Getting Started:
Motivation, Tools, Materials

Lothar Schmitz

lothar@informatik.unibw-muenchen.de

August 2004
1. Why Syntax Tools?

Some believe syntax to be a very complicated topic, which is best left to specialists: Only those “nerds” that implement programming language compilers need to know what is going on inside these tools or how they are built using so-called compiler-compilers. The rest of us can simply use compilers in happy ignorance of their complicated internal workings ...

For computer scientists, however, syntax and programming tools that deal with syntax are practically ubiquitous:

- Every time a number is “read in” by a program, a syntax tool will check whether the input string actually qualifies as a valid number description. For a valid number string, the syntax tool will also compute the number’s internal value.

- Similar steps are performed by a web browser whenever an HTML document is to be displayed.

- Authors using \texttt{\LaTeX} to produce a paper will start the \texttt{\LaTeX} compiler over and over again to watch their progress in a dvi previewing window — the same steps again.

- And of course programmers use compilers all the time. Again, the translation process is based on the same sequence of steps.

In the above situations a passive understanding of syntax will do, i.e., you only have to understand syntax developed by others for numbers, HTML or \texttt{\LaTeX} documents or the syntax of the programming language you use. As a user you do not have to know more!

Computer scientists often find them in situations where they have to develop syntax and syntax processors themselves, i.e., where an active understanding of syntax is required:

- Nowadays every one talks about XML and its potential as the “future language of the Internet” or as the common basis of E-Commerce or simply as a standard format for exchanging data. In every case you first have to decide on the syntax of your intended XML dialect by writing a DTD or an XML Schema — both are kinds of grammars. Technically, each XML application and each XML tool is based on a (validating) XML parser.
• If you want to convert large quantities of data from one format to another, you first have to specify precisely the syntax of both the source and the destination formats. The conversion job itself can then be carried out almost automatically by suitable syntax tools. A non-trivial special case is the conversion of old programs (so-called legacy software) into a modern programming language.

• You are developing a tool for generating programs and their documentation directly from UML models? Here, the syntax of the model description is a suitable starting point. Automated syntax tools will almost certainly be incorporated into your generator.

• Finally: If you intend to implement some language — no matter whether it is a programming language or a data base query language or some XML language like XPath or XSLT: For any one of these tasks syntax and syntax tools are indispensable!

Different tasks require different tools: E.g., for handling XML documents you can choose from different XML parser frameworks that are suitably parameterized and can be “plugged in” easily into your own application. For many tasks, script languages like Perl offer the most elegant way of syntax processing. In other situations a compiler compiler will be the most adequate tool.

All these tools are based on the same, well-known principles. So once we understand these principles we can determine which tool is suited best for the task at hand and how to proceed. Unfortunately, syntax tools are optimized for efficiency rather than ease of understanding and the underlying theory is typically explained in the special context of programming language implementation (with its intricate problems). In order to help you overcome these obstacles we offer

1. **visual compiler compilers** that give you plenty of feedback while you are developing your own syntax based programs (for more details see Our approach and the introductory tours of SIC and Jaccie);

2. a brief introduction into the **basics of syntax theory and tools** — outside the context of programming language implementation. Section What to find where contains pointers to our resources.

Our visual tools have been developed in a long series of Diploma theses and term projects. In section Who did it all? we give credit to contributions of the student developers.
2. Our Approach

Around 1960, compiler construction was one of computer science’s most important and successful research areas (in both theory and practice):

- The automatic translation of high level programming languages into machine code raised both the level of abstraction and — as a consequence — programmer productivity.
- The new techniques were firmly rooted on solid mathematical foundations from formal language theory.
- For the first time, programs were successfully applied as tools (compilers) for generating programs. Meta tools (compiler compilers) for efficiently generating those tools were to follow soon.

Below we see (on the right hand side) the internal structure of a compiler and how its components are generated from formal descriptions (left hand side) by corresponding generator components of the compiler compiler (middle):

![Compiler generated by a compiler compiler diagram]
Thus a compiler compiler generates the compiler components (scanner, parser, evaluator) from declarative language
descriptions (regular expressions and context-free grammars) and attribute evaluation rules.

