

Jaccie-Handbook

(Version 1.1)

Nico Krebs and Lothar Schmitz

lothar.schmitz@unibw.de

28. October 2011

Title Page



Page 1 of 99

Back

Full Screen on/off

Close

Quit

This handbook describes the compiler compiler **Jaccie** (acronym for 'Java-based compiler compiler with interactive environment').

Reasons why one needs compiler compilers and situations where you can profitably apply them yourself are shown in the first section **Why Syntax Tools?** of an accompanying paper (in file `introductionScreen.pdf`).

The first section **Example and Overview** of another accompanying paper (in file `theoryScreen.pdf`) explains how compiler compilers work in principle. The paper goes on to describe the components of compiler compilers and their underlying theory in some detail. **This is the background you need for seriously applying these tools.** (A hint: Hyperlinks can be traced back - even across document borders - by using the *Back* buttons.)

The handbook is organized as follows:

- Section **Jaccie-Tour** gives you a first impression of how **Jaccie** works. This essentially is a sequence of screenshots with a few explanations in between. As a running example we use an evaluator for simple arithmetic expressions that consist of numbers, operators, and variables.
- Section **Tool architecture** describes the structure of **Jaccie** and the components it is composed of. The three main phases of translation have a very similar structure and are, therefore, controlled using a common set of GUI controls. The common elements are explained in this section once and for all.
- Like other compiler compilers **Jaccie** generates compilers that operate in three successive phases: **Scanning**, **Parsing**, and **Attribute evaluation**. The corresponding compiler components are generated by **Jaccie** from formal descriptions (regular expressions, contextfree grammars and attribute evaluation rules). The three phases are covered in three sections where we describe both the *special editor* that is used to produce and modify the formal description and the *debugging environment* that is used for testing the component generated from the formal description.
- In section **Getting started** you are finally given a simple task for gaining hands-on experience with the tools. **Since working with compiler compilers is not a trivial task, we recommend that you at least skim through the preceding sections in order to avoid frustration!**
- Finally, section **Building Compilers** describes how to assemble the pieces of Java source code generated by **Jaccie** into a **standalone compiler**.

In future versions of this handbook, an additional section will describe how to set up Web pages with **Jaccie** scanner-, parser- and evaluator *applets* for demonstration purposes.

Jaccie-Tour

Tool architecture

Token Recognition

Grammars and Parsing

Attribute Evaluation

Getting Started

Building Compilers

Title Page

◀◀

▶▶

◀

▶

Page 1 of 99

Back

Full Screen on/off

Close

Quit

1. Jaccie-Tour

We¹ demonstrate [Jaccie](#) by producing a compiler that analyzes and evaluates **arithmetic expressions**. (As indicated in [section 5](#) of the theory paper you can essentially in the same way translate the expression into machine code or build a Kantorovic tree representation of the expression.) In order to make the example more interesting, we allow the expressions to contain variables. The user will be prompted to provide concrete values for the variables. And instead of simple Java number representations we use (reduced) fractions having each a numerator and a denominator (fractions are implemented using string representations).

E.g., for the expression

```
123 * x / ( alpha + 565767)
```

and the variable values $x = 20$ and $\alpha = -560067$ we would expect the compiler to compute somehow along the following lines

```
123 * 20 / ( -560067 + 565767)
= 2460 / 5700
= 41 / 95
```

In order to produce the **analysis part of a compiler**, we provide [Jaccie](#) with a formal description of the syntax of arithmetic expressions. Arithmetic expressions contain number constants, variable identifiers (names), parentheses, and operator symbols. For [Jaccie](#), these 'patterns' are specified by regular expressions as follows:

```
<addOpPattern>      := ($+ | $-)
<multOpPattern>     := ($* | $/)
<openingParenthPattern> := $(
<closingParenthPattern> := $)
<letterPattern>     := ({ $a-$z } | { $A-$Z })
<digitPattern>      := { $0-$9 }
<namePattern>       := <letterPattern> (<letterPattern> | <digitPattern>)*
<numberPattern>     := <digitPattern>[1-*]
```

¹The Jaccie-Tour was developed by the tree students, who have implemented [Jaccie](#) as part of their Diploma theses: Christoph Reich, Volker Seibt and Nico Krebs.

In the **regular patterns** the 'ASCII symbols you mean' are prefixed with a Dollar symbol. Alternatives are separated by vertical bars. Symbol sequences (like, e.g., alternatives) are grouped with parentheses. Thus an `<addOpPattern>` represents a single plus oder minus sign. A `<closingParenthPattern>` somewhat pompously describes a closing parenthesis symbol. The `<digitPattern>` consists of an interval which is enclosed in braces, meaning one of the ASCII symbols between 0 and 9, i.e., a single digit. At the end of the `<numberPattern>` we have (enclosed in brackets) a repetition clause which states that a number contains at least one digit (lower bound is 1) while the upper bound (*) allows any number of digits. More details will be given later.

The **token definition** below decides which of the above patterns are to be considered complete lexical tokens and by which name (left hand side) these tokens will be known (as terminal symbols of a grammar).

```

<addOp>           := <addOpPattern>
<multOp>          := <multOpPattern>
<openingParenth> := <openingParenthPattern>
<closingParenth> := <closingParenthPattern>
<name>            := <namePattern>
<number>          := <numberPattern>

```

Note that letters and digits are considered *incomplete* in this sense! All other patterns describe tokens: An `<addOp>` is defined by the `<addOpPattern>` and a `<number>` by the `<numberPattern>`.

Below we see the production rules of a **contextfree grammar** that defines the syntactical structure of arithmetic expressions. The rules contain *terminal symbols*, as defined by the tokens above, and *nonterminal symbols*. Each nonterminal appears on the left hand side of a production rule.

```

Expression -> Term
           -> Expression addOp Term

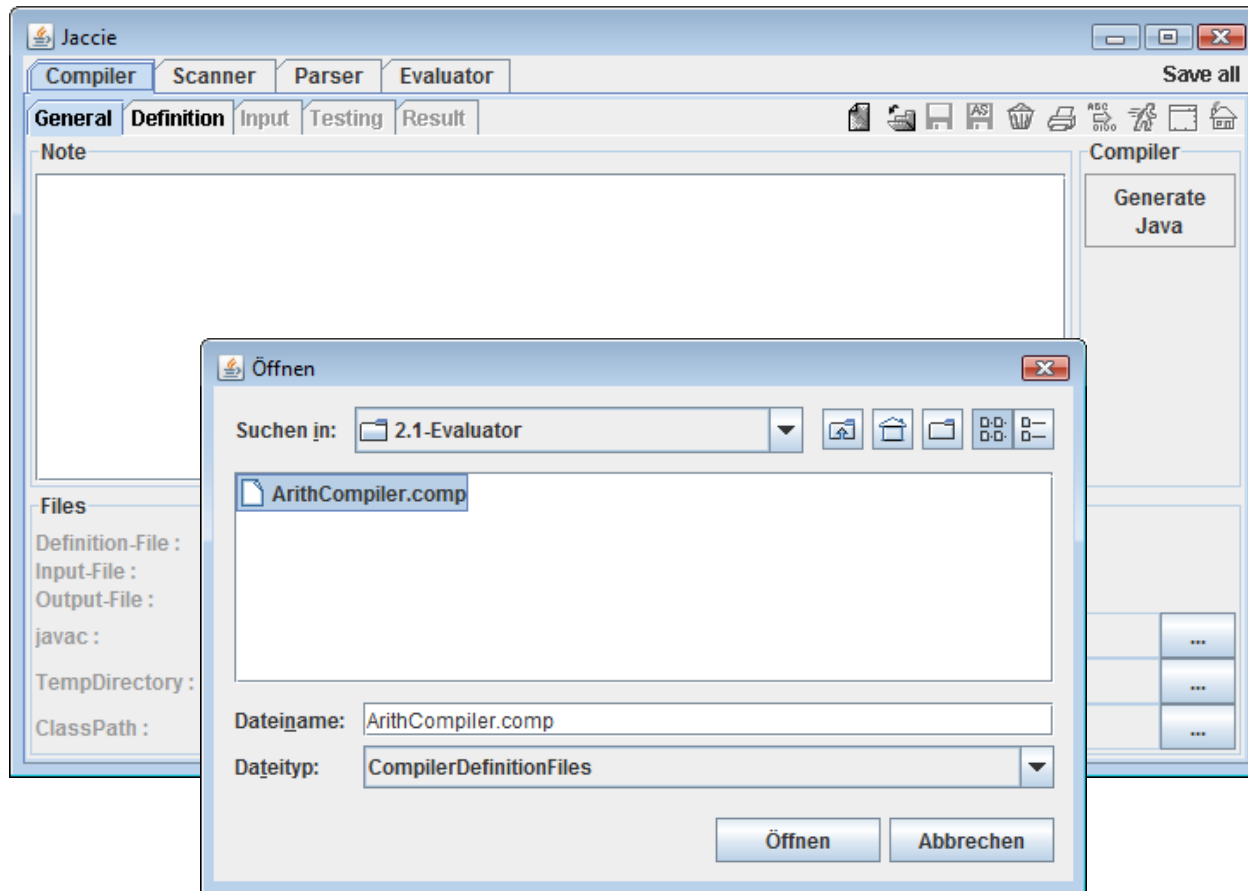
Term       -> Factor
           -> Term multOp Factor


Factor     -> number
           -> name
           -> openingParenth Expression closingParenth

```

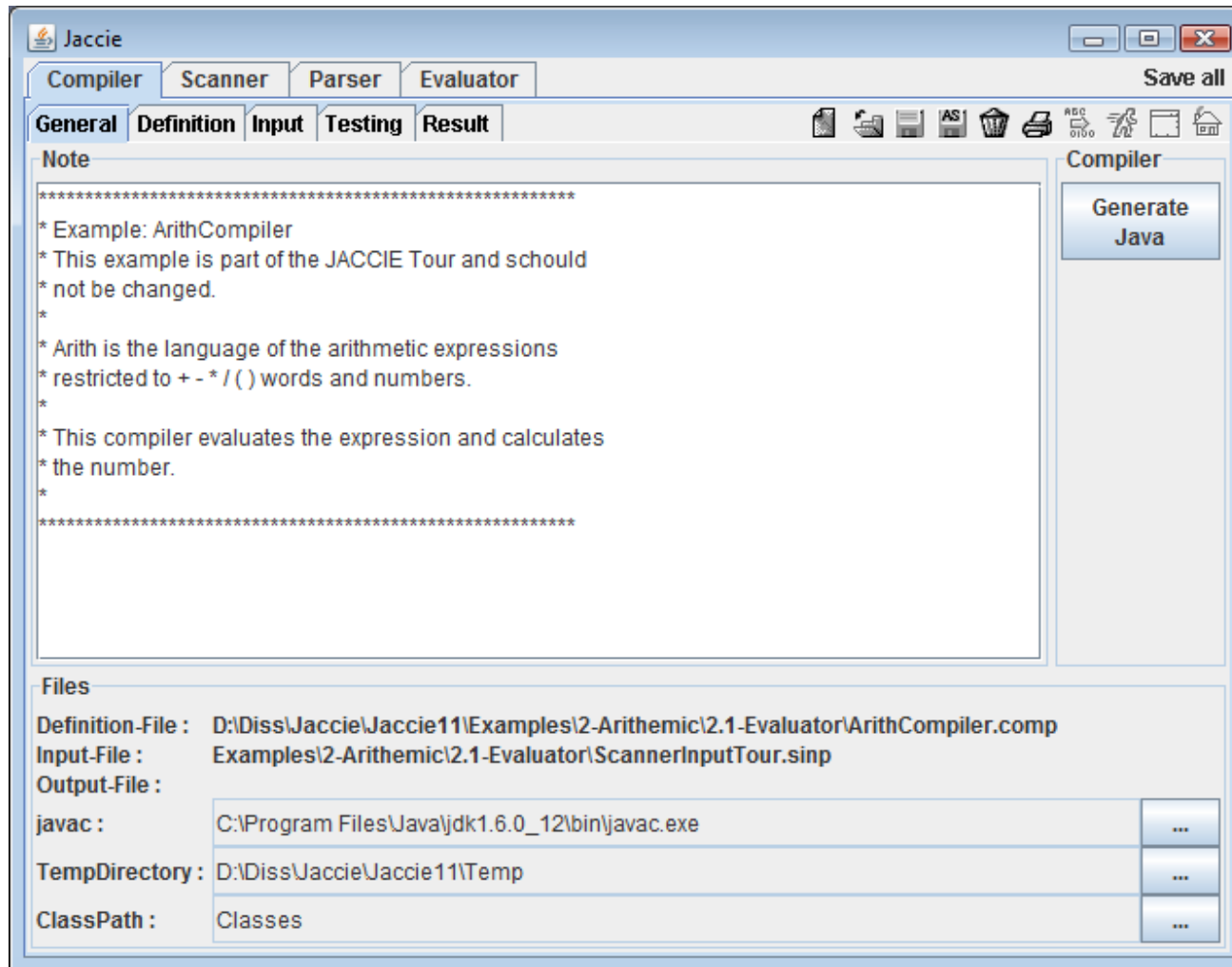
E.g., the three alternatives for nonterminal Factor state that a Factor either is a number or a name or an Expression enclosed by parentheses. Again, more details later.

