Science of Computer Programming 79 (2014) 101–115

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

JACCIE: A Java-based compiler–compiler for generating, visualizing and debugging compiler components

Nico Krebs, Lothar Schmitz [*]

Universität der Bundeswehr München, Fakultät für Informatik, München, Germany

ARTICLE INFO

Article history:
Received 17 January 2011
Received in revised form 24 February 2012
Accepted 8 March 2012
Available online 20 March 2012

Keywords:
Compiler–compiler
Compiler generation
Visualization of compiling techniques

ABSTRACT

Many programmers live in happy ignorance of their compilers' internal workings. Others may want to take a look at what is going on inside a compiler in much the same way that they use a debugger to watch their compiled programs execute. While conventional compilers are black boxes whose internals are hidden from the user, the JACCIE tool set helps to open up the box and have a look at what is going on inside.

Technically speaking, JACCIE is a compiler–compiler that, from suitable formal descriptions, generates the scanner, parser, and attribute evaluator components of a compiler and presents them in a visual debugging environment. It offers a number of alternative parser generators producing both top-down (LL) and bottom-up parsers of the LR variety, including SLR(1) and LALR(1) parsers, thus allowing users to experiment with different parsing strategies and to get a "feel" for their relative pros and cons. When designing JACCIE, the main emphasis was on two ergonomic goals that we considered important for educational software. Firstly, give user control over the program and not vice versa, e.g., our parsers (and other components) can be directed to go step-by-step forwards or backwards or to leap to some point in the input indicated by the users. Secondly, overcome the sometimes severe size limitations of computer displays by offering the same information in multiple representations that complement each other and by dividing information in smaller chunks that can be traversed in a meaningful way.

In this paper, after outlining the architecture of JACCIE, we discuss some of its technical and ergonomic aspects in detail, give a brief introduction into the use of JACCIE and its documentation, show an example application done with JACCIE, and finally discuss related work and future plans.

1. Introduction

At our institute, we have a long tradition in building compiler–compilers. The first systems (`coco`, implemented in PL/1 and `cosy`, implemented in Ada) date back to the late 1970s and 1980s, respectively. They served us mainly as test beds for theoretical investigations [1,2] concerning the correctness of parsing algorithms, in general, and of LR parsers optimized by the elimination of chain productions, in particular.

For novices, the theoretical background of compiler theory (especially deterministic parsing algorithms of the LR variety) is notoriously hard to understand. In a similar vein, Klint and others [3] deplore that "despite the pervasive role of grammars in software systems, the engineering aspects of grammarware are insufficiently understood". Conventional compiler-writing tool sets such as the well-known LEX/YACC combination [4] and the tools mentioned above are of little help to novices: they were developed for the efficient (batch-oriented) production of new compilers and, to be applied adequately, they require

[*] Corresponding author.
E-mail addresses: Nico.Krebs@unibw.de (N. Krebs), Lothar.Schmitz@unibw.de (L. Schmitz).

0167-6423/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2012.03.001

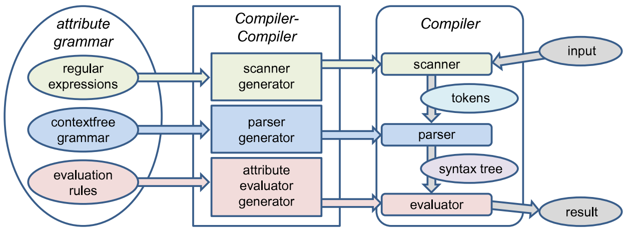


Fig. 1. Architecture of compilers and compiler–compilers.

some prior knowledge of compiler theory. The typical use of such a tool set starts with the preparation of some file(s) describing the (lexical and context-free) syntax of the intended source language and, possibly, a definition of the translation to be performed. From these definitions, the tool set either generates a compiler, or (more likely) a list of error messages that, for non-specialists, are hard to understand and act upon. Even if the source code is free of syntax errors and a compiler is generated successfully, it may still not do what it is expected to do. Therefore, to find any semantic errors, test inputs must be prepared from the source language, and, after applying the compiler, it must be checked whether the generated target language file meets the expectations. If it does not, the tool set offers little support for tracking down the causes of malfunction.

Compare this to using a compiler for your favourite programming language embedded in a modern IDE: even when preparing the source file, there is interactive support in the form of syntax highlighting. Syntax errors are not only listed by the compiler; when clicking on an error message, one is taken to the place in the source code where the error was found and provided with a (more or less) useful indication of what is wrong. For tracking down semantic errors, users are offered a debugging tool that allows them to continually observe, during execution, the values of so-called watch cells, i.e., expressions defined by users for this purpose. Also, users can control execution by either single-stepping through the program or by having the execution of the code interrupted at user-defined break points if some assertion (again defined by the user) is found invalid.

An obvious idea is to likewise embed a compiler–compiler in a GUI and provide debugging facilities, both for finding syntax errors and semantic errors. This also gives novices an opportunity to take a look at what is going on inside a compiler and thus develop a deeper understanding of compiling techniques. JACCIE, the *JAva-based Compiler–Compiler in an Interactive Environment*, was developed exactly for this purpose. It is an educational tool for automatically generating compiler components from suitable language descriptions and for executing these components in a visual debugging environment. The debugging environment continuously displays the components' internal states and is controlled with the mouse to move back and forth, in single or sets of steps. Auxiliary information derived from the source language descriptions can be viewed interactively any time. While conventional compilers are black boxes whose internals are hidden from the user, our tools let users take a look at what is going on inside.

The rest of this paper is organized as follows. In the next section, we outline the basic architecture of both compilers and compiler–compilers. Visual debugging of scanners and parsers requires to keep visible the close relationship between scanner and parser definitions (in the form of regular expressions and context-free grammars) on the one hand, and, on the other hand, the internal automata used by scanners and parsers that are derived from these definitions. Section 3 briefly describes how JACCIE derives "readable" scanning and parsing automata in a uniform way. Section 4 explains that the real challenges in designing and implementing our tools were not caused by theory, but rather by mundane ergonomic problems, some of them evoked by the severe size limitations of computer displays. Section 5 gives an introduction to JACCIE and its functionality by discussing some of its typical screens. We also describe where JACCIE, its handbook, and other useful documentation may be found. Section 6 describes a small application taken from the first author's work on a UVC (Universal Virtual Computer) for the long-term archiving of documents. Finally, we conclude with related work and future plans.

2. Architecture of compilers and compiler–compilers

Conceptually, the compilation process divides into three successive phases: During the lexical analysis phase, the scanner groups the characters of the given input text into a token sequence. During syntax analysis, the parser determines the syntactic structure of this token sequence in the form of a syntax tree. In the following synthesis phase, an attribute evaluator is applied to the syntax tree to achieve the desired translation result. For example, a C compiler would first determine the tokens in a given C program text, then build a syntax tree from this token sequence, and finally produce the resulting target code by performing semantic actions on the syntax tree. (For a programming language like C, the synthesis phase typically falls into a number of subphases like intermediate representation generation, optimization, and target code generation. While multiple evaluation subphases are supported by JACCIE, we do not cover these aspects here.)

Fig. 1 shows in its right-hand column the generated compiler. Reading this column from top to bottom, one finds the three compiler components (scanner, parser, evaluator) and the intermediate data (tokens, syntax tree) involved in translating some input into the result of the translation process. In the middle column the compiler–compiler is shown along with its three generators. The left-hand column indicates the definitions that must be provided as input to the generator components of the compiler–compiler: for the scanner generator, token definitions take the form of regular expressions; context-free grammars are input to the parser generator; attribute definitions and attribute evaluation rules must be added to each context-free production rule to define the desired translation. When reading the diagram from left to right, we see that, like in the compilation process shown in the right-hand column, some input (here, the attribute grammar definition) is transformed into a result (here, the compiler on the right-hand side). Because of this similarity, the tool set in the middle is often called a *compiler–compiler*, i.e., a compiler that generates compilers.

In Fig. 1, all data are enclosed by ovals and all program components are enclosed by rectangles. When reading the figure from left to right, the generated compiler components (scanner, parser, evaluator) are data. When reading the right-hand column from top to bottom, the compiler components are programs. To signify these double roles, compiler components are enclosed by rounded rectangles.

The above description applies in exactly the same way to production compiler–compilers like the Lex/Yacc tool set and to Jaccie. The main difference is that Lex/Yacc typically produces efficient black box compilers, while the internal structure and workings of compilers generated by Jaccie can be observed in detail at compiler run-time in the Jaccie debugger.

In the Jaccie tool set, the main components also correspond to Fig. 1: for each compiler component, there is a special editor for writing its definition, e.g., a direct manipulation style editor for the context-free grammars defining the parser. Also, for each compiler component, there is a combined tool for generating it from the definition and then executing it interactively, controlled by the user. In addition to these six main tools, there are quite a few specialized tools for viewing compiler information that was derived by the compiler–compiler from the compiler definition.