Since compiler compilers were developed in “pre-Windows” times, they are typically lacking modern graphical user
interfaces. They rather come with a command line interface which is more suitable for automated (batch style)
production runs. Unfortunately, however, we also have to do without a graphic debugging tool like the ones we have
become accustomed to in programming. Therefore, the effects of even small changes to the declarative descriptions
can only be tested after a complete production run. This can be rather cumbersome.

In order to improve matters for developers (and for beginners, in particular) we have developed **visual compiler
compilers** which allow new languages to be developed and tested interactively. Our tools **SIC** and **Jaccie** are rather
simple, but fully functional compiler compilers. Their graphical interfaces were designed to support language designers
by giving access to all relevant internal information as soon as possible and by allowing to control and watch every
execution detail in debugging mode. The short tours of our tools in the following two sections will give you a first
impression.

Since compiler construction is a rather well-understood topic, research interests in our maturing field have naturally
moved on to other new and exciting topics. Compiler construction courses do not necessarily belong to the core
syllabus any more. As a consequence, compiler construction looks like a topic suitable for language implementation
specialists only. Its being close to formal language theory does not help to increase its popularity.

This is unfortunate for two reasons:

- As indicated above syntax and syntax tools continue to be important base technologies and, therefore, belong
  into every software developer’s bag of methods and tools.

- The statement “*Nothing is more practical than a good theory*” is demonstrated most convincingly in this field:
  Concepts and algorithms like pushdown automata and Myhill’s subset construction provided by theorists are
  put to immediate good use in the construction of efficient, deterministic parsers.

If you are in a great hurry proceed directly to section **What to find where**. There you will find our tools (complete
with documentation and example applications) and a concise paper entitled „*What every computer scientist should
know about syntax and syntax tools*“. After reading this paper you should be able to adequately choose and employ
those syntax tools that are required to solve your practical problems.
3. Using SIC

The "Smalltalk-based Interactive Compiler compiler" SIC visualizes information and processes that are relevant for developing and using compilers. SIC differs from other compiler compilers by providing the functionality of a debugger. This makes it an ideal tool for everyone who wants to “take a look inside” a compiler compiler in order to better understand compilers and their internal organization.

As a complete compiler compiler SIC provides three kinds of generators: Scanner generators, parser generators, and attribute evaluator generators.

SIC’s control panel displayed on the left hand side allows you to choose between different parser generators (for LL(1), LR(0), SLR(1), LALR(1) and LR(1) parsing modes) and between different attribute evaluation strategies.

Using the control panel all three components (scanner — parser — attribute evaluator) can be operated in a similar way: one can set independently a syntax definition (Current) and the component’s input string (Input). After pushing one of the Browse buttons a File Browser offers a suitable choice of existing files or, alternatively, lets you create a new file.

If box Interactive is checked, the component will be executed interactively in debugging mode. If box Browse Result is checked, the result will be shown in a separate window afterwards.

The radio boxes in the parser and attribute evaluation components let you select a suitable parsing or attribute evaluation strategy.

Using the four buttons in the bottom part of the control panel you can direct SIC to perform any two or all three compilation steps in succession.
Let us now apply SIC to the well-known grammar of arithmetic expressions, which is determined by the following set of production rules:

\[
\begin{align*}
\text{Expression} & \rightarrow \text{Term} \\
& \mid \text{Expression addOp Term} \\
\text{Term} & \rightarrow \text{Factor} \\
& \mid \text{Term multOp Factor} \\
\text{Factor} & \rightarrow \text{openingBracket Expression closingBracket} \\
& \mid \text{number} \\
& \mid \text{name}
\end{align*}
\]

A suitable scanner definition (details of the notation can be found in section 3 of our “Theory” paper) is:

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

In the following screen shot we see the scanner being applied to the arithmetic expression:

\[(\alpha + 17) \ast (\beta - \alpha /17)\]
The top subwindow shows the current input line, with red colour indicating those characters that have already been processed. The ‘Accumulated token’ subwindow shows the tokens that have been recognized so far. E.g., the 4th line says that the input characters 17 have been found to be a number.