Now, let us start [Jaccie](#):

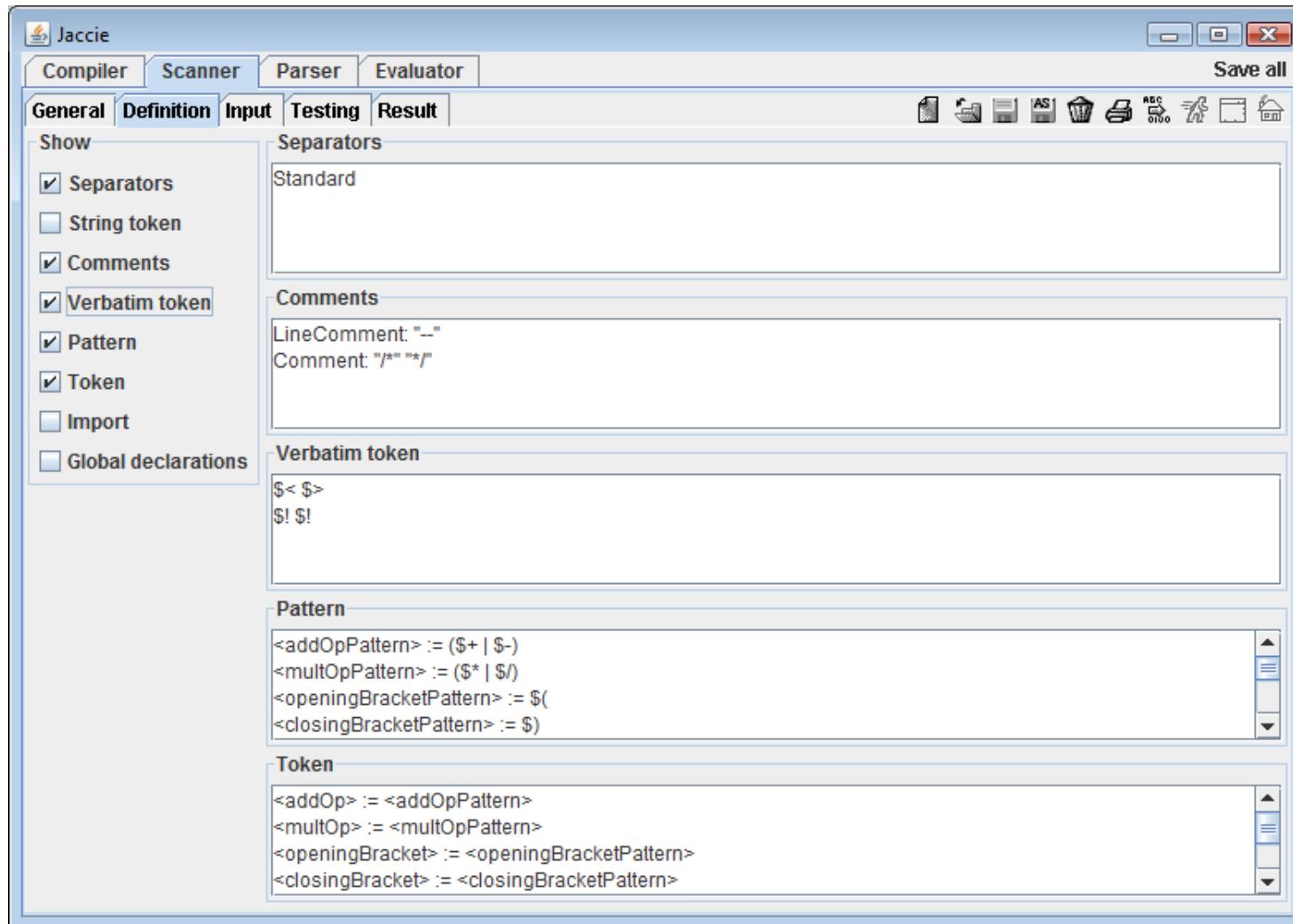


The layout and handling of the [Jaccie](#) GUI are explained systematically in subsequent sections of this handbook. Here, we simply open one of the compiler projects we have created. As usual, we click on an *open* icon () and thus start a file selection dialog where we select the ArithCompiler project.

As a side effect – as you can see in the lower part of the illustration – all required settings for the example have been loaded.

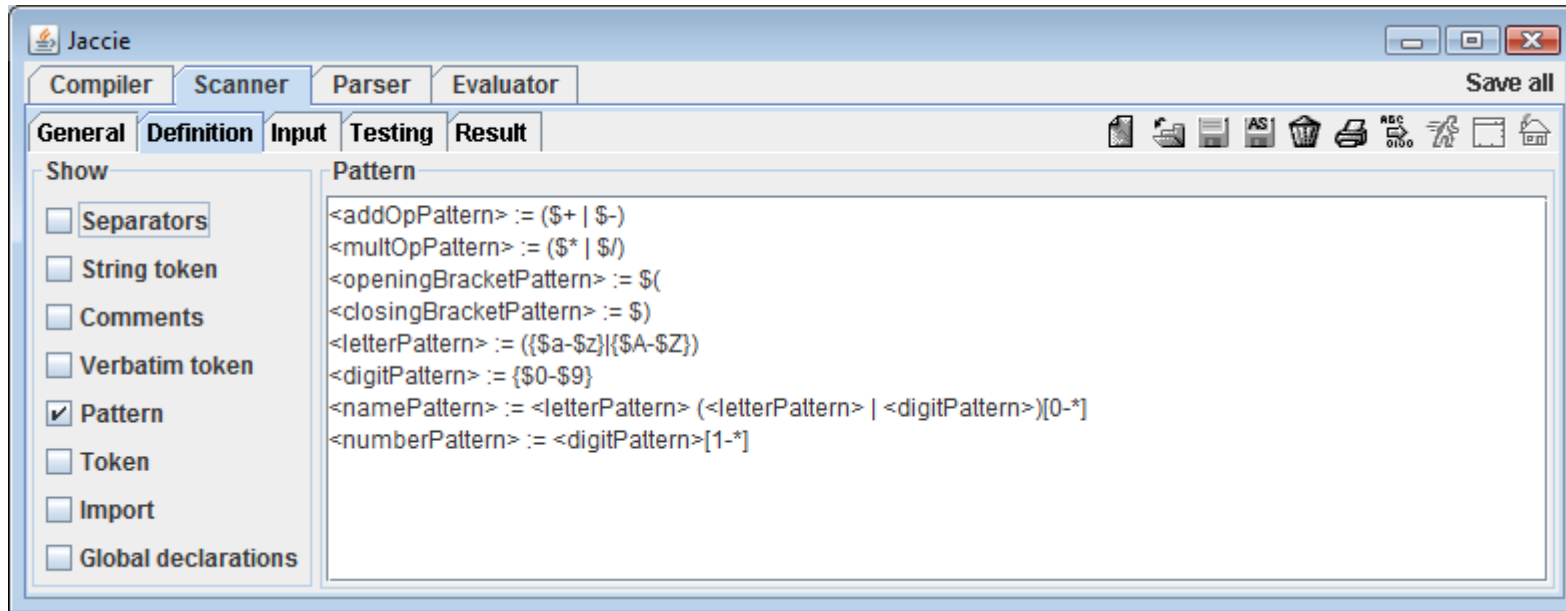


We first select the window area that is dedicated to lexical analysis (i.e. the **Scanner**). There, we find the pattern and token definitions described above:

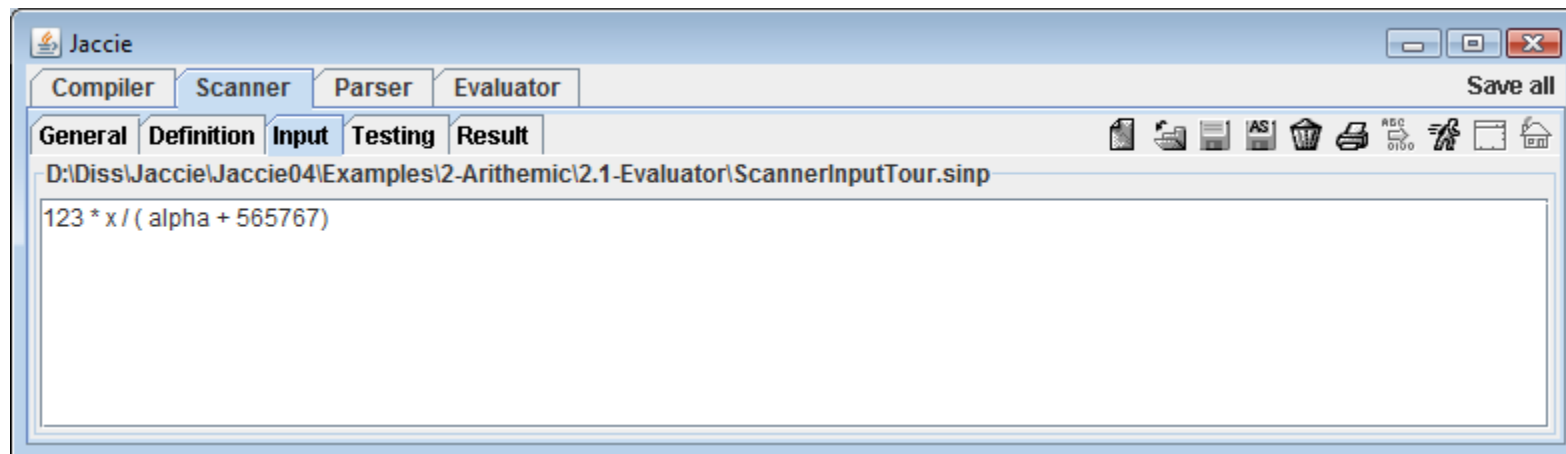



Here, we are shown an overview of the scanner definition. In order to see more details, we can deselect areas we are not interested in. We first have a look at the pattern definition.

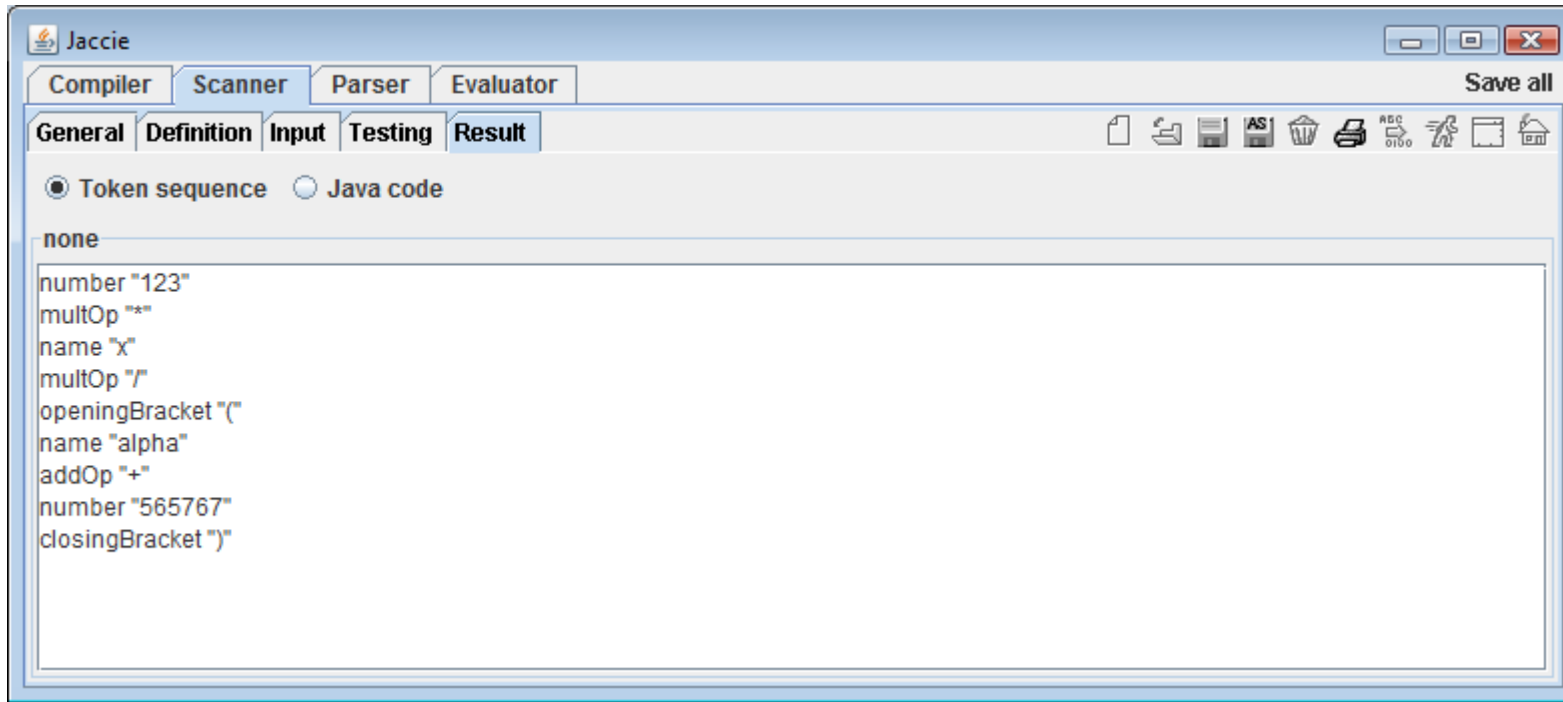
Obviously, these are exactly the patterns we have described above:



