CWI

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

M.H.H. van Dijk, J.W.C. Koorn

GSE, a generic syntax-directed editor

Computer Science/Department of Software Technology Report CS-R9045 September

GSE, a Generic Syntax-Directed Editor

M.H.H. van Dijk[1] & J.W.C. Koorn[2]
[1]*BSO Automation Technology bv,*
*P.O. Box 8052, 3503RH Utrecht**
[2]*Programming Research Group,*
University of Amsterdam,
Kruislaan 409, 1098SJ Amsterdam

GSE is a generic syntax-directed editor developed in the context of the GIPE project (Generation of Interactive Programming Environments). GSE is generic, i.e., it can be used for any language for which the syntax has been specified. Furthermore, GSE is syntax-directed, which means that it is aware of the syntax of the language in use and can thus assist the user both in building a syntactically correct program and in checking and correcting the syntax of existing (parts of) programs. Finally, free text editing and syntax-directed editing are integrated in a transparent way.

Key Words & Phrases: syntax-directed editing, generic syntax-directed editor, generation of programming environments.

1990 CR categories: I.7.1 **[Text processing]**: Text editing; D.2.6 **[Software engineering]**: Programming environments.

1985 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

An editor is a tool to produce documents at a computer terminal. Documents exist in a great many varieties, e.g., plain texts, computer programs, graphical objects, etc., and of course combinations of them. In many cases, the document to be produced must fulfill a number of requirements of a syntactic or semantic nature. For instance, computer languages require a document (in this case a computer program) to comply with very strict syntactic and semantic rules. Also, in the case of design languages like Jackson or Yourdon, which employ graphical notions, there are rules that limit the ways in which graphical primitives may be combined into one document. Finally, even plain texts may be subject to rules that define their layout or fix the partitioning into sections or chapters. All tools that enable the user to check whether the document he or she has created obeys certain rules, generally live a life separate from that of the editor. This state of affairs has two major drawbacks:

- The creation of a document requires an iteration between two activities: 1) creating and modifying the document by means of an editor, and 2) checking the correctness of the document by means of tools like parsers, compilers, format checkers etc.
- Too much work is done. If, for instance, a minor change is made to a computer program, essentially the whole program must be parsed again if we want to make sure that it is still syntactically correct.

GSE aims at removing these drawbacks as far as syntax checking tools are concerned. First, by integrating the edit and syntax checking operations into one tool, the syntax-directed editor, and,

* For the duration of his work on GSE, the first author was assigned to the Centre for Mathematics and Computer Science.

Report CS-R9045
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

secondly, by providing facilities for having only part of the document checked for syntactical correctness.

GSE is language independent. This means that any text can be edited using GSE, provided the syntax, with which the text must comply, has been specified. The syntax must be specified in SDF (Syntax Definition Formalism), that was developed within the GIPE project [HHKR89]. A parser is generated from the SDF definition by a parser generator [HKR89], GSE communicates with the parser to perform syntax checks on (part of) the (program) text and assists the user in building a syntactically correct (program) text. GSE is also language independent in the sense that the user interface is independent of the language used. That is, the commands of GSE and the screen layout are independent of the language.

Work on GSE started in the summer of 1987. Since that time the implementation has gone through various stages of sophistication. Experience with intermediate implementations has led to adjustments in our ideas on which GSE should be based and, vice versa, these new ideas led to improved implementations. A description of the basic principles on which GSE is based and a comparison with some other syntax-directed editors is given in [Log88]. Various aspects of the implementation of GSE are discussed in [DK89b], and the use of GSE is described in [DK89a]. GSE is implemented in a version of Lisp (see [LeLisp]) on a Sun3 or Sun4 workstation, a BULL DPX1000, and the IBM PC RT running Unix. Furthermore, the implementation of GSE assumes that X-windows and the graphical object system [CI88] are installed.

Not yet implemented are the structured search operation (Section 4.1), mouse pointing with strings (Section 6.2), a pretty printer for the expand command (Sections 4.2 and 7.1) and a file management system (the present implementation uses the Unix file system).

In Section 2 we discuss the environment in which GSE runs. After a short discussion of how GSE communicates with the graphical object system, the parser generator, and the parser we concentrate on GSE itself. Section 3 introduces the notion of focus. The focus is a concept that is indispensable for combining edit actions with syntax checking, but also for allowing the user to have parts of the (program) text parsed. In Section 4 we describe how syntax-directed edit actions can be based on the focus concept. Some remarks concerning the implementation of GSE are made in Section 5. A major concern in building a syntax-directed editor is the design of the user interface. Some insight in the basics underlying the user interface of GSE is provided in Section 6. Section 7 discusses some of the major problems that have been or still must be solved in GSE. Finally, in Section 8, we will consider some of the possible extensions of GSE like the integration of other tools such as type checkers, interpreters and pretty printers.

2. The environment of GSE

Figure 1 shows the environment in which GSE operates. The main purpose of this diagram is to show schematically the interaction of GSE with the tools that make up its environment. GSE is represented by the "bubble" labeled GSE. The externals of GSE are represented as rectangular boxes. The annotated arrows represent data flows.

The user has to provide a language definition which is used by the parser generator to generate a parser. Some user actions require GSE to parse a portion of text. The parser returns an abstract syntax tree that is further processed by GSE (see also sections 2.2, 3 and 4). The graphical object system is responsible for all displaying on the terminal screen. It keeps track of, for instance, the number of windows opened on some document, their location on the screen, their size and their contents. In the following two subsections we will discuss in what way the graphical object system and the parser generator and parser communicate with the editor.

2.1. The Graphical Object System

The graphical object system [CI87, CI88]. is used to define special objects (e.g., buttons, menus, dialog boxes, etc.) that make up the user interface.

Essentially, the specification process for graphical objects consists of three phases. The geometrical definition of the object, a specification of how it is displayed on the screen, and a definition of the functionality (behaviour) of the object.

The geometry of the graphical objects of the user interface is defined as a hierarchy or tree of objects. The geometric model is specified in the language Aida [Dev87]. Aida contains basic geometric forms referred to as atomic constructors (like string, icon, box, etc.) from which more complicated geometries may be built using assembly functions (like constructing a column or row) also provided by Aida. So, for instance, a menu bar might be built as a row of menu buttons, where every menu button is an overlay of a box and a string.