In the bottom subpane (under ‘Last recognized token’) you see the token that is currently being accumulated. In the given situation, the fist letter b of the name beta has been processed. This is confirmed by the ‘Active token’ subwindow which shows that a name token is being recognized. The points in the regular expression are progress markers. All the other tokens are currently passive.

Using the arrow buttons, processing can be controlled — forward or backward, in steps of one character or one token or to the end. A completed token sequence automatically becomes the input of the next (parsing) phase:
The screen shot below shows the **parser** operating on the token sequence (top left subwindow) produced by the scanner from the input string \((\alpha + 17) * (\beta - \alpha /17)\)

Blue highlighting indicates that in the current situation all tokens up to the first `closingBracket` (misnomer for closing parenthesis) have been processed. The next token is `mulOp`.

The SLR(1) parsing strategy chosen here builds the syntax tree ‘bottom up’, i.e., from the leaves (bottom) towards the root (top) of the tree. In intermediate situations like the one shown here you see a sequence of subtrees. In subsequent steps these (and the remaining input tokens) will be ‘grown’ into a complete parse tree. Using the arrow buttons processing can be controlled like in the previous (scanning) phase.

Using the check boxes (bottom left) one can choose the amount and form of information displayed in any one parsing situation. In the screen shot above, the parsing stack and a graphical representation of the (partial) parse tree have been chosen. Another option is to make visible the parser’s internal control information (also called the ‘parser states’).
We now show how to translate arithmetic expressions into machine code that uses a data stack to determine the value of the expression. For our running example the instruction sequence to be produced is:

```
 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
```

The first two lines reserve storage cells for two variables, alpha and beta. MOVE instructions ‘push’ variables or constant values into a data stack. Arithmetic operation instructions (ADD, SUB, MULT, and DIV) take two operands from the stack and replace them by the result of the operation. This leaves the value of the expression on the stack.

**Attribute evaluation** is used to carry out the translation. For this purpose, a code attribute is attached to each non-terminal (Expression, Term, Factor) of the grammar. Attribute evaluation rules describe how attributes are evaluated in the context of a grammar rule. E.g., for the grammar rule

```
Expression -> Expression addOp Term
```

we have the attribute evaluation rule

```
codeExpression1 := stringaddOp = '+'
    ifTrue: [codeExpression2, codeTerm, ''] ADD W !SP+, !SP+, -!SP
    ']
    ifFalse: [codeExpression2, codeTerm, '] SUB W !SP+, !SP+, -!SP
    ']
```

This rule states that one obtains the code codeExpression1 for the complete expression by concatenating the code codeExpression2 for the left subexpression with the code codeTerm for the right subexpression. Depending on whether the addOp is a + or not either an ADD or a SUB instruction is appended to complete the code for Expression1.
As in the preceding phases, execution can be controlled using arrow buttons.
Here, too, internal information can be displayed in different forms:
In the screen shot above, evaluation is shown from a ‘bird’s view perspective’. Alternatively, you can ‘zoom’ to the current tree node to have a detailed view of attribute values.
The button Dependencies produces a graphical representation of the attribute dependencies within the context of a grammar rule — as shown in the small window to the right.
The editor shown below is used to define attribute rules:

![Editor definition file editor on: F:\Programme\vwSi.4ne\bin\grammars\ARITH2ML.EDF]

Rules:
- Arithmetic -> Expression
- Expression -> Term
- Expression1 -> Expression2 addOp Term
- Factor -> openingBracket Expression closingBracket
- Factor -> number
- Factor -> name

Symbols:
- Expression1
- Expression2
- addOp
- Term

Attributes:
- variables
- code

Semantic rule:
- codeExpression1 := stringaddOp = '+'
  - ifTrue: [codeExpression2, codeTerm, 'ADD W ISP+, ISP+, ISP]
  - ifFalse: [codeExpression2, codeTerm, 'SUB W ISP+, ISP+, ISP']

Dependencies:
- variablesExpression1
- variablesExpression2
- codeExpression2

Edit help:
- codeExpression2
- stringaddOp
- codeTerm
- ifTrue: []
- ifFalse: []

Since writing language definitions is a laborious and error-prone task, special editors are provided for developing context-free grammars and for the definition of attributes and attribute evaluation rules on top of the context-free syntax. For the definition of scanner tokens, a simple ASCII text editor will suffice. The complex structure of the above editor window indicates how much support these special editors offer in order to help avoid erroneous definitions.