In the *Input* subarea we also find our example expression:

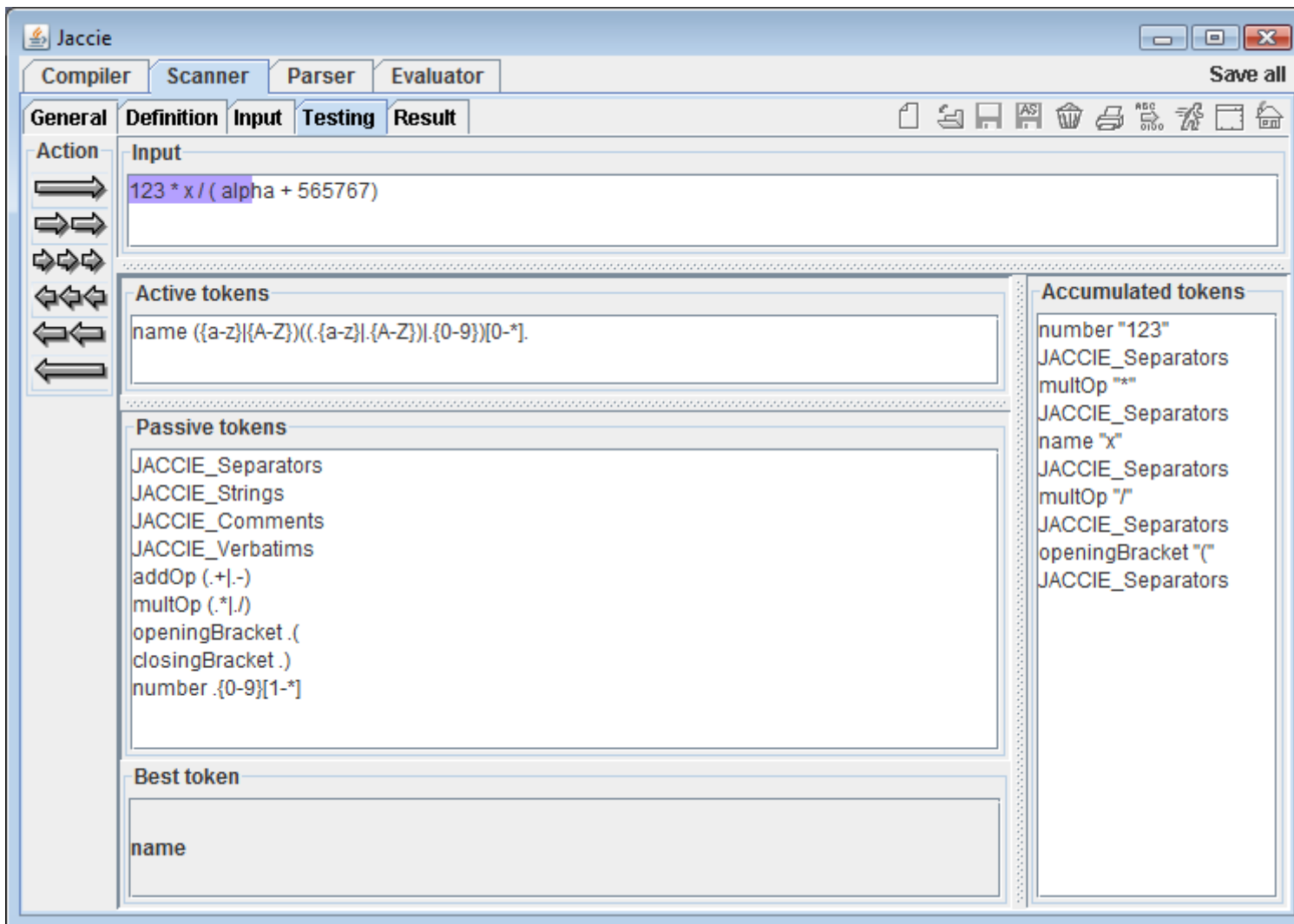


Using the *run* icon () we apply the scanner to this input. Then the expected token sequence appears in the *Result* subarea:



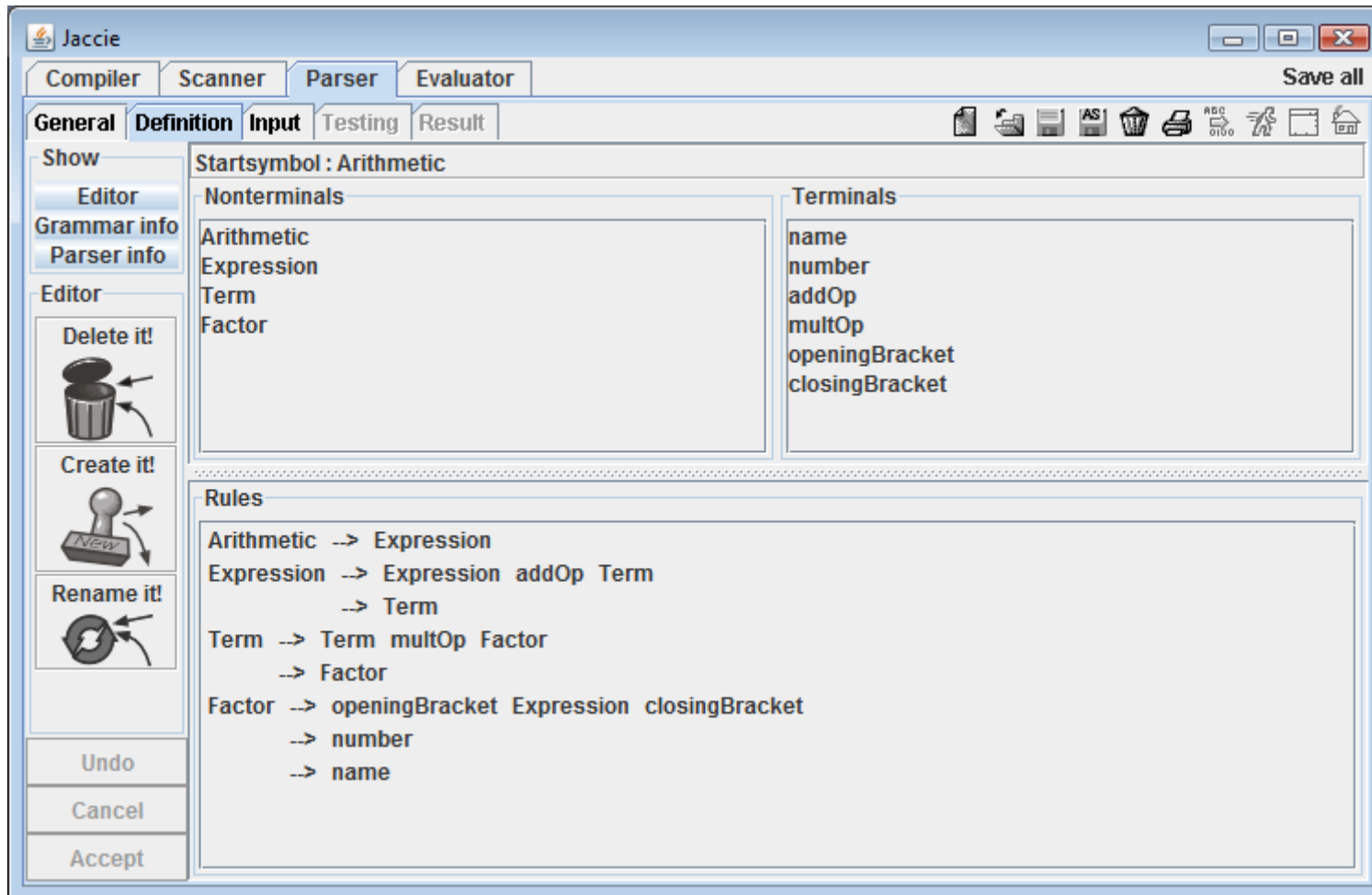
In the scanner result there is exactly one line per token. The token name is followed by the token text in double quotes. The complete expression starts with a (*number*), more precisely with 123. The first *multOp* is a multiplication symbol, the second one a division symbol. If you concatenate all the strings enclosed by quotes you obtain the original input text – generously ignoring blanks and other whitespace characters which are deemed irrelevant.

In order to watch the scanner at work, we use the *debugging* subarea (labelled *Testing*). Using the *Action* arrows on the left, you can direct the scanner to go in the direction (forward or backward) and at the rate (one input symbol or one token or to the end) you want it to go:

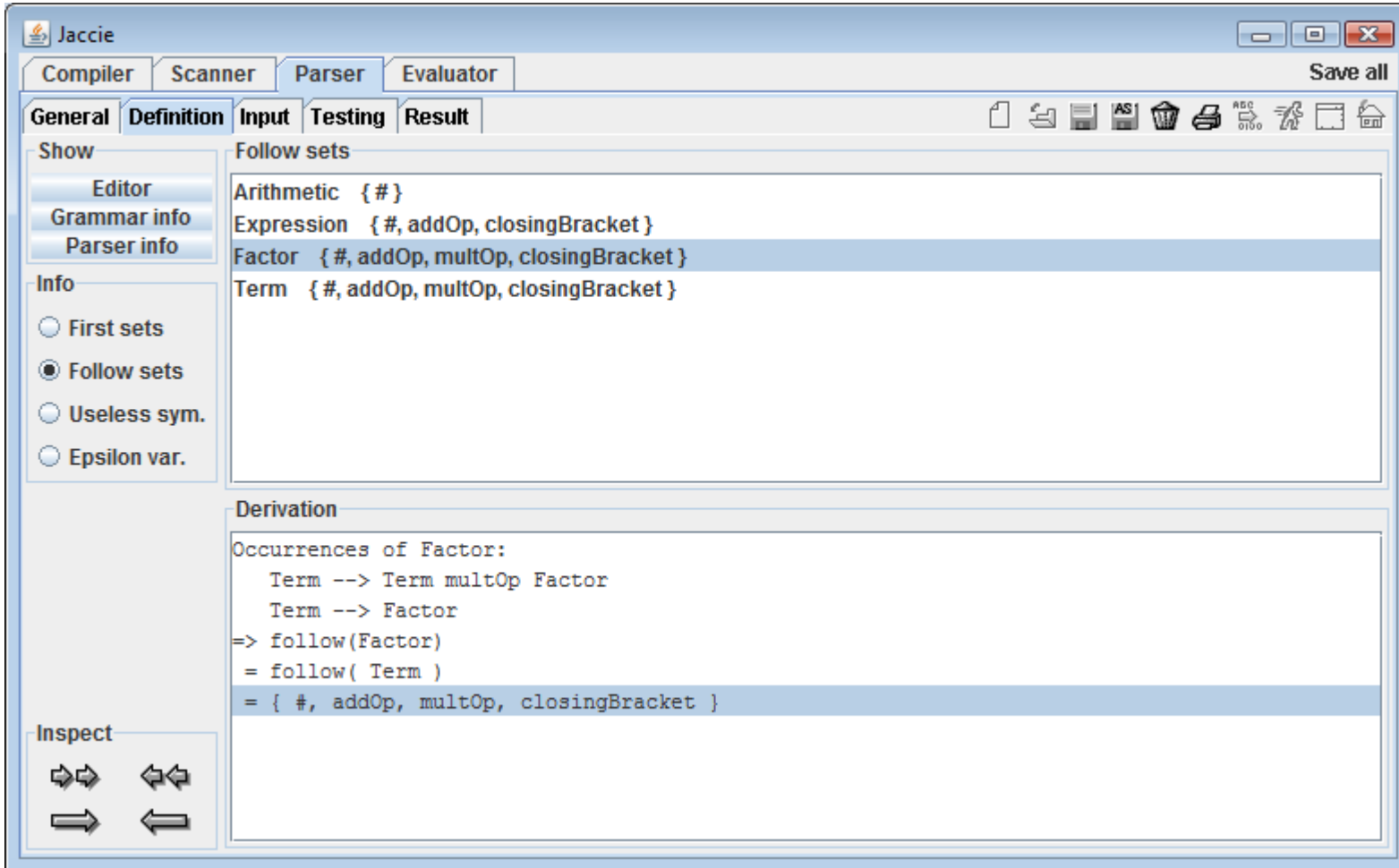


Highlighting (in blue) indicates that in the current situation the input sequence has been processed up to (and including) the third letter of the identifier `alpha`. In the *Active tokens* window pane you find the pattern defining the syntax of the `name` token being recognized. Dots within the regular pattern show all the currently active states of processing this pattern. Notice that within the pattern pattern names like `letter` and `digit` have been replaced by the patterns they denote. In the *Accumulated tokens* window pane the sequence of tokens recognized so far is shown. The `JACCIE_Separators` pseudotokens indicate irrelevant whitespace characters – they are shown for completeness but are stripped off before the token sequence is passed on to the parser.