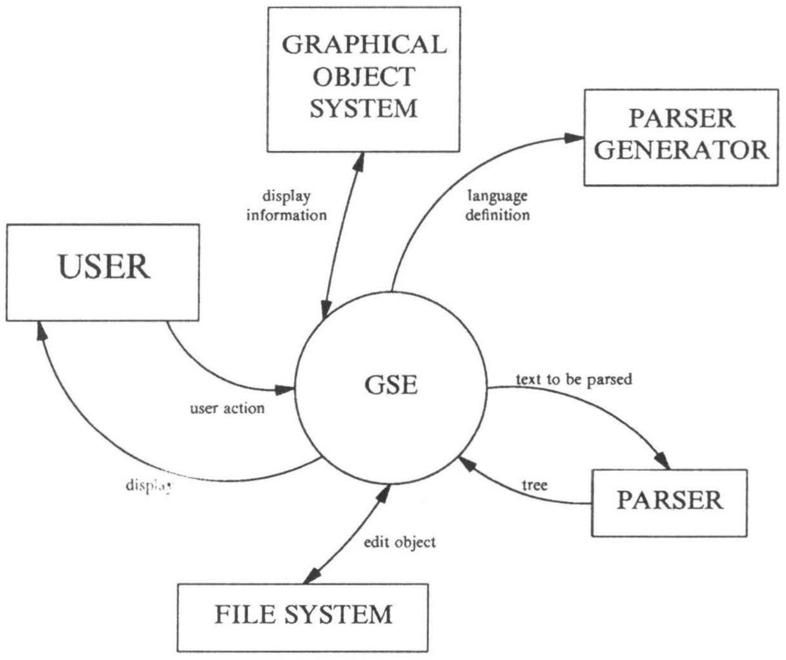


Figure 1

GSE in its environment. The externals forming the environment of the editor and the type of data exchanged between the editor and its environment are shown.

Having defined what some object should look like, it must be displayed on the screen. A basic display machine has been defined in the Lelisp virtual window system to be as independent as possible of the underlying window system. (The Lelisp window system has been implemented on top of a number of commercially available window systems, among which Sunviews and X-windows.) As new objects are defined the display machine must be extended with functions that know how to display the new object.

Finally, we require each object to know how to react to external stimuli like mouse events or keyboard actions. For instance, we want the menu buttons in the menu bar to display a pull down menu on a mouse down event in its associated box. The functionality of an object is defined in the language Esterel [Ber86].

The graphical objects are generic. The editor, GSE, can create instances of some graphical object by a suitable function call. For example, if an extra GSE window (see Figure 7) must be opened on the current program text a call that looks like "create-edit-window" with some parameters would be in order, or if the parser detects an error in some part of the progrm text, a call like "create-dialog-box" resulting in the display of a dialog box to show the error message of the parser, would be in order.

2.2. The parser generator and the parser

In this section, we briefly discuss the communication between the parser and the parser generator in this section so as to provide some insight into the principles underlying the syntax-directed part of GSE.

The parser is generated in a lazy manner. This means that only those parts of the parser are generated that are needed to parse some specific text that has been offered to the parser. Thus the parser is built stepwise during an edit session.

The parser generation process is also incremental. Since the GIPE system provides tools for language design, the syntax of the language that is used may be subject to change. The incrementality of the generation process aims at reusing as much as possible from the already generated part of the parser after a modification of the grammar. Notice that incrementality and laziness interact in a convenient way, since if, for instance, the modification of the grammar relates to a part of the parser that was not yet generated no work needs to be done.

Laziness and incrementality result in a situation where small parts of the parser are added or replaced at a time. This is, of course, a large advantage for interactive use as in an editor like GSE.

Finally, we note that the parsing algorithm which is used as a basis for the parsing, allows a parser to be generated for any context-free grammar. Consequently, one has great freedom in specifying the syntax of the language to be used. However, as we will see in section 7.3, an unexpected problem arises in retaining the property that not only programs, but also parts of programs can be parsed.

3. The focus

GSE, like most other-syntax directed editors, maintains a structured representation of the progrm text. In the case of GSE this structured representation is an abstract syntax tree. The abstract syntax tree is represented in the VTP (Virtual Tree Processor) formalism [Lan86].

Suppose, that we are editing a Pascal program. The program text might look like the text in Figure 2a. The corresponding abstract syntax tree is shown in Figure 2b. The little black triangle represents the cursor and marks the position where characters may be inserted in the text. In the following we will not show explicitly the construction of parse trees, and will also use the verb "parse" in the meaning of deriving the abstract syntax tree corresponding to some string.

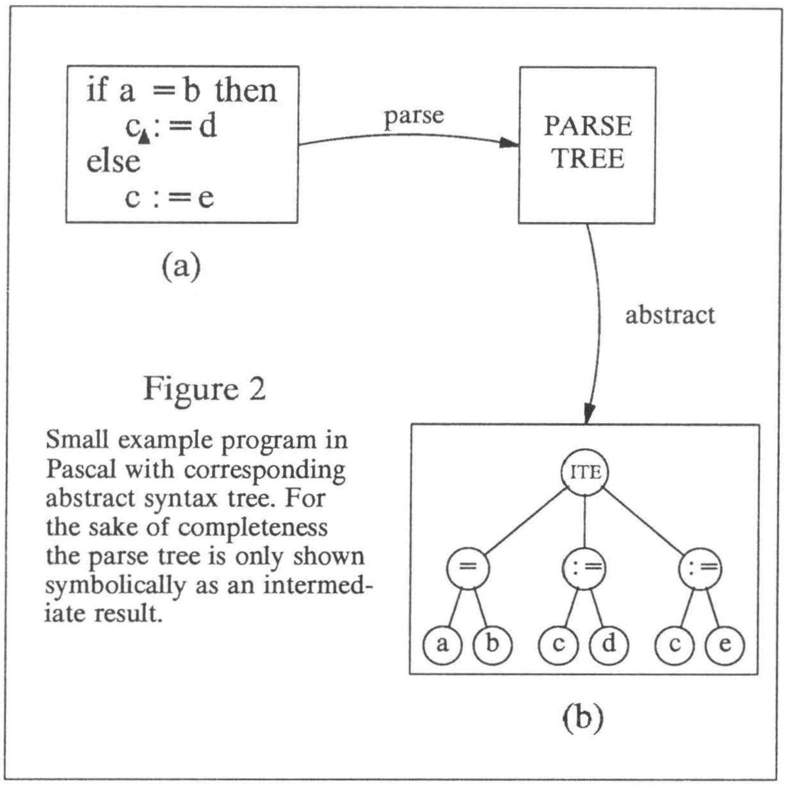


Figure 2

Small example program in Pascal with corresponding abstract syntax tree. For the sake of completeness the parse tree is only shown symbolically as an intermediate result.

