

SIC'95 Report and User Manual

Lothar Schmitz
Frank Schulter
Jens van Laak

Contents

- 1. Purpose and characteristics of SIC ... 2
- 2. A first look at SIC ... 4
- 3. Installation ... 9
- 4. Using the tools ... 11
 - 4.1 Overall organization ... 11
 - 4.2 Browsers and editors ... 13
 - 4.3 Control windows ... 18
 - 4.4 Information windows ... 23
- 5. Formats of definitions and data ... 26
- 6. Example applications ... 31
 - 6.1 Complete applications ... 31
 - 6.2 Demonstrating the parser ... 50
 - 6.3 Demonstrating the scanner ... 58
- 7. On the development of SIC ... 61
- Appendix: Smalltalk basics ... 66

1. Purpose and characteristics of SIC

Many programmers live in happy ignorance of their compilers' internal workings. Others may want to take a look at what is going on inside the black box in much the same way they use a debugger to watch their compiled programs execute.

The Smalltalk-based Interactive Compiler-compiler (SIC) was developed for exactly this purpose. It is an educational tool for visualizing modern compiling techniques which automatically generates demonstration compilers from suitable language descriptions. The generated compilers continuously display their internal states and are controlled with the mouse to move back and forth, in single or larger steps, etc. Auxiliary information derived from the language description can be viewed interactively at any time. While conventional compilers are black boxes whose internals are hidden from the user, SIC helps to open up the box and have a look at what is going on inside.

Conceptually, the compilation process divides into three successive phases: The lexical analysis phase groups the characters of the given input text into a token sequence. The syntactic analysis phase determines the syntactic structure of this token sequence in the form of a syntactic tree. In the following synthesis phase this tree is used to perform the desired translation. E.g. a Pascal compiler would first determine the syntax tree corresponding to a given Pascal program text, and then produce the resulting machine code from the syntax tree.

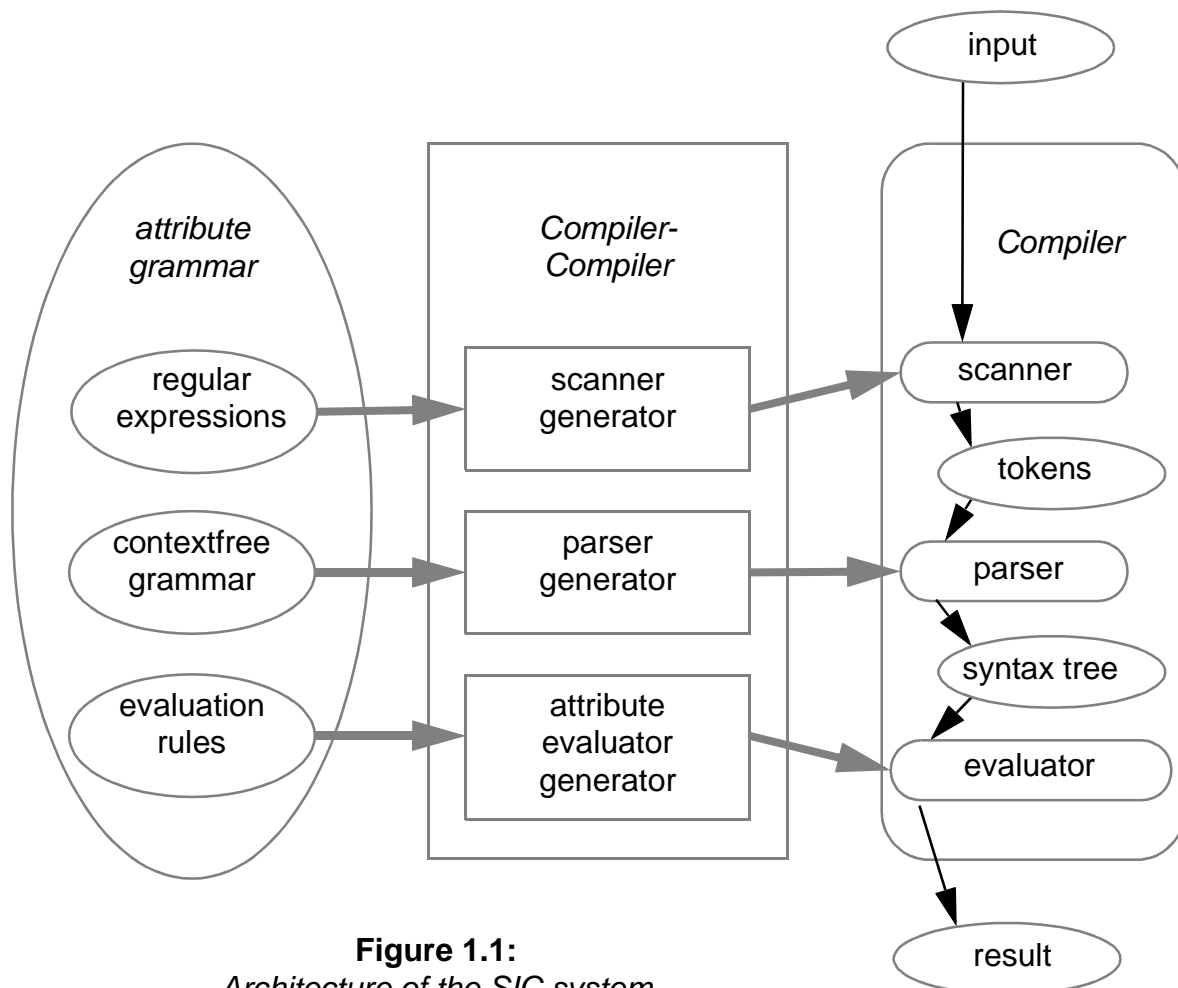


Figure 1.1:
Architecture of the SIC system

With SIC, a compiler is specified by an attribute grammar, which consists of a set of regular expressions defining the token structure, a context-free grammar and a set of attribute rules. As usual, the context-free grammar defines the syntax of the input language. The attribute rules determine the translation into the target language. Figure 1.1 shows that the syntax analyser, or parser, component of the compiler is generated automatically from the context-free grammar. Analogously, the compiler-compiler generates the scanner and the attribute evaluator components from the given sets of regular expressions and attribute rules, respectively.

Technically speaking, the major components of SIC are a scanner generator, a parser generator, and an attribute evaluator generator. Both top-down and bottom-up parsers are produced. When creating a parser, the user may choose between the LL(1), LR(0), LR(1), SLR(1), and LALR(1) parsing strategies, which are typical of modern compilers. SIC-parsers can be operated non-deterministically, e.g. an SLR(1) parser for a non-SLR(1) grammar. Ambiguities are then resolved either by backtracking or by taking advice from the user. The input to SIC-parsers may contain both terminal symbols and syntactic variables. In contrast, the input of conventional parsers may contain terminal symbols only. For attribute evaluation, different strategies are available, including one which works for all well-defined attribute grammars. Attribute evaluation can either be controlled by hand, by selecting one of a set of predefined strategies, or by preparing an individual evaluation sequence using a special editor.

The core of SIC as well as of any other compiler-compiler is a system of rather complicated algorithms that are based on a firm mathematical foundation called parsing and attribute evaluation theory. However, the real challenge when designing and implementing SIC was not theory but came rather from two ergonomic problems: Firstly, to give the user control over the program and not vice versa. Secondly, to overcome the sometimes severe size limitations of video screens.

(a) *How to provide sufficient user control?* Active learning (cf. [4]) within a simulation environment which encourages students to make their own experiences is much more effective than stepping through a predefined sequence of elementary lessons. Because of the nature of the translation process, the main phases lexical analysis, parsing and attribute evaluation have to be carried out essentially in that order. Still, SIC-users have plenty of room for experiments: They can define their own grammars and attribute evaluation rules, choose among several parsing techniques and attribute evaluation strategies, and can direct the parsing and attribute evaluation processes in almost any way they wish. E.g., our parsers can be made to single-step forwards and backwards, or leap to that point in the input where the user clicks the mouse.

(b) *How to cope with the problem of the restricted screen size?* Even for small grammars no video screen is large enough to hold all relevant information (the attributed grammar, the current parsing situation, the parsing automaton, first-/follow-sets etc.) at the same time. For each kind of information SIC provides a special kind of window. Users can open these windows whenever they wish. Large information objects like parsing automata are displayed in smaller chunks and can be traversed in a meaningful way. For some kinds of information, users may switch between different representations that complement each other, e.g. between a leftmost reduction sequence and the partial syntax tree corresponding to it.

In section 2 a small example translation problem, the evaluation of arithmetic expressions, is used to demonstrate the main features of SIC. Section 3 briefly explains how to install the current version, SIC'95, of SIC on top of VisualWorks. Section 4 describes the user interface of SIC. Users who wish to manipulate the definition files using text editors (instead of the special purpose editors provided by SIC) should consult section 5. A sampler of typical applications is given in section 6. Section 7 shortly describes the development of SIC and some of the extensions we plan to implement in future. The appendix introduces the Smalltalk

elements you need for reading and writing attribute evaluation rules.

2. A first look at SIC

As an introductory example we consider the evaluation of arithmetic expressions that contain both number constants and variable names. During evaluation, the variable names from a given expression will be collected and the user will be prompted to provide values for these unknowns. First, the lexical and syntactical structure of admissible inputs has to be defined.

Lexical structure:

The lexical structure is a sequence of numbers, variable names, parentheses, and operator symbols as defined by the following pattern definitions:

```
digitPattern  {$0-$9}
numberPattern digitPattern[1-*]
letterPattern {$a-$z}|{$A-$Z}
namePattern  letterPattern(letterPattern|digitPattern)[0-*]
openingBracketPattern $(
closingBracketPattern $)
addOpPattern $+|$-$-
multOpPattern $*|$/
```

The first line introduces the name „digitPattern“ for the decimal digits specified by the interval „{\$0-\$9}“ of ASCII symbols between zero and nine. Similarly, „letterPattern“ is specified by the union (symbol „|“) of the intervals of small letters and capital letters. In the second line, the repetition clause „[1-*]“ indicates that a „numberPattern“ is a sequence of digitPattern containing at least one element. The general form of a repetition clause is „[<lower bound> - <upper bound>]“. An upper bound of „*“ stands for „any natural number“. Repetition clauses are applied to the pattern expressions immediately preceding them. Therefore, a „namePattern“ consists of one letterPattern followed by zero or more elements of the union „(letterPattern/digit Pattern)“. Notice that new patterns are defined by using the names of pattern that have been introduced before. This merely is an abbreviation mechanism: Circular definitions are not allowed!

The second part of a scanner definition is a list of the token types to be recognized by the scanner. Each token type is defined by one of the patterns defined before .

In our example, we have six token types:

number	numberPattern
name	namePattern
openingBracket	openingBracketPattern
closingBracket	closingBracketPattern
addOpPattern	addOpPattern
multOp	multOpPattern

Application of the resulting scanner to the input text

```
(alpha + 17) * (beta - alpha /17)
```

will produce the following token sequence:

```
openingBracket«(»
name«alpha»
addOp«+»
number«17»
closingBracket«)»
multOp«*»
openingBracket«(»
name«beta»
addOp«-»
name«alpha»
multOp«/»
number«17»
closingBracket«)»
```

Each token is placed in a new line. The token text is enclosed in funny brackets („«, „>>“). Figure 2.1 shows the SIC scanner control window in a situation where in the above input the first character of „beta“ has just been processed. „Active“ are those token descriptions that may possible match the token currently being processed. „Passive“ are the token classes that have been excluded already. Within active token descriptions points represent the current state of the recognition process. The different point positions indicate that the first letter „b“ of the

current token may be followed by more letters and /or digits or not.

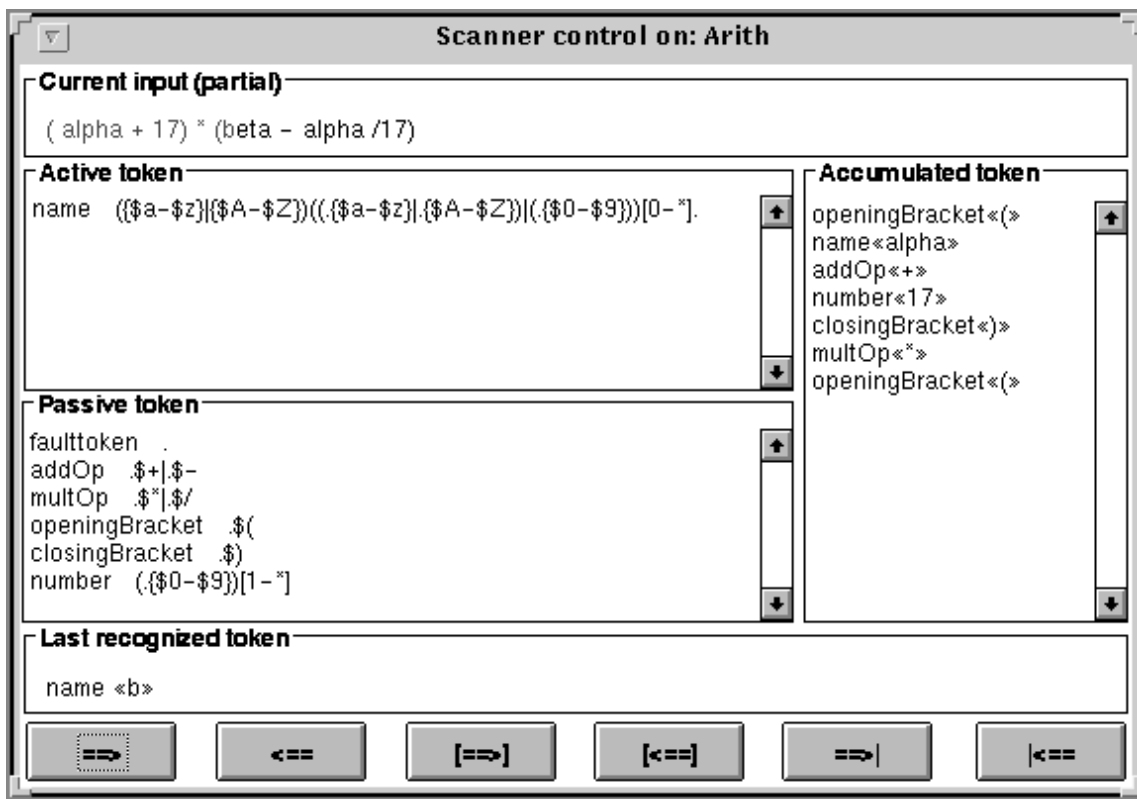


Figure 2.1: *The SIC scanner control window*

Syntactical structure:

Like other compiler-compilers, SIC requires the syntactical structure of the input to be specified by a context-free grammar (CFG). In Figure 2.2 below, the rules of a CFG defining

the language of arithmetic expressions are shown.

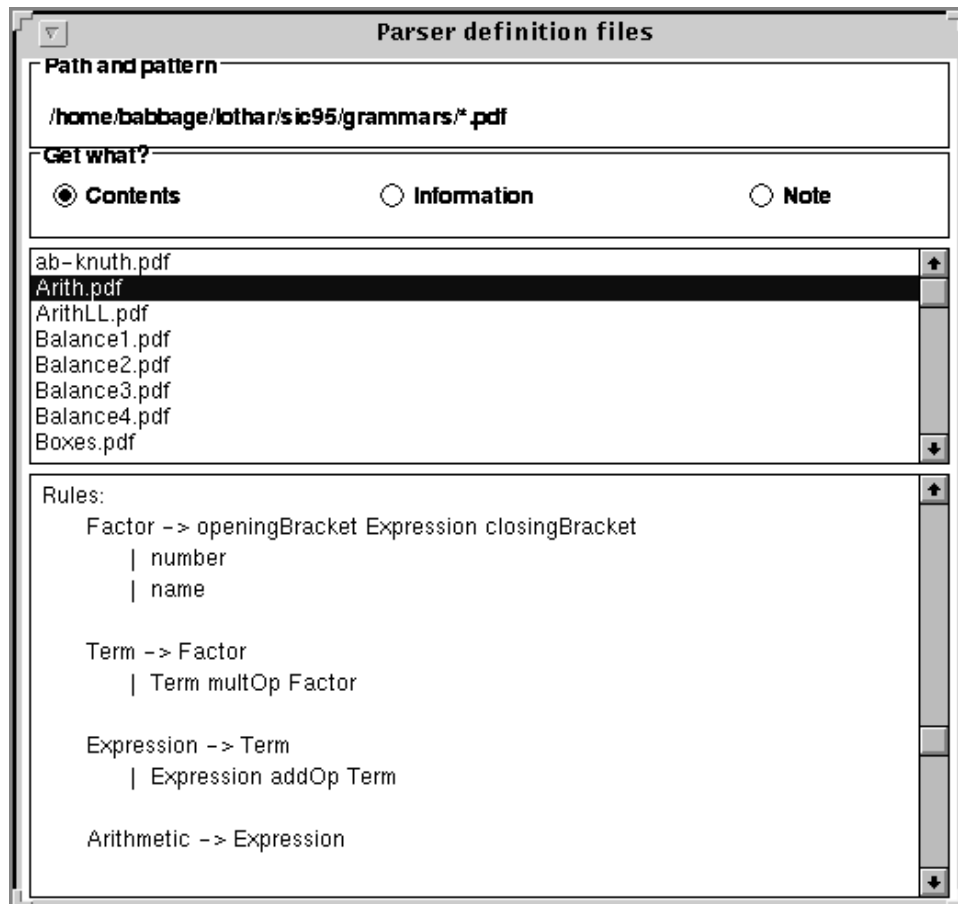


Figure 2.2: *The SIC parser definition browser*

The start symbol, Arithmetic, denotes complete arithmetic expressions. The other nonterminals (Expression, Term, and Factor) correspond to subexpressions and introduce operators in such a way that the resulting syntax trees correctly reflect the precedences of these operator symbols, i.e. multOps take precedence over addOps. The apparently redundant production rule „Arithmetic \rightarrow Expression” will prove use useful for describing the evaluation of arithmetic expressions. The other rules are read as usual. E.g.

$$\text{Expression} \rightarrow \text{Term} \mid \text{Expression addOp Term}$$

states that an Expression is either a Term (first alternative) or a sequence consisting of a sub-Expression, an addOp, and a Term (second alternative).