In the next phase the token sequence is processed by the **Parser**. Choosing its register card, we move (from the *Scanner*) to the *Parser* area and, similarly, within that area to the *Definition* subarea where a special editor for grammars is available. As indicated by the graphical buttons on the left hand side, this editor allows for *direct manipulation* including *drag-and-drop*. Details will be explained in section 4.

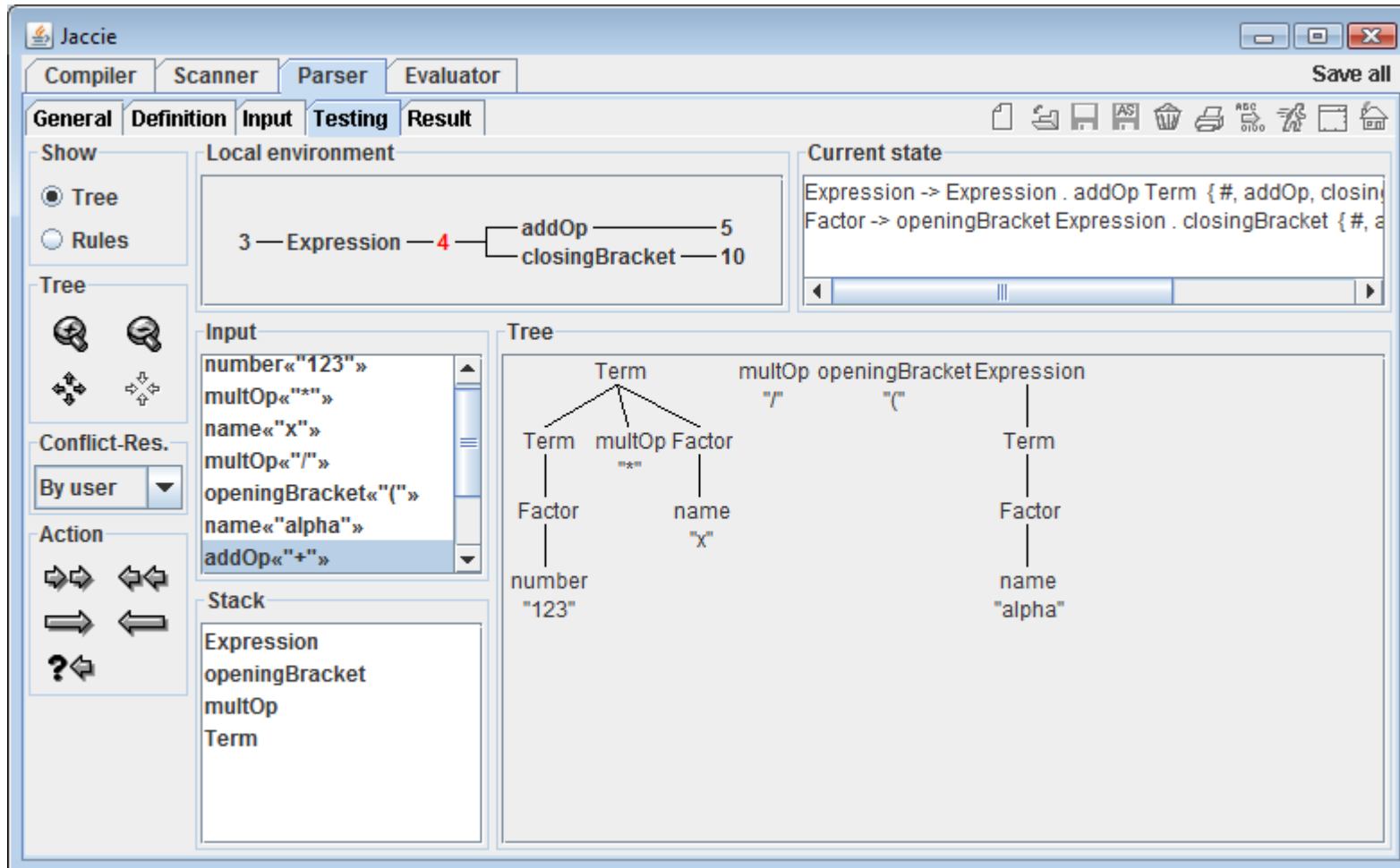


The entries in the *Show* panel allow you to view (any time) useful informations that are derived automatically from the grammar (including first / follow sets and parsing automata). Let us have a look at that!



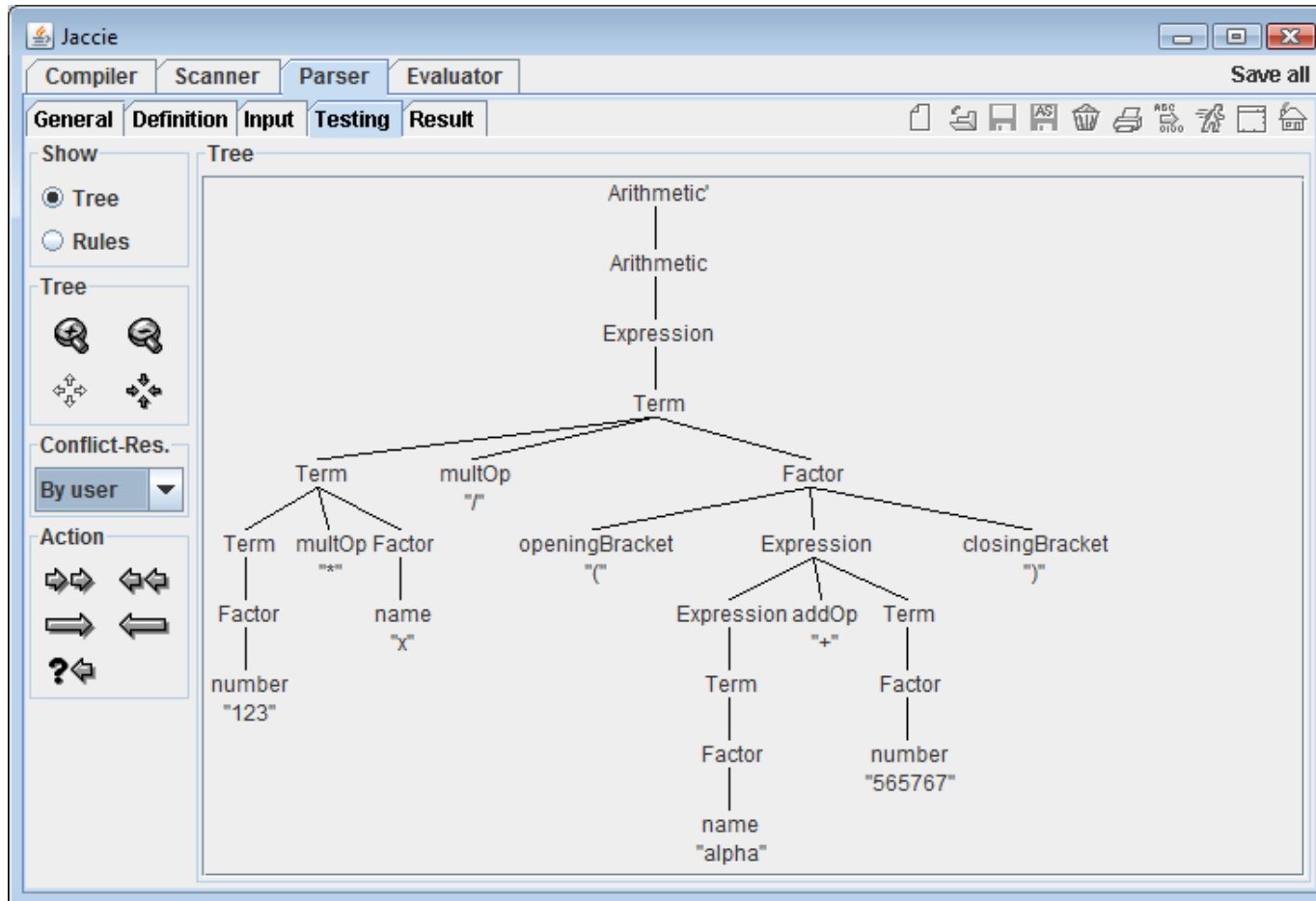
Here, the so-called *follow sets* are shown: For each nonterminal X , $\text{follow}(X)$ is the set of all terminals that may follow X in any context (i.e., sentential form). In the lower subwindow, a stepwise *Derivation* of $\text{follow}(\text{Factor})$ is given which explains how the set was computed.

Like the *Scanner* area, the *Parser* area offers a debugging facility in its *Testing* subarea. Here, you can control the parsing process and view all internal informations used by parsing algorithms (input, parsing stack, parsing automaton):



In the top row that part of the parsing automaton is shown which is relevant for the next parsing step (current state with predecessor and successors on the left, items of the current state on the right). Below that you see the input (token sequence), the parsing stack and those parts of the syntax tree that have been recognized so far.

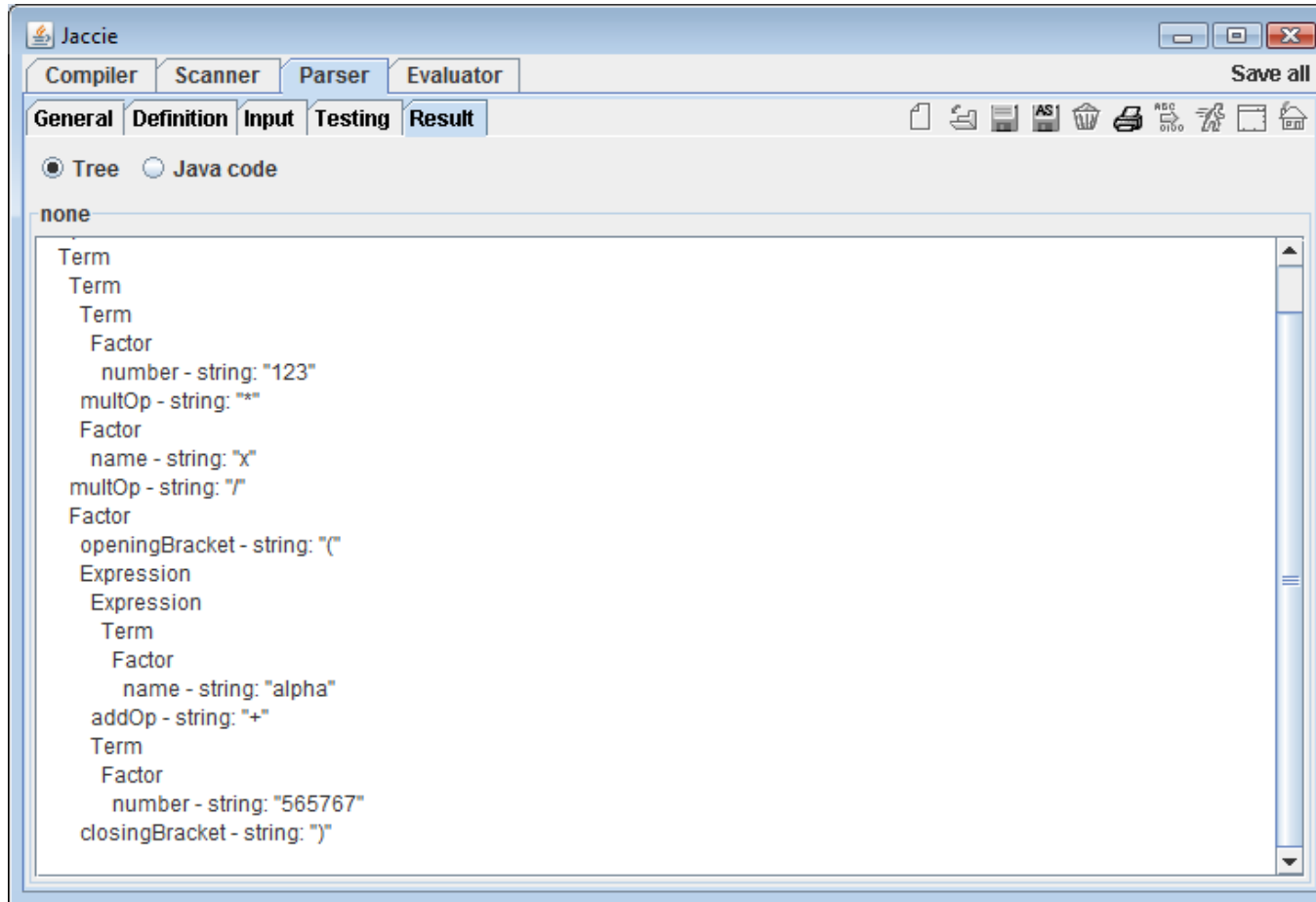
If that is information overload to you, you can concentrate on the tree being built by switching off everything else:



Watching closely, you will detect a slightly larger font in the tree. This is what you control by using the Zoom-in / Zoom-out (looking glass) buttons on the left hand side. Here, the syntax tree was made to just fit into its subwindow (this is possible for relatively small syntax trees only).

