is the system component for manipulating abstract syntax trees. The VTP contains functions to:

- define an abstract syntax of a language
- navigate through trees
- build trees
- modify trees
- search for a subtree
- extract information about the constructs of the language
- extract information about the (sub)trees

Trees are built both by construct manipulation actions (insert and delete subtrees) and by replacing the current subtree (or even the environment) by the parsed text of the focus after text editing.

### 5.5. Interface with the scanner and parser

The parser-scanner combination ([HKR87], [HKR88]) is able to

- parse the text in the focus or in a file and return all possible trees and their syntactic types (one string may be of several syntactic types, for example, an assignment- statement is of type *statement* but may also be a *list of statements* which forms one *statement*)
- recognize a placeholder and handle it as a subtree
- detect errors and generate appropriate error messages

When a text file is read into the editor or when the focus is moved by a structure oriented edit action, the parser will be called. The scanner scans the text for tokens. The parser finds all possible parses of the text and returns the corresponding abstract syntax trees together with their syntactic types. If a parse error occurs the parser returns an error message.

The editor decides how to handle the results of the parser: the editor updates the tree with one of the returned subtrees, decides to enlarge the focus to reparse the text with more environment or displays an error message.

### 5.6. Interface with the prettyprinter

A prettyprinter is used to unparse the tree representation to formatted text. The text is displayed on the screen or used to fill the focus with the textual representation of a language construct when text editing starts.

### 5.7. Interface with the user

A user may
- give commands to edit text, edit constructs, handle file input and output
- type text
- use the mouse to move cursor and focus
- scroll through the text

Commands may be given by key commands or by menu selection, or via command files. Each command invokes the appropriate edit function. The function calls are independent of the chosen user interface.

## 6. CONCLUSIONS AND FUTURE DIRECTIONS OF THE PROTOTYPE IMPLEMENTATION

### 6.1. Current situation

A prototype of the editing system has been made. Implemented are:
- a rudimental text editor within the focus
- most structure actions
- smooth integration between text and structure editing

Not yet implemented are:
- the user interface: the editor has not yet been connected to the (multi-) windowing system, menus (pull-down and pop-up), scrollbar, and mouse handling facilities as currently under development in the Gipe project. Several issues will have to be solved first: a data structure has to be found which provides quick scrolling facilities combined with mouse selection.
- only a simple form of the expand command is implemented. The user interface has to be extended so that all places where expanding is possible are shown to the user.
- automatically enlarging the focus. If the program structure is changed by text editing insuch a way that it influences the program structure surrounding the focus, it may be necessary to reparse its environment (the text surrounding the focus) to construct a correct tree. The system has to prevent the environment from becoming too large. This is necessary, since in the case of a syntax error the whole tree would have to be reparsed to detect that no correct tree can be formed. Depending on the environment the system must decide whether the environment of the focus must be reparsed.

Current SDF language definitions do not yet incorporate prettyprint rules. In the future the prettyprint rules will be defined in the language definition, so the prettyprinter can be generated automatically, and built and modified incrementally.

### 6.2. Conclusions and future directions

The prototype shows that the main design goals as defined in chapter 3.3 can be achieved. The editor can be used both by novice and experienced users. Experienced users use the structure edit actions because of the higher abstraction level to manipulate the text. The text editing facilities improve the flexibility in modifying a program text. Novices are supported by use of templates and placeholders and continuous syntax checking. Structure and text edit actions are integrated by use of the focus and the fragment parsing technique. Between two structure actions the text may become temporarily syntactically incorrect.

Possible extensions to the prototype editor are:
- ellipsis (or holophrasing, folding): to show the structure in the program and to obtain a overview of the

program, details can be suppressed by printing only "..." (3 dots) instead of the program text. This introduces the problems how such abbreviated text can be reparsed.

- extend the user interface with a command language to build new commands from existing commands
- use of multiple text edit areas in one file
- one file may be edited in more than one window at the time
- storing comments in abstract syntax trees. In most programming languages comment is permitted between the constructs. The parser cannot determine to which construct a comment belongs. One solution is to provide a special user interface to assign comments to constructs, for example, see [ABL81]
- multiple views of the same program, for example, both a graphic representation and a text representation. Multiple representations can be generated from the same data structure.

## ACKNOWLEDGEMENTS

This paper has profited from the helpful comments of the members of the GIPE project. In particular I would like to thank Paul Klint and Jan Rekers for their discussions on the subject and their comments. Also I would like to thank Roelof Vuurboom for his constructive criticism of the paper.