Notice that the terminal symbols of this CFG correspond exactly to the token types defined above. Therefore, the generated scanner and parser will cooperate smoothly. Figure 2.3 shows the SIC SLR(1)-parser working on the token sequence produced by the scanner. The parser is just about to read the second openingBracket token and (as a bottom up parser) has built a forest of trees on top of the input token processed so far. These trees will become

subtrees of the complete syntax tree for this token sequence.

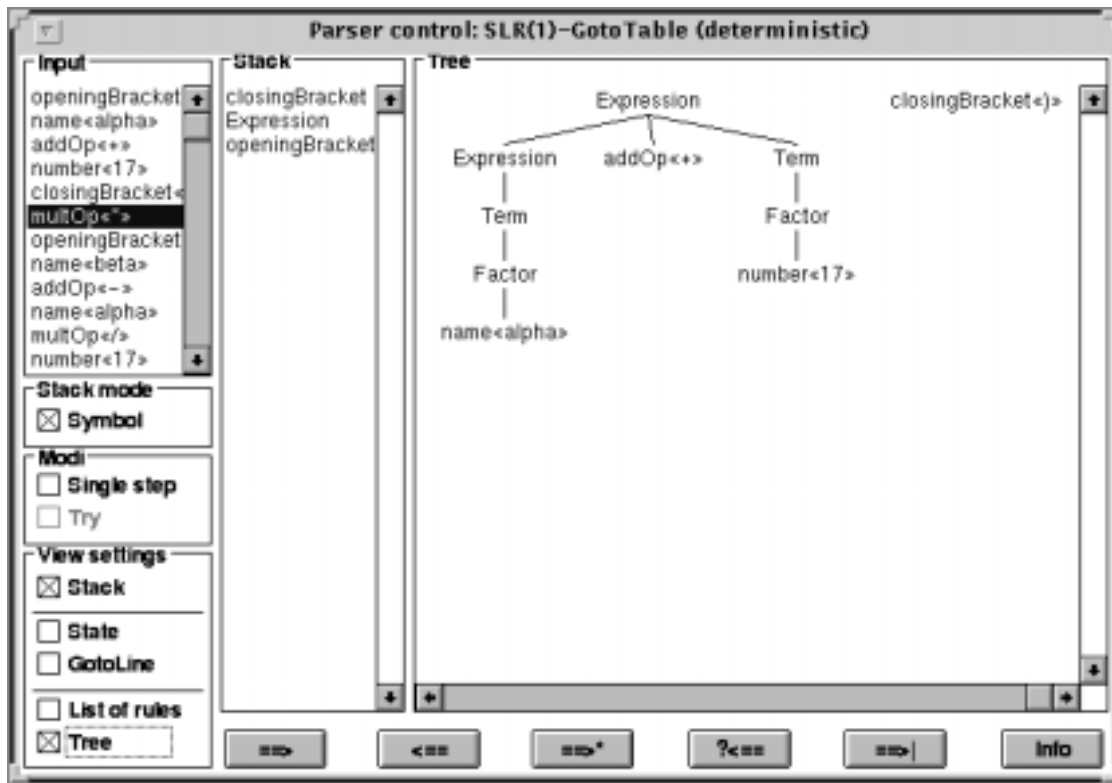


Figure 2.3: The SIC parser control window

Evaluation:

In order to determine the value of arithmetic expression we associate with every nonterminal a „value“ attribute and with every CFG production rule an attribute evaluation rule. The „value“ attributes of Expression and Term are called „valueExpression“ and „valueTerm“, respectively. For the CFG production rule

$$\text{Expression} \rightarrow \text{Expression addOp Term}$$

we have the following attribute evaluation rule (SIC attribute evaluation rules are written in Smalltalk):

```

valueExpression1 :=
  (stringaddOp := '+'
   ifTrue: [valueExpression2 + valueTerm]
   ifFalse: [valueExpression2 - valueTerm])
  
```

This means: If the text of the addOp token (contained in the pseudo attribute stringaddOp) is '+', then the value of the first Expression (the one on the left hand side of the CFG

production rule) is the value of the second Expression plus the value of the Term. Otherwise, (the text of the addOp token is '-' and) the value of the first Expression is the value of the second Expression minus the value of the Term.

For most of the other production rules there are similar evaluation rules. Since the value of a name cannot be determined otherwise, the user is prompted for it during evaluation. Figure 2.4 shows this.

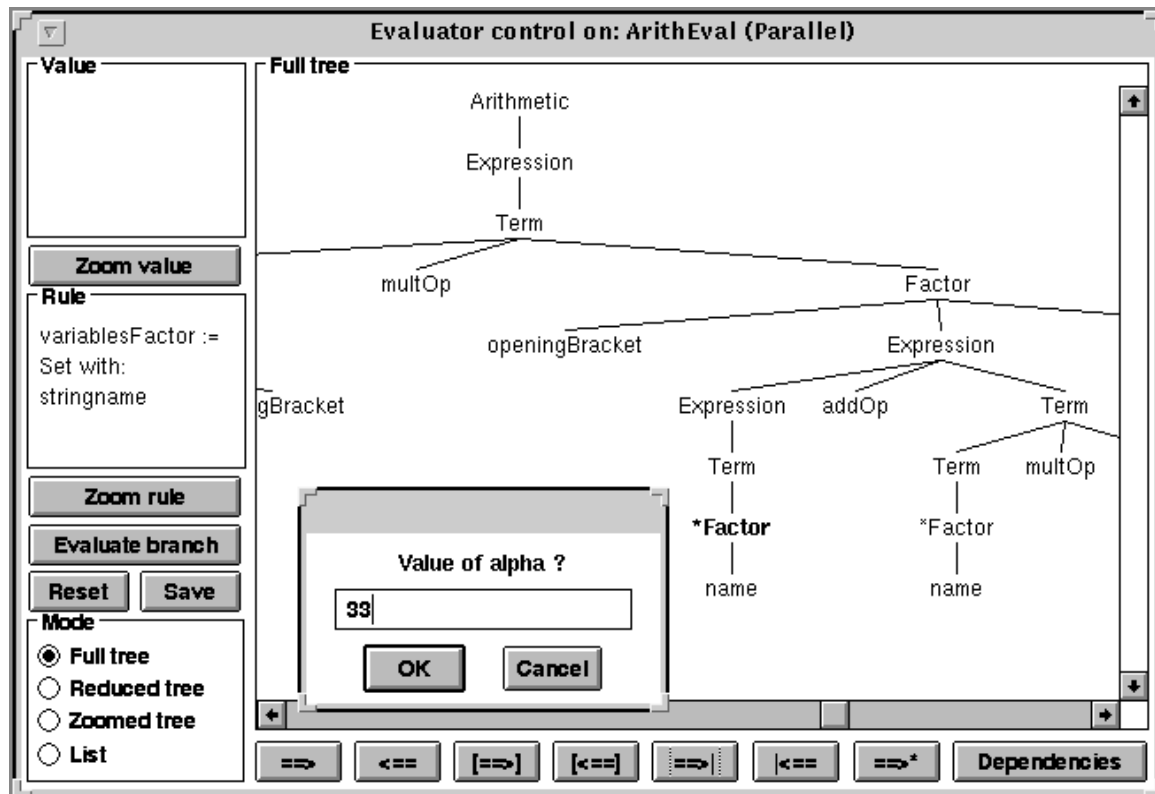


Figure 2.4: The SIC attribute evaluator control window

However, we do not want different occurrences of the same name to get different values. The names and their corresponding values have to be collected and propagated somehow.

For this purpose, two additional attributes are introduced:

- “variables” attributes contain all the names occurring in the subtree at whose root nodes they are placed. Like the “value” attributes they are computed from the leaves towards the root of the tree. Technically speaking, “value” and “variables” are *synthesized* attributes.
- At the root all the variables occurring in the tree are thus known. Here, an “environment” attribute is constructed by prompting the user for the variables’ values. This attribute is propagated from the root towards all the leaves of the tree. “environment” is called an *inherited* attribute.

The details of this and a related grammar will be given in section 6.

3. Installation

In order to install SIC95 you need a VisualWorks 2.0 system. SIC95 requires less than one MB of additional hard disk space. Because of the subtle differences of the underlying operating systems (file formats and packing tools) we describe two different installation procedures. The location and the name of the directory where SIC is placed do not matter. We will always refer to this directory as SIC95. Since the Smalltalk programs are identical on both platforms both installation procedures create the following subdirectories and files of the SIC directory:

```
SIC95
  IMAGE
    visual.im
    visual.cha
    [visual.sou]
  GRAMMARS
    <files with extensions sdf/pdf/edf>
  INPUTS
    <files with extensions sin/pin/ein>
  OUTPUTS
    <files with extensions ll1/lr0/slr/llr/lr1/out>
  sic95.st
```

Unix installation:

You start with either the tar-file `sic95.tar` or its gzipped version `sic95.tar.gz`. Move this file to the SIC95 directory. Unzip a gzipped file with `'gunzip -v sic95.tar.gz'` to obtain `sic95.tar`. Untar the tar file with `'tar -xvf sic95.tar'` the above directories and files.

Windows installation:

The installation disk contains three files

- `install.bat`
- `sic95.zip`
- `pkunzip.exe`

Create the SIC95 directory using e.g. `'mkdir c:\SIC95'`.

Call the installation batch job `'a:\install c:\SIC95'`.

Smalltalk installation (identical on both platforms):

By now, the above directories and files are set up as shown except for the contents of IMAGE which have to be obtained from your VisualWorks installation (see below). We still have to

- (1) make the Smalltalk system available for SIC
- (2) start Smalltalk
- (3) provide access to the Smalltalk source files
- (4) file the SIC programs contained in `'sic95.st'` into the Smalltalk image
- (5) make SIC available via the launcher's `'tools'` menu
- (6) save your installation image

- (1) Copy the VisualWorks files `visual.im` and `visual.cha` to the SIC95/IMAGE directory; access to `visual.sou` can be provided most conveniently using the Smalltalk 'settings' window, see below.

Before proceeding please consult the Smalltalk handbooks about usage of mouse, menus etc. because there typically are confusing differences with the surrounding windowing system.

- (2) Start VisualWorks **from the SIC95/IMAGE directory (!)** with `'oe20 visual.im'`.
- (3) From the VisualWorks launcher's file menu choose 'Settings'. On the 'sources' page fill in the path name of `'visual.sou'`. Close the settings window.
- (4) By clicking on the launcher's drawer icon start the File List. Into the field to the left of the 'auto read' check box type `'./.*'` (Unix) or `'..\.*'` (Windows). Hit the 'enter/return' key. Then in the middle subwindow the contents of the SIC95 directory are shown. From this list choose `'sic95.st'` (answer 'get info'). From the local subwindow menu choose 'file in'. Now, SIC is filed into the Smalltalk system. This will take a few minutes at most. Messages `'... is undeclared'` in the Transcript window can be ignored safely.
- (5) Into the Transcript window (lower part of the launcher) type the expression:

`SICInterface installToLauncher`

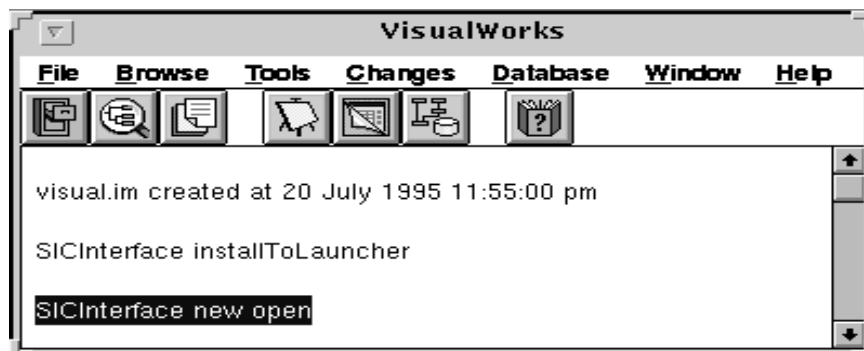
Then highlight the expression with the left mouse button and choose 'do it' (the Smalltalk interpreter) from the local menu. This will result in the old ('this') launcher being closed, and a new launcher being started instead (you will be asked to answer a few questions which you answer in the obvious way, most of them probably with 'yes'). In the new launcher SIC is available as the last item of the 'tools' menu.

- (6) Complete the installation by choosing 'save as' from the launcher's file menu (you can simply confirm the name `'visual.im'` of the image file by hitting 'enter' or you can name the file differently).

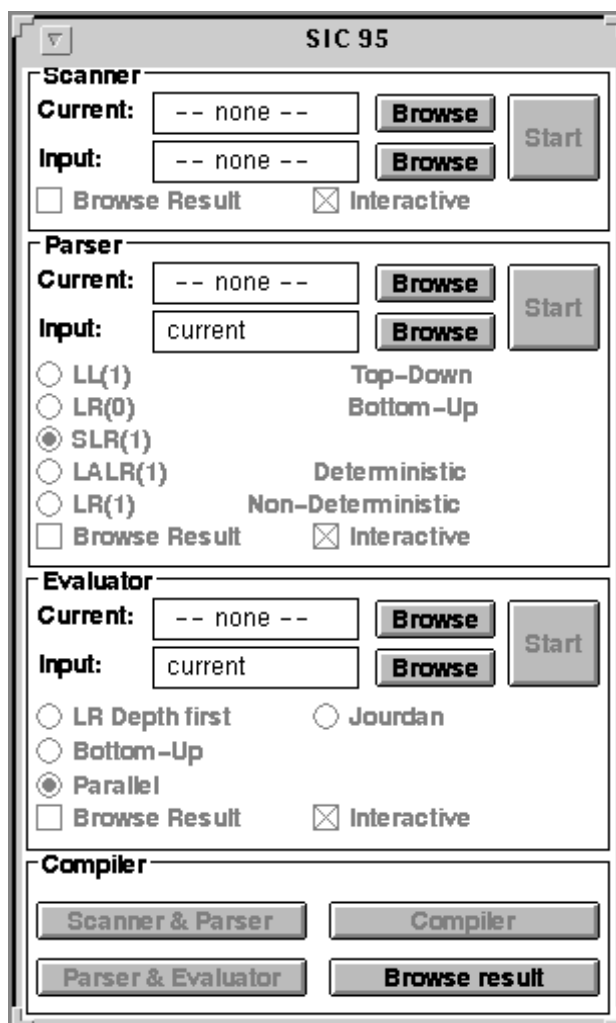
4. Using the tools

4.1 Overall organization

- **Start** VisualWorks **from the SIC95/IMAGE directory (!)** with 'oe20 visual.im'. The Smalltalk launcher window shown below will appear.



- **Start** the SIC95 control panel below by either choosing SIC95 from the Smalltalk launchers' **Tools menu** or by typing (and then highlighting and evaluating) the Smalltalk expression 'SICInterface new open' in any Smalltalk text window.



The SIC95 control window

In the SIC95 control window there are four areas called 'Scanner', 'Parser', 'Evaluator' and 'Compiler', respectively. The upper three areas correspond to the three phases of lexical analysis, syntactic analysis and attribute evaluation. These areas have a similar structure:

- To the right of 'Current:' the name of the current scanner/parser/evaluator is shown.
- To the right of 'Input:' the name of the current input to the scanner/parser/evaluator is shown.
- The 'Browse' buttons open browser windows where you can select an appropriate tool (scanner/parser/evaluator definitions) or input.
- Once a tool and its input have been chosen the corresponding 'Start' button will become active. Pressing this button will start the tool. Depending on the state of the check box 'Interactive' the tool's control windows will open or the input will be processed in 'batch mode'. Depending on the state of the check box 'Browse Result' the result of this phase will be presented in a separate window or not.
- SIC95 offers five different parsing strategies (LL(1), LR(0), SLR(1), LALR(1), LR(1)). When you select one of them by choosing its radio button the required parsing information (first/follow sets, parsing automata etc.) is computed. SIC95 informs you whether the chosen strategy is top-down or bottom-up and whether the parser will operate deterministically.
- SIC95 offers four different attribute evaluation strategies (LR depth first, Bottom-up, Parallel, and Jourdan). The last two are general strategies that work for any cycle-free attribute grammar: a data driven evaluator showing which attributes can be evaluated in parallel, and the demand driven evaluator which was described by Jourdan.
- The result of one phase (scanner or parser) automatically becomes the input of the following phase (parser or evaluator, respectively).
- All inputs and results can be stored to and read from files. Therefore, the tools can be used in almost any order.

In the bottom area there are buttons for starting any combination of two or three successive phases and for viewing the result.

Access to SIC95 windows

There are four categories of SIC95 windows: browsers, (special) editors, control windows and information windows.

The **browsers** all have the same interface and are all started from the SIC95 control window. They all include a simple text editor for manipulating the text representations of input and definition files (formats described in section 5).