3. A unified approach to scanner and parser generation

Both scanners and LR parsers are controlled by deterministic finite automata. In both, these automata are derived from non-deterministic finite automata using the well-known subset construction (see, e.g., [4]) from formal language theory. Also, both constructions employ marker dots to indicate the recognition progress. In the subsections below, we explain these constructions in some detail. Throughout these constructions the original definitions (regular expressions defining scanner tokens and context-free production rules defining syntax) remain visible. This visibility is an important prerequisite for users to understand the relationship between the language definitions (originally provided by themselves) and the automata controlling compilation processes (as displayed by the visual debuggers).

3.1. Constructing finite automata for token recognition

Jaccie token definitions take the form of named regular expressions as in:

```
letter  {$a-$z}|{$A-$Z}
digit   {$0-$9}
name    letter(letter|digit)[0-*]
```

The first line says that a `letter` is either a lower case letter or an upper case letter of the Latin alphabet. The second line likewise states that a `digit` is a symbol from the ASCII range 0–9. The first two lines are auxiliary definitions that help to define `name` tokens in the third line: a `name` starts with a `letter`, which is followed by a sequence of `letter`s and `digit`s of arbitrary length.

Obviously, one can get rid of the above auxiliary definitions by substituting them into the token definition. This substitution gives the (slightly less readable) token definition:

```
name    ({$a-$z}|{$A-$Z})(({$a-$z}|{$A-$Z})|{$0-$9})[0-*]
```

The standard textbook construction for turning a regular expression into a finite automaton (again see [4]) recursively breaks the expression into ever smaller parts and systematically labels an automaton's edges with these subexpressions. This break-down results in a non-deterministic finite automaton (with ϵ-transitions), which by the subset construction mentioned above (and an additional minimization step) is turned into an efficient deterministic finite automaton. The resulting automaton is well-suited for scanning `name`-tokens, but the connection to the original token definition is lost completely. This loss is a problem when trying to identify errors in token definitions by using the scanner debugging component.

Therefore, we use an alternative construction invented by McNaughton and Yamada[1] [6]. This construction heavily relies on **marker dots** that are inserted into regular expressions and separate the part that already has been matched to the input character sequence from what still must be matched. In the beginning, the dot is placed immediately to the left of the regular expression, because no input has been matched so far. This results in the following *initial state*:

```
.({$a-$z}|{$A-$Z})(({$a-$z}|{$A-$Z})|{$0-$9})[0-*]
```

[1] In a more recent paper [5], alternative constructions of deterministic finite automata from regular expressions have been described in a unified framework. It appears that our construction results in so-called Antimirov automata.

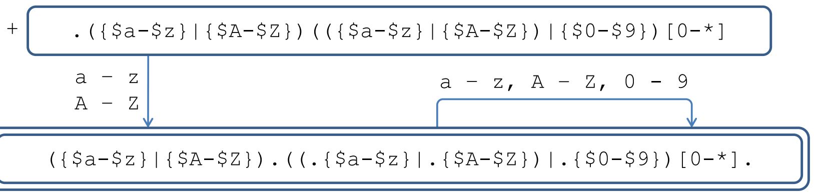

+
.({$a-$z}|{$A-$Z})(({$a-$z}|{$A-$Z})|{$0-$9})[0-*]
a - z
A - Z
a - z, A - Z, 0 - 9
({$a-$z}|{$A-$Z}).((.{$a-$z}|.{$A-$Z})|.{$0-$9})[0-*].

Fig. 2. Deterministic finite automaton for `name` tokens.

If in the initial state, a (lower case or upper case) letter is found, the letter is "consumed" and a transition into the following state (named B for further reference) occurs:

```
({$a-$z}|{$A-$Z}).(({$a-$z}|{$A-$Z})|{$0-$9})[0-*]
```

Since the rest of the expression is optional (may be repeated *zero* or more times), from B an ϵ-transition leads to another state:

```
({$a-$z}|{$A-$Z})(({$a-$z}|{$A-$Z})|{$0-$9})[0-*].
```

Similarly, it can be argued that ϵ-transitions also lead from B to the following states:

```
({$a-$z}|{$A-$Z})((.{$a-$z}|{$A-$Z})|{$0-$9})[0-*]
({$a-$z}|{$A-$Z})(({$a-$z}|.{$A-$Z})|{$0-$9})[0-*]
({$a-$z}|{$A-$Z})(({$a-$z}|{$A-$Z})|.{$0-$9})[0-*]
```

Now, application of the standard **subset construction** to B and its ϵ-successors results in the following set of "dotted regular expressions", which by construction is a state of an equivalent deterministic finite automaton:

```
{({$a-$z}|{$A-$Z}).(({$a-$z}|{$A-$Z})|{$0-$9})[0-*] ,
 ({$a-$z}|{$A-$Z})(({$a-$z}|{$A-$Z})|{$0-$9})[0-*]. ,
 ({$a-$z}|{$A-$Z})((.{$a-$z}|{$A-$Z})|{$0-$9})[0-*] ,
 ({$a-$z}|{$A-$Z})(({$a-$z}|.{$A-$Z})|{$0-$9})[0-*] ,
 ({$a-$z}|{$A-$Z})(({$a-$z}|{$A-$Z})|.{$0-$9})[0-*]
}
```

Finally, the above set of *five* regular expressions with *one* marker dot each may more concisely be represented by *one* regular expression with *five* marker dots:

```
({$a-$z}|{$A-$Z}).((.{$a-$z}|.{$A-$Z})|.{$0-$9})[0-*].
```

Continuing this construction, we obtain a deterministic finite automaton that can be used for analysing `name` tokens and whose states indicate progress with respect to the original token definition — you just have to look where the marker dots are.

The complete automaton is shown in Fig. 2. It only has the two states that we have derived above. In the figure, the initial state is pointed out with a plus sign. With our construction, *final states* are all states where a marker dot appears at the end of the regular expression. In the figure, the final state is doubly enclosed. Admissible transitions between states are represented by labelled arrows.

3.2. *Constructing LR(0) automata for parsing*

In analogy to the previous subsection and to obtain a consistent formal and visual representation for both scanners and parsers, we construct deterministic finite parsing automata whose states indicate progress in terms of the underlying parser definition, i.e., a context-free grammar. By again applying the marker dot technique and the subset construction, we obtain LR(0) parsing automata, which are the core components of the SLR(1) and LALR(1) automata that are often used in practical parsers. For debugging and visualization, the explicit representation of LR states is better suited than the more compact action and goto tables derived from these states. LR automata tend to be rather voluminous. Therefore, we demonstrate the LR(0) construction on a tiny grammar only.

The *language* of all valid balanced sequence of parentheses may be formally defined by a *context-free grammar* with the production rules

$$S \rightarrow (S)S \mid \varepsilon$$

For the LR(0) construction, we have to add a new *start symbol*, S', and a new *start production*, $S' \rightarrow S$.

Next, we insert **marker dots** into the right-hand sides of production rules (like we did for regular expressions) to separate the part that has already been recognized from the rest. Production rules with marker dots are called *items*. They represent the states of a non-deterministic finite automaton. Its initial state *A* is the start rule with the marker dot at the beginning of its right-hand side to indicate that nothing has been recognized yet and that we expect to find a sequence derivable from *S*:

```
S' -> . S
```

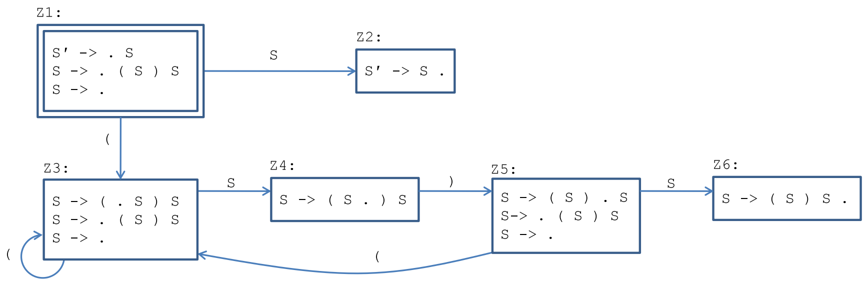


Fig. 3. LR(0) automaton for balanced sequences of parentheses.

According to the grammar this sequence is either empty or must match $(S)S$. Therefore, in the non-deterministic automaton, there are ε-transitions from state A to the states B_1:

```
S -> .
```

and B_2:

```
S -> . ( S ) S
```

If in state B_2 a left parenthesis is found in the input, then it can be matched in a transition leading to state C:

```
S -> ( . S ) S
```

In C, the marker dot is in front of symbol S and there are again ε-transitions to the states B_1 and B_2.