It is beyond the scope of this short introductory tour to list (let alone explain) all components and features of the SIC system. For details, take a look at the comprehensive SIC hand book.

Or, even better, install and try out SIC on your own!
4. Using Jaccie

The “Java-based compiler compiler in an interactive environment” Jaccie has similar features as SIC: Jaccie comprises a scanner generator and the same set of parser generators as offered by SIC. An attribute evaluator generator has been completed recently. Unlike SIC Jaccie also has a specialized scanner editor. Another very useful feature: Only a mouse click is required to generate the Java source code of generated compiler components. The code is displayed in a text subwindow and can be copied easily into other applications.

For ease of comparison, we again use arithmetic expressions as a running example. Attribute evaluation is shown in the Jaccie handbook.

The first screenshot shows the new scanner editor which helps to avoid errors in the scanner definition:

\[
\begin{align*}
\text{Pattern} & : = (\text{<addOpPattern>} | \text{<digitPattern>} | \text{<numberPattern>} | \text{<namePattern>} | \text{<letterPattern>} | \text{<digitPattern>} | \text{<namePattern>} ) \text{ [0-9]}
\end{align*}
\]
Here we see the Java source code generated from the scanner definition:

```java
public class ScannerJava {
    public static void main(String[] args) {
        Scanner scanner = new Scanner();
        scanner.add(new Token(Tokens.NUMBER));
        scanner.add(new Token(Tokens.IDENTIFIER));
        scanner.add(new Token(Tokens.OP));
        scanner.add(new Token(Tokens.OPERATOR));
        scanner.add(new Token(Tokens.NEWLINE));
        scanner.add(new Token(Tokens.COMMENT));
    }
}
```

In the bottom half of the Java text an automaton for the `addOp` pattern is created. Then the states and transitions between states are added one by one.

In a similar way, automata for all token classes are generated.

Let us again look at a scanner in operation:
The SIC and Jaccie scanner windows are almost identical. Again red input symbols are those that have been processed, and again accumulated tokens appear in the right hand subwindow. There is, however, a small difference: Syntactically irrelevant symbols (spaces, tabs, newlines) are grouped into JACCIESeparator tokens and appear temporarily among the other tokens. Before the token sequence is passed on to the parser, the JACCIESeparator tokens are stripped off.

The other Jaccie scanner subwindows also correspond to their SIC scanner counterparts. Again, the active token subwindow shows that a name token is about to be recognized, and all the other tokens are passivated.

The arrows buttons have been replaced by nicer — and more intuitive — graphic controls.
Here we have one of the Jaccie information windows. It displays information derived from the user defined context-free grammar (ε variables, first and follow sets, useless symbols). This information is 'clickable', i.e., explanations are available via mouse click.

In the above screen shot the derivation of the set follow(Factor) is explained in more detail.
This screen shot shows the Jaccie parser in a situation where it has just successfully finished its job:

In the largest subwindow we see (part of) the complete syntax tree. As in the SIC parsing window there are subwindows for the input token sequence and the current stack. The two additional Jaccie parser subwindows (in the top) show the current parser state (top right) and its position (in red) within the parsing automaton (top left).

On the left hand side, we again have nice graphic controls instead of arrow buttons. Also, one mouse click will switch off all subwindows except the tree representation.
In another information window you can find out details about the parsing automata derived from a grammar (this includes parsing conflicts, too):

In the screenshot above, „Parser Info“ on a grammar for arithmetic expressions is shown: Since its LR(0) automaton has conflicts, the grammars is not LR(0). We have selected the third conflict to see both the parser state containing the shift reduce conflict (top middle) and the conflicting items (top right hand side).