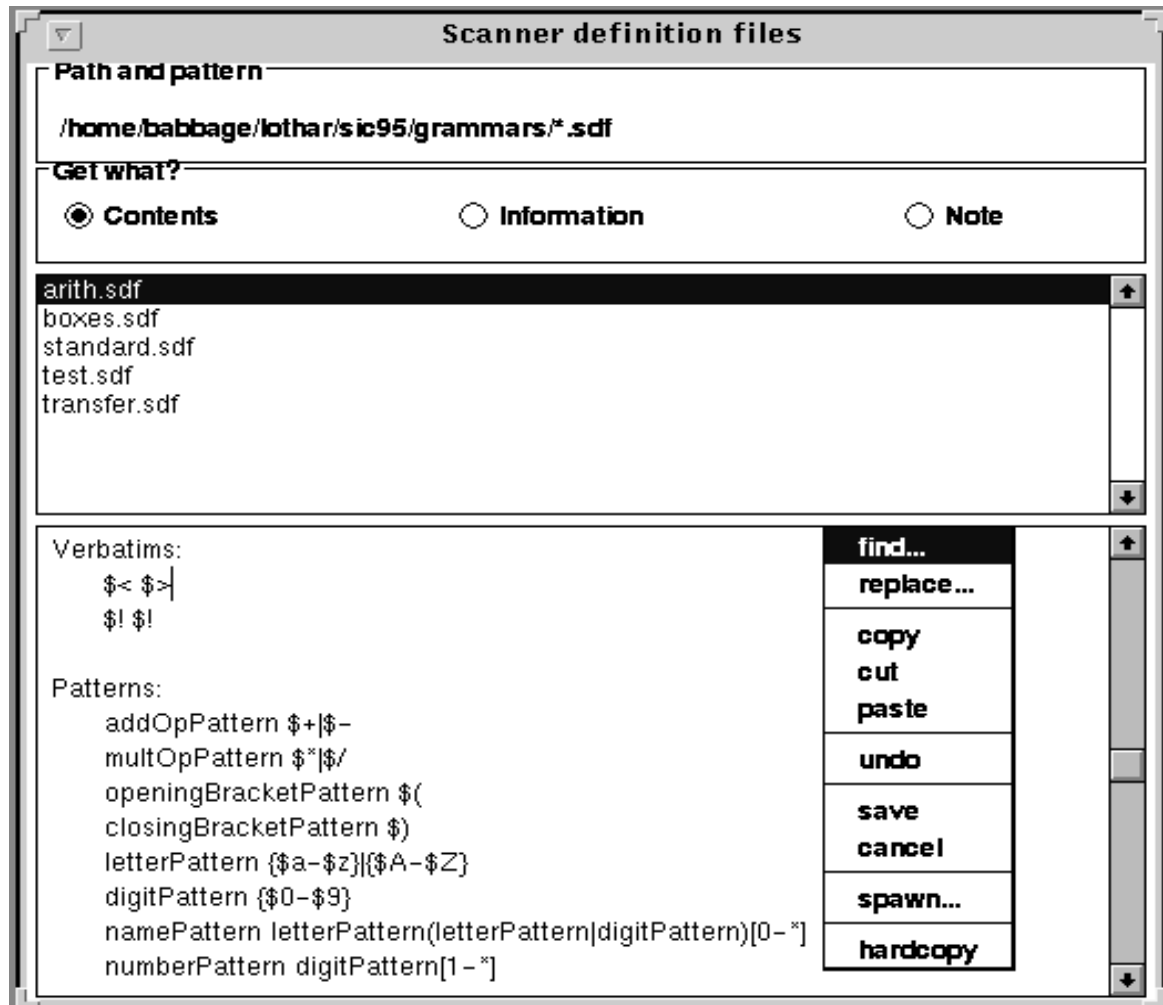
There are **special editors** for context-free grammars, for parser inputs, and for attribute grammar definitions. These are activated from the standard browsers above by choosing 'spawn' (or 'new', respectively) from the middle mouse menu.

The **control windows** (scanner/parser/evaluator control) are started from the SIC95 control window.

Parser-related **information windows** can be started either from the CFG editor or from the parser control window. Likewise, windows showing attribute dependency graphs can be started alternatively from the attribute grammar editor or the evaluator control windows.

4.2 Browsers and editors

The standard SIC95 browser is very similar to the Smalltalk File List window. It is used for browsing input and definition files (see section 5). Depending on which of the seven 'Browse' buttons of the SIC95 control window has been used to start it it will display the corresponding list of files in its middle (scrollable list) subwindow.



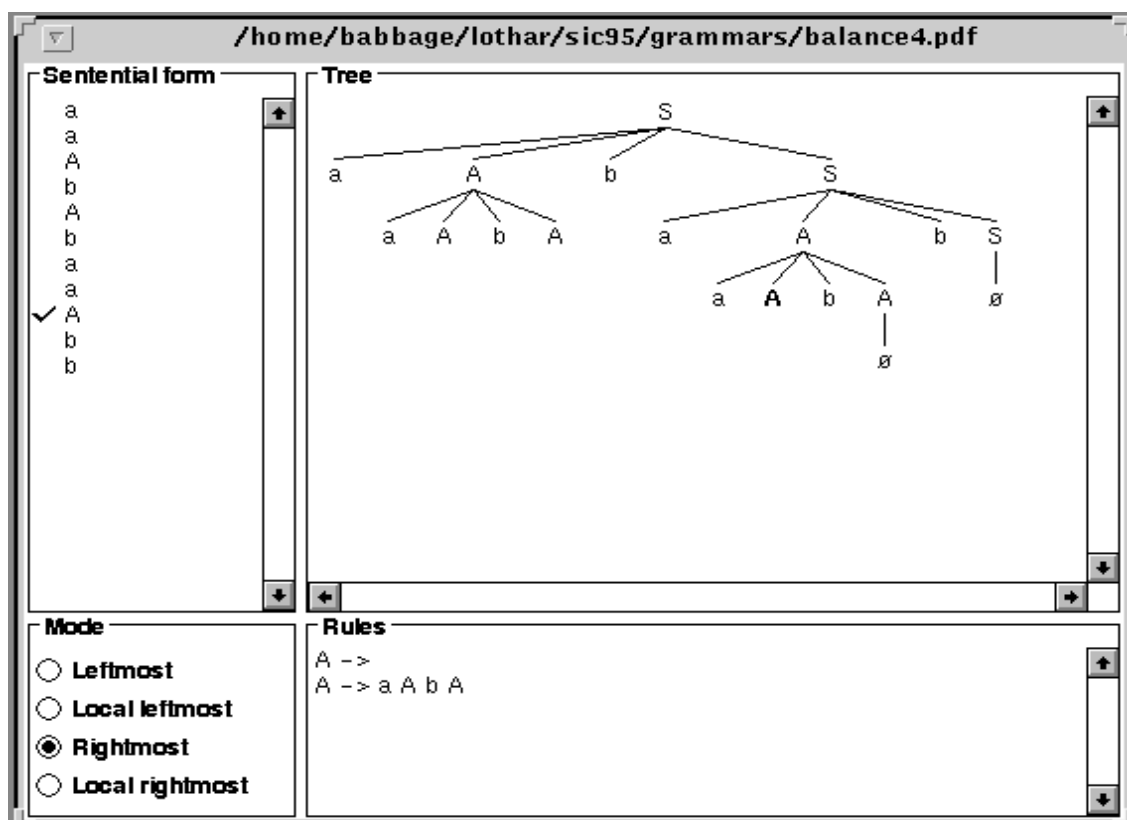
E.g. when pressing the topmost 'Browse' button the above 'Scanner definition files' window will appear. Via the radio boxes 'Contents', 'Information' and 'Note' you can select the information that is shown in the bottom (text editor) subwindow for the file chosen from the list: the file's contents, some file information (size, creation, last access dates), or a user-provided note.

The text editor's pop-up menu indicates its (simple) functionality. By choosing 'spawn' a separate editing window can be started. In most cases, this will be a simple text editor again. For parser definitions or attribute evaluator definitions, this will be one of the special editors to be described below. The (CFG) derivation editor is started from the parser input file browser by the middle mouse menu item 'new'.

When closing a browser window (either by doubleclicking the file name in the list or by choosing 'close' from the window menu) the selected information will be loaded from the file; its name will appear in the SIC95 control window to the left of its 'Browse' button.

The list subwindow's menu allows you to create, delete, copy or rename files as you wish.

The Sentential form editor



The Sentential form editor lets you build syntactically correct parser input files by constructing syntax trees. Starting with the root node of the syntax tree which is labelled with the CFG's start symbol the tree is expanded stepwise into a complete syntax tree.

The 'Tree' subwindow shows the tree constructed so far. The label of the current node is written bold face. In the 'Rules' subwindow the production rules with the current node symbol on the left hand side are listed. When you choose one of these, the syntax tree is expanded accordingly.

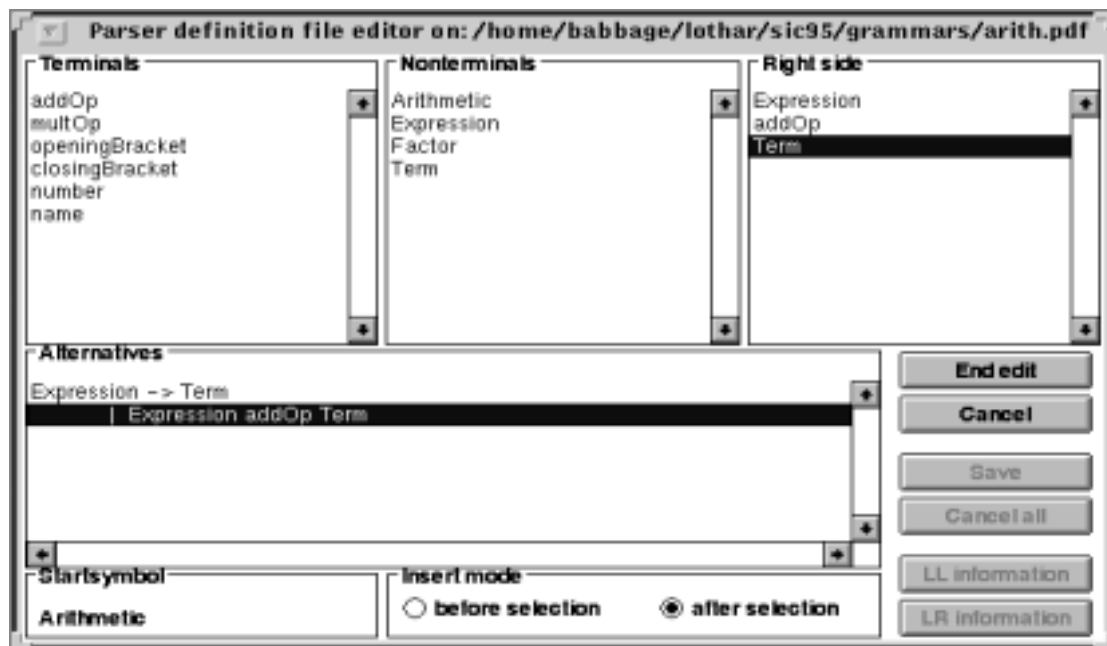
The current symbol can be selected by mouse-click. After an expansion the new current symbol is determined by the derivation window's mode (see radio buttons in the bottom left corner): In any case the new current node is a leaf marked with a nonterminal symbol - if there is such a node. In mode 'Leftmost' ('Rightmost') it is the leftmost (rightmost) such node. In mode 'Local leftmost' ('Local rightmost') it is the leftmost (rightmost) such node within the subtree emanating from the previous current node.

Using the 'Tree' subwindow's local menu you can copy, cut and paste subtrees of any size. The paste operation requires that the label of the root of the tree on the clipboard agrees with the label of the node the subtree is pasted on. Otherwise, the tree resulting from the paste operation might not be a valid syntax tree.

In the 'Sentential form' subwindow the current sentential form (i.e. the left-to-right sequence of leaf labels) is shown with a mark on the label corresponding to the current node.

On closing the window the current sentential form is written to the file whose name was provided by the user before this window was opened.

The GrammarEditor



The GrammarEditor helps to build syntactically correct parser definition files (i.e. CFG definitions) without superfluous typing. The only thing you have to type are the names of nonterminal and terminal symbols which then appear in the corresponding subwindows. All operations on symbols (create, rename, delete) are provided in the local menus. This, too, is the way a chosen nonterminal becomes the start symbol of the CFG.

When you select a nonterminal symbol, say A, then the alternatives of A (i.e. the rules with A on the left hand side) are shown in the Alternatives subwindow. When you select one of the alternatives, the GrammarEditor switches to 'edit Alternative' mode and the symbols of the right hand side are presented in the Right side subwindow for manipulation: The selected symbol can be deleted via the local menu; depending on the Insert mode (see radio buttons at the bottom of the GrammarEditor) nonterminal and terminal symbols you choose by clicking at them with the mouse appear in the right side list before or after the currently selected symbol. There are three ways of leaving the 'edit Alternative' mode:

- by clicking at the alternative in the Alternatives subwindow;
- by pressing the 'End edit' button and
- by pressing the 'Cancel' button.

'Cancel' will cancel the edits you did in 'edit Alternative' mode. This is the first level of a two-level undo mechanism. The second level is the 'Cancel all' button which resets the CFG to the state when the GrammarEditor was opened. Changes are made permanent by pressing 'Save' and then closing the GrammarEditor window.

In order to add a new alternative instead of selecting one of the existing alternatives choose 'new alternative' from the Alternatives local menu: an empty right hand side appears and the GrammarEditor is in 'edit Alternative' mode.

By pressing one of the buttons in the bottom right hand corner of the GrammarEditor you can open an information window which is described in section 4.4. The information will be computed for the CFG in its current editing status.

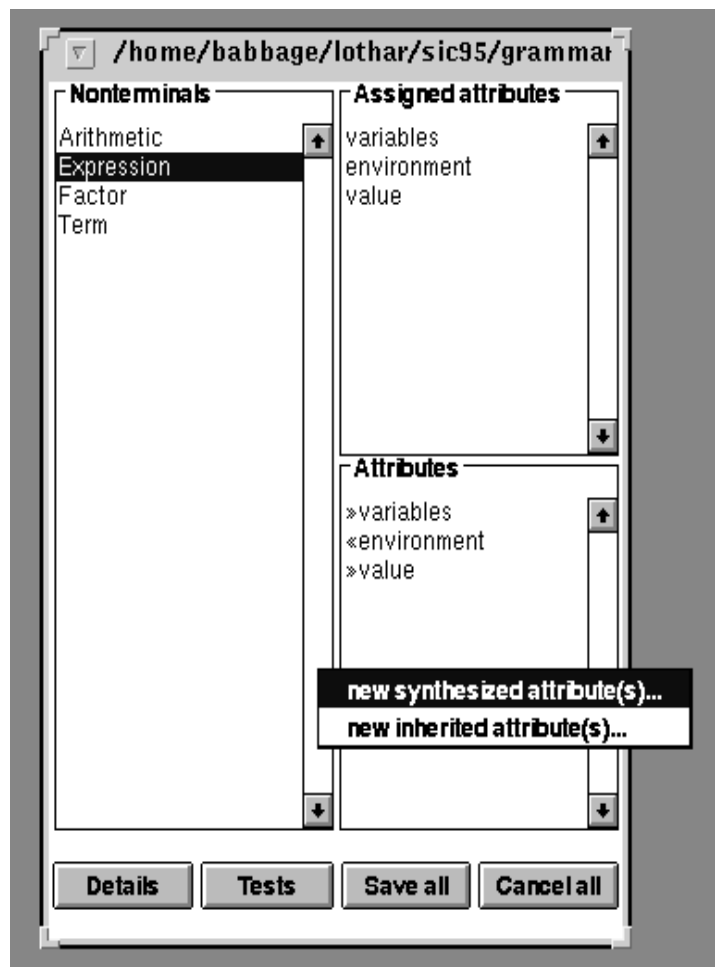
Note: Only identifiers are admissible as terminal symbols. If you need other character combinations use a scanner!

The AttributeEditor

The AttributeEditor helps you to edit the attributes' part of an attribute grammar (AG). It consists of three separate windows which are activated from the StandardBrowser (version Evaluator definition files) and/or from each other.

In the first window attributes are defined, deleted and assigned to the AG's nonterminal symbols. In the second window attribute dependencies and attribute evaluation rules are defined and deleted in the context of the underlying CFG's production rules. Using the third window it can be tested whether the current AG is absolutely cyclefree, cyclefree, or contains cycles.

Below, the first window is shown for the AG from section 2.

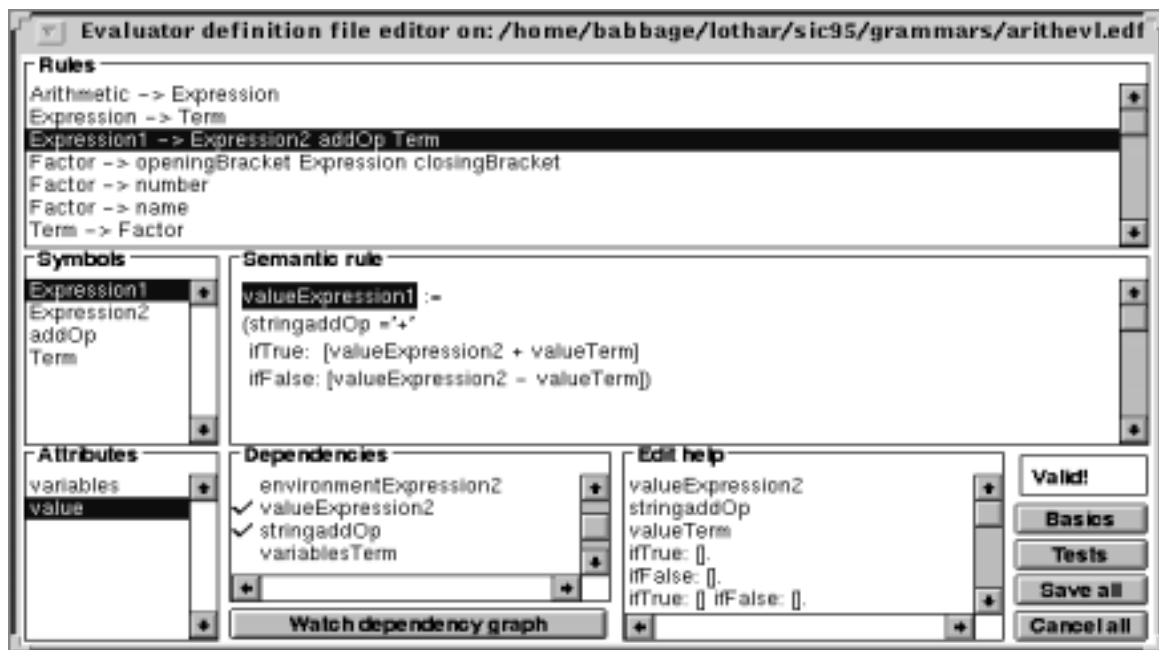


Using the local menu of the Attributes subwindow new synthesized/inherited attributes can be introduced; the selected attribute can be removed (whereby it is removed *completely* from the whole AG). If one of the nonterminal symbols is selected you can assign it an attribute by simply clicking at the attribute in the Attributes subwindow. It will then appear in the subwindow Assigned attributes. By clicking at an attribute in Assigned attributes this assignment is cancelled.

By pressing 'Save all' the current state is saved. 'Cancel all' restores the state which was saved last. By pressing 'Details' or 'Tests' you can switch to one of the other two attribute editor windows (typically one would proceed to 'Details' from here).

The Details window

The Details window serves to edit attribute dependencies and attribute evaluation rules.



In the Rules subwindow the production rules of the underlying CFG are shown. On choosing one of them the symbols of this rule appear in the Symbols subwindow. On choosing one of the symbols its attributes (synthesized attributes of the left hand side, inherited attributes of the right hand side) appear in the Attributes subwindow. On choosing one of the attributes its evaluation rule and its dependencies on other attributes become visible in the Semantic Rule and Dependencies subwindows, respectively.

Typically, one would first define the attribute dependencies. In the Dependencies subwindow all attributes available in the context of the current production rule are shown. By clicking into this list the currently chosen attribute is made dependent of other attributes (clicking toggles between dependency and non-dependency). These attributes are marked with ticks in the Dependencies subwindow and are made available in the Edit help subwindow.