Any textual modification of the program will result in the need to reparse the whole program if one wants to find out whether the program is still syntactically correct. It is, however, evident that most textual changes will affect only a small part of the program structure or abstract syntax tree. We try to minimize the amount of reparsing by limiting the region for free text editing. Suppose that in the example of Figure 2 we select the statement "c := d" as the region for free text editing and mark the corresponding subtree of the abstract syntax tree as the current node. This amounts to a situation that we are actually editing the statement "c := d" and after having finished modifying this statement we only need to reparse the modified string and replace the subtree corresponding to the current node by the abstract syntax tree that results from parsing the modified string. This sequence of actions has been shown in Figure 3. The region for free text editing is shown as a box surrounding the selected text. The textual modification in Figure 3 amounts to replacing the string "c := d" by the string "d := c - e". After the textual modification has been completed it suffices to parse the modified string and replace the subtree corresponding to the current node by the abstract syntax tree corresponding to the modified string.

By selecting some part of the program text and marking the corresponding subtree in the abstract syntax tree we have actually introduced the *focus*. For the focus to be effective we need the guarantee that text lying outside the focus is syntactically correct and that only the subtree corresponding to the current node will change as a result of a textual modification.

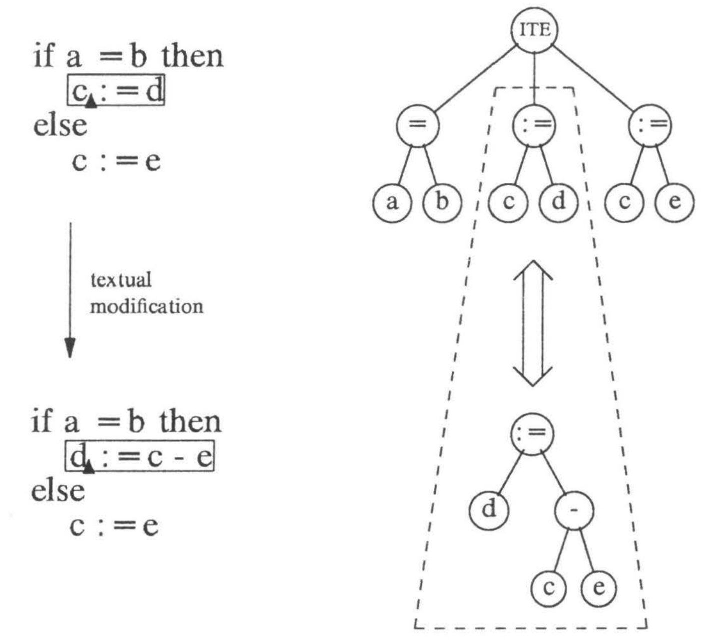


Figure 3

Example of the replacement of part of the abstract tree when the contents of the focus has changed as a result of free text edit actions.

It is easily seen that the introduction of the focus mixes parsing and editing. If after modifying the statement "c := d" of the example in Figure 3 we want to edit the statement "c := e", the focus must be moved to some part of the text that includes this statement (preferably but not necessarily the statement itself). However before the focus can be moved we must first parse the text in the focus. If we would just move the focus to "c := e" there would be no guarantee that the text outside the focus would still be syntactically correct. GSE takes care of this automatically. If a request to move the focus is issued by the user (see Section 4 on how this is done in GSE) the text in the old focus is parsed and the request is only granted if it is found to be syntactically correct.

4. Syntax-directed edit actions in GSE

In this section we will discuss the syntax-directed edit actions provided by GSE. Section 4.1 discusses the navigation actions to move the focus through a syntactically correct text as well as the syntax-directed search action (not yet implemented). Section 4.2 is concerned with those commands that can be used to build a progrm in a syntax-directed way.

4.1. Navigation actions

GSE knows four basic navigation actions to move the focus to other parts of the program. They are: *zoom-in to first child, zoom-out to parent, go to next child of same parent* and *go to previous child of same parent*. These actions are shown schematically in Figure 4.

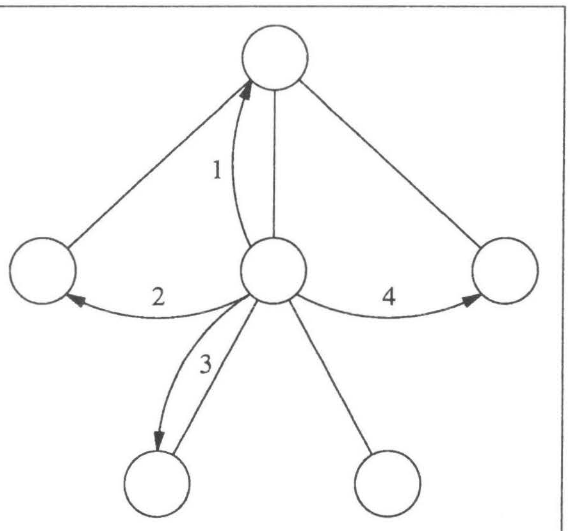


Figure 4
The node from which all arrows emanate is the current node. The arrows point to the nodes that will become current node as a result of one of the following basic navigation actions,
arrow 1 = zoom-out ; arrow 2 = go to previous child
arrow 3 = zoom-in ; arrow 4 = go to next child.

The four navigation actions enable the user to manoeuvre the focus to any node of the tree and its corresponding text part. Clearly, editing would be cumbersome if only these four navigation actions were available for moving the focus. Therefore, a more powerful navigation operation has been added to GSE that enables the user to point at a certain character in the text with the effect that GSE searches for the smallest language construct that still contains this character and moves the focus to that language construct.