Notice, that the leaves of the tree – read from left to right – contain the token sequence that was produced by the scanner. The token texts (i.e. the character sequences from the original input) are shown as well.

In the *Result* subarea you will find a textual representation of the same syntax tree that was shown graphically in the *Testing* subarea:



As mentioned before, the leaves of the syntax tree contain chunks of the original character input sequence. These pieces of information are called *pseudo attributes*.

In the following **Synthesis**, essentially the inner nodes of the syntax tree are augmented systematically with attributes (different kinds of information pieces) in such a way that some root node **attributes** will contain the result of the desired translation. How that works will hopefully become obvious in the example below!

What **attributes** are useful for the evaluation of arithmetic expressions with variables?

- The purpose of this example is to compute the **value** of an expression. At the leaves, values are available as Strings in the pseudo attributes. We therefore require every inner node to have a value attribute of type String. Every inner node is the root of a subtree; its value is defined to be that of its subtree. Values are computed in steps starting at the leaves and proceeding towards the root of the syntax tree.
- For computing the value of an expression with variables we first have to ask the user for the concrete variable values. Variables occurring more than once in an expression always denote the same value. Therefore, we initially collect the **set of variable names** that occur in the expression using variables attributes: The variables attribute of a node contains the set of variables in its subtree. These attributes also are computed starting at the leaves and proceeding towards the root of the syntax tree.
- For actually evaluating the expression we need a variable binding, i.e. a set of (name, value) pairs. Since in programming language terminology this is called an **environment**, we use environment attributes. At the root node the environment is constructed by asking the user for concrete values of each element of the variables attribute. The environment attribute thus obtained is propagated into the tree, this time proceeding from the root towards the leaves.

Depending on the direction of evaluation attributes are called either *synthesized* (towards the root like value and variables) or *inherited* (towards the leaves like environment).

Using the **Attribute editor** on the next page, attributes are introduced with their type and direction. Also, they are assigned to a subset of nonterminals. This uniquely defines a set of attributes for each node of a syntax tree!

We will also use the attribute editor to define **attribute evaluation rule** that describe how an attribute value is computed either directly or from other attribute value at neighbour nodes. The details of the attribute editor GUI will be explained in section 5. Here, we only catch a few short glimpses of this rather complex tool.

The attribute types, the attribute evaluation rules and other, globally defined variables and subprograms are all written in the programming language **Java** which is also the language **Jaccie** is implemented in.

Title Page

◀◀

▶▶

◀

▶

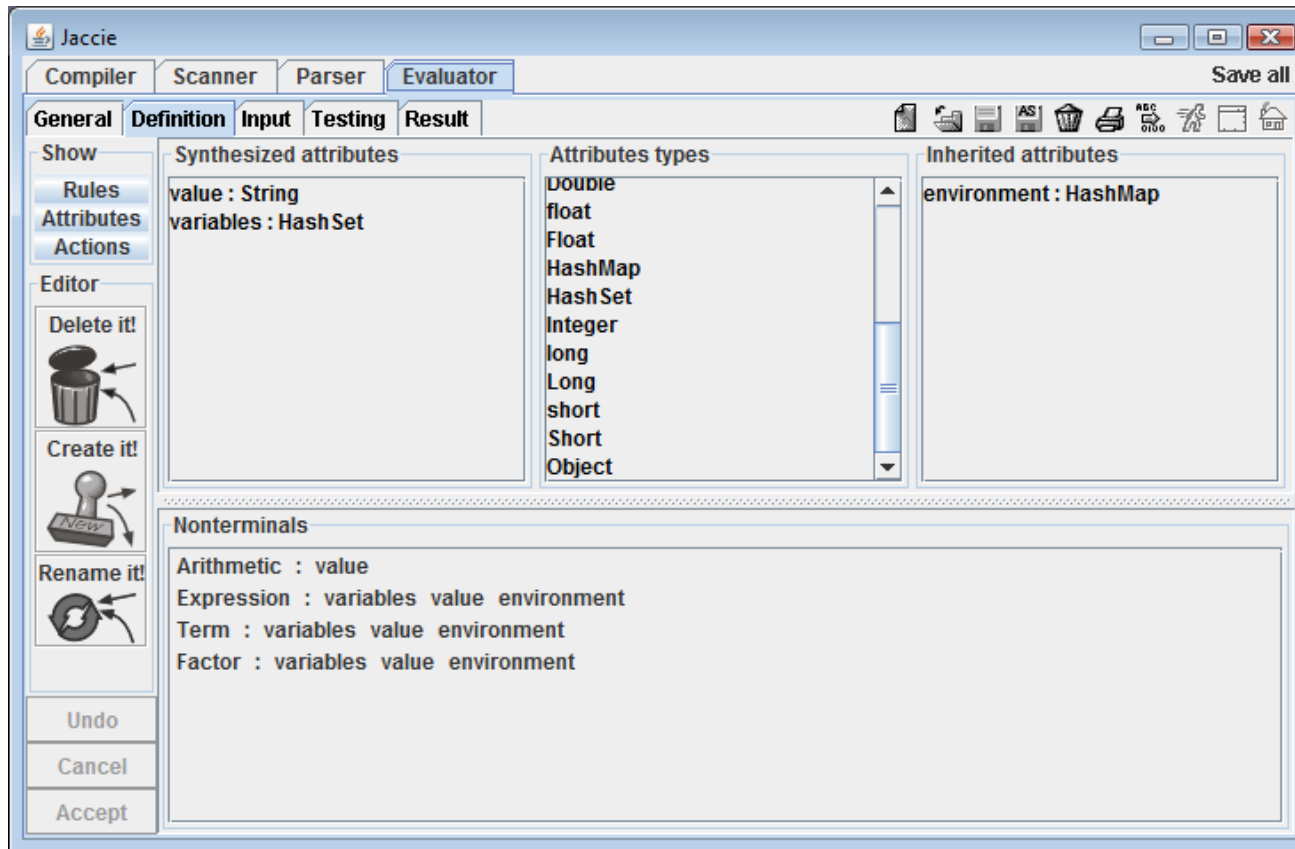
Page 15 of 99

Back

Full Screen on/off

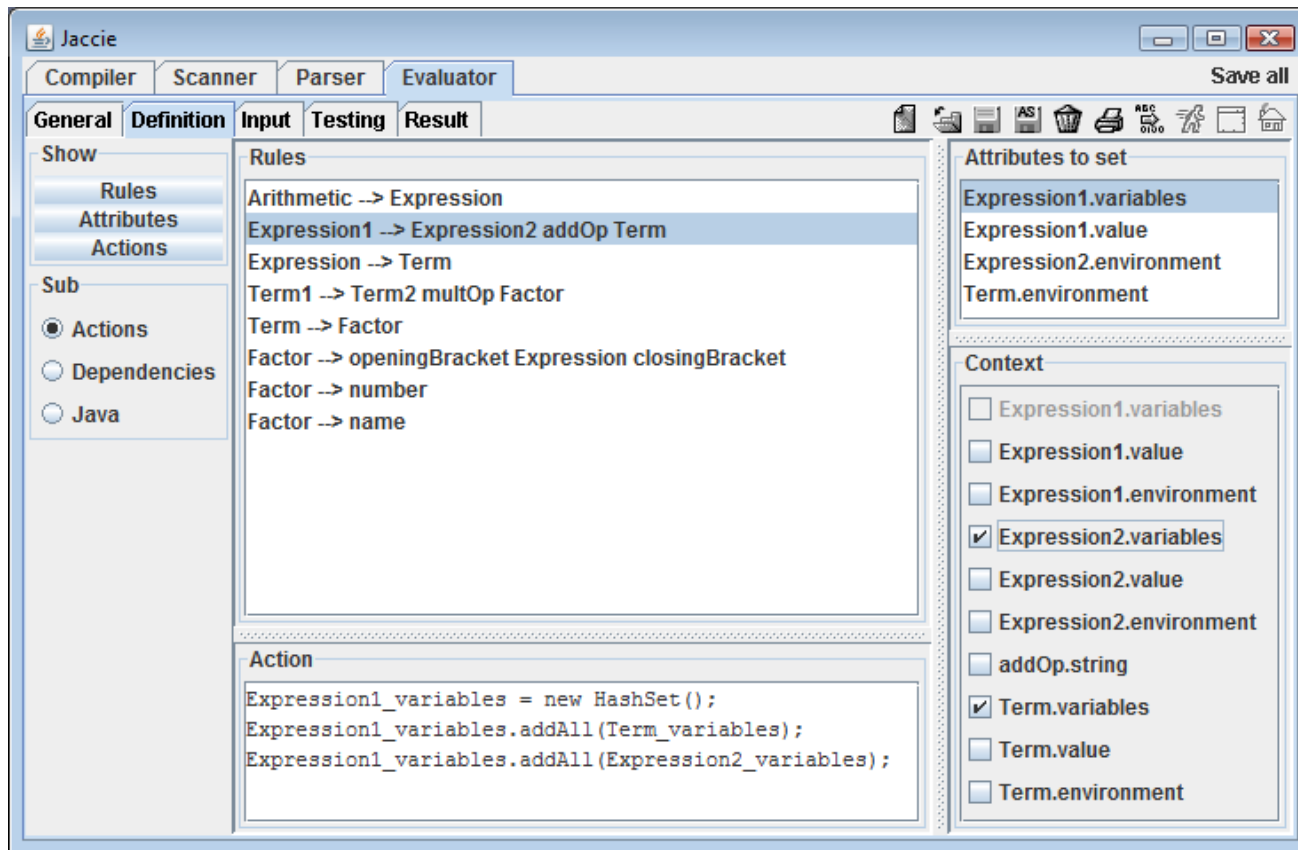
Close

Quit



In this editors you can comfortably (using *direct manipulation* with *drag-and-drop*)

- define and delete attributes and typen
- rename attributes
- assign Java types to attributes
- assign attributes to nonterminals