When pressing the 'Watch dependency graph' button an information window is opened which shows all direct attribute dependencies at the current production rule.

The Semantic rule subwindow is a text editor for writing and modifying attribute evaluation rules (to be written in the Smalltalk programming language). Some hints on Smalltalk and, in particular, on compiling attribute evaluation rules are given in the appendix on Smalltalk (the 'Valid!' on the right hand side of the window signals that the text of this evaluation rule has been compiled successfully). Notice that the Smalltalk local variable 'valueExpression1' denotes the 'value' attribute of the Expression nonterminal symbol on the left hand side of the production rule. Similarly, 'valueTerm' denotes the 'value' attribute of Term and 'stringaddOp' denotes the 'string' pseudoattribute of the terminal addOp as recognized by the scanner (i.e. either '+' or '-'). For ease of use, code templates and attribute identifiers can be inserted at the cursor position by choosing them from the Edit help subwindow.

As before, the 'Save all' button saves the current state of the AG and 'Cancel all' restores the state that was saved last. By closing the window you return to the AG browser.

Alternatively, you can switch to the other attribute windows by pressing 'Basics' or 'Tests'.

The Tests window

The Test window serves to test for cyclic attribute dependencies.



The tests are triggered by pressing the 'Start' button. Then first the AG is tested for absolute cyclefreeness. If this test is successful then both 'nil's are replaced by 'true'. If the first test is not successful then (the computationally much more expensive) second test is performed. By pressing 'Basics' or 'Details' you can switch back to the other attribute windows.

4.3 Control windows

All the control windows have been shown in section 2. We first consider their common features. The main control mechanism in all three control windows is the set of control buttons in the bottom. Similar buttons in different windows provide similar functionality:

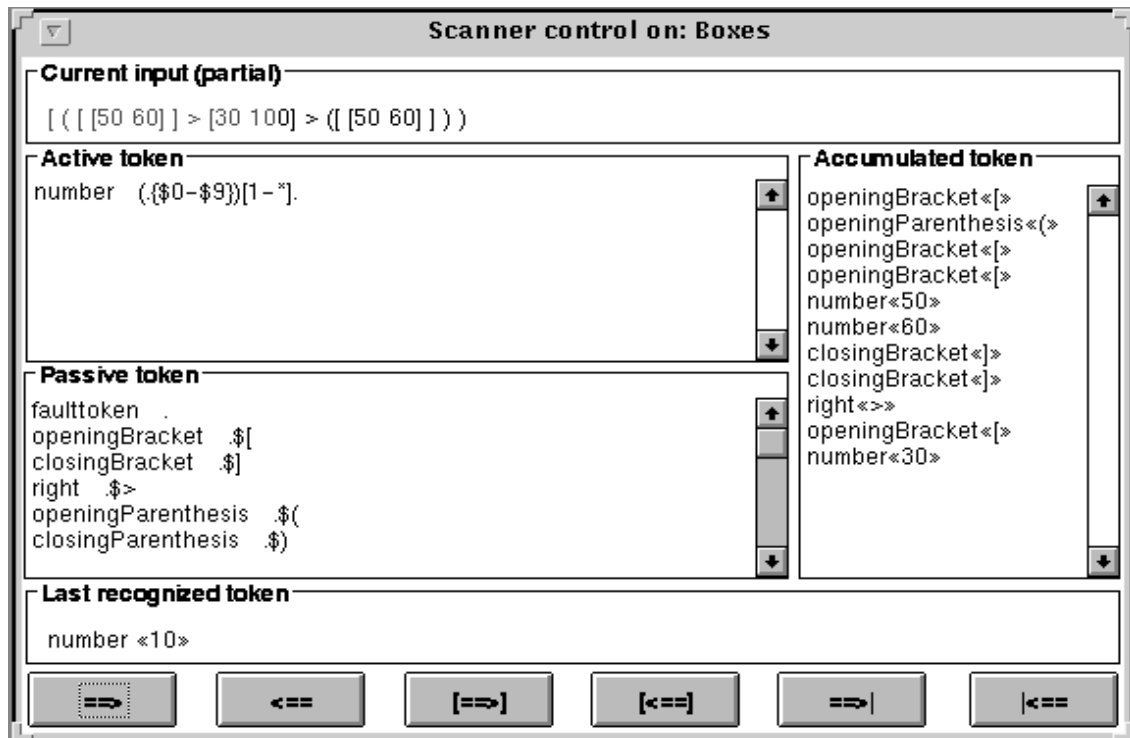
- '=>' one step forward
- '<=' one step backward
- '[=>]' one block forward
- '[<=]' one block backward
- '=>|' forward to the end
- '|<=' back to the beginning
- '=>*' forget the rest
- 'Info' / 'Dependencies' open information windows

Alternatively, users can move the control to a certain point by clicking at it with the mouse. In this respect, the three windows behave very differently.

All the time the control windows display as much information as useful to understand what is going on. Not only the inputs and outputs are shown but also control information used by the controlled process to decide on its next action (active tokens of the scanner, parsing automata, stack contents).

You can interrupt processing any time and go back to editing or get information.

The Scanner control window



Here, one step processes one input character; a block consists of accumulating one token.

The top subwindow shows the current input line. The '(partial)' after 'Current input' indicates that there may be more than one input line (as is the case here). The characters that have already been processed are red, the others are black.

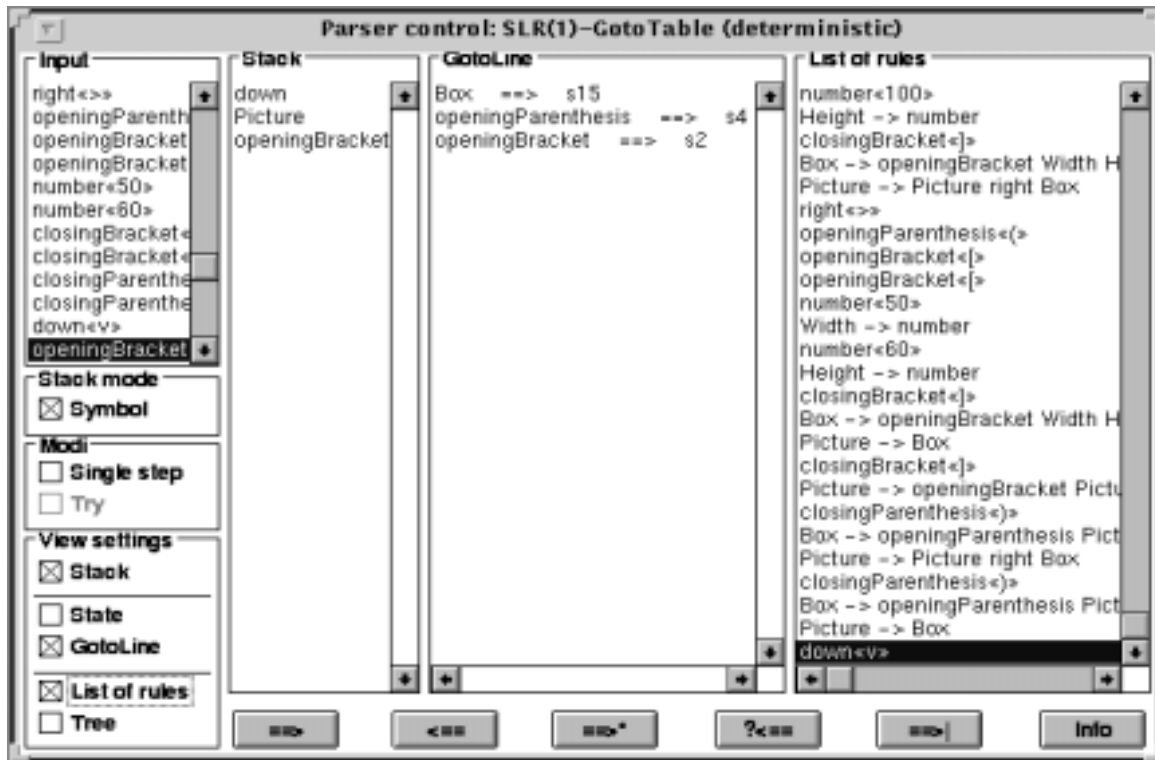
The 'Accumulated token' subwindow shows the tokens that have been recognized so far.

In the bottom you see the token that is currently being accumulated (under the heading 'Last recognized token'). Here, the scanner has processed the first two digits of the input number '100'.

The subwindows 'Active token' and 'Passive token' contain all the tokens defined in the scanner definition (for syntax see section 5). Auxiliary pattern names have been eliminated by replacing them with the patterns they denote. When processing starts or when some token has just been recognized then token recognition starts (again) and all token definitions are active. On processing a character all those tokens are excluded (made passive) that do not match this character. Within an active token points indicate how far processing of this token has progressed (typically there is more than one point in a token).

Except for the buttons in the bottom there is little user control for this window: You can click to a token in the 'Accumulated token' subwindow. This corresponds to a 'move back to this point and forget the rest'. The other subwindows provide no user control at all.

The Parser control window



Here, one step processes one input token; there are no block steps.

The amount and type of information displayed in this window is controlled by the settings of the checkboxes under 'View settings': Instead of the two subwindows 'Stack' and 'Tree' in figure 2.3 above we have three subwindows labeled 'Stack', 'GotoLine', List of rules'. Using the checkboxes you switch on/off (or alternate between) the following information subwindows (for *bottom up parsers*):

- 'Stack' displaying the contents of the parser's stack;
- 'State' displaying the current parser state in symbolic form;
- 'GotoLine' displaying the current parser state in reduced (tabular) form;
- 'List of rules' displaying the current parser output as leftmost reduction sequence;
- 'Tree' displaying the current parser output as forest of trees.

For *top down parsers*:

- there are no parser states;
- 'List of rules' displays the current parser output as rightmost reduction sequence;

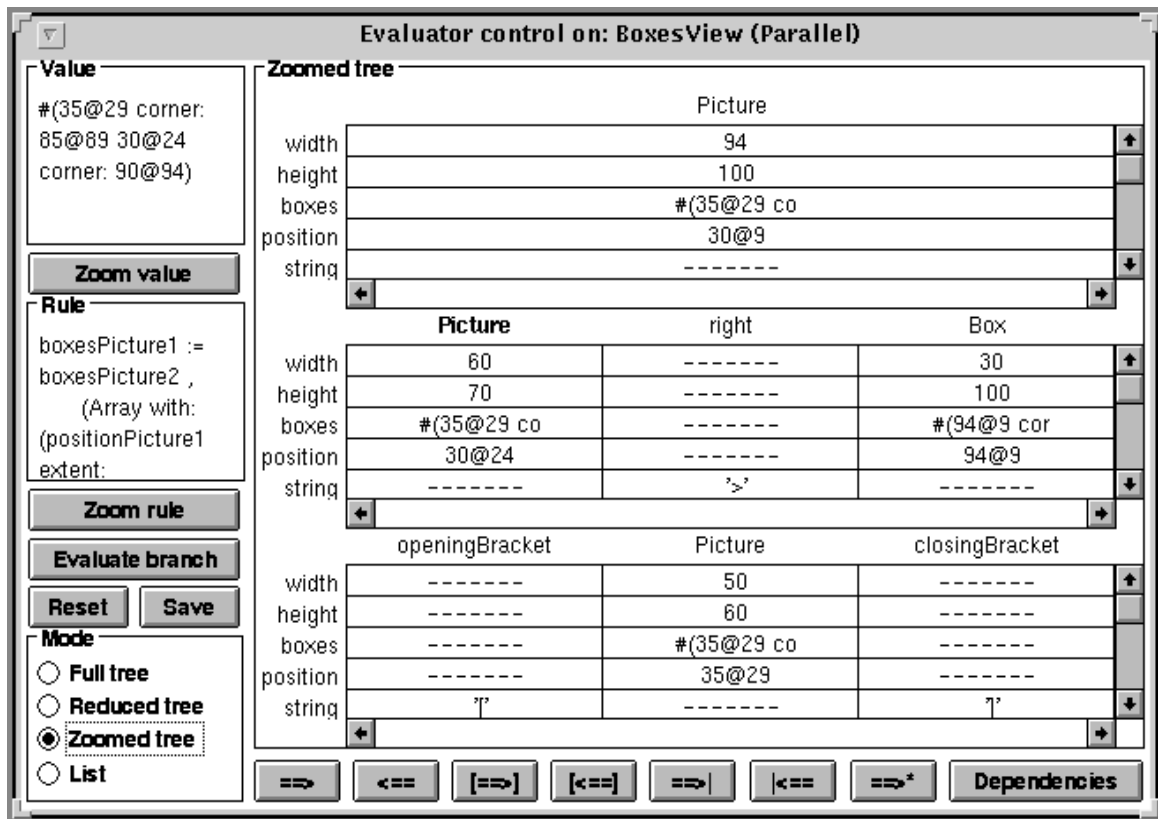
Here, one step evaluates one attribute occurrence in the tree; a block evaluates a sequence of attributes that do not depend on each other (strategy 'Parallel').

When comparing figure 2.4 with the pictures above and below you will find that there are rather different representations of attribute trees for the Evaluator control window. The radio buttons in the bottom left hand corner are used to switch between these different representations:

In the modes 'Full tree' and 'Reduced tree' a graphical tree representation is shown where the currently selected node is written with bold face letters and asterisks mark nodes that contain attributes which are ready for evaluation. In a reduced tree only subtrees which contain as yet unevaluated attributes are shown.

In the 'List' mode all the tree's attributes are shown in the order determined by the attribute evaluation strategy as chosen in the SIC95 control window. In this list every line corresponds to one attribute occurrence in the tree. On the left hand side the value or its evaluability status ('*' for 'immediately evaluable', '-' for 'not yet evaluable') are given.

In mode 'Zoomed tree' the current node and its neighbours are shown along with their attributes: The middle row contains the current node (bold face) and its brothers. The upper row contains the father of the current node and the lower row the immediate sons of the current node.



The control buttons work as explained for the other control windows. They refer to the attribute list generated by the chosen strategy.

Alternatively, in 'Full tree' mode attribute evaluation can be controlled completely 'by hand'. As a side effect, this will produce a reorganized attribute evaluation list where all the attributes are listed in the sequence in which they have been evaluated. In order to evaluate an

attribute click at its node to make it current and from the local menu choose the attribute to be evaluated (must be marked with '*', i.e. be ready for evaluation). As a result, the attribute value and evaluation rule appear in the 'Value' and 'Rule' subwindows.

Since attribute values and attribute evaluation rules can become quite voluminous the contents of these two subwindows can be transferred to separate (resizable) windows with 'Zoom value' and 'Zoom rule'.

By pressing the button 'Evaluate branch' you can evaluate all the attributes of the subtree whose root is the current node (*and* all those attributes outside this subtree that the attributes inside the subtree depend on directly or indirectly).

The 'Reset' button restores the Evaluator control window to the state when it was opened.

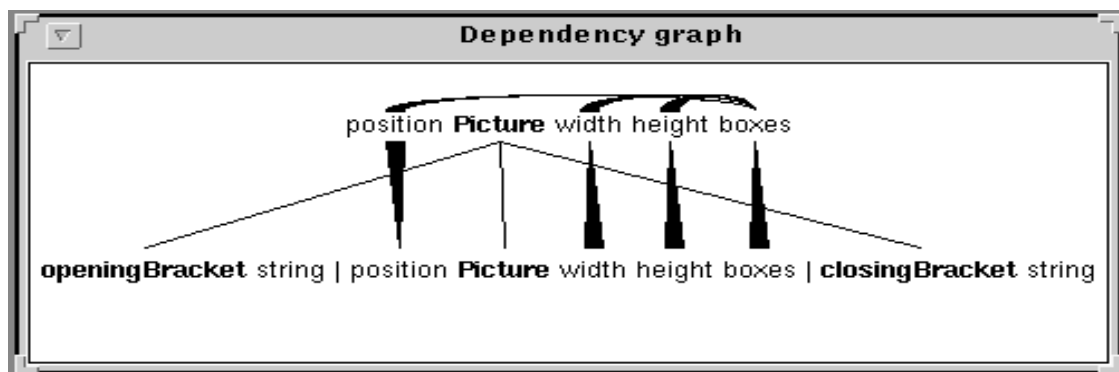
'Save' stores the tree and the attribute evaluation list (plus index describing the point to which the evaluation had progressed) in a file. The user is prompted for a file name.

'Dependencies' opens an information window showing all direct attribute dependencies in the production rule that was applied at the current node (see below).

4.4 Information windows

The Dependency graph window

The 'Dependency graph' window shown below is accessible from the Attribute editor and the Evaluator control windows.



The symbols of the CFG production rule are shown bold face. There are straight lines from the left hand symbol (upper node) of the production rule to the right hand symbols (lower nodes).

To the left of a nonterminal symbol its inherited attributes are listed; to the right its synthesized attributes. Adjacent symbol-and-attributes lists are separated by '|' symbols. To the right of a terminal symbol its pseudo string attribute (value from scanner) is placed

Straight and/or curved arrows describe attribute dependencies. The pointed end of the arrow is near the attribute which depends on the attribute near the blunt end of the arrow.