Proceeding in this way, a non-deterministic *parsing automaton* is obtained. By applying the standard **subset construction** to this automaton, finally the deterministic *LR(0) parsing automaton* is derived. Its initial state is the following set of items:

```
{ S' -> . S ,
  S -> . ( S ) S ,
  S -> . }
```

The initial state's (-successor is:

```
{ S -> ( . S ) S ,
  S -> . ( S ) S ,
  S -> . }
```

The complete LR(0) automaton is shown in Fig. 3. Thus, we can obtain similar and consistent formal and visual representations for both scanners and parsers.

4. On visualizing compiler execution

As stated in the introduction, the real challenges in the design and implementation of JACCIE were not caused by theory but came from two rather mundane ergonomic problems:

1. *How to provide sufficient user control?* Active learning within a simulation environment that encourages students to make their own experiences is much more effective [7] than stepping through a predefined sequence of elementary lessons. So active learning is what we are aiming at. Yet, because of the nature of the translation process, the main phases, lexical analysis, parsing, and attribute evaluation, are deterministic processes that must be carried out essentially in that order. Still, JACCIE users have plenty of room for experimentation:
   - Users can direct the processes of scanning, parsing, and attribute evaluation in almost any way they wish, e.g., JACCIE scanners can be made to single step forwards and backwards, in units of characters or whole tokens, or leap to a place in the input where the user clicks the mouse.
   - Users may freely choose between different parsing strategies: LL(1), LR(0), LR(1), SLR(1), and LALR(1), not detailed here.
   - Likewise, different strategies for attribute evaluation are available to users. One strategy allows users to direct attribute evaluation at run-time (where attributes that are "ready for evaluation" are shown in blue).
   - The ordinary flow of data and control is that the results from one phase are available as input for the next phase. Alternatively, results can be stored in files — for use in other programs or for resuming processing later on. If users are not interested in the details of some phase, they can have it executed in "batch mode" in the background.
   - All the time, additional information derived by JACCIE from the compiler definition files may be viewed interactively. Examples of such information are the first-/follow-sets and the different parsing automata derived from a context-free grammar.
   - One important debugging feature of JACCIE parsers is their support for *non-deterministic parsing*, e.g., SLR(1) parsing for non-SLR(1) grammars. Non-deterministic parsing may either proceed automatically (exploring different alternatives

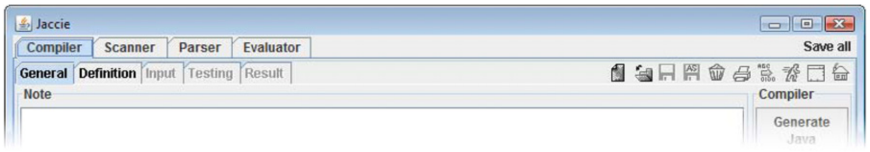


Fig. 4. Jaccie double tab bar.

in backtracking mode) or directed by the users. This feature is meant to be used for tracking down (and eliminating!) sources of non-determinism[2]: if a parser is deterministic, its underlying grammar is unambiguous, ensuring that for each valid parser input there is exactly one syntax tree, which in turn is a prerequisite for successful attribute evaluation.

- Finally, Jaccie is a full-fledged compiler–compiler, giving users the opportunity to explore any definitions and inputs that they can conceive.

2. *How to cope with the problem of the restricted screen size?* Even for small grammars no computer display is large enough to present all relevant information (the attributed grammar, the current parsing situation, the parsing automaton, first-/follow-sets etc.) at the same time. Obviously, modern GUIs somewhat alleviate these problems by providing multiple windows for switching between different data sets. Also, scrolling mechanisms help to explore very large data sets. For some of the artefacts mentioned above, such standard windowing techniques do not suffice. Jaccie therefore offers a number of additional mechanisms tailored to the information to be presented to users:
   - Large syntax trees are not well-suited to standard scrolling mechanisms because substantial portions of the drawing area tend to be empty, and when looking at some detailed section of (the rather uniformly built) tree structure users may get lost easily. For attribute evaluation, this getting-lost-effect is exacerbated, because each node of the tree is decorated with any number of attributes that carry the relevant (and sometimes extensive) information. Some of the complementary mechanisms offered by Jaccie to support orientation are: a zoom mechanism (good for relatively small trees); a mechanism for traversing the tree (up or down) along its structure or in some attribute evaluation order; a mechanism for switching on and off the details of attributes, including an alternative view that shows the current node and its neighbours in full detail (including a graphic representation of all attribute dependencies at these nodes).
   - LR automata tend to become extremely large even for context-free grammars of moderate size — a single state may easily fill the display. Scrolling through a list of all states is not a meaningful way of traversing an automaton. Instead, Jaccie allows users to traverse the states of an automaton by following links. During parsing, the current state of the parsing automaton is kept visible. In addition to the item sets described in the last section, reduced action-/goto-table representations, i.e., conventional parsing tables are available. For tracking down problems, a list of error states gives immediate access to all the states that contain shift-/reduce- or reduce-conflicts.
   - Core information for LL parsers (and for LR parsers with lookahead) are the so-called first-/follow-sets. Jaccie not only shows these sets; it also allows users to trace back their computation from the underlying context-free grammar.

   Essentially, all these solutions are variations of two themes: The first theme is to offer alternative representations that complement each other, giving the user a choice to use what is suited best for solving the problem at hand. The second theme is to unfold large data sets over time and to provide meaningful traversal mechanisms.

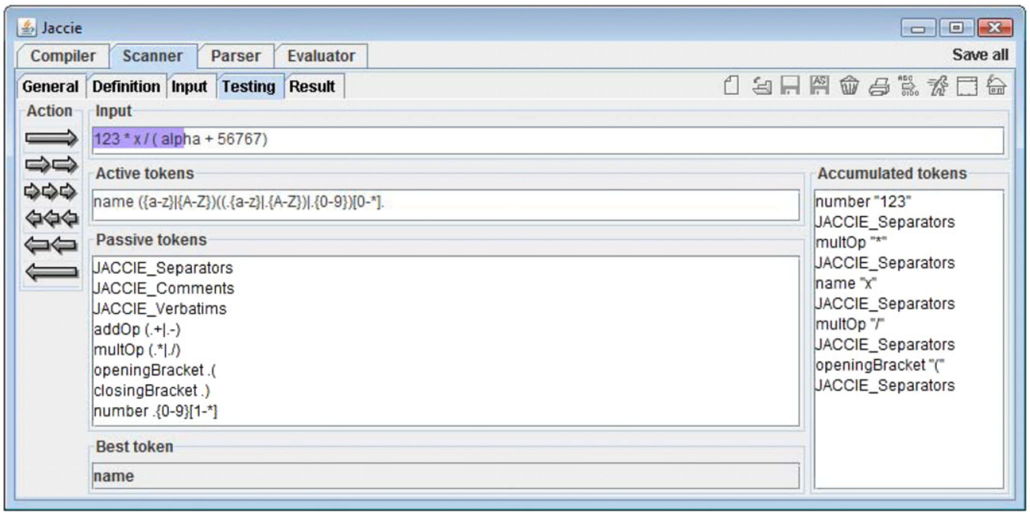
The Jaccie grammar editor is designed to *avoid user errors*: it allows the name of each (terminal or non-terminal) symbol to be typed in exactly once. Thereafter, production rules are assembled from the sets of symbols using direct manipulation, i.e., drag-and-drop. This strategy effectively avoids any typos on the users' part. Symbols may be renamed (and automatically replaced consistently throughout the grammar). Also, new symbols may be created and existing ones deleted (the latter operation deletes all instances of the symbol from the whole grammar).

5. Jaccie in action

The double tab bar shown in Fig. 4 is visible in the Jaccie GUI all the time. It reflects the tool architecture described in Section 2: in the top row, there is one tab for the compiler as a whole and for each compiler component. With Jaccie, users can not only extensively debug compilers and their components; they can also output the tested compiler in the form of Java source code, i.e., after submitting it to a standard Java compiler, use it as an ordinary compiler. Having selected one component in the top row, the tabs in the second row allow its definition (in tab "Define") and a suitable input file ("Input") to be loaded and edited, the component to be debugged interactively on the input ("Testing"), and the result to be viewed ("Result").

The graphical buttons – most of them are not active initially – let you create, load, save, delete, and print artefacts (definitions, inputs, outputs). The fourth button from the right is for generating Java source as indicated above; the third button is for running the compiler (component) in batch mode and will bring you directly from the input to the result tab.

[2] This Jaccie feature should not be confused with parsing strategies, like Tomita's [8], which allow for the parsing of non-LR grammars.