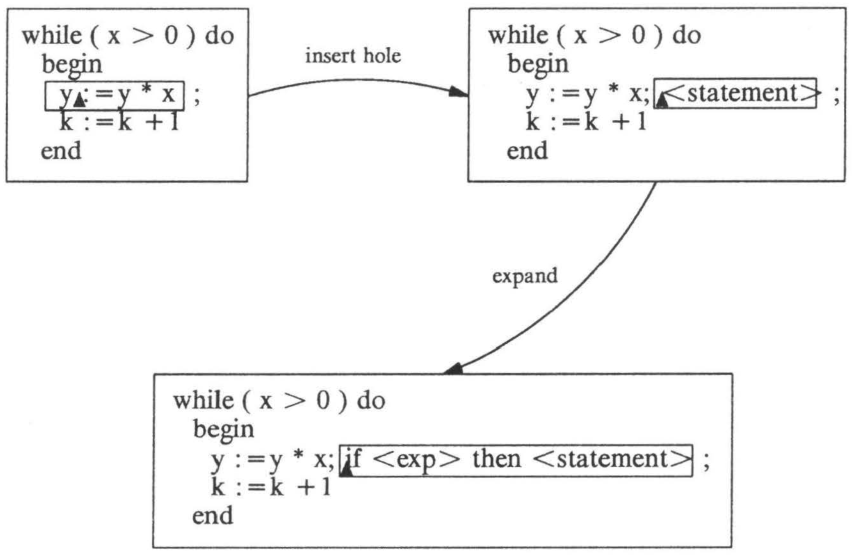
Finally, the syntax-directed editor provides a structured search operation that searches for the next occurrence of a certain language construct selected by the user. This operation has not yet been implemented.

4.2. Syntax-directed creation of programs

GSE provides facilities for actually building a progrm using syntax-directed creation operations. Consider the example shown in Figure 5. A small part of a Pascal program is shown with the focus positioned at the statement "y := y * x".

Suppose we want to insert a statement after the statement "y := y * x". We might do this by first zooming out to the statement list (or begin-end construct) and just type the required statement at the desired place. A more elegant and faster way is shown in Figure 5.

The statement "y := y * x" is a child of a variable arity or list node. We may therefore execute the "insert-hole-after" operation. This operation inserts a meta-variable string and separator ("<statement>", ";", respectively in the example of Figure 5) after the text of the current node and inserts a meta-variable in the abstract syntax tree at the appropriate place. The focus is placed at the meta-variable string. The new situation is shown in Figure 5 as the result of the arrow labeled "insert hole". This way of inserting a statement in the list evidently minimizes the amount of parsing.



```
while ( x > 0 ) do
  begin
    y := y * x ;
    k := k + 1
  end
```

```
while ( x > 0 ) do
  begin
    y := y * x; <statement> ;
    k := k + 1
  end
```

```
while ( x > 0 ) do
  begin
    y := y * x; if <exp> then <statement> ;
    k := k + 1
  end
```

Figure 5
Programme creation using insert hole and expand

The focus is now positioned at the hole. GSE, from its knowledge of the grammar, is capable of determining which language constructs are allowed at this place. Thus a list of possible substitutions of the meta-variable string "<statement>" may now be requested. An item from this list may be selected and substituted. For instance, if the "if-then" statement would have been selected, the string "<statement>" is replaced by the string "if <exp> then <statement>". This is shown in Figure 5 as the text resulting from the arrow labeled with "expand". The present implementation of the expand command does not insert layout into the text. This must still be done by hand. We will discuss this problem in more detail in Section 7.1.

The expand operation may be used with any rule from the context-free syntax. Lexical tokens cannot be generated in this way and consequently must be typed by the user. The large scale structure of the program may thus be built top down by executing *insert hole before, insert hole after* and *expand* commands. Furthermore, the user does not need to be intimately acquainted with the syntax since GSE inserts syntactically correct templates or place holders for each language construct selected during an expand operation.

5. The implementation of GSE

In this section we briefly discuss some aspects of the implementation of GSE. Section 5.1 describes the data structure on which GSE operates, and Section 5.2 describes how this data structure is updated in response to user actions.

5.1. The GSE data structure

The data structure on which GSE is based consists of four major parts: a text representation, an abstract syntax tree, a focus, and a view. The abstract syntax tree is a VTP tree. The text representation is a list of strings, where each string represents a line from the text. The focus part of the structure keeps track of the text region associated with the focus, the cursor position and whether the focus is still syntactically correct (i.e., whether a text edit operation has occurred). The view part contains display information.