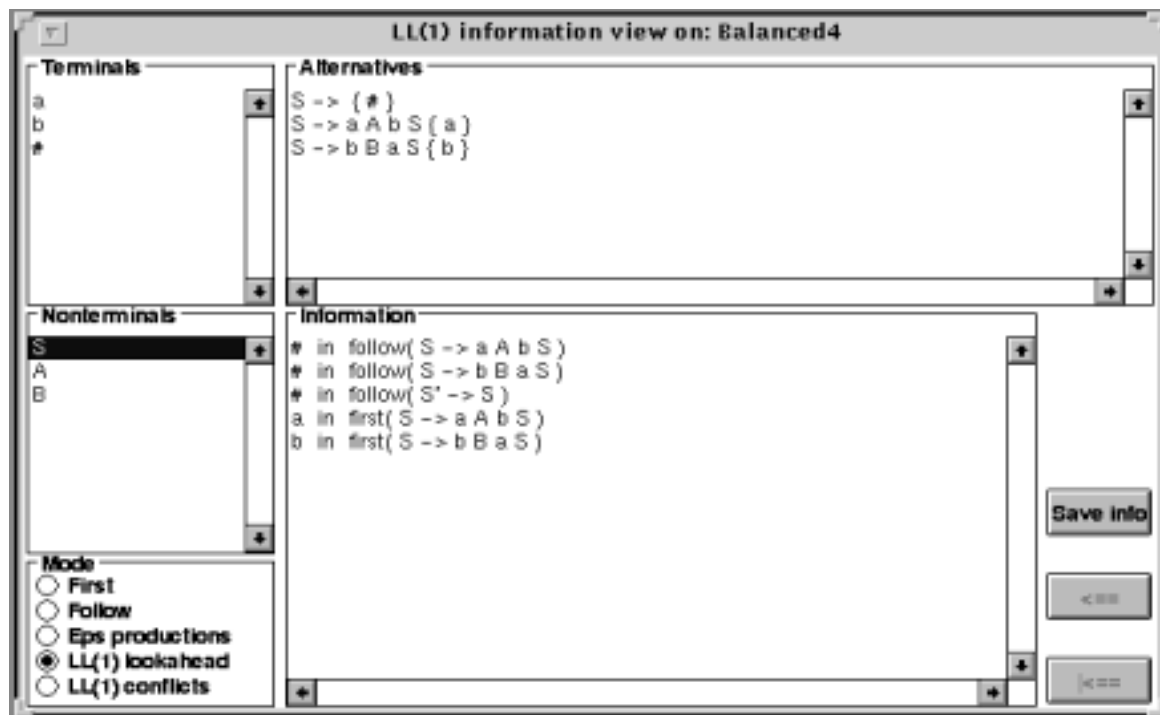
E.g., in the above dependency graph two attributes depend directly on the left hand side's

inherited position attribute: the left hand side's synthesized boxes attribute and inherited position attribute of the right hand side's Picture symbol.

For large dependency graphs a horizontal scroll bar is provided.

The LL(1) information window

The 'LL(1) information view' window shown below is accessible from the Grammar editor and the Parser control windows.



First choose one of the nonterminals from the 'Nonterminals' subwindow. We will refer to it as the 'current nonterminal'. By setting the window's mode (see radio buttons in the bottom left hand corner) you determine what kind of information is to be displayed in the 'Information' subwindow: The first/follow sets of the current nonterminal, its ϵ -production (if any), its LL(1) lookaheads or its LL(1) conflicts.

All the time, in the 'Alternatives' subwindow the alternatives of the current nonterminal are shown plus some information related to the current mode. Above, the LL(1) lookahead sets for the three alternatives of S are shown (in braces).

The 'Information' subwindow not only displays the required information but also gives hints on why terminal symbols belong to a given set. These hints are detailed and traversible (see below) but may be confusing at first. Above the last line indicates the obvious fact that the right hand side of the alternative 'S -> bBaS' starts with a 'b'. The third line indicates that # is in follow(S) because it is in follow(S') (and that is true by definition). The first two lines state that # is in follow(S) because S is the last symbol on the right hand side and because # is in follow(left hand side) (= follow(S)).

For lines of the form 'a in follow(...)' or 'a in first(A->B...)' (i.e. the right hand side starts with a nonterminal B) the computation of 'first' and 'follow' can be traced back by clicking on these lines. With the '<=' button you travel in the opposite direction; with '<|=' right back to

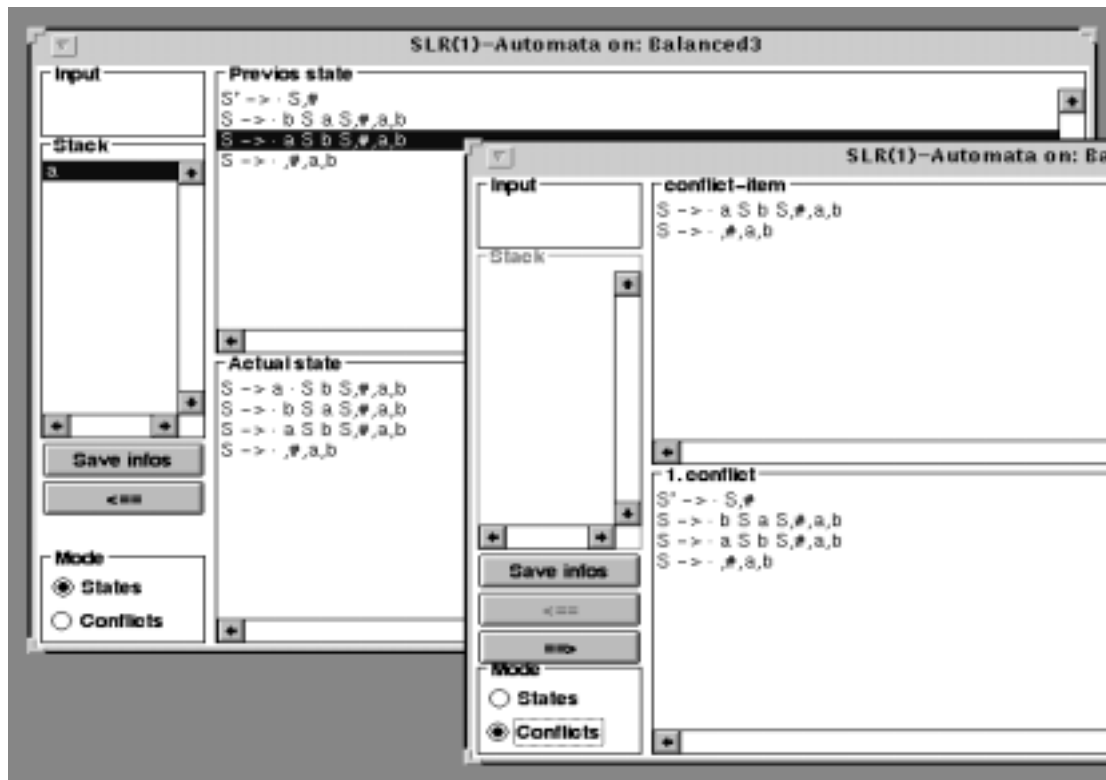
the beginning.

The 'Terminals' subwindow is a filter: By selecting one of the terminals you can reduce the amount of information displayed in the 'Alternatives' and 'Information' subwindows.

'Save info' saves all the LL(1) informations in a file (name provided by user).

The LR information window

The 'LR information view' window two instances of which are shown below is accessible from the Grammar editor and the Parser control windows.



This window is used for viewing the states and conflicts of a variety of LR automata (LR(0), SLR(1), LALR(1), and LR(1)).

In mode 'Conflicts' (see radio buttons in the bottom left hand corner) the list of conflicts (states and conflicting items in separate subwindows) can be traversed using the '<=' and '=>' buttons.

In mode 'States' the states of the automaton can be traversed. Initially, the start state of the automaton is shown under 'Actual state'. An *item* is a CFG production rule with a point separating that (left) part of the right hand side which has been matched successfully with the input from the rest. Since a bottom-up parser can process several CFG rules in parallel, a state is a list of items. If in an item the point is immediately to the left of a terminal symbol 'a', then this item defines a '*shift a*' action ('a' is transferred from input to stack). If the point is at the end of the right hand of the production rule 'r', then this item defines a '*reduce according to r*' action where in the parser stack the symbols of the right hand side of r are replaced by its left hand side.

If in 'Actual state' you click on a 'shift a' item then this shift is performed: The current

state becomes the new 'Previo(u)s state' and its a-successor is the new 'Actual state'. If you click on a 'reduce according to r' item then this reduction is performed: The current state becomes the new 'Previo(u)s state' and the new 'Actual state' is determined according to the LR parser definition. '<=' undoes actions.

'Save info' saves all infomations on the chosen LR automaton in a file (name provided by user).

5. Formats of definitions and data

All files containing SIC data and definitions are tagged with three-character file name extensions which uniquely determine their type. The diagram below shows in which subdirectory of SIC95 you find which file type and what the name extension acronyms stand for.

INPUTS	OUTPUTS	GRAMMARS
.sin (scanner input)	.ll1 (LL(1) information)	.sdf (scanner definition)
.pin (parser input)	.lr0 (LR(0) information)	.pdf (parser definition)
.ein (evaluator input)	.slr (SLR information)	.edf (evaluator definition)
	.llr (LALR information)	
	.lr1 (LR(1) information)	

To the files in GRAMMARS we refer to as 'definition files'. All the others we simply call 'data files'. The files in INPUTS and GRAMMARS are processed by SIC; the files in OUTPUTS are produced by SIC when required by the user (for text editing / documenting results of work). For examples look at section 6.

Data file formats

Scanner inputs are ASCII files (no special format required).

Parser inputs are token sequences. Each token is on a separate line. Optionally, the text of the token may be appended to the token name (in 'funny brackets' « and », corresponding to ASCII codes 171 and 187, respectively).

Evaluator inputs are textual representations of (partially evaluated) syntax trees. They consist of three parts (separated by blank lines):

Tree:
<tree representation>

ListIndex:
<number>

VisitList:
<attribute list>

The <tree representation> of a syntax tree is defined recursively as follows:

- a terminal node is represented by a token name optionally followed by the token text (with funny brackets)
- a nonterminal node is represented by the nonterminal's name, followed by a list (enclosed in parentheses) of the representations of the immediate successors.

The <attribute list> is a list of attribute names (in the same form as can be seen in the attribute evaluator window).

The <number> satisfies $1 \leq \text{<number>} \leq \text{length(<attribute list>)}$ and indicates how far processing of the <attribute list> had proceeded when the tree was saved.

Definition file formats

The formats of definition files are much more complicated than the data file formats. In order to describe these formats we use a mixture of verbal explanations and context-free syntax rules.

Scanner definition:

As shown in the box below, a scanner definition divides into nine *paragraphs*. Paragraphs start with a header line like 'Patterns:' and are terminated by a blank line. Paragraphs may be written in any order.

```
Name:
    <scanner name>

Note:
    <any textual explanation>

Separators:
    Standard      -- consists of TAB, SPACE, CR, LF
    <in extra lines separator symbols can be added in
    the form +(ddd) where ddd is the symbols's ASCII code;
    similarly, -(ddd) removes a separator>

States:
    #Start
    <plus optionally other user-supplied states
    to be used in conditions and actions below>

Strings:
    <string delimiter symbols written like '$' or '$",
    i.e. in Smalltalk character notation>

Verbatims:
    <verbatim delimiter - similar syntax as string delimiters
    but left and right delimiters may be different>

Comments:
    <either Ada style end-of-line-comment starting with given
    string, e.g. defined by '--' LineComment or normal comment
    defined like '/*' Comment '*/'>

Patterns:
    <lines each with one pattern name followed by a pattern>

Tokens:
    <lines each with one token name followed by a pattern name>
```

In the Tokens paragraph the sequence of token definitions is important when definitions overlap: The first token definition has lowest priority, the last one highest. E.g. if a given substring matches both the second and the fifth token definition then this substring will be accepted as an instance of token number five. In a typical programming language we have identifiers and keywords ('reserved identifiers'). In order to give keywords higher precedence than identifiers one would place all the keyword token definitions *after* the identifier token definition.

The *syntax* for patterns (regular expressions) is:

```

<pattern> ::= $<character>
           | <interval>
           | <alternative>
           | <iteration>
<character> ::= < any ASCII character you can type in from the keyboard>
<interval> ::= { $<character> - $<character> }
<alternative> ::= ( <pattern> | <pattern> | ... )
<iteration> ::= <pattern> [ <number> - <number> ]
              | <pattern> [ <number> - * ] | <pattern> [ <number> ]
<number> ::= <nonnegative integer>

```

Above, some symbols are typed bold face to indicate that they appear ‘as is’ in the patterns. This also applies to hyphens and to the ‘|’ symbols within an alternative.

String, comment, verbatim and token definitions may (for each item) optionally contain condition and action definitions of the form:

```

start: <Smalltalk block that yields a Boolean>
end: <Smalltalk block that yields a Boolean>
action: <Smalltalk block>

```

Smalltalk blocks are introduced in the appendix.

A *start condition* is evaluated when token recognition starts. Only those tokens whose start condition evaluates to ‘true’ (or that do not have a start condition at all) are *active* in the beginning. If at the end of a token the *end condition* evaluates to ‘false’ then this token will not be accepted in this place. On accepting a token its associated *action block* (if present) is evaluated. All the Smalltalk code in the conditions and actions may access and update a dictionary ‘internals’ which at least contains entries for the following (symbol) keys:

```

#line           -- current line number
#column        -- current column number
#string        -- current token text
#state         -- current state

```

Of these, the last one is controlled by the user (who writes condition and action code). The other three are updated automatically by the scanner but can be ‘overruled’ in user code. Users may add further entries in the ‘internals’ dictionary.

Parser definition:

A parser definition (i.e. a context-free grammar description) divides into six *paragraphs* that are written in the order shown below.

A right hand side of a rule is any sequence of terminal and/or nonterminal symbols. This sequence may be empty (for an ϵ -production).

For a parser and a scanner to cooperate smoothly, the parser’s terminal symbols have to match exactly the token names defined by the scanner.

For a parser and an evaluator to cooperate smoothly, the underlying context-free grammars must be the same.

```
Name:
    <parser name>

Note:
    <any textual explanation>

Terminals:
    <first terminal symbol>
    <second terminal symbol>
    ...

Nonterminals:
    <first nonterminal symbol>
    <second nonterminal symbol>
    ...

Startsymbol:
    <one of the nonterminal symbols>

Rules:
    <nonterminal 1> -> <first right hand side for nonterminal 1>
                       | <second right hand side for nonterminal 1>
                       | ...

    <nonterminal 2> -> <first right hand side for nonterminal 2>
                       | <second right hand side for nonterminal 2>
                       | ...

    ...
```

Evaluator definition:

The format of an evaluator definition as shown below naturally resembles that of a parser

```

Nonterminals:
  <first nonterminal symbol> <its attributes>
  <second nonterminal symbol> <its attributes>
  ...

Tests:
  <'true' if attribute grammar is absolutely cycle-free>
  <'true' if attribute grammar is cycle-free>

Rules:
  <nonterminal 1> -> <first right hand side for nonterminal 1>
                    <attribute evaluation section>
                    | <second right hand side for nonterminal 1>
                    | <attribute evaluation section>
                    | ...
  ...

```

definition.

```

Name:
  <attribute grammar name>

Note:
  <any textual explanation>

Attributes:
  <first attribute>
  <second attribute>
  ...

Terminals:
  <first terminal symbol>
  <second terminal symbol>
  ...

```

(continued on next page)

In the Attributes paragraph the names of synthesized and inherited attributes are prefixed with » and «, respectively.

Within the Rules paragraph different occurrences of one (non)terminal symbol in one rule are distinguished by indices. Following each rule there is an attribute section which for each synthesized attribute of the left hand side nonterminal and for each inherited attribute of the symbols on the right hand side contains an attribute dependency line, an attribute evaluation rule, and a Boolean stating the validity of the evaluation rule.

The attribute dependency line first names an attribute and then all the attributes it depends on immediately (i.e. that are used in its evaluation rule).

The attribute evaluation rule is given as a Smalltalk block. It is up to the user to guarantee that the attribute to be evaluated will be assigned a (correct) value by the Smalltalk code.

The validity Boolean is 'true' if the evaluation rule only uses attributes listed in the

attribute dependency line.

6. Example applications

In this section, we present the definition, the input and the result files for three groups of examples. The “complete applications” are moderate size examples that use all the features of SIC: the scanner (generator), the parser (generator), and the attribute evaluator (generator). There are two CFGs with one scanner definition, one parser definition, and two attribute evaluator definitions each. Two of the attribute sets define typical translations of a source into a target language; the other two define interpreters and make use of the interaction and graphical features of Smalltalk.

The second group of examples demonstrates aspects of syntax. There are four different grammars all describing the same formal language. With respect to parsing, these grammars fall into different categories. Also, there are some tiny grammars showing that the hierarchy of SLR(1), LALR(1), and LR(1) grammar classes is proper.

The third group consists of just one example which demonstrates some of the possibilities of the SIC scanner generator which are not used in the other examples but might prove useful in some applications.

All the examples are included in the SIC distribution files. Apart from a few introductory remarks we just present some relevant file contents.

6.1 Complete applications

“Arith” is the name of the scanner and the parser discussed in section 2. First, the scanner:

Arith.sdf (first part)

```
Name:
      Arith

Note:
      This is part of the compiler examples. Assemble
      compiler from files with same name. Look for input files.

Separators:
      Standard

States:
      #Start

Strings:
      $"

Comments:
      '--' LineComment
      '/*' Comment '*/'

Verbatims:
      $< $>
      $! $!
```

Arith.sdf (second part)

Patterns:

```

addOpPattern $+|$-
multOpPattern $*|$/
openingBracketPattern $(
closingBracketPattern $)
letterPattern {$a-$z}|{$A-$Z}
digitPattern {$0-$9}
namePattern
    letterPattern(letterPattern|digitPattern)[0-*]
numberPattern digitPattern[1-*]

```

Tokens:

```

addOp addOpPattern
multOp multOpPattern
openingBracket openingBracketPattern
closingBracket closingBracketPattern
name namePattern
number numberPattern

```

And now the parser definition:

Arith.pdf (first part)

Name:

Arith

Note:

This is part of the compiler examples. Assemble compiler from files with same name. Look for input files.

Terminals:

```

addOp
multOp
openingBracket
closingBracket
number
name

```

Nonterminals:

```

Arithmetic
Expression
Factor
Term

```

Startsymbol:

Arithmetic

Arith.pdf (second part)

Rules:

```
Factor -> openingBracket Expression closingBracket
        | number
        | name

Term -> Factor
      | Term multOp Factor

Expression -> Term
           | Expression addOp Term

Arithmetic -> Expression
```

The purpose of first set of attributes and attribute evaluation rules was already indicated in

section 2. Some details will be explained below.

ArithEval.edf (first part)

Name:

ArithEval

Note:

This is part of the compiler examples. Assemble compiler from files with same name. Look for input files.