Input

```
123 * x / ( alpha + 56767)
```

Active tokens

```
name ({a-z}|{A-Z})((.{a-z}|.{A-Z})|.{0-9})[0-*].
```

Passive tokens

```
JACCIE_Separators
JACCIE_Comments
JACCIE_Verbatims
addOp (.+|.-)
multOp (.*|./)
openingBracket .(
closingBracket .)
number .{0-9}[1-*]
```

Best token

```
name
```

Accumulated tokens

```
number "123"
JACCIE_Separators
multOp "*"
JACCIE_Separators
name "x"
JACCIE_Separators
multOp "/"
JACCIE_Separators
openingBracket "("
JACCIE_Separators
```

Fig. 5. JACCIE's scanner debugging tool.

We now briefly look at a selection of characteristic JACCIE windows: the scanner and LR parser debugging tools, the direct manipulation grammar editor, and the grammar information window.

5.1. Two debugging tools

Fig. 5 shows the scanner debugger working on an arithmetic expression. The characters that have been processed so far are highlighted in the input subwindow. On the right-hand side, the resulting tokens are accumulated. The arrow controls allow the debugger to be moved to the end of the (input / next token / next character) and back. Alternatively, users may click directly in the input subwindow to where they want the debugger to move.

Arithmetic expressions are made up of names, operators, numbers and parentheses. Regular expressions defining these tokens (more precisely, states of the finite automata derived from these definitions) are shown in the middle subwindows. Because the prefix "alp" of the current token only matches the `name` token definition, this is the only "Active token". Recall that, in Section 3.1, we have constructed this very automaton.

The LR parser debugging tool in Fig. 6 has a similar structure and similar arrow controls. The question mark arrow is for non-deterministic parsing and allows one to back up to the nearest decision point (and to reconsider the choice).

In bottom-up parsing, the intermediate result is a forest of subtrees of the syntax tree to be constructed. Because these trees tend to be very large, using the Tree controls, users can zoom in and out and, additionally, switch off all other information subwindows.

The two top right information subwindows show the current state (no. 2) of the LR automaton that controls the parser's operation and this state's context within the automaton, i.e., its immediate predecessor and successor states.

The input is the token sequence produced by the scanner.

The bottom left subwindow shows the LR stack. This stack always contains (in reverse order) the root labels of the trees to its right. Therefore, when joining some trees in a reduce step, the top of this stack is where the symbols of the right-hand side of the production rule applied in the reduction step can be found.

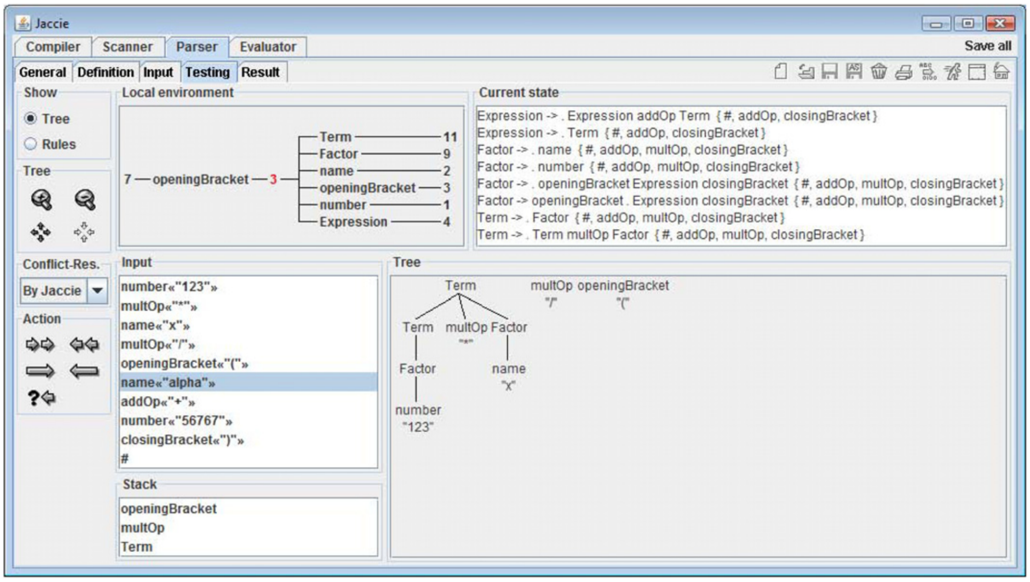
Under "Show", users can choose the rightmost derivation to be shown instead of the (equivalent) graphical tree representation; this may be sensible for very large trees.

5.2. The grammar editor

The grammar editor in Fig. 7 heavily relies on direct manipulation to avoid any typing errors. There are four information subwindows: two subwindows in the top show the available non-terminal and terminal symbols, respectively. In a line above the non-terminals, the start symbol of the grammar (a non-terminal) is placed. The large subwindow in the bottom collects the grammar's production rules.

The predominant controls are the big buttons on the left: information is deleted by dragging it to the trash bin. A new object is created by dragging the stamp button into the subwindow where the new object must appear. If this new object is a symbol, the user is prompted to type its name. A symbol is renamed by dragging it to the Rename button and typing the new name into a prompter.

Other operations use direct manipulation as well: to assign a new start symbol, simply one of the non-terminals is dragged to the start symbol's place. For adding some new symbol (instance) to the right-hand side of a production rule, this symbol is selected from one of the symbol subwindows and dragged into the rule. Also, symbols can be moved within and between



Local environment:

```
7 — openingBracket — 3 —┬— Term ———————— 11
                        ├— Factor —————— 9
                        ├— name ———————— 2
                        ├— openingBracket — 3
                        ├— number —————— 1
                        └— Expression —— 4
```

Current state:

```
Expression -> . Expression addOp Term {#, addOp, closingBracket}
Expression -> . Term {#, addOp, closingBracket}
Factor -> . name {#, addOp, multOp, closingBracket}
Factor -> . number {#, addOp, multOp, closingBracket}
Factor -> . openingBracket Expression closingBracket {#, addOp, multOp, closingBracket}
Factor -> openingBracket . Expression closingBracket {#, addOp, multOp, closingBracket}
Term -> . Factor {#, addOp, multOp, closingBracket}
Term -> . Term multOp Factor {#, addOp, multOp, closingBracket}
```

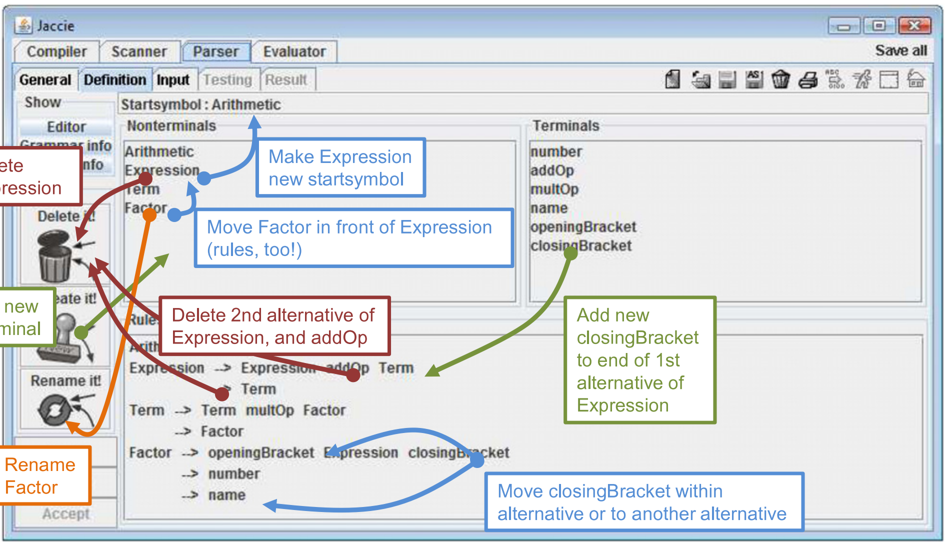
Input:

```
number«"123"»
multOp«"*"»
name«"x"»
multOp«"/"»
openingBracket«"("»
name«"alpha"»
addOp«"+"»
number«"56767"»
closingBracket«")"»
#
```

Stack:

```
openingBracket
multOp
Term
```

Fig. 6. JACCIE's LR parser debugging tool.



```
Arith...
Expression  -->  Expression  addOp  Term
            -->  Term
Term        -->  Term  multOp  Factor
            -->  Factor
Factor      -->  openingBracket  Expression  closingBracket
            -->  number
            -->  name
```

Fig. 7. JACCIE's direct manipulation grammar editor.

right-hand sides of production rules using direct manipulation. Finally, rearranging the order of non-terminals by direct manipulation will also rearrange the production rules according to the new ordering of its left-hand sides.