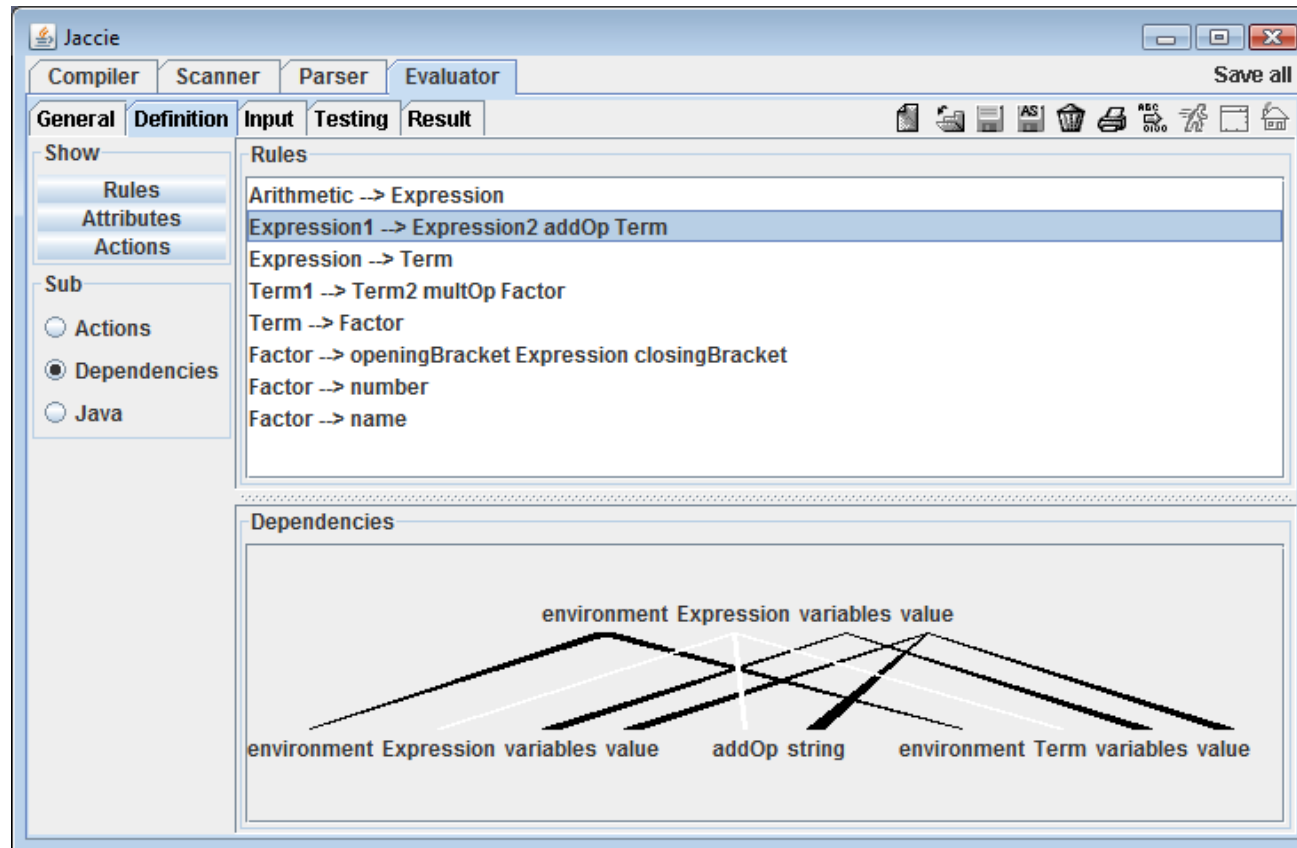


Here we see the attribute evaluation rule for `Expression1.variables` in the context of the production rule

`Expression -> Expression addOp Term`

Jaccie automatically distinguishes different occurrences of the same symbol using indices: `Expression1` therefore denotes the occurrence of `Expression` on the left hand side of the rule.

The bottom right panel lists all attributes that are available in the context of this production rule. Marks indicate that the attribute evaluation rule (i.e. the Java statements shown on the left) will use the attribute values `Term.variables` and `Expression2.variables`. This should be obvious since the `variables` of `Expression1` are obtained as the union of these two variable sets! (If you forget to mark attributes you want to use in an attribute evaluation rule, they will be undefined. This results in a Java compiler error as soon as you try to save the rule.)

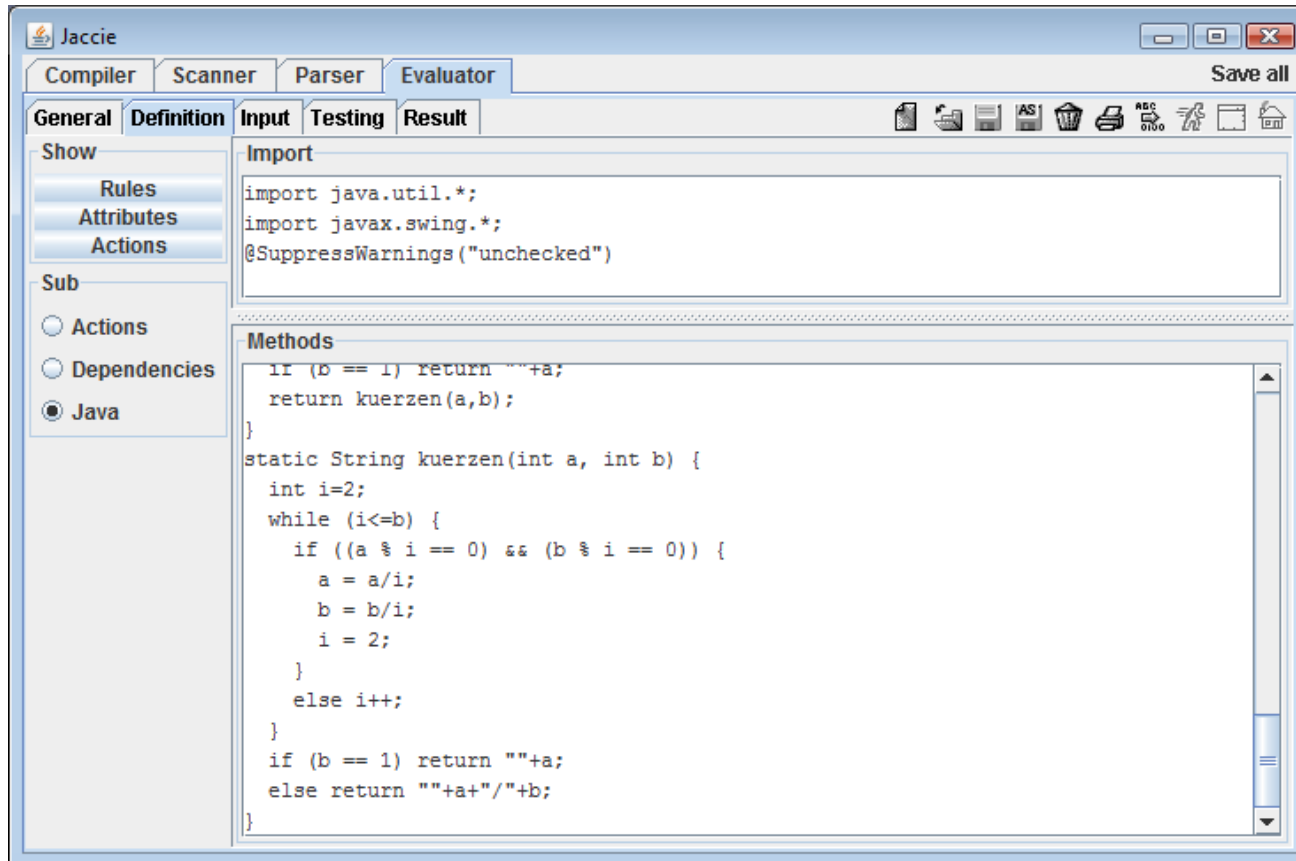


In the *Dependencies* subarea Jaccie visualizes all attribute dependencies (defined by marks as shown on the previous page) for one production rule. In general, these dependencies originate from different attribute evaluation rules. Here we see the dependencies for production rule

`Expression -> Expression addOp Term`

White edges link the left hand side of the production rule (top row) with every grammar symbol on the right hand side (bottom row).

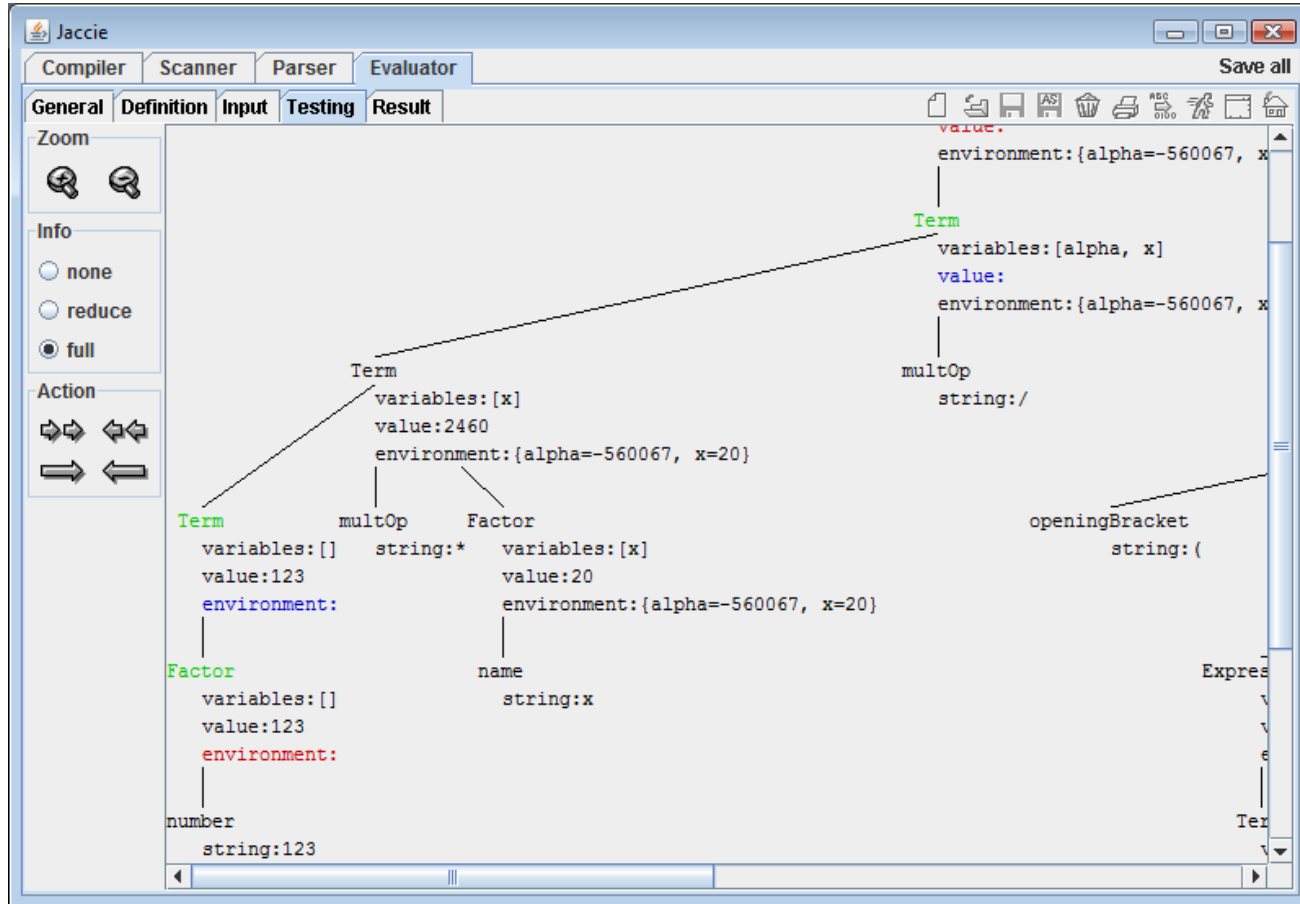
Black, pointed arcs represent attribute dependencies. As you can see, environment attributes are propagated top-down, all other attributes bottom-up. In other words, environment is an inherited attribute and the other attributes are synthesized.



In the *Java* subarea additional definitions are made available to the attribute evaluation rules. In this example, we have decided to use reduced (i.e. cancelled) fractions. Since this is not available in standard Java, we provide our own definitions: among other things, a method *kuerzen* (German for *cancelling*).

Classes from the Java class library are made available by ordinary Java import statements: Among other things, we import Swing classes, since when asking the user for concrete variable values we need a Prompter.