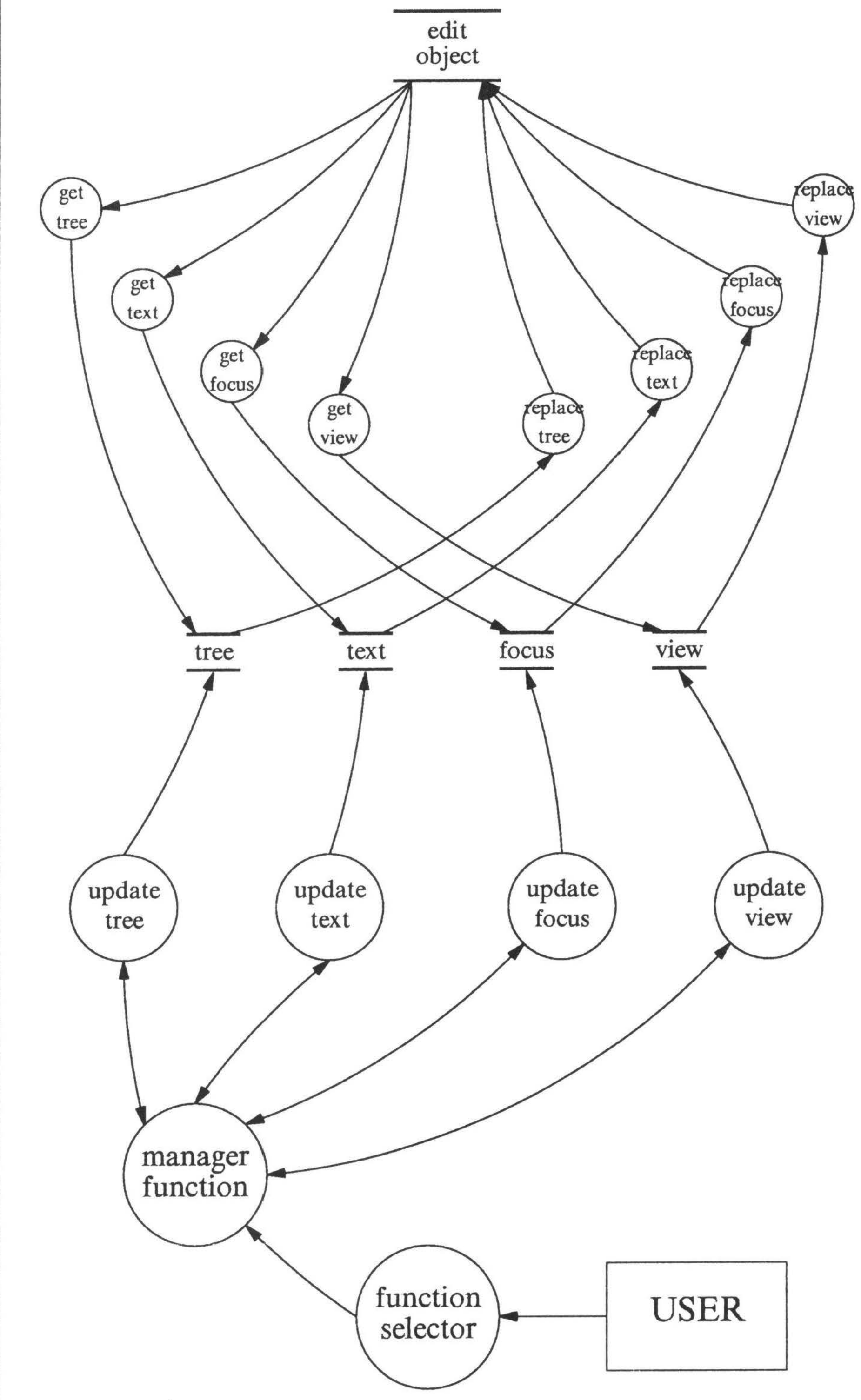


Figure 6
Schematic representation of the processing of a user action by GSE.

5.2. Updating the data structure

GSE considers all user actions as events in the outside world. The events are placed in an event queue. Every event is labeled with a number of items from which GSE is able to reconstruct the nature of the user action. Every event has a "manager" function associated with it, that operates on the data structure of GSE. Any updates of the data structure are performed by specialized lower level functions that are controlled by the manager function. The processing of a user action is shown in Figure 6.

A specific user action invokes an associated update function of the manager group. This function analyses the action and determines which parts of the data structure need to be updated. The parts that need to be updated are retrieved from the data structure and are passed to specialized functions that know how to perform the update. After updating the specialized function returns the updated data part to the manager function, after which the updated data part is replaced in the data structure.

There are two obvious advantages to this organization. First, the integrity of the data is ensured. Second, GSE's functionallity may easily be extended. If, for instance, GSE must be extended with actions that need data not yet present in the data structure this may be done without affecting the existing parts of GSE. Examples of such extensions, like the integration with a pretty printer or coupling GSE with type checkers and interpreters, are discussed in Sections 7.1, 7.4 and 8.

6. The user interface

In this section we will give an impression of what the user interface of GSE looks like. For a more detailed description we refer to [DK89a].

Figure 7 shows a GSE edit window. At the top of the window a menu-bar is displayed. Each item of the menu-bar is a pull-down menu, i.e., when it is activated a column of buttons will be displayed each of which has a specific action associated with it. Along the right-hand side and the bottom of the window scroll-bars in the style of the Macintosh interface are displayed.

In the next two subsections we describe the various components of the GSE edit window in more detail. In Section 6.3 we briefly discuss the use of multiple windows.

6.1. The menu and scroll bars

The items in the menu bar are activated by pressing the mouse button, while the mouse is in the box associated with the desired menu item. A pull-down menu appears containing actions that may be selected from this menu item. The pull-down menu for the tree menu is shown as an example in Figure 8a. To select an action move the mouse down while keeping the mouse button depressed and release the mouse button when the mouse has entered the box associated with the desired action. On moving the mouse down the box associated with the action that will be executed on releasing the mouse button will be shaded. Figure 8b shows the pull-down menu just previous to releasing the mouse button when selecting the go to next child operation.

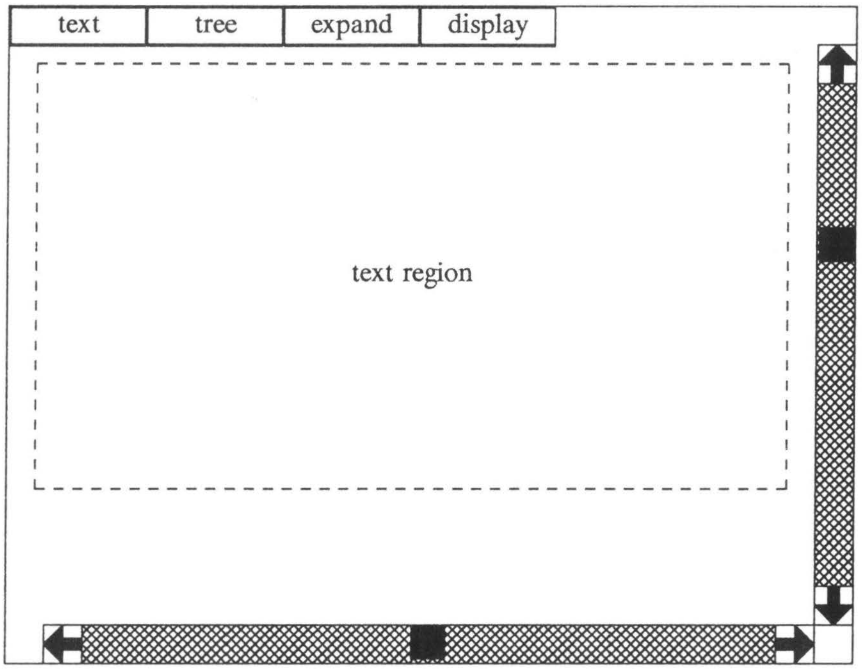
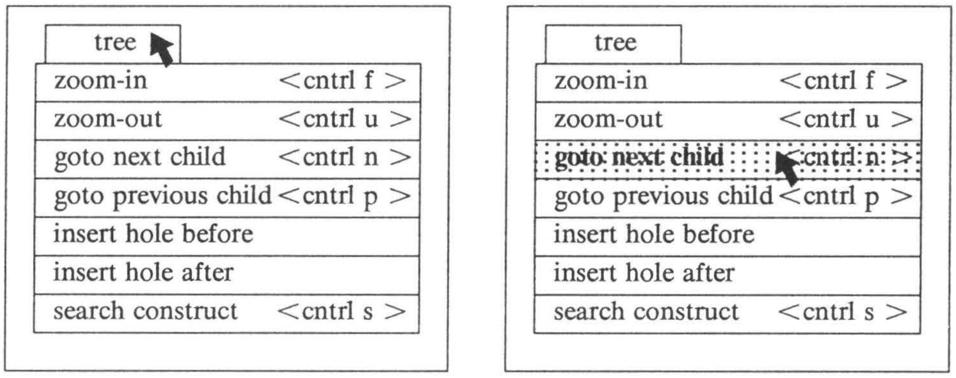