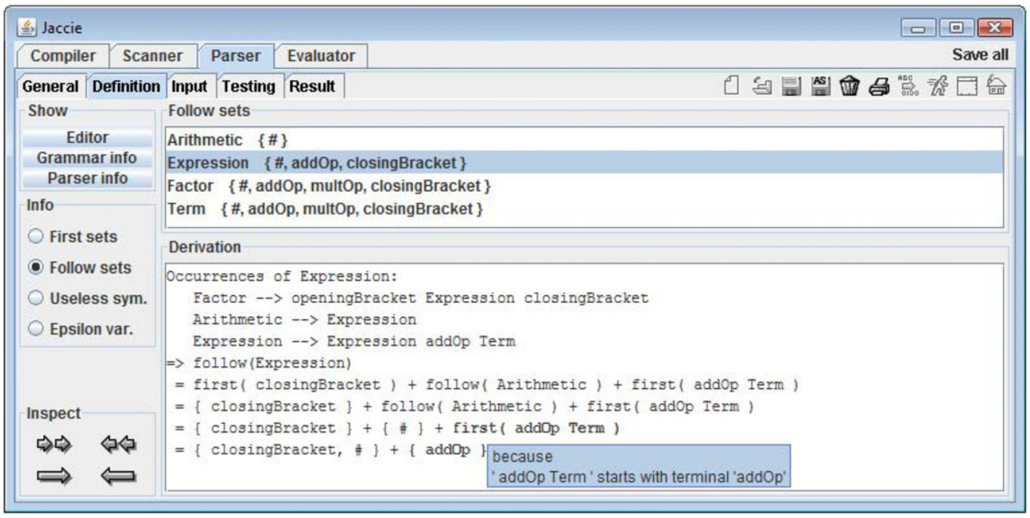
Using the Undo-button, the last editing operation can be undone[3] (repeatedly). At the end of an editing session, either all changes must be made permanent (by pressing Accept) or be dismissed (by pressing Cancel).

From the grammar editor window two different information windows can be reached by users: one for browsing parser information derived from the grammar (LL decision sets, all different types of LR automata, and LL and LR conflicts); the other shows basic information related to the grammar.

5.3. The grammar information window

The grammar information window in Fig. 8 gives a choice of what information is to be shown: first sets, follow sets, useless symbols, or ε-variables. Here, the follow sets were chosen. For each non-terminal, the set of terminal symbols that can possibly follow it in any sentential form is shown.

[3] In all JACCIE text subwindows, the standard CTRL-Z combination will effect an undo.



```
Arithmetic   { # }
Expression   { #, addOp, closingBracket }
Factor   { #, addOp, multOp, closingBracket }
Term   { #, addOp, multOp, closingBracket }
```

```
Occurrences of Expression:
   Factor --> openingBracket Expression closingBracket
   Arithmetic --> Expression
   Expression --> Expression addOp Term
=> follow(Expression)
 = first( closingBracket ) + follow( Arithmetic ) + first( addOp Term )
 = { closingBracket } + follow( Arithmetic ) + first( addOp Term )
 = { closingBracket } + { # } + first( addOp Term )
 = { closingBracket, # } + { addOp }
```

```
because
' addOp Term ' starts with terminal 'addOp'
```

Fig. 8. Grammar information is given and justified.

After selecting one of these sets, Jaccie can be made to explain how the set was computed. Here, the follow set for "Expression" was selected. The "Inspect" arrow controls help to trace back the computation of this set. When the mouse pointer hovers over the last line, Jaccie provides additional explanations in the form of tool tips. The tool tip in Fig. 8 explicitly states the (trivial) reason why `first(addOp Term)` reduces to `{addOp}`.

Obtaining and using Jaccie

The Jaccie tool and its extensive handbook are available on-line.[4]

For non-commercial purposes, Jaccie is available for free. Under the same premise, the source code is available on request from the developers.

The Jaccie system comes in the form of a single JAR file. Installation (which essentially boils down to double-clicking the JAR file and then setting some path info) and all its features are described at great length in the Jaccie handbook.

On the same site,[5] two tutorials explain why syntax tools are useful in general and what users must know about them.

6. Implementing a UVC assembler using Jaccie

Today, most documents are created, stored, and communicated using computers. A program that displays digital documents of some format on a computer screen for human consumption is called a *rendering program*, e.g., Adobe's Acrobat Reader for PDF documents. Jeff Rothenberg's famous quote[6] pointedly describes the long-term archiving problems thus entailed:

"Digital documents last forever – or five years, whichever comes first (1999)".

One of the main approaches to overcome these problems [9] is *emulation*: keeping digital documents alive by preserving their bit stream representation and porting suitable rendering software to future computers.

A variant of this approach was proposed by Lorie [10]: defining a *Universal Virtual Computer (UVC)* that is both sufficiently powerful for serving as a target machine for all conceivable rendering programs and simple enough to be ported to all future computers.

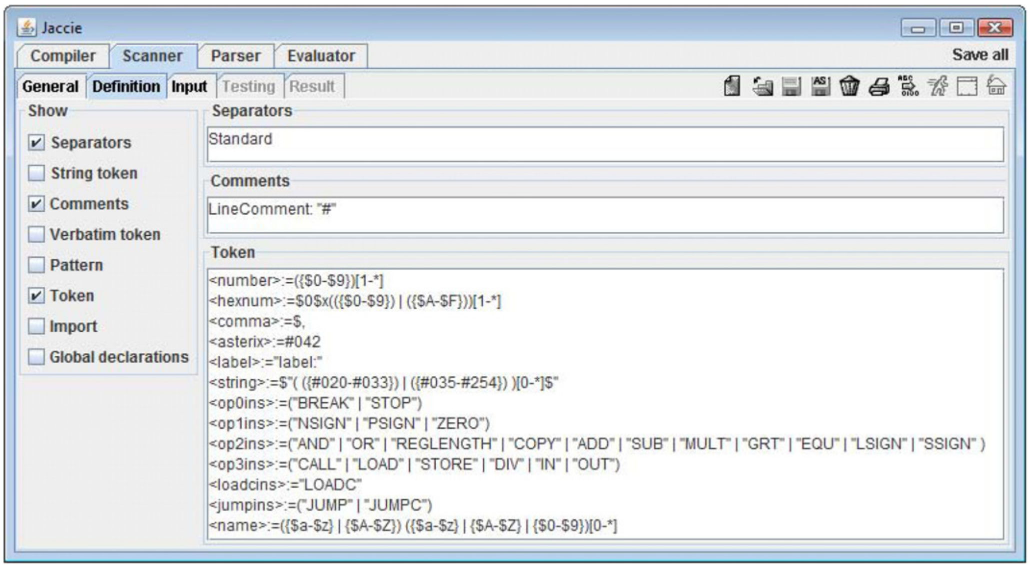
A precise, albeit not formal, specification of UVC was given in Lorie [11]. The first author's research on long-term preservation of digital documents included assessing the feasibility of the UVC approach, in particular, its ease of implementation.

Thus, a prototype UVC assembler was created using Jaccie as an implementation tool. From Lorie's UVC description, formal Jaccie definitions were derived and implemented as described below.

6.1. UVC scanner

The UVC is a RISC with extremely powerful storage characteristics: There are no fixed bounds on either the number of storage cells and registers or the size of a register, leaving implementers the freedom to find appropriate solutions according to circumstances.

4 http://inf2.w3.rz.unibw-muenchen.de/Tools/Syntax/english/download.html.
5 http://inf2.w3.rz.unibw-muenchen.de/Tools/Syntax/english/index.html.
6 http://www.clir.org/pubs/reports/rothenberg/introduction.html.



Show: [✓ Separators] String token [✓ Comments] Verbatim token Pattern [✓ Token] Import Global declarations