In the same window you can view and traverse all the states of the LR(0) automaton: just select the radio button „Automata“ instead of „Conflicts“!
Again, only a mouse click is required to view the generated parser’s Java source code in a text window:

The best way to learn more about Jaccie is to install it and test it on your own!
5. What to find where

Our materials fall into three **Categories**:

**Tool-independent documentation**
- the overview you are just reading
- a short tutorial on the theoretical foundations of our tools
  *“What every computer scientist should know about syntax and syntax tools“*
  (recommended reading if you seriously consider to use syntax tools)

**SIC-specific materials**
- a comprehensive handbook
- the SIC system (Smalltalk-Parcels, to be loaded into VisualWorks Smalltalk)
- a lot of examples showing how SIC can be used

**Jaccie-specific materials**
- a short handbook
- the Jaccie system (executable Java jar-file)
- a lot of examples showing how Jaccie can be used

All materials are available on the download page!
6. Who did it all?

In our institute, the development of compiler-compilers has a long tradition starting in the late 70ies with the coco system which was written in PL/1 and served as a testbed for practically exploring parser optimization techniques (like chain production elimination from LR parsers).

When experimenting with Smalltalk and graphical user interfaces 10 years later, the idea was born to develop an interactive compiler compiler with extensive debugging facilities for teaching purposes. That was the beginning of a long-lived development effort. The implementation was done by students in a series of term projects and Diploma theses:

1989 GrammarBrowser (term project, J. Kröger)
1990 Interactive Parser Generator (Diploma thesis, J. Kröger)

At this point we won the Software Award of the German Academic Software Cooperation which was presented by German Federal Minister R. Ortleb at the CIP-Kongress 1991 in Berlin.

1992 Standalone components for SIC (term project, V. Niespor, E. Zschau)
1992 Attribute evaluation and tree transformation (Diploma thesis, O. Dörre, F. Gräfe)
1993 Improved tree representation (term project, D. Kassebaum, O. Laduch)
1993 A scanner generator (term project, M. Worch, S. Zimmermann)
1994 An editor for attributed syntax trees (term project, F. Schulter, J. van Laak)

Thus in 1995 the final version (called SIC95) of the Smalltalk-based development line was completed.
The different versions of SIC were spread considerably and found many users, not only in Germany, but also in far-away regions like Siberia (Irkutsk) and even Mongolia. We have presented SIC at international conferences such as STACS 91 in Hamburg and OOPSLA 92 (Academic Symposium) in Vancouver, Canada.

In spite of these successes we had to overcome many problems:

- When we started the required Smalltalk licenses were rather expensive. Meanwhile, Smalltalk vendors have changed their licensing policy considerably: for non-commercial purposes, the VisualWorks platform we are using is available for free.

- The Digitalk Smalltalk version we started with was available on PC platforms only. Users asked for versions that could be used on Unix as well. That was the reason for porting SIC to VisualWorks which is available on PCs, Macs, and Unix workstations.

- Eventually, users complained that prior to using the SIC attribute evaluation mechanism they would have to learn the Smalltalk language with its strange syntax and low installation base. In academic circles, Smalltalk was criticized for being untyped.

Although we did not share these reservations we considered the new „Internet language“ Java an interesting alternative: Java’s excellent tool support, its comprehensive class library and the similarity of the Smalltalk and Java architectures (which both make use of a virtual machine and are, therefore, highly portable) made it an ideal alternative implementation platform. Thus in 1996 we started to reimplement our visual compiler-compiler in Java:

1996  Jaccie: A Java-based compiler-compiler (Diploma thesis, Ch. Reich, V. Seibt)

2002  An applet version of the Jaccie scanner generator (term project, N. Krebs)

2003  Attribute evaluator components for Jaccie (Diploma thesis, N. Krebs)

Because of other activities, the resulting Jaccie system was completed only recently. It was first used by students in a course on compiler-compilers in January through March 2004 (and proved to be very stable in that test).