Figure 7
The layout of a GSE window on the screen


tree
zoom-in <cntrl f >
zoom-out <cntrl u >
goto next child <cntrl n >
goto previous child <cntrl p >
insert hole before
insert hole after
search construct <cntrl s >
tree
zoom-in <cntrl f >
zoom-out <cntrl u >
goto next child <cntrl n >
goto previous child <cntrl p >
insert hole before
insert hole after
search construct <cntrl s >

(a) (b)

Figure 8
The pull-down menus of the menu bar at the top of a GSE window

Many actions may also be executed by means of key strokes. The key stroke that performs the same action as the item from the pull-down menu is also shown. For instance, the go to next child action may also be executed by striking "n" with the control key depressed.

The vertical scroll bar consists of four parts. Two arrow boxes to move the window up and down over the text line by line. A scroll block, shown as a black rectangle, is used to move the window up and down over the text by dragging. The location of the scroll block in the shaded bar is a measure of the position of the window on the text. Single mouse clicks in the shaded bar above and below the scroll block move the window up and down respectively by some fixed amount of lines.

Since lines in GSE are of unlimited length the scroll block in the horizontal scroll bar has a different functionality and meaning. It is always located in the middle of the shaded region. It may be dragged all the way to the left of the shaded bar to place the window all the way to the left on the text.

The arrow boxes and shaded bar have a functionality which is similar to that of the vertical scroll bar.

6.2. Mouse pointing.

Mouse pointing was mentioned in Section 4.1. A slightly more general variant of mouse pointing is required for the purpose of correct string searching. It might be that the user wants to search for a certain string and requests GSE to position the focus such that it contains this string. For this purpose we need an operation that is capable of finding the smallest subtree whose associated text contains the selected string.

Both the navigation operations and the (generalized) mouse pointing operation rely on how the mapping between the text and the tree representation is organized. We will consider this mapping more closely in Section 7.2.

6.3. Multiple windows

The user may open more than one window on the same progrm text. In this way it is possible to look at different parts of the text at the same time. It also enables the user to copy parts from one part of the text to another in a simple way using the copy and paste operations (see [DK89a]). Multiple windows have been implemented in such a way that actions are only performed when their effects are visible on the screen. For instance, a character insert is only executed when the cursor is visible. This also implies that the window scrolls automatically to keep the focus visible when navigation actions are performed.

7. Problems in building GSE

It is, of course, not surprising that many problems have been and are being encountered along the way to implementing a syntax-directed editor, that indeed has the functionality described in the previous sections. An anthology of the major problems encountered will be presented in the next four subsections.

7.1. Prettyprinting

We will here briefly consider the problem of layout and text formatting. The discussion will concentrate on the correct functioning of the expand operation.

Most syntax-directed editors keep the program being edited in the form of an abstract syntax tree and recreate the textual form of the program by means of a prettyprinter.

A prettyprinter is a program to format the text of a progrm to improve its readability. A prettyprinter is preferably language independent. Formalisms to specify pretty printers can be divided into two classes. In one class the pretty printer takes a structured representation of the program as input (e.g. an abstract syntax tree) and the prettyprinter rules are essentially mappings from trees to formatted text ([MC86] and [Bor89]). The other class of prettyprinters are mappings from text to formatted text (e.g. [Jok89]).

GSE, however, keeps the program in both forms and does not use prettyprinting. This makes the editor more predictable for the user, but also has some disadvantages, as can, for instance, be seen from the behaviour of the expand command (Section 4.2). The simple expand operation described in Section 4.2 suffers from the drawback that no layout is inserted when building the program skeleton using expands only. The readability of a program text is greatly enhanced when it is supplied with a suitable layout, in particular consistent indentation. Using the expand from Section 4.2 would result in a progrm text consisting of one line containing all language constructs that make up the program text separated by a minimum of layout.

A first step towards improving the mapping from abstract syntax trees to text is to supply every syntax rule with layout information, from which GSE can derive the format that must be used to print the text associated with that rule.

Although such local (rule oriented) pretty-print functions constitute an improvement over a situation with no pretty printing at all, they by no means suffice to provide sufficient layout for producing pretty enough text formats. Rule oriented pretty-print functions provide local layout and formatting information, whereas the pretty-printers we would like to have need global information. For instance, they need to keep track of current indentation levels. We would like to have the keywords "if" and "else" of the Pascal "if-then-else" rule to be aligned with statements that lie directly before or after it, or to be indented with respect to the enclosing compound ("begin ... end") statement. Another example where the pretty printer needs to be supplied with more global information is the case of a list of assignment statements. If, for instance, we want all the assignment operators in the list to be in the same column, the pretty printer must be able to determine the size of the longest left hand side in the list. Within the GIPE project pretty printers are currently under development (e.g. [MCC86] and [Bor89]).