Separators

```
Standard
```

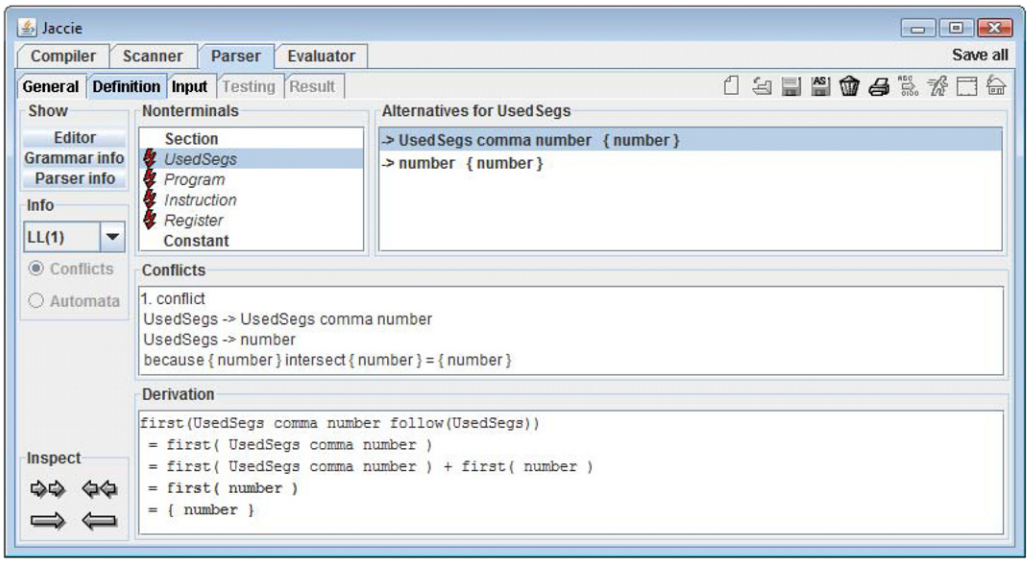
Comments

```
LineComment: "#"
```

Token

```
<number>:=({$0-$9})[1-*]
<hexnum>:=$0$x(({$0-$9}) | ({$A-$F}))[1-*]
<comma>:=$,
<asterix>:=#042
<label>:="label:"
<string>:=$"( ({#020-#033}) | ({#035-#254}) )[0-*]$"
<op0ins>:=("BREAK" | "STOP")
<op1ins>:=("NSIGN" | "PSIGN" | "ZERO")
<op2ins>:=("AND" | "OR" | "REGLENGTH" | "COPY" | "ADD" | "SUB" | "MULT" | "GRT" | "EQU" | "LSIGN" | "SSIGN" )
<op3ins>:=("CALL" | "LOAD" | "STORE" | "DIV" | "IN" | "OUT")
<loadcins>:="LOADC"
<jumpins>:=("JUMP" | "JUMPC")
<name>:=({$a-$z} | {$A-$Z}) ({$a-$z} | {$A-$Z} | {$0-$9})[0-*]
```

Fig. 9. UVC Scanner definition in Jaccie.



Nonterminals

```
Section
UsedSegs
Program
Instruction
Register
Constant
```

Alternatives for UsedSegs

```
-> UsedSegs comma number  { number }
-> number  { number }
```

Conflicts

```
1. conflict
UsedSegs -> UsedSegs comma number
UsedSegs -> number
because { number } intersect { number } = { number }
```

Derivation

```
first(UsedSegs comma number follow(UsedSegs))
 = first( UsedSegs comma number )
 = first( UsedSegs comma number ) + first( number )
 = first( number )
 = { number }
```

Fig. 10. Conflicts within intuitive parser definition in Jaccie.

The UVC instruction set consists of 25 instructions with different characteristics: apart from the few load and jump instructions, all other operations were grouped according to their number of operands (0, 1, 2, or 3) for unified processing by parsers. This grouping also helped to reduce the size of the context-free grammar describing the UVC assembly language.

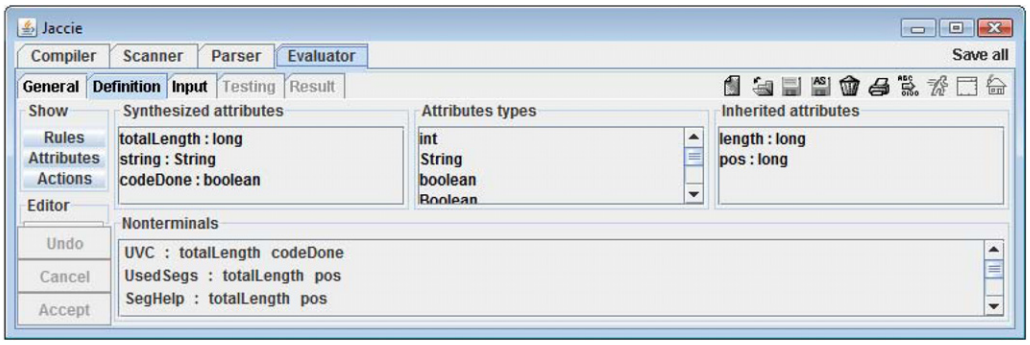
Other token definitions define UVC's numbers, hex numbers, strings, and names. All resulting UVC token definitions are shown in Fig. 9.

6.2. UVC parser

In the UVC community, it was decided early on to describe the assembly language with an LL(1) context-free grammar. However, such a grammar was not made available; it must be derived from the specification text. As was to be expected, the context-free grammar distilled from the text did not have the LL(1) property. Here, Jaccie helped by showing all LL(1) conflicts and their causes (see Fig. 10).

Converting this grammar into an LL(1) grammar turned out to be relatively simple; it only required two standard transformation techniques (see, e.g., [4]) to be applied consistently:

- Eliminate left recursion.
- Factor out common prefixes of different alternatives.



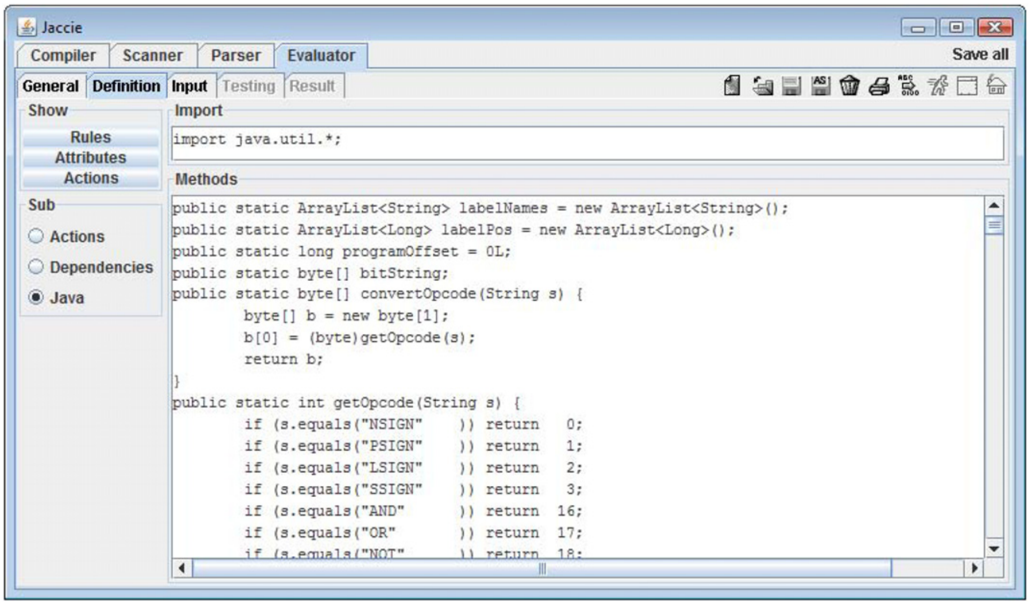
Synthesized attributes
totalLength : long
string : String
codeDone : boolean

Attributes types
int
String
boolean
Boolean

Inherited attributes
length : long
pos : long

Nonterminals
UVC : totalLength codeDone
UsedSegs : totalLength pos
SegHelp : totalLength pos

Fig. 11. Defining JACCIE attributes.



Sub: Actions Dependencies [✓ Java]

Import
```
import java.util.*;
```

Methods
```
public static ArrayList<String> labelNames = new ArrayList<String>();
public static ArrayList<Long> labelPos = new ArrayList<Long>();
public static long programOffset = 0L;
public static byte[] bitString;
public static byte[] convertOpcode(String s) {
        byte[] b = new byte[1];
        b[0] = (byte)getOpcode(s);
        return b;
}
public static int getOpcode(String s) {
        if (s.equals("NSIGN"    )) return   0;
        if (s.equals("PSIGN"    )) return   1;
        if (s.equals("LSIGN"    )) return   2;
        if (s.equals("SSIGN"    )) return   3;
        if (s.equals("AND"      )) return  16;
        if (s.equals("OR"       )) return  17;
        if (s.equals("NOT"      )) return  18;
```

Fig. 12. Java frame providing methods and variables.

6.3. UVC translator

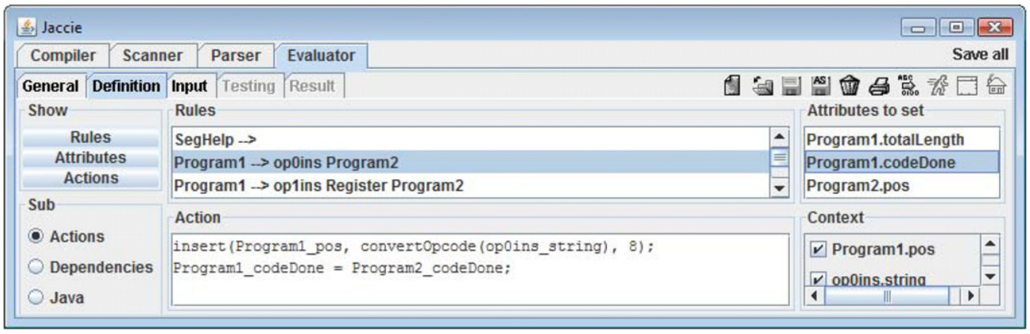
In JACCIE and other compiler–compilers based on attribute grammars, translation processes rely on data associated with the non-terminals of the grammar (and thus with each node of the syntax tree) – *attributes* – and on rules for computing and systematically propagating these attributes — *attribute evaluation rules*. Attribute evaluation rules are provided by users for each production rule of the context-free grammar.