Attributes:

variables
environment
value

Terminals:

addOp
multOp
openingBracket
closingBracket
number
name

Nonterminals:

Arithmetic value
Expression variables environment value
Factor variables environment value
Term variables environment value

Tests:

true
true

ArithEval.edf (second part)

```

Rules:
Arithmetic -> Expression
valueArithmetic valueExpression
[valueArithmetic := valueExpression]!
true
environmentExpression variablesExpression
[environmentExpression := Dictionary new .
variablesExpression do:
  [ :v | environmentExpression at: v put:
    (Dialog request: 'Value of ' , v , ' ?'
    initialAnswer: '0'
    for: nil) asNumber ]]!
true

Expression -> Term
variablesExpression variablesTerm
[variablesExpression := variablesTerm]!
true
valueExpression valueTerm
[valueExpression := valueTerm]!
true
environmentTerm environmentExpression
[environmentTerm := environmentExpression]!
true

Expression -> Expression addOp Term
variablesExpression1 variablesExpression2 variablesTerm
[variablesExpression1 := Set new .
variablesExpression1 addAll: variablesExpression2 .
variablesExpression1 addAll: variablesTerm]!
true
valueExpression1 valueExpression2 stringaddOp valueTerm
[valueExpression1 :=
(stringaddOp = '+'
  ifTrue: [valueExpression2 + valueTerm]
  ifFalse: [valueExpression2 - valueTerm])]!
true
environmentExpression2 environmentExpression1
[environmentExpression2 := environmentExpression1]!
true
environmentTerm environmentExpression1
[environmentTerm := environmentExpression1]!
true

Factor -> openingBracket Expression closingBracket
variablesFactor variablesExpression
[variablesFactor := variablesExpression]!
true

```


ArithEval.edf (last part)

```

valueFactor valueExpression
[valueFactor := valueExpression]!
true
environmentExpression environmentFactor
[environmentExpression := environmentFactor]!
true

  | number
variablesFactor
[variablesFactor := Set new]!
true
valueFactor stringnumber
[valueFactor := stringnumber asNumber]!
true

  | name
variablesFactor stringname
[variablesFactor := Set with: stringname]!
true
valueFactor environmentFactor stringname
[valueFactor := environmentFactor at: stringname]!
true

Term -> Factor
variablesTerm variablesFactor
[variablesTerm := variablesFactor]!
true
valueTerm valueFactor
[valueTerm := valueFactor]!
true
environmentFactor environmentTerm
[environmentFactor := environmentTerm]!
true

Term -> Term multOp Factor
variablesTerm1 variablesTerm2 variablesFactor
[variablesTerm1 := Set new .
variablesTerm1 addAll: variablesTerm2 .
variablesTerm1 addAll: variablesFactor ]!
true
valueTerm1 valueTerm2 stringmultOp valueFactor
[valueTerm1 :=
(stringmultOp = '*'
  ifTrue: [valueTerm2 * valueFactor]
  ifFalse: [valueTerm2 / valueFactor])]!
true
environmentTerm2 environmentTerm1
[environmentTerm2 := environmentTerm1]!
true

```

The rules' part starts with:

```

Arithmetic -> Expression
valueArithmetic valueExpression
[valueArithmetic := valueExpression]!
true
environmentExpression variablesExpression
[environmentExpression := Dictionary new .
 variablesExpression do:
   [ :v | environmentExpression at: v put:
     (Dialog request: 'Value of ' , v , ' ?'
      initialAnswer: '0'
      for: nil) asNumber ]]!
true

```

In the context of the production rule `Arithmetic -> Expression` (first line) two attributes are evaluated: the synthesised 'value' attribute of `Arithmetic` and the inherited 'environment' attribute of `Expression`. Line 2 says that 'valueArithmetic' (the 'value' attribute of `Arithmetic`) depends on the attribute 'valueExpression' which is computed in some other context. Line 3 shows the (trivial) attribute evaluation rule in the form of a parameterless Smalltalk block (see appendix for an explanation).

Line 5 states that 'environmentExpression' depends on the attribute 'variablesExpression'. The corresponding attribute evaluation rule is contained in the block in lines 6 - 11. The environment of `Expression` is a dictionary containing a value for every variable occurring in the expression. This environment is constructed by enumerating all the variables (contained in 'variablesExpression'), each time prompting the user for the corresponding value ('Dialog request: ...'), converting the user-supplied string into an integer ('asNumber'), and then inserting the variable-value pair into an (initially empty) dictionary.

Lines 4 and 12 contain organizational information regarding attribute dependencies.

In the context of the production rule `Term -> Term multOp Factor` (among others) the synthesised 'value' and 'variables' attributes of the left hand `Term` are evaluated. In the first attribute evaluation rule below the 'value' attribute is computed by multiplication or division depending on the contents of the pseudo 'string' attribute of the terminal symbol 'multOp'. In the second attribute evaluation rule the union of 'variablesTerm2' and 'variablesFactor' is computed by adding them to initially empty set 'variablesTerm1' (for an explanation of Smalltalk control and data structures we again refer to the appendix):

```

valueTerm1 :=
(stringmultOp = '*'
 ifTrue: [valueTerm2 * valueFactor]
 ifFalse: [valueTerm2 / valueFactor])

variablesTerm1 := Set new .
variablesTerm1 addAll: variablesTerm2 .
variablesTerm1 addAll: variablesFactor

```

Based on the same CFG "Arith" there is another attribute grammar, "Arith2Mi", which translates expressions to machine code. E.g. the expression

$$(\alpha + 17) * (\beta - \alpha / 17)$$

is translated to

```

beta:      DD 0
alpha:     DD 0
           MOVE W alpha , -!SP
           MOVE W I 17 , -!SP
           ADD W !SP+, !SP+, -!SP
           MOVE W beta , -!SP
           MOVE W alpha , -!SP
           MOVE W I 17 , -!SP
           DIV W !SP+, !SP+, -!SP
           SUB W !SP+, !SP+, -!SP
           MULT W !SP+, !SP+, -!SP

```

Notice that for each variable occurring in the expression exactly one data definition (DD) line is created. Number constants (17) translate into 'immediate operands' (I 17). The expression is evaluated using the machine stack: Variables and numbers are pushed onto the stack with 'MOVE W <variable/number>, -!SP' commands. Arithmetic operations ('ADD', 'DIV', 'SUB', 'MULT') each pop two arguments off the stack ('!SP+') and push the result onto the stack ('-!SP').

In order to avoid duplicate 'DD' lines, all variables are collected in 'variables' attributes as above. In the synthesized 'code' attributes the target code is created and propagated towards the root of the syntax tree.

This example demonstrates Smalltalk string handling.

Arith2Mi.edf (first part)

```

Name:
      Arith2Mi

Note:
      This is part of the compiler examples. Assemble
      compiler from files with same name. Look for input files.

Attributes:
      variables
      code

Terminals:
      addOp
      multOp
      openingBracket
      closingBracket
      number
      name

Nonterminals:
      Arithmetic code
      Expression variables code
      Factor variables code
      Term variables code

Tests:
      true
      true

```


Arith2Mi.edf (second part)

```

Rules:
Arithmetic -> Expression
codeArithmetic variablesExpression codeExpression
[| h |
h := '' .
variablesExpression do:
  [ :v | h := ((v , ':DD 0
  ') , h) ] .
codeArithmetic := h , codeExpression)!
true

Expression -> Term
variablesExpression variablesTerm
[variablesExpression := variablesTerm]!
true
codeExpression codeTerm
[codeExpression := codeTerm]!
true

Expression -> Expression addOp Term
variablesExpression1 variablesExpression2 variablesTerm
[variablesExpression1 := Set new .
variablesExpression1 addAll: variablesExpression2 .
variablesExpression1 addAll: variablesTerm]!
true
codeExpression1 codeExpression2 stringaddOp codeTerm
[codeExpression1 :=
(stringaddOp = '+'
ifTrue: [codeExpression2 , codeTerm , 'ADD W !SP+, !SP+, -!SP
']
ifFalse: [codeExpression2 , codeTerm , 'SUB W !SP+, !SP+, -!SP
'])]!
true

Factor -> openingBracket Expression closingBracket
variablesFactor variablesExpression
[variablesFactor := variablesExpression]!
true
codeFactor codeExpression
[codeFactor := codeExpression]!
true
  | number
variablesFactor
[variablesFactor := Set new]!
true
codeFactor stringnumber
[codeFactor := '          MOVE W I ' , stringnumber , ' , -!SP
']!
true

```

Arith2Mi.edf (last part)

```

    | name
variablesFactor stringname
[variablesFactor := Set with: stringname]!
true
codeFactor stringname
[codeFactor := '          MOVE W ' , stringname , ' , -!SP
' ]!
true

Term -> Factor
variablesTerm variablesFactor
[variablesTerm := variablesFactor]!
true
codeTerm codeFactor
[codeTerm := codeFactor]!
true

Term -> Term multOp Factor
variablesTerm1 variablesTerm2 variablesFactor
[variablesTerm1 := Set new .
variablesTerm1 addAll: variablesTerm2 .
variablesTerm1 addAll: variablesFactor ]!
true
codeTerm1 codeTerm2 stringmultOp codeFactor
[codeTerm1 :=
(stringmultOp = '*'
ifTrue: [codeTerm2 , codeFactor , ' MULT W !SP+ , !SP+ , -!SP
' ]
ifFalse: [codeTerm2 , codeFactor , ' DIV W !SP+ , !SP+ , -!SP
' ])]!
true

```

In the “Arith” examples above arithmetic expressions were analysed lexically and syntactically and then either evaluated or translated to machine code (MI assembly language).

In the other two examples a “**Boxes**” input language defines pictures made up of boxes whose sides are parallel to the geometrical x- and y-axes. Boxes may be nested or placed side to side (either horizontally ‘>’ or vertically ‘v’). Like in the “Arith” examples there are two attribute grammars based on “Boxes”. The first interprets the input and draws a corresponding picture into a graph window (using Smalltalk window and graph primitives). The second translates the input string into a postscript program which can be sent to a laser printer to produce the picture.

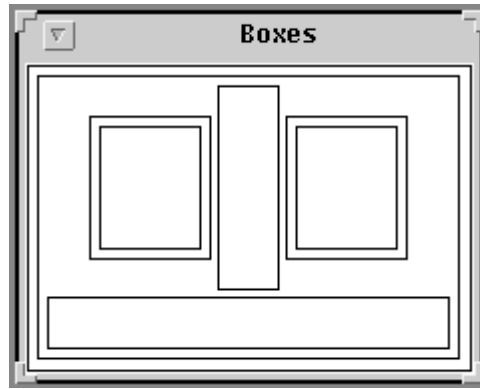
For the input

```

[ ( [ [ 50 60 ] ] > [ 30 100 ] > ( [ [ 50 60 ] ] ) )
v [ 200 25 ] ]

```

evaluation according to the first attribute grammar results in the graph window:



Below, the non-empty parts of the “Boxes” scanner definition are shown.

Boxes.sdf

Name :

Boxes

Note :

This is part of the compiler examples. Assemble compiler from files with same name. Look for input files.

Separators :

Standard

States :

#Start

Patterns :

```
leftBracketPattern $[
rightBracketPattern $]
greaterPattern $>
leftParenthesisPattern $(
rightParenthesisPattern $)
vPattern $v
digitPattern {$0-$9}
numberPattern digitPattern[1-*
```

Tokens :

```
openingBracket leftBracketPattern
closingBracket rightBracketPattern
right greaterPattern
openingParenthesis leftParenthesisPattern
closingParenthesis rightParenthesisPattern
down vPattern
number numberPattern
```

The “Boxes” parser definition is:

Boxes.pdf

Name:

Boxes

Note:

This is part of the compiler examples. Assemble compiler from files with same name. Look for input files.

Terminals:

openingBracket
closingBracket
number
right
down
openingParenthesis
closingParenthesis

Nonterminals:

Picture
Box
Width
Height
Painting

Startsymbol:

Painting

Rules:

```
Picture -> Box
        | Picture right Box
        | Picture down Box
        | openingBracket Picture closingBracket

Box -> openingBracket Width Height closingBracket
     | openingParenthesis Picture closingParenthesis

Height -> number

Painting -> Picture

Width -> number
```

Notice again that the scanner's token definitions exactly match the terminal symbols of the CFG.

BoxesView.edf (first part)

Name:

BoxesView

Note:

This is part of the compiler examples. Assemble compiler from files with same name. Look for input files.

Attributes:

width
height
boxes
position

Terminals:

openingBracket
closingBracket
number
right
down
openingParenthesis
closingParenthesis

Nonterminals:

Picture width height boxes position
Box width height boxes position
Width width
Height height
Painting boxes

Tests:

true
true

Rules:

Picture -> Box
widthPicture widthBox
[widthPicture := widthBox]!
true
heightPicture heightBox
[heightPicture := heightBox]!
true
boxesPicture boxesBox
[boxesPicture := boxesBox]!
true
positionBox positionPicture
[positionBox := positionPicture]!
true

BoxesView.edf (second part)

```

Picture -> Picture right Box
widthPicture1 widthPicture2 widthBox
[widthPicture1 := widthPicture2 + 4 + widthBox]!
true
heightPicture1 heightPicture2 heightBox
[heightPicture1 := heightPicture2 max: heightBox]!
true
boxesPicture1 boxesPicture2 boxesBox
[boxesPicture1 := boxesPicture2 , boxesBox]!
true
positionPicture2 positionPicture1 heightPicture2 heightBox
[| yPos yDiff |
yPos := positionPicture1 y. yDiff := heightBox - heightPicture2.
yDiff > 0 ifTrue: [yPos := yPos + (yDiff // 2)].
positionPicture2 := positionPicture1 x @ yPos]!
true
positionBox positionPicture1 widthPicture2 heightPicture2
heightBox
[| yPos yDiff |
yPos := positionPicture1 y. yDiff := heightPicture2 - heightBox.
yDiff > 0 ifTrue: [yPos := yPos + (yDiff // 2)].
positionBox := positionPicture1 x + 4 + widthPicture2 @ yPos]!
true

| Picture down Box
widthPicture1 widthPicture2 widthBox
[widthPicture1 := widthPicture2 max: widthBox]!
true
heightPicture1 heightPicture2 heightBox
[heightPicture1 := heightPicture2 + 4 + heightBox]!
true
boxesPicture1 boxesPicture2 boxesBox
[boxesPicture1 := boxesPicture2 , boxesBox]!
true
positionPicture2 positionPicture1 widthBox widthPicture2
[| xPos xDiff |
xPos := positionPicture1 x. xDiff := widthBox - widthPicture2.
xDiff > 0 ifTrue: [xPos := xPos + (xDiff // 2)].
positionPicture2 := xPos @ positionPicture1 y]!
true
positionBox positionPicture1 widthPicture2 heightPicture2 widthBox
[| xPos xDiff |
xPos := positionPicture1 x.
xDiff := widthPicture2 - widthBox.
xDiff > 0 ifTrue: [xPos := xPos + (xDiff // 2)].
positionBox := xPos @ (positionPicture1 y + 4 + heightPicture2)]!
true

```


BoxesView.edf (third part)

```

    | openingBracket Picture closingBracket
widthPicture1 widthPicture2
[widthPicture1 := 10 + widthPicture2]!
true
heightPicture1 heightPicture2
[heightPicture1 := 10 + heightPicture2]!
true
boxesPicture1 widthPicture1 heightPicture1 positionPicture1
boxesPicture2
[boxesPicture1 := boxesPicture2 ,
      (Array with: (positionPicture1 extent: widthPicture1 @
heightPicture1)) ]!
true
positionPicture2 positionPicture1
[positionPicture2 := positionPicture1 + (5 @ 5)]!
true

Box -> openingBracket Width Height closingBracket
widthBox widthWidth
[widthBox := widthWidth asNumber]!
true
heightBox heightHeight
[heightBox := heightHeight asNumber]!
true
boxesBox widthBox heightBox positionBox
[boxesBox := Array with: (positionBox extent: widthBox @
heightBox)]!
true

    | openingParenthesis Picture closingParenthesis
widthBox widthPicture
[widthBox := widthPicture]!
true
heightBox heightPicture
[heightBox := heightPicture]!
true
boxesBox boxesPicture
[boxesBox := boxesPicture]!
true
positionPicture positionBox
[positionPicture := positionBox]!
true

Height -> number
heightHeight stringnumber
[heightHeight := stringnumber]!
true

```

BoxesView.edf (last part)

```

Width -> number
widthWidth stringnumber
[widthWidth := stringnumber]!
true

Painting -> Picture
boxesPainting boxesPicture widthPicture heightPicture
[| windowWidth windowHeight window component |
Cursor execute
  showWhile:
    [boxesPainting := boxesPicture.
     windowWidth := widthPicture + 14 min: 640.
     windowHeight := heightPicture + 14 min: 480.
     (window := ScheduledWindow new) component:
       (BorderDecorator on:
        (component := CompositePart new)).
     window minimumSize: windowWidth @ windowHeight.
     window maximumSize: windowWidth @ windowHeight.
     windowWidth = 640 ifTrue:
       [window component useHorizontalScrollBar].
     windowHeight = 480
       ifTrue: [window component useVerticalScrollBar]
       ifFalse: [window component noVerticalScrollBar].
     boxesPicture do: [:box | component add: box asStroker].
     window open.
     window label: 'Boxes']]!

true
positionPicture
[positionPicture := 4 @ 4]!
true

```

The extensions of all components of a picture are determined bottom-up in the synthesized attributes ‘width’ and ‘height’. Using these extensions and starting from left upper corner with x- and y-coordinates both 4 pixels (in Smalltalk such a point is written as 4 @ 4) the coordinates of the left upper corners of all the picture’s components are computed and propagated via inherited ‘position’ attributes. The positions and extensions uniquely determine the boxes contained in the picture. The Smalltalk expression

```
(positionBox extent: widthBox @ heightBox)
```

constructs a box. All the drawing of boxes is done in the line

```
boxesPicture do: [:box | component add: box asStroker].
```

of the biggish attribute evaluation rule shown above. Before that, a window of a suitable size is created; scroll bars are supplied when needed.

For the input shown in the beginning application of the second attribute grammar based on “Boxes” will produce the following Postscript code:

```
/box {
  /bwidth exch def
  /bheight exch def
  0 bwidth rlineto
  bheight 0 rlineto
  0 bwidth neg rlineto
  closepath
} def
newpath
  131 125 moveto 50 60 box
  126 120 moveto 60 70 box
  190 105 moveto 30 100 box
  229 125 moveto 50 60 box
  224 120 moveto 60 70 box
  105 209 moveto 200 25 box
  100 100 moveto 210 139 box
stroke
showpage
```

The attribute grammar is shown below. Instead of collecting Smalltalk representations the corresponding Postscript code is generated and collected in the synthesized 'postscript' attributes.

BoxesPostscript.edf(first part)

Name:

BoxesPostscript

Note:

This is part of the compiler examples. Assemble compiler from files with same name. Look for input files.