Designing techniques for specifying pretty printers is thwarted by the requirement that specifying a pretty printer for a language should be easy. Having a separate formalism for the specification of pretty printers does not make for easy use. A different approach for obtaining a pretty printer is to make it intelligent and let it learn the formatting instructions from sample texts. This approach is described, for instance, in [WC89], but pretty printers of this type are still very primitive.

7.2. Mappings between text and tree

In order for GSE to be able to position the focus on the correct part of the text, a mapping has to be defined between the text and the tree representation. For instance, when one of the four basic navigation actions is performed, the current node corresponding to the new focus is immediately found. However, GSE has to find the text part corresponding to the new current node. Conversely, in the case of mouse pointing GSE must be able to determine what subtree corresponds to a certain character or string in the text.

Since GSE is text oriented, we have chosen for a solution to this problem that leaves the text representation unchanged. The tree representation is extended with information from which the mapping may be derived. An extensive description of how the correspondence between text and tree is achieved is given in [Dijk89]. A new and improved method for the implementation of such a mapping is described in [Koo]. Here we briefly describe the principles on which such a mapping is based and how the information that represents the mapping is maintained.

Every node in the abstract syntax tree is annotated with four numbers, from which the absolute positions of the first and last character of the text associated with the node may be derived. We refer to these numbers as text pointers, since we are able to derive from them which portion of text corresponds to each node in the tree. Navigation and mouse pointing can now be realised as follows. As soon as the new current node is known in case of navigation, the absolute coordinates of the text associated with this node are derived from the text pointers, and the location of the new focus is known. For mouse pointing GSE performs a depth first search of the tree to locate the smallest subtree (this will become the new current node) that contains the character or string position and next retrieves the positions of the first and last character of the text associated with the new current node from its text pointers.

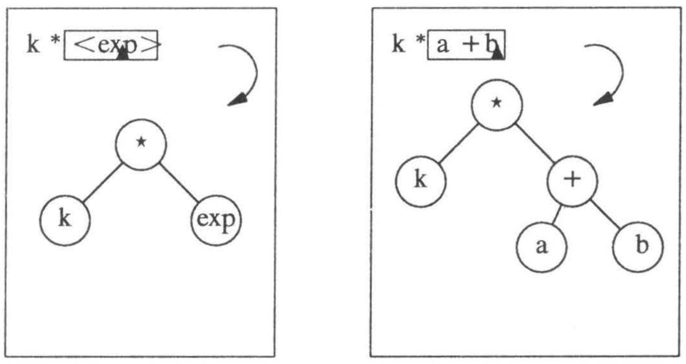
Some work is involved in maintaining the text pointers. If text editing has occurred, the location and shape of text regions of nodes, whose associated text parts are either located after the focus or include the focus text, have changed. Consequently, these text pointers have to be updated accordingly. The procedure for updating text pointers is described in [Koo89].

The decision to store all mapping information in the tree and keep the text representation clean has the evident advantage that text edit operations and operations like scrolling may be implemented in a very efficient way.

7.3. Dealing with priorities in ambiguous language definitions

In Section 2.2 we hinted at a problem arising from the fact that SDF allows arbitrary context-free grammars to be specified. In this section we will discuss this problem in some detail.

There may be ambiguities in context-free grammars, i.e., situations in which the same progrm has two or more different parses. If none of the parses has precedence over the others the only possibility to remove the ambiguity is to attempt to rephrase the syntax rules that define the grammar such that the ambiguities are removed. One way of removing ambiguities is to introduce *priority relations* between different rules of the grammar. The use of priority relations for disambiguation purposes introduces, however, a serious problem, that threatens to defeat the goal of an editor that supports fractional parsing of programs (i.e. allowing not only programs, but also parts of programs to be parsed). We will consider an example.



(a) (b)

Figure 9

Example of a priority conflict. Figures a and b show the abstract syntax trees before and after editing <exp>. The tree in (b) contains a priority conflict when * has greater precedence than +.

Suppose that the text shown in Figure 9a is being edited and that the focus is positioned on "<exp>". The abstract syntax tree (also shown in Figure 9a) corresponding to the text has the multiplication operator (*) as top node. Now we replace the string "<exp>" by the string "a + b" leading to the text shown in Figure 9b. Parsing the new focus text will result in the subtree corresponding to "<exp>" being replaced by the abstract syntax tree corresponding to "a + b". The operator of the top node will still be the multiplication operator. Now suppose we would have specified a priority relation between the multiplication and addition operator, saying that multiplication has higher precedence than addition (which is a sensible thing to do). We then have a problem. The syntax tree corresponding to the new text should have the addition operator as its top node. This shows that in some cases it is not correct to just replace the subtree corresponding to the text of the focus by its new version after some edit actions have been performed. The problem we are alluding to here is a subtle one that occurs in many disguises.

It is not evident what GSE should do in cases like this. In the case of there are two options. One can either "reshuffle" the tree to remove the pri also feasible that brackets are placed around the string "a + b" if that is allow in order to shield the addition construct from the stronger precedence of the multi The second solution seems to correspond more closely to the wishes of the user. Th the string "<exp>" by "a + b" the user seems to have expressed the wish to multip "a + b" by "k". However, it can easily be shown that the placing of brackets will not intended shielding effect under all circumstances.

7.4. Editing in the meta-environment

The GIPE environment can, in particular, be used for designing programming languages. The us GSE for this purpose introduces complications, that we will discuss here in some detail. A mo elaborate discussion is given in [Kli].

A complete language specification generally consists of a syntax definition in SDF and a specification of the semantics in ASF (algebraic specification formalism). Currently a combined formalism ASF+SDF is under development with which both the syntax and semantics of a language may be specified. A discussion of ASF+SDF and a fairly large example specification are given in [Meu88] An environment (the set of programming tools like parsers, type checkers, and so on) is generated from these language specifications. We call the development environment for language specification the "meta-environment". Clearly, we would like to use GSE for constructing a language specification. In order to discuss the problems related to editing in the meta-environment we will consider an example specification in ASF+SDF.

```
module Booleans
exports
begin
    sorts BOOL
    context-free syntax
        true                    → BOOL
        false                   → BOOL
        BOOL ∧ BOOL             → BOOL
        BOOL ∨ BOOL             → BOOL
        ¬ BOOL                  → BOOL
variables
    x                       → BOOL
equations
[B1]    x ∨ true = true
[B2]    x ∧ false = x
[B3]    ¬ true = false
[B4]    ¬ false = true
end Booleans
```

Figure 10

Example of a specification of the Booleans in ASF+SDF.