The UVC translator translates the assembly language program (that was analysed by the parser and, hence, is available as a syntax tree) into the bit string representation used by the UVC. For recursively computing these bit strings from substrings, we need each substring's length and position within the final result, i.e., in the root attributes `totalLength` and `pos`. The substrings are kept in `string` attributes. The `codeDone` attributes help to synchronize attribute evaluation in the presence of global data (to be discussed immediately). Fig. 11 shows how these attributes are defined and associated with the non-terminals of the grammar.

For efficiency reasons, it is desirable to complement a pure attribute grammar with global data that exist only in one place. Fig. 12 shows how such global data (and any methods needed for processing these data) can be defined in JACCIE. The attribute evaluation rule defined in Fig. 13 uses the global data. To ensure the correct order of computation, the `codeDone` attribute is used to signal the availability of the corresponding piece of code.

6.4. UVC compiler and editor

The overall aim of this development was to produce a UVC assembler system implemented in Java and embedded in a GUI as shown in Fig. 14.



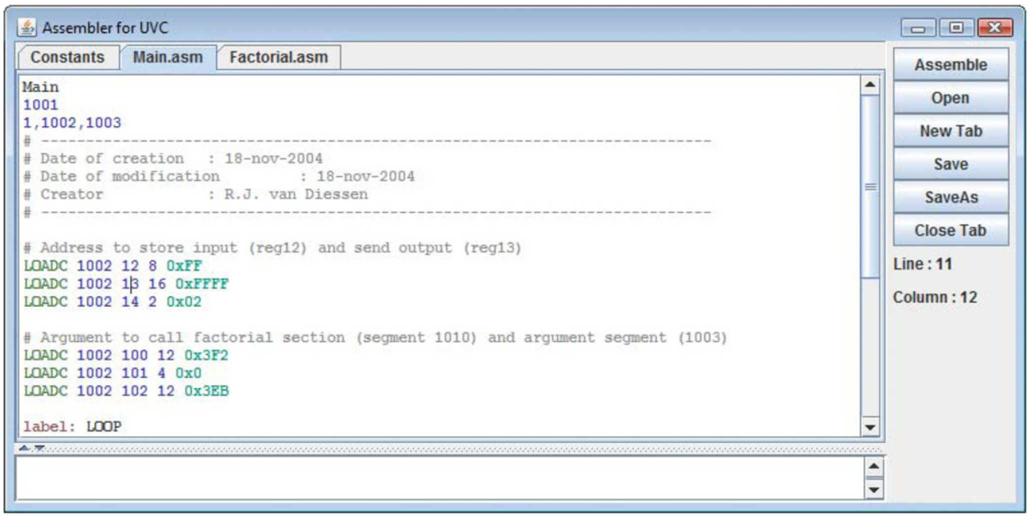
```
SegHelp ->
Program1 -> op0ins Program2
Program1 -> op1ins Register Program2
```

```
Program1.totalLength
Program1.codeDone
Program2.pos
```

```
insert(Program1_pos, convertOpcode(op0ins_string), 8);
Program1_codeDone = Program2_codeDone;
```

✓ Program1.pos
✓ op0ins.string

Fig. 13. Efficient computation combining global data and attribute evaluation.



```
Main
1001
1,1002,1003
# ---------------------------------------------------------------------------
# Date of creation   : 18-nov-2004
# Date of modification         : 18-nov-2004
# Creator            : R.J. van Diessen
# ---------------------------------------------------------------------------

# Address to store input (reg12) and send output (reg13)
LOADC 1002 12 8 0xFF
LOADC 1002 13 16 0xFFFF
LOADC 1002 14 2 0x02

# Argument to call factorial section (segment 1010) and argument segment (1003)
LOADC 1002 100 12 0x3F2
LOADC 1002 101 4 0x0
LOADC 1002 102 12 0x3EB

label: LOOP
```

Fig. 14. Editor for the UVC assembler toolkit.

Therefore, we organized the components described above into one compiler – using JACCIE's main tab "Compiler" – and produced the Java source code of the compiler class (named `Compile`) by simply pressing the appropriate "generate code" button after all the components had been finished and tested.

The generated `Compile` class finally was linked into a UVC assembler application using the following code snippet:

```
Compile compiler = new Compile();

compiler.setObserver(this);        // to receive errors or acks
compiler.setInput( tabbedPanes.get(i).getText() );

compiler.run();                    // let it run

long l = compiler.getLength();     // helpful modifications grant ...
byte[] currentStream = compiler.getResult();      // ... easy access
```

While the Java GUI shown in Fig. 14 must be implemented using standard Java tools, the editor's syntax highlighting is supported by again calling the scanner.

7. Conclusion: related work and future plans

The theoretical foundation for compiler–compilers like JACCIE was laid by Donald E. Knuth in his two seminal papers [12, 13] on LR parsing and attribute grammars. A standard text book on compilers and compiler–compilers is the well-known "Dragon book" [4]; as a characteristic example of a compiler–compiler, it describes the popular LEX/YACC system.

There are legions of compiler generation tools like YACC [14]. Indeed, its name is an acronym (chosen in 1978!) for "Yet Another Compiler–Compiler." One comprehensive list of production compiler-generating tools (comparing them by feature) is maintained on Wikipedia.[7] This list is split into four categories: the first comprises tools for regular languages, i.e., scanner

[7] http://en.wikipedia.org/wiki/Comparison_of_parser_generators.

generators; the second category lists tools for deterministic context-free languages, most of them LALR(1)-parser generators; the third category consists of "packrat" or recursive descent parser generators; generators in the fourth category support more general parsing techniques (like Earley's [15] or GLR [16]) and often do not distinguish scanning and parsing. Details on all kinds of parsing algorithms can be found in the comprehensive text book [17].

One advantage of the "conventional" tools from the second category is that a generator can decide whether the resulting parser is deterministic and, hence, will produce a unique syntax tree for *each* valid parser input. Also, these tools strictly separate parser specification and generated parsers; for recursive descent parsers and the parser combinators that have become popular recently, there is a tendency to replace the language specification (i.e., the context-free grammar) by the parser's code and, even worse, to insert the code for syntax tree evaluation into the parser as well. This mixing of different phases impedes modifications and should, therefore, be avoided. LR parsing is notoriously hard to understand for novices, and even experts may sometimes find it difficult to remove unwanted non-determinism from LALR(1) parsers by modifying their underlying grammars. To help novices and experts overcome these problems, we have created JACCIE: JACCIE's parser generator, therefore, produces deterministic (LL(1), LR(0), SLR(1), LALR(1), and LR(1)) parsers. Its scanning, parsing, and attribute evaluation phases are strictly separated.

While there are many compiler generation tools and also many demonstration tools that visualize certain aspects of the compilation process to be found in the Internet, there are relatively few systems that, like JACCIE, aim to combine the generation of *conventional* compilers with comprehensive debugging features. We are aware of only the following systems of this kind.

Early on, the YTRACC parse browsing system [18] was presented, which automatically instruments parsers produced by YACC, 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 report on experiences made in two consecutive compiler classes (of about 50 students each) where (instrumented versions of) YTRACC and YSHOW were offered for voluntary use in a multi-user Unix environment; and — arguing from the data thus obtained — they discuss different patterns of student behaviour in detail. Our own classes are much smaller and students work on their own private computers; hence, statistically relevant *quantitative* data would be very hard to collect. Yet, we learn most from how students solve problems using JACCIE. Their solutions and comments provide the *qualitative* feedback required to continually improve our tool; e.g., all the examples that come with JACCIE have been worked out as training exercises by the participants of one course.

An even earlier and more complete educational compiler generator than YTRACC, the Visible Attributed Translation System (VATS), is described in [19]. Like its 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., grammars must not contain left recursive rules) but it blends better with embedded attribute evaluation, because 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 are described. Some of the extensions to VATS, that were termed "under development", 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. All these features (and more) are provided by JACCIE.

More recently, a commercial system became available with characteristics rather similar to JACCIE. Sandstone Inc. have marketed the VISUALPARSE++ system described in [20]; it supports LALR parsing. Parse trees are visualized either in a 3D representation (where nodes are shown as spheres and are connected by poles) or in a 2D "stack" representation (which allows trees to be folded and unfolded), better suited for viewing large trees. As would be expected from a commercial product, VISUALPARSE++ offers a choice of five target languages (C, C++, Java, Delphi, and VB). There is, however, no attribute evaluator generator: VISUALPARSE++ consists (like many other tool sets) of a scanner generator and a parser generator only. Unfortunately, development of VISUALPARSE++ appears to have been abandoned.