As for the preceding phases, there is a **debugger** for *Testing* attribute evaluation in detail:

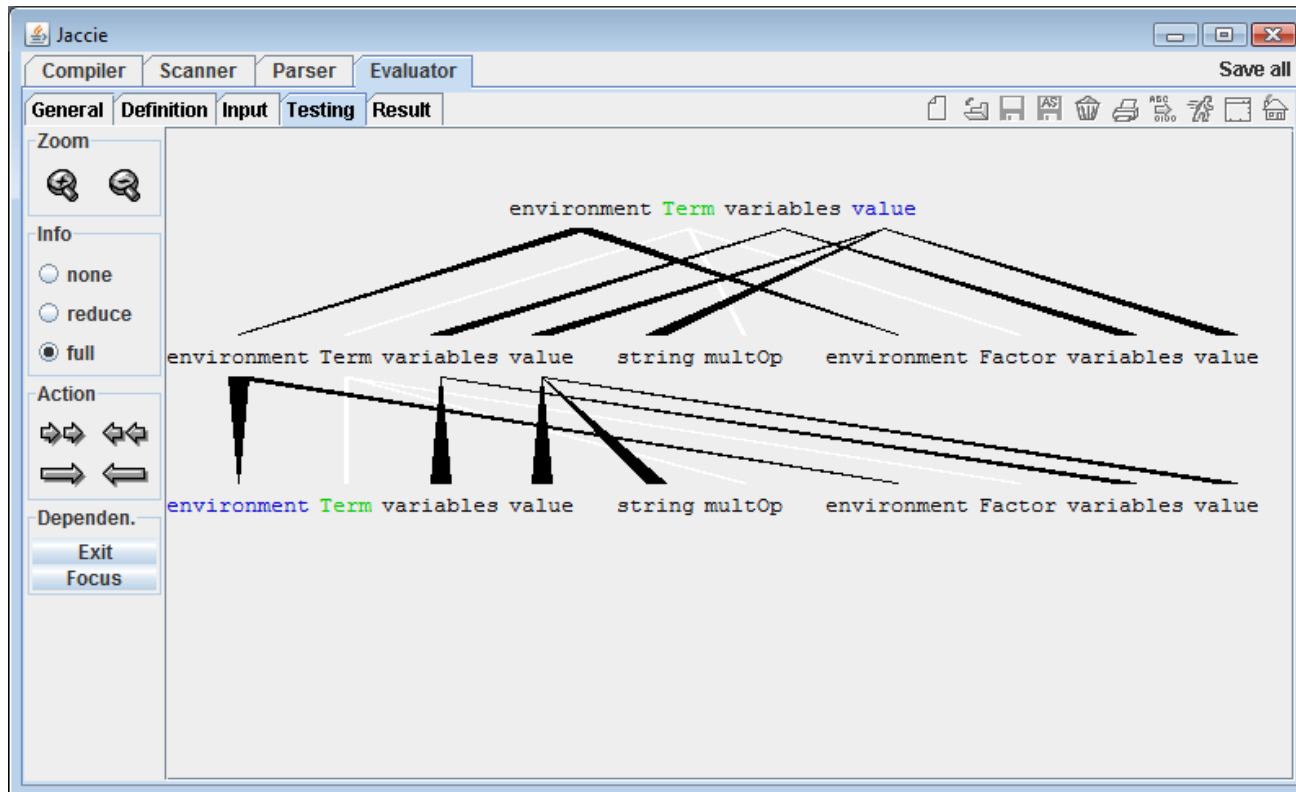


The arrows in the *Action* panel again serve to direct the process forth and back – either in single steps or to the very end.

The other controls let you vary the scale of the tree graphic (and thus its size) and the amount of information shown for each node (no attributes – with attributes – with attributes and attribute values).

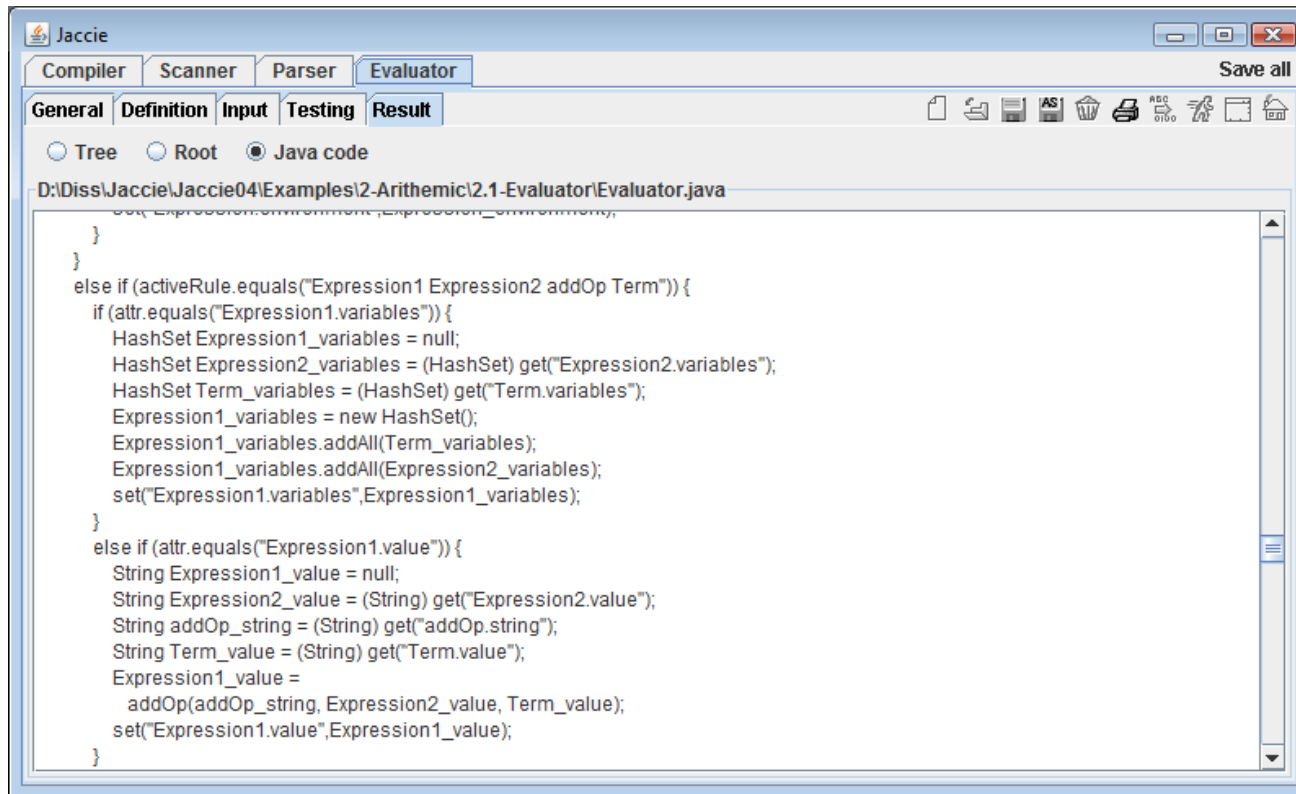
A mouse click on a tree node will show this node, its neighbours and all dependencies between these nodes in detail (see next page)!

This representation is even traversable: Clicking at a node will put this node into the centre position (i.e. give it the *focus*). Thus we can move from the focus node to any one of its neighbours.



Having thoroughly tested your attribute grammar, you would like to 'wrap everything up', i.e., produce a compiler that can either be integrated into other applications or be used as a standalone program.

In the *General* subarea there is a *Generate* button, which generates the Java source code of the current compiler component. The text is shown in the *Result* subarea - ready for takeaway. In the screenshot below we see that part of the attribute evaluator that contains the attribute evaluation rule we have discussed on page 17 (before the rule is evaluated, the required attributes are fetched from the tree using 'get(...)'; after evaluation, the computed attribute value is stored into the tree using 'set(...)'):



The screenshot shows the Jaccie IDE interface. The 'Evaluator' tab is selected, and the 'Result' subarea is active. The 'Java code' radio button is selected. The code displayed is the implementation of the attribute evaluator, showing the logic for evaluating expressions based on the active rule. The code is as follows:

```

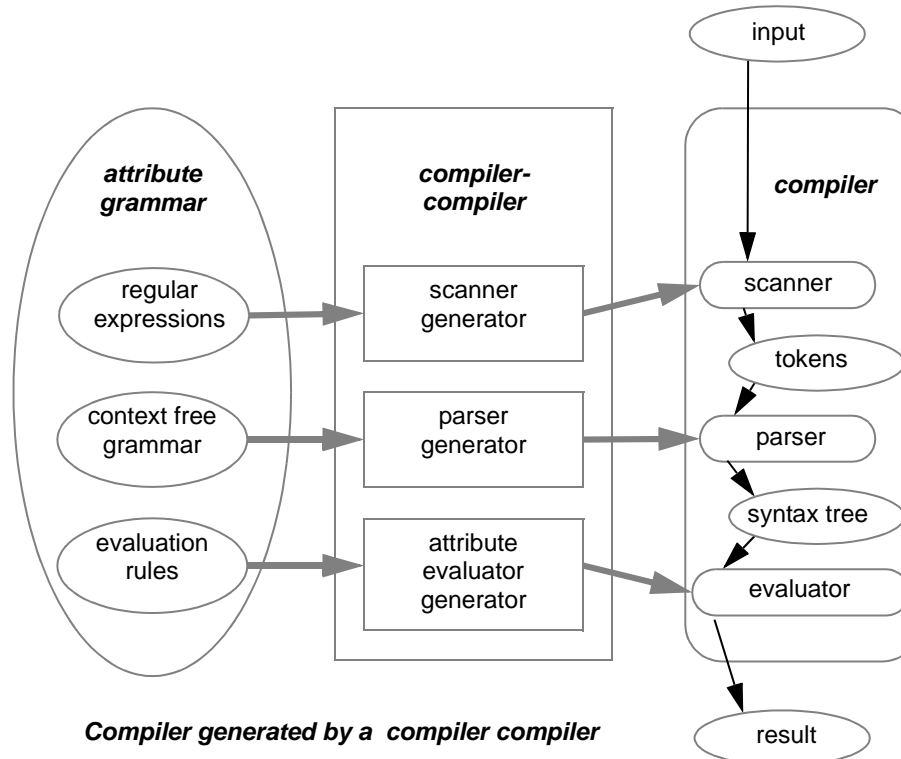
}
}
else if (activeRule.equals("Expression1 Expression2 addOp Term")) {
    if (attr.equals("Expression1.variables")) {
        HashSet Expression1_variables = null;
        HashSet Expression2_variables = (HashSet) get("Expression2.variables");
        HashSet Term_variables = (HashSet) get("Term.variables");
        Expression1_variables = new HashSet();
        Expression1_variables.addAll(Term_variables);
        Expression1_variables.addAll(Expression2_variables);
        set("Expression1.variables", Expression1_variables);
    }
    else if (attr.equals("Expression1.value")) {
        String Expression1_value = null;
        String Expression2_value = (String) get("Expression2.value");
        String addOp_string = (String) get("addOp.string");
        String Term_value = (String) get("Term.value");
        Expression1_value =
            addOp(addOp_string, Expression2_value, Term_value);
        set("Expression1.value", Expression1_value);
    }
}

```

This completes our [Jaccie Tour](#)!

2. Tool architecture

The architecture of **Jaccie** (and other compiler compilers) is shown below:



Jaccie consists of

- the **generator components** of the compiler compiler (middle block)
- **special editors** for defining these components formally (on the left hand side)
- **debuggers** for testing the components of the generated compiler (on the right hand side)

The development of a compiler with [Jaccie](#) typically falls into three successive **phases** (corresponding to horizontal slices of the architecture diagram): *scanner generation*, *parser generation*, and *attribute evaluator generation*. In the resulting compiler, output from the component of one phase is input for the component of next phase.

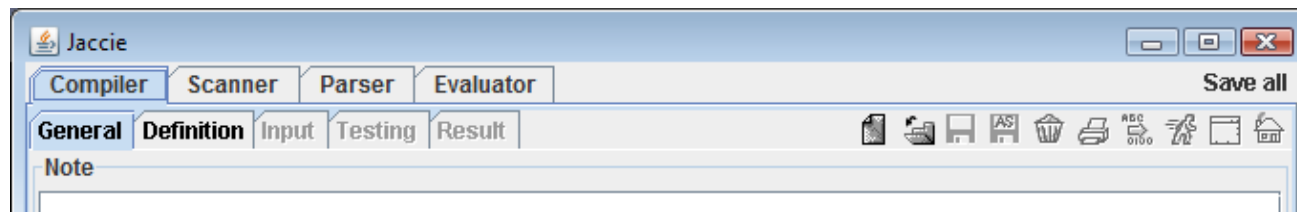
The next three sections each describe *one* phase of the compiler development with [Jaccie](#). The current section describes the organization and elements common to all phases.

[Jaccie](#) is implemented in Java. Also, compiler components produced by [Jaccie](#) as well as the attribute evaluation rules and attribute types in attribute evaluator definitions provided by [Jaccie](#)-users all are written in Java.

There exist two variants of [Jaccie](#): the *applet variant* consists of three Java applets where each applet provides the [Jaccie](#) tools for one phase; the *application variant* combines all [Jaccie](#) tools in one program and thus allows for better integration, e.g., by automatically passing information from one debugging tool to the next. Actually, the applet variant was developed for advanced users who want to build Internet demos with [Jaccie](#). *This initial version of the handbook only describes the application variant.*

Overall GUI organization

Near the top of the [Jaccie](#) window, there are two rows with register cards and icons:

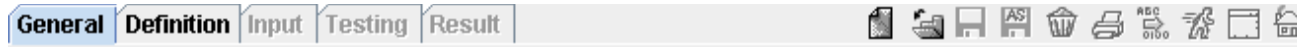


The register cards in the first row allow you to select an **area** with the tools for one of the three phases (*Scanner - Parser - Evaluator*). The tab of the **current area** is shown in a lighter gray (here: *Compiler*). The *Compiler* area contains general settings and allows you to combine components (details later).

The second row of register cards allows you to select a **subarea** of the current area. Since each area has the same set of subareas, we describe them in the next subsection below.