Attributes:

width
height
position
postscript

Terminals:

openingBracket
closingBracket
number
right
down
openingParenthesis
closingParenthesis

Nonterminals:

Picture width height position postscript
Box width height position postscript
Width width
Height height
Painting postscript

BoxesPostscript.edf(second part)

Tests:

```

    true
    true

```

Rules:

Picture -> Box

widthPicture widthBox

[widthPicture := widthBox]!

true

heightPicture heightBox

[heightPicture := heightBox]!

true

postscriptPicture postscriptBox

[postscriptPicture := postscriptBox]!

true

positionBox positionPicture

[positionBox := positionPicture]!

true

Picture -> Picture right Box

widthPicture1 widthPicture2 widthBox

[widthPicture1 := widthPicture2 + 4 + widthBox]!

true

heightPicture1 heightPicture2 heightBox

[heightPicture1 := heightPicture2 max: heightBox]!

true

postscriptPicture1 postscriptPicture2 postscriptBox

[postscriptPicture1 := postscriptPicture2 , postscriptBox]!

true

positionPicture2 positionPicture1 heightPicture2 heightBox

[| yPos yDiff |

yPos := positionPicture1 y.

yDiff := heightBox - heightPicture2.

yDiff > 0 ifTrue: [yPos := yPos + (yDiff // 2)].

positionPicture2 := positionPicture1 x @ yPos]!

true

positionBox positionPicture1 widthPicture2 heightPicture2

heightBox

[| yPos yDiff |

yPos := positionPicture1 y.

yDiff := heightPicture2 - heightBox.

yDiff > 0 ifTrue: [yPos := yPos + (yDiff // 2)].

positionBox := positionPicture1 x + 4 + widthPicture2 @ yPos]!

true

BoxesPostscript.edf(third part)

```

    | Picture down Box
widthPicture1 widthPicture2 widthBox
[widthPicture1 := widthPicture2 max: widthBox]!
true
heightPicture1 heightPicture2 heightBox
[heightPicture1 := heightPicture2 + 4 + heightBox]!
true
postscriptPicture1 postscriptPicture2 postscriptBox
[postscriptPicture1 := postscriptPicture2 , postscriptBox]!
true
positionPicture2 positionPicture1 widthBox widthPicture2
[| xPos xDiff |
xPos := positionPicture1 x.
xDiff := widthBox - widthPicture2.
xDiff > 0 ifTrue: [xPos := xPos + (xDiff // 2)].
positionPicture2 := xPos @ positionPicture1 y]!
true
positionBox positionPicture1 widthPicture2 heightPicture2 widthBox
[| xPos xDiff |
xPos := positionPicture1 x.
xDiff := widthPicture2 - widthBox.
xDiff > 0 ifTrue: [xPos := xPos + (xDiff // 2)].
positionBox := xPos @ (positionPicture1 y + 4 + heightPicture2)]!
true

    | openingBracket Picture closingBracket
widthPicture1 widthPicture2
[widthPicture1 := 10 + widthPicture2]!
true
heightPicture1 heightPicture2
[heightPicture1 := 10 + heightPicture2]!
true
postscriptPicture1 widthPicture1 heightPicture1 positionPicture1
postscriptPicture2
[postscriptPicture1 :=
  (postscriptPicture2 ,
   ' ' ,
   positionPicture1 x printString , ' ' ,
   positionPicture1 y printString , ' moveto ' ,
   widthPicture1 printString , ' ' ,
   heightPicture1 printString , ' box
  ' )]!
true
positionPicture2 positionPicture1
[positionPicture2 := positionPicture1 + (5 @ 5)]!
true

```


BoxesPostscript.edf(fourth part)

```

Box -> openingBracket Width Height closingBracket
widthBox widthWidth
[widthBox := widthWidth asNumber]!
true
heightBox heightHeight
[heightBox := heightHeight asNumber]!
true
postscriptBox widthBox heightBox positionBox
[postscriptBox :=
  ('' ,
   positionBox x printString , ' ' ,
   positionBox y printString , ' moveto ' ,
   widthBox printString , ' ' ,
   heightBox printString , ' box
  ' )]!
true

| openingParenthesis Picture closingParenthesis
widthBox widthPicture
[widthBox := widthPicture]!
true
heightBox heightPicture
[heightBox := heightPicture]!
true
postscriptBox postscriptPicture
[postscriptBox := postscriptPicture]!
true
positionPicture positionBox
[positionPicture := positionBox]!
true

Height -> number
heightHeight stringnumber
[heightHeight := stringnumber]!
true

Width -> number
widthWidth stringnumber
[widthWidth := stringnumber]!
true

Painting -> Picture
positionPicture
[positionPicture := 100 @ 100]!
true

```

BoxesPostscript.edf(last part)

```

postscriptPainting postscriptPicture
[postscriptPainting :=
'/box {
  /bwidth exch def
  /bheight exch def
  0 bwidth rlineto
  bheight 0 rlineto
  0 bwidth neg rlineto
  closepath
} def
newpath
' ,
postscriptPicture ,
'stroke
showpage
']!
true

```

6.2 Demonstrating the parser

In this section, six small grammars are shown:

Balanced1, Balanced2, Balanced3, and Balanced4

all describing the same language of balanced strings over the alphabet {a,b} where each word contains an equal number of a's and b's;

Balanced1 and Balanced3 are not LR(1). The other two are not LR(0) but SLR(1) and LL(1)

notSLR

the grammar is not SLR(1) but LALR(1)

notLALR

the grammar is not LALR(1) but LR(1)

For these grammars, the SIC parser definition files are shown plus additionally SIC-generated LL(1) information files and / or LR automata / conflict states justifying the above claims.

Balanced1.pdf

Name:
Balanced1

Note:

Terminals:
a
b

Nonterminals:
S
A
B

Startsymbol:
S

Rules:

```

B -> b
   | a B B
   | b S

S -> a B
   | b A
   |

A -> a
   | b A A
   | a S

```

This grammar is obviously not LL(1) because for A there are two alternatives starting with a. There are more LL(1) conflicts as shown by the following LL(1) lookahead sets.

Lookahead sets:

```

S ->      { # a b }
S -> b A   { b }
S -> a B   { a }

A -> a S   { a }
A -> a     { a }
A -> b A A { b }

B -> a B B { a }
B -> b S   { b }
B -> b     { b }

```

SIC reports four LR(1) conflicts. The first one is:

```
1. conflict:
   conflict-items for: a

   B -> b . ,a,b
   S -> . ,a,b
   S -> . a B,a,b

   Conflict-state: I16

   B -> b . S,a,b
   B -> b . ,a,b
   S -> . b A,a,b
   S -> . ,a,b
   S -> . a B,a,b
```

Balanced2.pdf

```
Name:
    Balanced2

Note:

Terminals:
    a
    b

Nonterminals:
    S
    A
    B

Startsymbol:
    S

Rules:
    B -> b
      | a B B

    S -> a B S
      | b A S
      |

    A -> a
      | b A A
```

LL(1) Lookahead sets (no conflicts):

```
S -> { # }
```

```
S -> a B S   { a }
S -> b A S   { b }
A -> b A A   { b }
A -> a       { a }

B -> b       { b }
B -> a B B   { a }
```

SIC reports six LR(0) conflicts. The last one is:

```
6. conflict:
  conflict-items for: b

  S -> .
  S -> . b A S

  Conflict-state: I14

  S -> b A . S
  S -> .
  S -> . b A S
  S -> . a B S
```

The SLR(1) automaton below is conflict-free.

Balanced2.slr (first part)

```
I0:  S' -> . S,#
      S -> . ,#
      S -> . b A S,#
      S -> . a B S,#

      S -> I4
      b -> I7
      a -> I10

I1:  S -> a B . S,#
      S -> . ,#
      S -> . b A S,#
      S -> . a B S,#

      S -> I5
      b -> I7
      a -> I10

I2:  B -> a B . B,#,a,b
      B -> . a B B,#,a,b
      B -> . b,#,a,b

      B -> I3
      a -> I12
      b -> I9

I3:  B -> a B B . ,#,a,b

I4:  S' -> S . ,#

I5:  S -> a B S . ,#

I6:  S -> b A S . ,#
```

Balanced2.slr (last part)

```

I7:  S -> b . A S,#
      A -> . b A A,#,a,b
      A -> . a,#,a,b

      a -> I11
      b -> I8
      A -> I13

I8:  A -> b . A A,#,a,b
      A -> . b A A,#,a,b
      A -> . a,#,a,b

      a -> I11
      b -> I8
      A -> I14

I9:  B -> b . ,#,a,b

I10: S -> a . B S,#
      B -> . a B B,#,a,b
      B -> . b,#,a,b

      B -> I1
      a -> I12
      b -> I9

I11: A -> a . ,#,a,b

I12: B -> a . B B,#,a,b
      B -> . a B B,#,a,b
      B -> . b,#,a,b

      B -> I2
      a -> I12
      b -> I9

I13: S -> b A . S,#
      S -> . ,#
      S -> . b A S,#
      S -> . a B S,#

      S -> I6
      b -> I7
      a -> I10

I14: A -> b A . A,#,a,b
      A -> . b A A,#,a,b
      A -> . a,#,a,b

      a -> I11
      b -> I8
      A -> I15

I15: A -> b A A . ,#,a,b

```

The following two grammars are rather different from the above but have the same formal

properties as `Balanced1` and `Balanced2`, respectively.

Balanced3.pdf

Name :

Balanced3

Terminals :

a
b

Nonterminals :

S

Startsymbol :

S

Rules :

```
S ->
  | a S b S
  | b S a S
```

Balanced4.pdf

Name:

Balanced4

Terminals:

a

b

Nonterminals:

S

A

B

Startsymbol:

S

Rules:

B ->
| b B a B

S ->
| a A b S
| b B a S

A ->
| a A b A

The following grammar is not SLR(1) but LALR(1):

notSLR.pdf

```

Name:
    notSLR
Terminals:
    a
    b
    c
    d
Nonterminals:
    S
    A
    B
    F
Startsymbol:
    S
Rules:
    B -> b
    S -> F
        | B d
        | c A d
        | d F
    F -> A c
    A -> a
        | B
  
```

In the SLR(1) automaton there is just one conflict:

```

conflict-items for: d
S -> B . d,#
A -> B . ,c,d
Conflict-state: I11
S -> B . d,#
A -> B . ,c,d
  
```

The corresponding LALR(1) state has no conflict:

```

I10:  S -> B . d,#
      A -> B . ,c
  
```

The following grammar is not LALR(1) but LR(1):

notLALR.pdf

```

Name:
    notLALR
Terminals:
    a
    b
    c
    d
    e
Nonterminals:
    S
    A
    B
Startsymbol:
    S
Rules:
    B -> c
    S -> a A d
      | b B d
      | a B e
      | b A e
    A -> c
  
```

In the LALR(1) automaton there is just one conflict state:

```

conflict-items for: d and e
A -> c . ,d,e
B -> c . ,d,e
Conflict-state: I7
A -> c . ,d,e
B -> c . ,d,e
  
```

The corresponding LR(1) states have no conflicts:

```

I6:   A -> c . ,d
      B -> c . ,e

I7:   A -> c . ,e
      B -> c . ,d
  
```

6.3 Demonstrating the scanner

The following example demonstrates some of the SIC scanners' features that are useful for processing formatted input, e.g. assembly code lines or fill-in-forms. The scanner shown below is to process fill-in-forms for money transfer orders.

Technically, this scanner has internal **states**, #Normal and #FamilyNameRecognized. The scanner's behaviour depends on the current state. Tokens are recognized only when at the beginning (in the moment when the token is about to be processed) a '**startcondition**' (is provided in the token definition in the form of a Smalltalk block - see appendix on Smalltalk) holds *and* in the end (right after reading the last character of the token) an '**endcondition**' holds. On successfully recognizing a token its '**action**' block (if present) is evaluated.

transfer.sdf (first part)

Name:

Transfer

Note:

This is part of the scanner examples. Usage of start and end conditions and actions is demonstrated. Look for input files.

Separators:

Standard

States:

#Normal

#FamilyNameRecognized

Strings:

Comments:

Verbatims:

Patterns:

```
digitpattern {$0-$9}
smallletterpattern {$a-$z}
capitalletterpattern {$A-$Z}
symbolpattern {(0)-(255)}
namepattern capitalletterpattern smallletterpattern[0-*]
accountpattern digitpattern[1-*]
zipcodepattern digitpattern[8-8]
wordpattern symbolpattern[1-*]
dmpattern digitpattern[1-*]
pfennigpattern digitpattern[2-2]
```

transfer.sdf (last part)

Tokens:

```

FamilyName namepattern
  start:[((internals at: #line) = 5)
        & ((internals at: #column) >= 6)
        & ((internals at: #state) = #Normal)]
  end:[(internals at: #column) <= 59]
  action: [internals at: #state put: #FamilyNameRecognized.]

ChristianName namepattern
  start:[((internals at: #line) = 5)
        & ((internals at: #column) >= 6)
        & ((internals at: #state) = #FamilyNameRecognized)]
  end:[(internals at: #column) <= 59]

AccountNo accountpattern
  start:[((internals at: #line) = 7)
        & ((internals at: #column) >= 6)]
  end:[(internals at: #column) <= 26]
  action: [internals at: #state put: #Normal.]

ZipCode zipcodepattern
  start:[((internals at: #line) = 7)
        & ((internals at: #column) >= 44)]
  end:[(internals at: #column) <= 59]

DM dmpattern
  start:[((internals at: #line) = 11)
        & ((internals at: #column) >= 36)]
  end:[(internals at: #column) <= 55]

Pfennig pfennigpattern
  start:[((internals at: #line) = 11)
        & ((internals at: #column) >= 57)]
  end:[(internals at: #column) <= 59]

word wordpattern
  start:[(#(9,13,15) includes: (internals at: #line))
        & ((internals at: #column) >= 6)]
  end:[(internals at: #column) <= 59]

```

The action mechanism is extremely powerful. In principle, not only Chomsky-3 languages can be processed but also any other formal language: Smalltalk like most programming languages is Turing-complete.

This, however, was not the reason for including actions, start and end conditions. We just wanted to provide a practical, easy-to-use tool.

Below, a possible input and the resulting token sequence are shown.

transfer.sin

Schulter Frank

0003307492

79090000

Volksbank-Raiffeisenbank Wuerzburg und Umgebung

10000 00

Please use this
for voluntary donations

transfer.pin

FamilyName<<Schulter>>
ChristianName<<Frank>>
AccountNo<<0003307492>>
ZipCode<<79090000>>
word<<Volksbank-Raiffeisenbank>>
word<<Wuerzburg>>
word<<und>>
word<<Umgebung>>
DM<<10000>>
Pfennig<<00>>
word<<Please>>
word<<use>>
word<<this>>
word<<for>>
word<<voluntary>>
word<<donations>>

7. On the development of SIC

We briefly describe the SIC development history, list the extensions that are planned for the (near?) future, comment on the benefits object-oriented implementation using Smalltalk, and list related efforts and references.

Milestones of the development

Although we have developed compiler-compilers since the late seventies (first in PL/1 and then in Ada) SIC was not a planned effort in the beginning. Rather, in order to compare Ada and Smalltalk two students were asked to do competitive implementations of a grammar browsing tool. While the Ada program never worked, the Smalltalk prototype was even more powerful than required. So we went on with what became the SIC project. The version of SIC which won the German Academic Software Prize in 1991 was completed within less than two and a half man years.

We then added new features that we and the students who used the system missed most: A true scanner generator based on regular expressions, a more efficient implementation of parsing based on action tables (as an alternative), more elaborate attribute evaluation mechanisms (which were too hard to use and unstable and were therefore not distributed) and trees that can grow to virtually any size.

Since SIC was not planned to grow the way it did it became apparent after some time that major reorganization efforts were required before we could go on with the development. Also in the last years the Smalltalk systems were integrated into the surrounding windowing system (Windows 3.x for Digitalk's Smalltalk/V and X-windows for ParcPlace Smalltalk 80 which was renamed to become VisualWorks). Since our MS-DOS based Smalltalk/V system became obsolete, we first started to port it to the Windows 3.x based version of Smalltalk/V. Also, we were asked many times for a Unix version of SIC. Unfortunately, Digitalk did not 'go Unix'. So we resolved to change our platform and turned to VisualWorks.

The **implementation** was done entirely by students:

Oct 1989 GrammarBrowser (Programming assignment, J. Kröger)

Sept 1990 Interactive Parser Generator (Diploma thesis, J. Kröger)

Oct 1990 Interactive Attribute Evaluator (Diploma theses, D. Mörs, M. Schmidt)

Sept 1992 Standalone Components for SIC (Programming assignment, V. Niespor, E. Zschau)

Oct 1992 Attribute Evaluation and Tree Transformations (Diploma theses, O. Dörre, F. Gräfe)

Oct 1993 Reverse Engineering applied to SIC (Diploma theses, V. Niespor, E. Zschau)

Nov 1993 Tree Reimplementation (Programming assignment, D. Kassebaum, O. Laduch)

Dec 1993 Scanner Generator (Programming assignment, M. Worch, S. Zimmermann)

July 1994 Attribute Tree Editor (Programming assignment, F. Schulter, J. van Laak)

July 1995 Reimplementation of SIC (Diploma theses, F. Schulter, J. van Laak)

The Interactive Parser Generator was presented in Jan 1991 at the **STACS'91** conference in Hamburg.

After that the name SIC (Smalltalk-based Interactive Compiler-Compiler) was introduced for the collection of tools created so far. In Sept 1991 SIC was awarded the **1991 German Academic Software Prize** for educational computer science software.

At **OOPSLA'92** an extended version of SIC was presented.

SIC'95 is a much improved version of the SIC system. F. Schulter and J. van Laak have completely reorganized the user interface and the program code, and ported the system from Smalltalk/V to **VisualWorks**.

The different versions of SIC were employed successfully in several courses and practical on compiler construction techniques both as a demonstration tool and for implementing small projects. Several SIC-related papers have been published (including [7],[8],[9],[10], most of them in German).

Future plans

SIC'95 lacks some features that earlier versions did have. These will be the first things to be supplemented:

- Input and output of derived informations such as parsing automata, compiled attribute evaluation rules etc.
- Resizable windows. At present, SIC windows are fixed size (mainly for aesthetic reasons).
- A bird's eye view on large trees.
- Sorting capabilities for the GrammarEditors.

Some developments that have already been started but not yet completed, in particular:

- Advanced attribute evaluation mechanisms.

Also, we would like to make SIC useful as a production system for small applications by:

- Adding a compiled (C++, Ada, Modula, ... ?) attribute evaluation language;
- Generating 'standalone' compilers using the attribute evaluation language as a target language.