**REFERENCES**


[ABL81] C.N. Alberga, A.L. Brown, G.B. Leeman Jr., M. Mikelsons & M.N. Wegman, "A Programming Development Tool", *Eight Annual ACM Symposium on Principles of Programming Languages*, January 1981, pp.92-104.

[BS85] R. Bahlke & G. Snelting, "The PSG-Programming System Generator", *ACM Sigplan Notices*, volume 20(7), July 1985, pp.28-33.

[CDF84] M. Chesi, E. Dameri, M.P. Franceschi, M.G. Gatti & C. Simonelli, "ISDE: An Interactive Software Development Environment", *ACM Sigplan Notices*, volume 19(5), May 1984, pp.81-88.

[Cha86] J. Chailloux, *Le_Lisp Version 15.2, Le Manuel de Référence*, INRIA, May 1986.

[CHI87] D. Clement, J. Heering, J. Incerpi, G. Kahn, P. Klint, B. Lang & V. Pascual, "Preliminary Design of an Environment Generator", GIPE, Esprit project 348, Second annual review report, January 1987.

[CK84] R.H. Campbell & P.A. Kirslis, "The SAGA Project: A System for Software Development", *ACM Sigplan Notices*, volume 19(5), May 1984, pp.73-80.

[Dec85] *VAX Language-Sensitive Editor, User's guide*, Digital equipment corporation. Maynard, Massachusetts, July 1985.

[DMS84] N.M. Delisle, D.E. Menicosy & M.D. Schwartz, "Viewing a Programming Environment as a Single Tool", *ACM Sigplan Notices*, volume 19(5), May 1984, pp.49-56.

[FPS84] C.N. Fischer, A. Pal, D.L. Stock, G.F. Johnson & J. Mauney, "The POE Language-Based Editor Project", *ACM Sigplan Notices*, volume 19(5), May 1984, pp.21-29.

[GM80] C. Ghezzi & D. Mandrioli, "Augmenting parsers to support incrementally", *Journal of the ACM*, Vol 27(93), July 1980, pp.564, 579

[GM84] D.B. Garlan & P.L. Miller, "GNOME: An Introductory Programming Environment Based on a Family of Structure Editors", *ACM Sigplan Notices*, volume 19(5), May 1984, pp.65-73.

[HKR87] J. Heering, P. Klint & J. Rekers, "Incremental generation of lexical scanners", Centre for Mathematics and Computer Science, Amsterdam, Report CS- R8761, 1987.

[HKR88] J. Heering, P. Klint & J. Rekers, "Incremental generation of parsers", Centre for Mathematics and Computer Science, Amsterdam, Report to appear in 1988.

[HM84] J.R. Horgan & D.J. Moore, "Techniques for Improving Language-Based Editors", *ACM Sigplan Notices*, volume 19(5), May 1984, pp.7-14.

[Kli87] P. Klint, "Programming Environments" (in Dutch), Course Notes, University of Amsterdam, 1987.

[Lan86] B. Lang, "The Virtual Tree Processor", GIPE, Esprit project 348, Third review report, September 1986.

[MMF81] R. Medina-Mora & P.H. Feiler, "An Incremental Programming Environment", *IEEE Transactions on software engineering*, vol. se-7, no. 5, September 1981, pp.472-481.

[MW80] M. Mikelsons & M. Wegman, *PDE1L: The PL1L Program Development Environment -- Principles of Operation*, IBM RC 8513, Yorktown Heights, New York, November 1980.

[PR87] V. Pascual & L. Rideau, "Current status of the Centaur interface", GIPE, Esprit project 348, Second annual review report, January 1987.

[Rei84] S.P. Reiss, "Graphical Program Development with PECAN Program Development Systems", *ACM*

*Sigplan Notices*, volume 19(5), May 1984, pp.30-41.

[ST84] T.A. Standish & R.N. Taylor, "Arcturus: A Prototype Advanced Ada Programming Environment", *ACM Sigplan Notices*, volume 19(5), May 1984, pp.57-64.

[TR81] T. Teitelbaum & T. Reos, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", *Communications of the ACM*, Vol 24 (9), September 1981, pp.563-573.

[Zel84] M.V. Zelkowitz, "A Small Contribution to Editing with a Syntax Directed Editor", *ACM Sigplan Notices*, volume 19(5), May 1984, pp.1-6.