A team of researchers from the university of Maribor together with international partners have developed the LISA [21] system, whose list of features is very similar to that of JACCIE: there is a number of parser generators, attribute evaluator generators, and visualization tools. Also, LISA supports aspect-oriented implementation of compiler components. From the homepage, the LISA tool can be downloaded as a Java `JAR` file. By running the tool, however, details could not be checked for want of suitable documentation: the tool's on-line help says that "on-line help is not yet available" and points to the LISA homepage for more information. There, in the reference manual, another pointer "For using LISA in graphic mode-IDE look at tool demo section of tutorial" directs readers to a PDF-document which contains a series of annotated screenshots that in no way suffice to explain LISA's GUI and, in particular, its many options. The facts that the homepage has not been updated for five years and that it is obviously incomplete indicate that the development of LISA has been abandoned, too.

Even more recently, Hielscher and Wagenknecht have created the tool set AtoCC[8]: a collection of loosely connected tools for generating and visualizing compiler components. A particularly nice feature for demonstration purposes is that finite

[8] http://www.atocc.de/.

automata can not only be visually edited but also can be animated. (Similar remarks apply to the VLex tool [22], which visualizes in detail the standard derivation of scanning automata from regular expressions, and to the visual construction of LL(1) and SLR(1) parsers with the JFLAP[9] demonstration tool, as shown in the JFLAP tutorial.) AtoCC also provides standalone visual editors for defining scanners and parsers. Likewise, T-Diagrams (as introduced by Niklaus Wirth) can be visually edited and generated. However, like for VisualParse++, there is no attribute evaluator generator.

The popular ANTLR (ANother Tool for Language Recognition) tool set[10] uses extended LL(k) parsing techniques as a more natural alternative to LALR(1) parsing. Still, as noted by Bovet and Parr in [23], ANTLR users need a sophisticated development environment that supports them by resolving sources of parser non-determinism. Such support is provided by AntlrWorks, a visual front end to the ANTLR tool. The AntlrWorks debugger is connected to ANTLR via a socket mechanism and "displays input streams, parser lookahead, parse trees, parse stacks, and ASTs as they change during recognition". In the Syntax Diagram window, non-determinism is shown to grammar developers by highlighting ambiguous paths. However, AntlrWorks and ANTLR are tailored to a special parsing strategy that is not used by other tool sets.

From the many tool sets that offer more rudimentary visualization and debugging features, we only mention JS/CC,[11] an LALR(1) parser generator that uses JavaScript both as implementation and target language and also offers an "online live installation" for online testing.

Summing up, we feel that among currently available grammar tool sets of its kind, Jaccie's combination of features – a full-fledged compiler–compiler including alternative (LL and LR style) parser generators and attribute evaluation, plus a plethora of visualization components, all smoothly integrated into one uniform debugging environment, and comprehensive documentation including many worked-out examples – is unique.

Still, there is a "wish list" of features that would help to further improve Jaccie and that we intend to include in future versions:

- output of automata and syntax trees in the dot format for *external rendering* with Graphviz[12]; likewise, XML output and input for *data exchange* with other tools;
- support for generating *error handlers*; this is particularly important for parsing, but would be a helpful extension of other components as well;
- flexible support for generating source code of compiler components not only in Java, but also in *other target languages*; this would allow users to integrate generated compiler components into their own software;
- providing more sophisticated and *efficient attribute evaluation* strategies, e.g., support for the OAGs (Ordered Attribute Grammars) defined in [24].

Acknowledgments

We would like to thank the four anonymous reviewers for their detailed and knowledgeable comments and for their many constructive and helpful suggestions to improve this paper.

Appendix. Supplementary data

Supplementary data to this article can be found online at http://dx.doi.org/10.1016/j.scico.2012.03.001.

References

[1] S. Heilbrunner, A parsing automata approach to LR theory, Theoretical Computer Science 15 (1981) 117–157.
[2] L. Schmitz, On the correct elimination of chain productions from LR parsers, International Journal of Computer Mathematics 15 (1984) 99–116.
[3] P. Klint, R. Lämmel, C. Verhoef, Toward an engineering discipline for grammarware, ACM Transactions on Software Engineering Methodology 14 (3) (2005) 331–380.
[4] A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools (2nd ed.), Pearson, Reading, 2007.
[5] C. Allauzen, M. Mohri, A unified construction of the Glushkov, Follow, and Antimirov automata, in: Mathematical Foundations of Computer Science 2006, in: LNCS, vol. 4162, Springer, 2006, pp. 110–121.
[6] R. McNaughton, H. Yamada, Regular expressions and state graphs for automata, IEEE Transactions on Electronic Computers 1 (9) (1960) 39–47.
[7] D.H. Jonassen, Integrating Learning Strategies into Courseware to Facilitate Deeper Processing, L. Erlbaum Associates Inc, Hillsdale, NJ, USA, 1988, 151–181.
[8] M. Tomita, An efficient context-free parsing algorithm for natural languages, in: A.K. Joshi (Ed.), Proceedings of the 9th International Joint Conference on Artificial Intelligence, IJCAI, Morgan Kaufmann, 1985, pp. 756–764.
[9] U.M. Borghoff, P. Rödig, J. Scheffczyk, L. Schmitz, Long-Term Preservation of Digital Documents: Principles and Practices, Springer-Verlag New York, Inc, Secaucus, NJ, USA, 2006.
[10] R.A. Lorie, A methodology and system for preserving digital data, in: JCDL '02: Proceedings of the 2nd ACM/IEEE-CS Joint Conference on Digital Libraries, ACM, New York, NY, USA, 2002, pp. 312–319. http://doi.acm.org/10.1145/544220.544296.

9 http://www.cs.duke.edu/csed/jflap/.
10 http://www.antlr.org.
11 http://jscc.jmksf.com.
12 http://www.graphviz.org.

[11] R.A. Lorie, R.J. van Diessen, UVC: A universal virtual computer for long-term preservation of digital information., Tech. rep., IBM Res. rep. RJ 10338. IBM, Yorktown Heights, NY. (2005).
[12] D.E. Knuth, On the translation of languages from left to right, Information and Control 8 (6) (1965) 607–639. http://dx.doi.org/10.1016/S0019-9958(65)90426-2.
[13] D.E. Knuth, Semantics of context-free languages, Theory of Computing Systems 2 (1968) 127–145. http://dx.doi.org/10.1007/BF01692511.
[14] J. Levine, T. Mason, D. Brown, Lex & yacc, 2nd ed., O'Reilly, 1992.
[15] J. Earley, An efficient context-free parsing algorithm, Communications of the ACM 13 (2) (1970) 94–102.
[16] A. Johnstone, E. Scott, G. Economopoulos, The grammar tool box: A case study comparing GLR parsing algorithms, Electronic Notes in Theoretical Computer Science 110 (2004) 97–113.
[17] D. Grune, C.J.H. Jacobs, Parsing techniques: A practical guide, in: Monographs in Computer Science, 2nd ed., Springer, Berlin, 2008.
[18] R. Furuta, P.D. Stotts, J. Ogata, Ytracc: a parse browser for Yacc grammars, Software - Practice and Experience 21 (1991) 119–132. http://portal.acm.org/citation.cfm?id=103815.103816.
[19] A. Berg, D.A. Bocking, D.R. Peachey, P.G. Sorenson, J.P. Tremblay, J.A. Wald, VATS - the visible attributed translation system, in: Proceedings of the 1985 ACM SIGSMALL Symposium on Small Systems, SIGSMALL '85, ACM, New York, NY, USA, 1985, pp. 70–81. http://doi.acm.org/10.1145/317164.317173.
[20] Visual parse++, Tech. rep., Sandstone Technologies (2001). http://visualparse.s3.amazonaws.com/pvpp-fix.pdf.
[21] M. Mernik, M. Lenic, E. Avdicausevic, V. Zumer, LISA: An interactive environment for programming language development, in: N. Horspool (Ed.), 11th International Conference Compiler Construction, CC 2002, in: LNCS, vol. 2304, Springer, 2011, pp. 1–4.
[22] A. Jorgensen, R. Economopoulos, B. Fischer, VLex: visualizing a lexical analyzer generator – tool demonstration, in: C. Brabant, E. Van Wyk (Eds.), LDTA'11 Language Descriptions, Tools, and Applications, Saarbrücken, Germany, March 26–27, 2011, ACM, New York, NY, USA, 2011.
[23] J. Bovet, T. Parr, ANTLRWorks: an ANTLR grammar development environment, Software–Practice and Experience 38 (2008) 1305–1332.
[24] U. Kastens, Ordered attribute grammars, Acta Informatica 13 (3) (1980) 229–256.