Remarks on the implementation

We feel that without using an object-oriented system like Smalltalk [3], it would have been difficult to achieve the functionality of SIC with the available manpower. The main benefits of the object-oriented approach (and the Smalltalk system we used) for the SIC project were: A natural way to model the problem domain; a large library of interface elements and data structures; powerful mechanisms for adapting reusable code to our needs.

First of all, we were surprised to see that even a notoriously hard piece of theoretical computer science software like a compiler-compiler can be *modeled very naturally* by hierarchies of classes and objects. E.g. a parsing automaton object contains a set of state objects which in turn are lists of item objects etc. Different kinds of parsing automata such as LALR automata are implemented by different subclasses of one abstract parsing automata class which contains all the common code. Grammars and attribute grammars, different kinds of parsers, scanners, derivations, rules, alternatives etc. also fit very well into the class

hierarchy.

Secondly, using the existing *class library* reasonably mature prototypes rich in functionality can be developed rapidly. This is obvious for the pluggable user interface building blocks. The ergonomic goals outlined in section 1 could not have been achieved without such support. For SIC, the collection classes were most valuable: they provided the high level of abstraction which made it comparatively easy to describe the computation of mathematical objects such as first-/follow-sets or parsing automata. Instead of manipulating pointers, one can concentrate on operations on sets or sequences (called *OrderedCollections*) as required by the problem. Efficient implementations of these data types and an abundance of operations on them are provided by the system. In fact, using the collection classes can be much more efficient than using ad hoc representations such as linked lists or arrays which are often used to implement these data types.

Finally, *code reuse* and the *adaptations* this requires are greatly facilitated by the inheritance and redefinition mechanisms. However, these benefits do not come about without effort. System developers have to look for opportunities to factor out common functionality required in different places. This is the hardest (and most rewarding) part of the implementation.

A word on *efficiency*: Smalltalk is an interpretative system, so that some run time overhead appears to be inevitable. Taking a closer look we find that Smalltalk programs are compiled into very compact intermediate code which is then interpreted by the Smalltalk virtual machine. Primitive operations (graphic, arithmetic) are implemented by assembler routines. As a result, Smalltalk and the SIC system built on top of it are highly reactive. Delays occur only when 'expensive' operations (loading or computing large amounts of information) are triggered by the user. SIC was designed for understandability rather than efficiency: E.g. as an alternative to the compressed GOTO tables SIC uses the explicit textual representation one finds in text books. Since SIC works fast enough for small projects there is little reason to trade explicit information for faster response times.

We believe that for applications like SIC another tradeoff, *programmer productivity* versus run time efficiency, should not always be resolved by using C.

Related work and references

The theoretical foundation for compiler-compilers like SIC was laid by D.E. Knuth in his two seminal papers, [5] and [6], on LR parsing and attribute grammars. A standard text book on compilers and compiler-compilers is the one by Aho, Hopcroft, and Ullman [1]; as a characteristic example of a compiler-compiler, it describes the popular LEX/ YACC system. YACC produces the C source code of an LALR(1) parser for a given context-free grammar. Actions in the form of user-supplied source code are inserted into the parser at the places indicated by action symbols in the grammar. Since attribute evaluation takes place during the parse, the order of attribute evaluation must be compatible with the parsing process: YACC works for L-attributed grammars and allows only one attribute per symbol.

A few years ago, the YTRACC parse browsing system [2] was presented, which automatically instruments YACC-produced parsers, so that the successive states of a parse are captured in a file as they are carried out. The captured parse can then be replayed forwards or backwards, step-by-step, or subtree-by-subtree. The viewing tool YSHOW continuously displays five successive snapshots of the parsing stack, the input line where the current input token is highlighted, the rule by which the next reduction will be carried out, and a command

line. The authors also report on experiences made in two consecutive compiler classes where YTRACC and YSHOW were offered for voluntary use. Different patterns of student behaviour are discussed in detail.

An earlier and more complete educational compiler generator, the Visible Attributed Translation System (VATS), is described in [12]. Like its conventional predecessor ATS it is built around an LL(1) parser generator. Compared to LALR(1) parser generators the LL(1) type is less general (e.g. context-free grammars must not contain left recursive rules) but it blends better with embedded attribute evaluation, since LL(1) parsers establish larger portions of the syntax tree sooner than other parsers. During a parse, ATS-generated compilers evaluate both inherited and synthesized attributes. VATS was created to 'provide a window on the compiler for tutorial and debugging purposes'. The visual parser in VATS displays the input with the most recently scanned token marked by a token cursor, the parse stack containing both grammar and action symbols, and message areas where the actions of the parser and the error recovery algorithm are described. Some of the extensions to VATS, that were termed 'under development' in [12], are: capabilities to single step the parsing algorithm, to scroll the parse stack, to support backtracking and the parsing of ambiguous grammars, and to display the flow of attribute values.

In [11] Standish and Bajaj describe SMALLGOL, an animated educational compiler, and experiences from a compiler construction course using it. SMALLGOL was not produced by a compiler-compiler but rather implemented conventionally in Lisa/Macintosh Pascal. SMALLGOL displays several windows which can be individually scrolled, repositioned, and closed to create different displays convenient for watching how separate parts of the compiler work. The windows contain the source code, the current state of the parser, an animated state diagram for the lexical analyser, the code emitted so far, the symbol table, and a trace of the parsing actions. In the course, the students progressed from using and watching SMALLGOL to extending it by some new features and on to implementing a new compiler by reusing and adapting components from the given one. According to an opinion survey, the students considered animation very helpful not only for understanding how compilers work but also in debugging their own compilers.

It is interesting to note that these efforts were obviously unrelated. We, too, came across the other work on visual compiling only after the implementation of SIC had been completed.

In order to get as much feedback as possible we have demonstrated SIC many times to colleagues at our university and other places; free copies of SIC are available on request.

References

- [1] A.V. Aho, J.E. Hopcroft, J.D. Ullman : Compilers - Principles, Techniques, and Tools, Addison-Wesley, Reading(Mass.), 1986.
- [2] R. Furuta, P.D. Stotts, J. Ogate : Ytracc: a Parse Browser for Yacc Grammars, Software - Practice and Experience 21(2), pp. 119-132, 1991.
- [3] A. Goldberg, D. Robson : Smalltalk80: The Language and its Implementation, Addison-Wesley, Reading(Mass.), 1983.
- [4] D.H. Jonassen : Integrating Learning Strategies into Courseware to Facilitate Deeper Processing, in: D.H. Jonassen (ed.) : Instructional Designs for Microcomputer Courseware, Lawrence Erlbaum Associates, Hillsdale(New Jersey), 1988.

-
- [5] D.E. Knuth : On the Translation of Languages from Left to Right, Information and Control 8,6, pp.607-639, 1965.
 - [6] D.E. Knuth : Semantics of context-free Languages, Mathematical Systems Theory 2, pp.127-145, 1968.
 - [7] J. Kröger, L. Schmitz : IPG - An Interactive Parser Generator, in: C. Choffrut, M. Jantzen (eds.) : STACS 91, LNCS 480, Springer, Berlin, 1991.
 - [8] J. Kröger, D. Mörs, M. Schmidt, L. Schmitz : SIC - Ein Smalltalk-basierter, Interaktiver Compiler-Compiler für den Unterricht. Universität der Bundeswehr München, Fachbereichsbericht Nr. 9009, November 1990.
 - [9] L. Schmitz: Watching the Objects within a Compiler Universität der Bundeswehr München, Fachbereichsbericht Nr. 9112, November 1991.
 - [10] L. Schmitz: Gläserne Compiler generieren. unix/mail, Heft 6, 1992.
 - [11] T.A. Standish, A.S. Bajaj : Using Animation to Teach Compiler Construction, Wheels for the Mind, 1986.
 - [12] J.-P. Tremblay, P.G. Sorenson : The Theory and Practice of Compiler Writing, McGraw-Hill, New York, 1986.

Appendix: Smalltalk basics

Smalltalk is both a small language and a big system. For understanding and writing attribute evaluation rules you need to know the syntax of the former (which looks strange at first sight but is quite easy to learn) and some base classes (data structures) and tools (interpreter, debugger and inspector) of the latter. For a deeper understanding of Smalltalk and object oriented programming in general, we recommend the book:

David N. Smith
 Concepts of object-oriented programming
 McGraw-Hill, 1991.

Let us start with the basic elements of the language:

“This is a comment“
 'This is a string'

4711.0, 47.11e2 and 4711 are **number constants**: two floating point constants (which will not appear in our examples but are there just as in any other programming language) and one integer constant, all of them having essentially the same value. The nice thing about Smalltalk integers is that there is virtually no upper limit. E.g. you can compute the factorial of 100 which causes problems in most other languages.

nil, true and false are the only elements of the classes UndefinedObject, True and False, respectively.

\$A \$> \$9 \$) \$, \$\$ are **character constants** denoting the characters: A > 9) , \$. I.e., for a character constant you type the prefix \$ and then any character (including non-printing characters like 'carriage return').

Symbol constants like #new, #do:, #Anything, #at:put: are introduced by just typing them. All method (OO term roughly similar to procedure or function) identifiers are symbols. In SIC scanners they are used as state identifiers. Syntactically, a symbol constant starts with prefix # which is followed either by a single identifier or a sequence of identifier-colon pairs.

Array constants also start with prefix # except when they are enclosed by another array constant. Like a LISP list, a Smalltalk array may contain any sequence of constants mixed as you wish and nested to any degree. Some examples are:

#(3 2 1 3)	“contains four integers“
#('I' 'go' 'home')	“contains three strings“
#('at' 4 'o''clock')	“a string, an integer, a string with an enclosed apostrophe“
#((2 1)(3 4))	“contains two enclosed arrays“
#(#at: (1 'ltr') 5.0 \$%)	“symbol, array, floating point number, character“

Typical Smalltalk **variable names** are:

aLocalVariable	“starts with a lower case letter“
AGlobalVariable	“starts with a capital letter“

Any **Smalltalk program code** is a sequence of assignments and messages separated by full stops (the semicolon used by other languages has a different meaning which will be explained later). A **message** is *both* an expression (returning a value) *and* a statement (producing a side effect). Similarly, an **assignment**

<variable> := <expression>

assigns the current value of the expression to the variable on the left hand side *and* returns the value of the expression.

All **messages** have the following syntax (which gives Smalltalk programs their particular outward appearance):

<object> <message text>

I.e. first you name the object which will receive the message and then you state the details of your order to that receiver object. This order contains the name of one of the services (methods) that the receiver object provides, and a list of parameters. In Smalltalk, depending on the number of parameters the message text is either a unary message, a binary message, or a keyword message.

In a **unary message**, the message text contains no parameters and thus consists of just a method identifier, e.g. in

7 even “will return false“
3.87 cos “will return the cosine of 3.87“

In a **binary message**, the message text contains one binary operator symbol and one parameter, e.g. in

17 + 4 “will return the integer 21“
'hello' , 'world' “will return the string 'helloworld' “

In a **keyword message**, the message text contains an alternating sequence of keywords and parameters (the method name being the catenation of all the keywords), e.g. in

5 to: 33 by: 2 “will return the sequence 5 7 9 11 ... 29 31 33“
'letter' at: 1 put: \$L “will return the string 'Letter' “
#(0 2) at: 1 put: 3 “will return the array #(3 2)“

Message expressions may be enclosed in parentheses and can be nested to any degree. Within nested message expressions unary messages have highest **precedence** and keyword messages have lowest precedence. Therefore,

5 + x abs * y max: 100 - 4

is an abbreviation for

((5 + (x abs)) * y) max: (100 - 4).

Notice that multiplication does not take precedence over addition! Binary expressions are always evaluated from left to right.

In Smalltalk, everything is an **object** and belongs to exactly one **class** of objects (with identical structure and identical methods but possibly different states). Even methods and classes are objects. Similarities between classes are used to define specialization (or **inheritance**) relations on classes and thereby prevent duplication of identical code. We will not go into this.

The following **code segment** is a Smalltalk version of the Pythagorean law. The first line introduces two local variables. No types are mentioned because Smalltalk variables are untyped. On the other hand, every object knows its (immutable) class.

```
| a b |
a := 3 . b := 4 .
Transcript cr; show: (a*a +(b*b)) sqrt printString
```

The last line contains a **message cascade**: the same object (the Transcript window) receives two messages that are separated by a semicolon. The first is the unary message 'cr' which causes Transcript's write cursor to move to the beginning of a new line. Then Transcript is asked to display at this position the string representation (printString) of the expression

$$\sqrt{a^2 + b^2}$$

By enclosing it in square brackets, any Smalltalk code segment can be turned into a **block**. Blocks are like unnamed procedures and can be stored in variables. A block object can be evaluated (any number of times) by sending it the unary message 'value'. Below, the Pythagorean formula is stored as a block in the global variable P and then evaluated twice. Therefore, the value '5' will be displayed in two consecutive lines within the Transcript window.

```
P := [ | a b | a:=3 . b:=4. Transcript cr; show: (a*a +(b*b)) sqrt printString ].
P value .
P value .
```

Obviously, **parameterized blocks** (like procedures with parameters) are much more useful. Formal parameters can be introduced (with colon prefixes) in the beginning of a block. Actual parameters are supplied when evaluating the block. For evaluating a two-parameter block the keyword message 'value: <aktPar1> value: <aktPar2>' is used. In the above example, we turn the local variables, a and b, into parameters:

```
P := [ :a :b | Transcript cr; show: (a*a +(b*b)) sqrt printString ].
P value: 3 value: 4 . “displays value 5 in Transcript“
```

Smalltalk **control structures** are based on blocks. The table below shows some conventional control structures and their Smalltalk equivalents:

conventional	Smalltalk
if <cond> then <stats>	<cond> ifTrue: [<stats>]
if <cond> then <stats1> else <stats2>	<cond> ifTrue: [<stats1>] ifFalse: [<stats2>]
while <cond> do <stats>	[<cond>] whileTrue: [<stats>]
for i:=a by b to c do <stats>	(a to: b by: c) do: [:i <stats>]
for x in set do <stats>	set do: [:x <stats>]

Smalltalk control structures are more flexible than their conventional counterparts. E.g. 'ifTrue:ifFalse:' is an if-statement and an if-expression at the same time; it can be used as the right hand side of an assignment statement. More important, the loop in the last line of the table can be used not only to enumerate the elements of a set but also (in exactly the same form) to enumerate the elements of *any* data collection (i.e. of any element of any subclass of Collection).

In order to be able to read and write attribute evaluation rules a working knowledge of the most important **data structure classes** is indispensable. We have already seen how to write array and string constants yielding objects of class Array and of class String, respectively. **Array** and **String** belong to a group of classes (the subclasses of IndexedCollection) permitting data elements to be accessed by their index. For *reading* the i-th element of an indexed data structure use the keyword message

```
'at: i'
```

and for *updating* the i-th element use

```
'at: i put: <newValue>'.
```

Since every data collection can be *enumerated* with the 'do:'-loop shown above, indices are typically not used for this purpose. With the unary message 'size' you can ask any data collection for its current *size*. Array and String belong to a subgroup of indexed data structures that have a fixed size. However, any indexed data structures of the same class can be *concatenated* using the binary “,” (comma) operator. For strings, we have seen the 'helloworld' example. For arrays, it works exactly the same way.

Objects of class **OrderedCollection** are indexed data structures of flexible size. A new `OrderedCollection` is created by the message expression `'OrderedCollection new'`. Existing components can be accessed using `'at:'` and `'at: put:'`. Let the variable `seq` contain an `OrderedCollection`. Then `'seq add: <element>'` appends an `<element>` at the end of `seq` and the expression `'seq addAll: <collection>'` appends all the elements of the parameter `<collection>` to `seq`. For indexed data structures there are lots of other methods including the unary `'first'` and `'last'` methods that access the first and last components, respectively.

Objects of classes **Set** and **Bag** are non-indexed data structures. New `Set` or `Bag` objects are created by evaluating expressions like `'Set new'` or `'Bag new'`. Like mathematical sets, objects of class `Set` may contain at most one copy of a value (object) `x`. Objects of class `Bag` like shopping bags may contain several copies of `x`. Let the variable `s` contain a `Set` or a `Bag` object. Then `'s add: x'` adds `x` to `s` and `'s remove: x ifAbsent: [<exception>]'` removes `x` from `s` if present and executes the block `[<exception>]` otherwise. The test `'s includes: x'` returns `'true'` if `x` is a component of `s` and `'false'` otherwise. The expression `'s addAll: <collection>'` appends all the elements of the parameter `<collection>` to `s`.

Objects of class **Dictionary** are flexible size data structures that contain key-value pairs. Evaluation of `'Dictionary new'` creates a new `Dictionary` object. Like indexed data structures `Dictionary` objects understand messages `'at:'` and `'at: put:'` the difference being that the `'at:'` parameter need not be an integer from the interval `[1:size]` but can rather be any (key) object. Let the variable `d` contain a `Dictionary` object. If the key `k` is already present in `d`, then the expression `'d at: k put: v'` updates the `k` component of `d` with the new value `v`. Otherwise, a new `k` component with value `v` is inserted into `d`. In addition to the method `'do:'` which enumerates all the values in `d` there is the method `'keysDo:'` which enumerates all the keys in `d`.

Now a few remarks about **tools**.

The **Smalltalk interpreter** is accessible from almost every text subwindow: Just highlight a (complete) Smalltalk code segment and choose `'printIt'` from the text menu to see the result.

Attribute evaluation rules are normally composed in the **attribute grammar editor** (details window). Choosing the text menu entry `'save'` not only saves the rule text but also adds code before (that extracts the required attribute values from the tree) and after it (that inserts the computed attribute value into the tree) and submits the resulting text to the Smalltalk compiler. If there is a **syntax error** in the attribute evaluation rule the compiler will catch it and insert an appropriate error message in the text. *Prior to correcting an error please restore the original text (using the text menu entry `'cancel'`)* because otherwise on the next `'save'` the text will be embedded in another pair of header and trailer texts which will inevitably produce syntax errors!

In Smalltalk **run time errors** automatically invoke the debugger. The **debugger** window lets you inspect the run time call stack, the source code of the methods in the stack and the values of all parameters and local variables. If you want to inspect the internals of a parameter, a local variable or the receiver (self) of the message, just click it with the mouse in the debugger and invoke (by choosing the middle mouse menu item `'inspect'`) an **inspector** window on this object. Within the inspector you can invoke inspectors on components of the inspected object recursively.

If you do not need a debugger or an inspector or some other Smalltalk tool any more, just close it.