Figure 10 shows an ASF+SDF specification of the module "Booleans". This specification is not complete, it just serves to illustrate the type of problem one encounters while editing in the meta-environment.

The syntax of Boolean expressions is defined in the "context-free syntax" section of the module. This syntax is used in the equations section for defining the meaning or semantics of Boolean expressions. Evidently, a modification of the syntax of the Booleans influences the way in which the semantics part (equations section) must be parsed. For instance, if we decided to replace the symbols "∧" and "∨" (the logical "and" and "or" operations) by the strings "and" and "or", the parser for the equations section must be modified in order to recognize equations with this new syntax correctly.

ASF + SDF also allows the user to have modules imported by other modules. If some module A is imported by module B then the functions of module A with their syntax and semantics are available within module B. In this case a modification of the syntax of module A may even influence the way in which another module (in our example module B) must be processed.

ASF + SDF provides many other facilities, like parametrized modules and renaming of functions and sorts that introduce similar problems. [Kli] proposes a strategy for tackling those problems.

8. Conclusions and future developments

The present version of GSE meets the demands of a well designed syntax-directed editor. Users, programming in a language they are not very well acquainted with, are assisted by GSE in using the language correctly (in particular by means of the "expand" command). More experienced users are offered facilities to build their programs in a fast and easy way.

Some of the possible extensions of GSE have already been discussed in previous sections (4.3, 6.2, 7.1 and 7.4). In this section we will discuss extensions in a more general context.

GSE is part of the interactive software development environment of GIPE. This environment consists of a variety of tools for designing programming languages and for processing programs, like parsers, type checkers, interpreters, pretty printers, and window managers. At the moment, GSE has been integrated only with a parser (generator) and a window manager to provide the user with easy access to syntax checking tools. It would be desirable to integrate all the tools from the environment with GSE. In this way all facilities offered by the tools in the environment are made available to the user in a homogeneous way. Furthermore, the user would, just as in the case of syntax-directed commands, benefit from the knowledge that the environment has about, for instance, type checking and pretty printing, and thus would be assisted in building a program that is syntactically correct and is also semantically correct as well as correctly formatted.

The present implementation of GSE can be used for editing documents that contain only text. A totally different direction in which GSE may be extended is adding facilities for the processing of graphics. This would allow the user, for instance, to create a view of a document containing figures. An example of such a facillity is the Unix PIC tool [Ker79] in combination with a previewer. This is particularly of interest when producing documents in software design languages like Jackson and Yourdon. An even more powerful, and consequently more difficult to realise, extension would be to allow the user to directly edit (i.e., modify) the graphical representation of the document as well. An extension of this nature will, in particular, necessitate essential additions to the specification formalism ASF + SDF.

Acknowledgements

We would like to thank Jan Heering, Paul Hendriks and Paul Klint of the Centre for Mathematics and Computer Science, Amsterdam for their minute reading of earlier versions of this manuscript and their many useful suggestions.

References

[Ber86] G. Berry et al., "ESTEREL v2.2 System Manuals," Collection of Technical Reports, Ecole des Mines, Sophia Antipolis (1986).

[Bor89] P. Borras, "Displaying trees and mouse location in the Centaur system," in: 4th Review Report Esprit Project no. 348 (1989).

[CI87] D. Clément and J. Incerpi , "Graphic objects: Geometry, Graphics and Behaviour," in: 3rd Review Report Esprit Project no. 348 (1987).

[CI88] D. Clément and J. Incerpi, "Specifying the behaviour of graphical objects using Esterel," Rapport de Recherche 836, INRIA, Sophia-Antipolis (1988).

[Dev87] M. Devin et al., "Aida: environnement de developpement d'applications," ILOG internal report, Paris (1987), [in French].

[Dijk89] M.H.H. van Dijk , "Maintaining text pointers in the tree representation of a syntax-directed editor," in: 4th Review Report Esprit Project no. 348 (1989).

[DK89a] M.H.H. van Dijk and J.W.C. Koorn, "Generic syntax-directed editor," in: The CENTAUR User's Guide - Version 0.9 I (1989).

[DK89b] M.H.H. van Dijk and J.W.C. Koorn, "Implementation of a syntax-directed editor," in: 4th Review Report Esprit Project no. 348 (1989).

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers, "The syntax definition formalism SDF - reference manual," *SIGPLAN Notices* **24**(11), pp. 43-75 (1989).

[HKR89] J. Heering, P. Klint, and J. Rekers, "Incremental generation of parsers," in: Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, SIGPLAN Notices, pp. 179-191 (1989).

[Jok89] M.O. Jokinen, "A language independent prettyprinter," *Software - Practice and Experience* **19**(9), pp. 839 - 856 (1989).

[Ker79] B.W. Kernighan, "PIC - a graphics language for type setting, revised user manual," AT&T Bell Laboratories, Murray Hill, New Jersey 07974 (1979).

[Kli] P. Klint, "A meta-environment for generating programming environments," in: Proc. METEOR Workshop on Methods Based on Formal Specifications, Mierlo, The Netherlands, September 1989, to appear.

[Koo] J.W.C. Koorn, to appear.

[Lan86] B. Lang , "The virtual tree processor, ," in: 3rd Review Report ESPRIT Project no. 348 (1986).

[LeLisp] *LeLisp, Version 15.21, le manuel de référence*, INRIA, Rocquencourt (1987).

[Log88] M.H. Logger, "An integrated text and syntax-directed editor," Report CS-R8820, Centre for Mathematics and Computer Science, Amsterdam (1988).

[Meu88] E.A. van der Meulen, "Algebraic specification of a compiler for a language with pointers," Report CS-R8848, Centre for Mathematics and Computer Science, Amsterdam (1988).

[MCC86] E. Morcos-Chounet and A. Conchon, "PPML, a general formalism to specify prettyprinting," in: Proceedings of the IFIP Congres Dublin, Springer-Verlag, North Holland (1986).

[WC89] K.A. Winter and C.R. Cook, "A prototype intelligent prettyprinter for Pascal," *SIGPLAN Notices* **24**(9), pp. 116-123 (1989).