Common GUI elements

On top of each area, you have the same row of register cards (left hand side) and graphical icons (right hand side):



Shading indicates, which **subarea** is currently active (tab light gray) and which other subareas are currently available for selection (black font). In the screenshot above, the current subarea is **General**. Subarea **Definition** is available for selection, while the other subareas are *inactive*, i.e., not available. Likewise, shading indicates which of the ten icons are available and which are not.

What is the purpose of the different subareas?

- In subarea **General** general settings may be made. The most important of these are the location of the Java parser (which is required by most components) as well as the location of a temp directory and the ClassPath. As described below, [Jaccie](#) remembers these settings so that they need only be made when installing [Jaccie](#) the first time. Also, in these subareas we find *Generate Java* buttons for generating the source code of compiler components and radio buttons for selecting between different tool variants (different parser types or attribute evaluator types).
- In each phase, subarea **Definition** offers a special editor for creating and editing the corresponding formal description (i.e., an editor for regular expressions, or for grammars, or for the attribute definitions and attribute evaluation rules belonging to an attribute grammar).
- In subarea **Input** the input for the current phase is established: either by typing it or by loading it from a file. This subarea is available as soon as a definition has been created or opened.
- In each phase, subarea **Testing** offers a suitable debugging environment which allows you to test (and watch all relevant internal data of) the current component. This subarea is available as soon as both the definition and a suitable input have been established.
- In the **Result** subarea results can be viewed as soon as they are available. Generated Java source code of compiler components is shown in this subarea, too.

Jaccie- Tour

Tool architecture

Token Recognition

Grammars and Parsing

Attribute Evaluation

Getting Started

Building Compilers

Title Page

◀◀

▶▶

◀

▶

Page 25 of 99

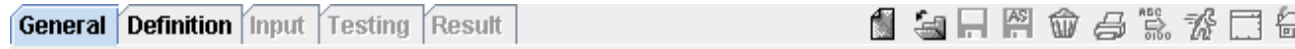
Back

Full Screen on/off

Close

Quit

As mentioned above, shading of the ten icons on the right hand side of



indicates which icons are available and which are not. This depends on the currently active subarea. As a rule of thumb, just those icons are available that can actually be used in the current situation.

The six leftmost icons correspond to well-known file operations ('objects' are definitions, input files etc.):

- creates a new object
- opens an existing object
- saves an object (under its name)
- saves an object (under a new name)
- deletes an object
- prints an object

The next two icons allow you to compile () Java code and to run () compiler components.

The icons () and () belong to the applet variant of [Jaccie](#) and are, therefore, not explained here.

Intermediate products

During a [Jaccie](#) session some **intermediate products** are created that will probably be useful in future and, therefore, can be made persistent by storing them in files with the following **file extensions**:

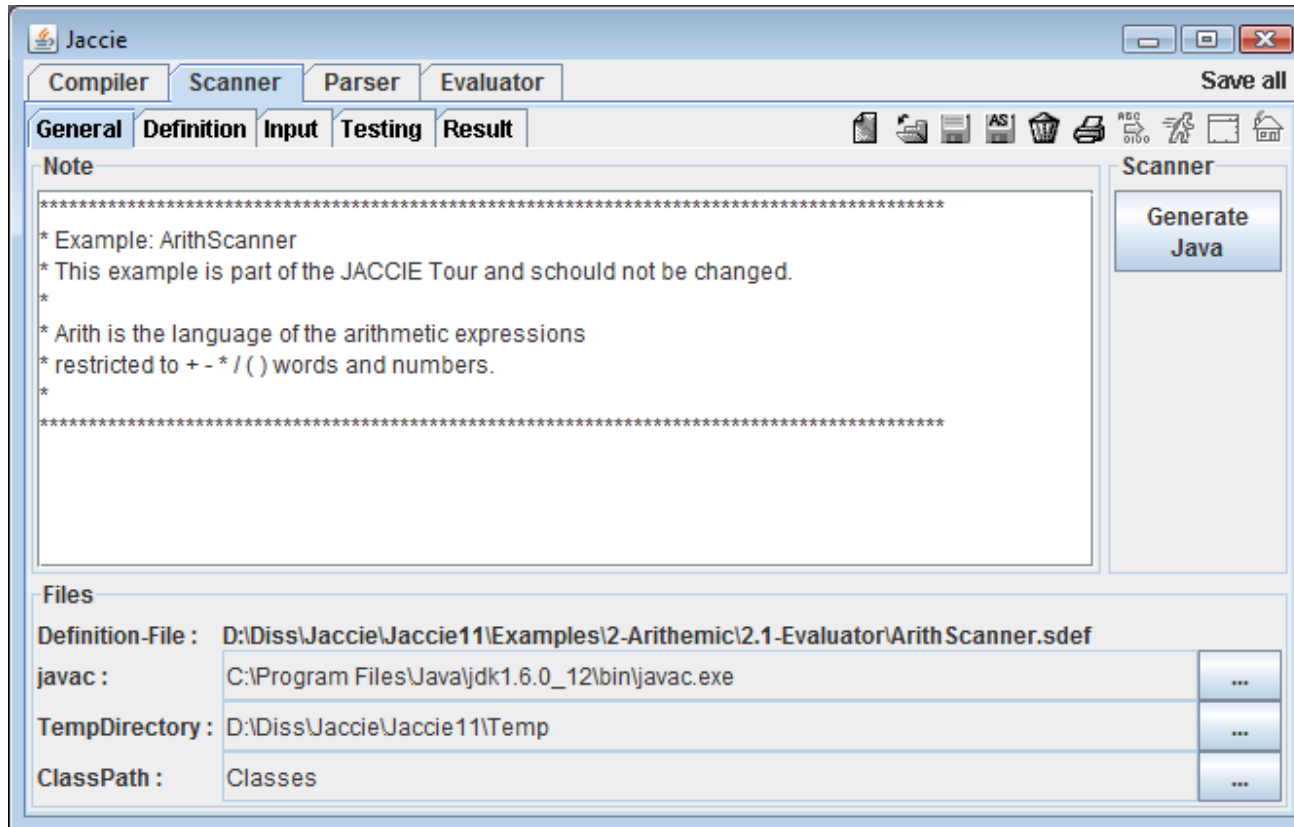
sdef / pdef / edef /comp: definition of Scanner / Parser / Evaluator / Compiler

sinp / pinp / einp: input for Scanner / Parser / Evaluator

tree: syntax tree, possibly containing attribute values

General settings

In the screenshot below, all possible settings for scanners have already been provided:



Definition-File: Path to scanner definition (here only shown; established in *Definition* subarea)

javac: Path to (Sun) Java compiler

TempDirectory: Path to your own Java classes (source code)

ClassPath: Path to compiled Java classes (binary code)

Using the (...) buttons on the right hand side, you can start the dialogs for changing the corresponding settings.

The current configuration

How does [Jaccie](#) store configuration information like the above settings?

The [Jaccie](#) system is stored in an executable jar file, `Jaccie.jar`. The directory this file resides in is called the [Jaccie home directory](#), e.g. `C:\myJaccie` containing `C:\myJaccie\Jaccie.jar`.

Within the `Config` subdirectory of the home directory there is a file, `Compiler.cfg`, which stores the current [Jaccie](#) configuration information in an XML format, e.g., for the above configuration we would have:

```
<Configuration>
  <javac>C:\j2sdk1.4.0\bin\javac.exe</javac>
  <ClassPath>C:\myJaccie</ClassPath>
  <JavaTempFile>C:\myJaccie\Temp\Temp.java</JavaTempFile>
  <lastWorkDirectory>C:\myJaccie\Examples\Arith</lastWorkDirectory>
</Configuration>
```

In addition to the three general settings [Jaccie](#) 'remembers' the path to its most recent **working directory**.

Installation of [Jaccie](#)

The [Jaccie](#) executable jar file, `Jaccie.jar`, and the four directories below are contained in the archive `jaccie.zip`:

Config - contains the configuration file `Compiler.cfg`.

Data - contains the precompiled, fixed parts of all compiler component.

Examples - contains a subdirectory tree with small example compiler definitions and inputs.

Temp - the directory where to place Java source code files for inclusion in [Jaccie](#) projects.

In order to install [Jaccie](#) you have to

- => create a home directory (say, `C:\myJaccie`)
- => unzip the archive `jaccie.zip` in the home directory
- => provide values for 'general settings' (choosing home directory for `ClassPath`)

For more details see section [Getting started](#).

3. Token Recognition

Lexical analysis is performed by the **Scanner** component of a compiler. Token definitions are given in the form of regular expressions that were introduced in section **Describing syntax** in the theory paper. Section **Lexical analysis** of this paper explains how deterministic finite automata are derived from regular expressions and how a set of automata is combined into one scanner. In the **Jaccie** tour, we already have seen a scanner at work.

Scanner definition

Details of the **Jaccie** notation for regular expressions are given in the tables below. Notice, that this notation slightly differs from the notation used in the theory paper.

regular expression	name	meaning
<name>	variable	Naming mechanism, supports clear and concise definition of regular expressions. Pattern or token identifier, respectively.
$(\alpha \mid \beta \mid \gamma)$	alternative	Alternatives α , β and γ are constituents of the alternative expression. The complete alternative expression must be enclosed by parentheses as shown.
$\$a$	character	Meta symbol \$ precedes the character a (as typed on the keyboard).
$\alpha\beta$	concatenation	Regular expressions α and β are combined into a new regular expression by juxtaposition.
"abc"	word	"abc" abbreviates \$a \$b \$c.
$\alpha[\text{min-max}]$	iteration 1	corresponds to concatenation of at least min and at most max instances of the regular expression α ; min and max are natural numbers (including 0); the value of max may be * ('unbounded'). For min > max only the empty word matches this expression.
$\alpha[k]$	iteration 2	corresponds to concatenation of exactly k instances of α .

Every **Jaccie** regular expression is terminated by a newline character.

regular expression	name	meaning
#xxx	ASCII Code	xxx is the ASCII code of a character.
#xxxx	Unicode	xxxx is the Unicode of a character.
{x-y}	range	A range defines a set of characters. The bounds x and y may be <i>characters</i> , <i>ASCII – Codes</i> , or <i>Unicodes</i> .
(α)	expression	A regular expression enclosed by parentheses.

We consider each part of a scanner definition in turn (some parts are *optional*):

Patterns

A pattern definition

```
<name> := <regular expression>
```

associates a `<regular expression>` with a `<name>`. Example:

```
<decimalNumber> := ($-|$(+)[0-1]{$1-$9}{$0-$9}[1-*
```

According to this definition, a `<decimalNumber>` is a decimal integer constant with an optional sign prefix, no leading zeroes (this excludes the number 0) and an unbounded number of decimal digits (at least two digits in all).

Pattern names may be used in subsequent pattern definitions, e.g. as in:

```
<numbers> := <decimalNumber> ($, <decimalNumber>)[0-*
```

Thus `<numbers>` denotes a sequence of `<decimalNumber>`s containing at least one `<decimalNumber>` where every two adjacent `<decimalNumber>`s are separated by a comma.

Note, that 'subsequent' excludes recursive pattern definitions: Iteratively replacing pattern names by the expressions they denote we can eliminate all pattern names from a regular expression. The only purpose of a pattern definition is to keep token definitions (see below) clear and concise.

Tokens

Token definitions introduce the lexical elements to be recognized by the scanner. They have the same structure as pattern definitions and are listed in order of **falling priorities** (thus deviating from the convention on priorities used in the theory paper near the end of section **lexical analysis**). Therefore, the above pattern definitions would also have been valid token definitions. However, patterns and tokens are handled differently by the scanner: Recognition of tokens is reported by the scanner while patterns are ignored. Actually, during scanner generation all pattern names are eliminated by replacing them with the regular (sub)expressions they denote. No pattern names are left for lexical analysis.

If, e.g., we had introduced `<decimalNumber>` as a pattern and `<numbers>` as a token, then during scanner generation the token definition

```
<numbers> := <decimalNumber> ($, <decimalNumber>)[1-*
```

would have been expanded into

```
<numbers> := ($-|$(+)[0-1]{$1-$9}{$0-$9}[1-*] ($, ($-|$(+)[0-1]{$1-$9}{$0-$9}[1-*]))[1-*
```

For better controlling lexical analysis so-called **start blocks** and / or **end blocks** can be added to token definitions: Start blocks are conditions (written as boolean Java expressions), that must be fulfilled when token recognition starts; otherwise, the token definition is (temporarily) set 'inactive', i.e., is not considered as a candidate when analyzing the next token. Similarly, **end blocks** must evaluate to true for a token to be recognized successfully. A third optional supplement to token definitions are **action blocks**: Java statement sequences that are executed on successful recognition of a corresponding token.

Start / end / action blocks are advanced elements, which increase the power of **Jaccie** scanners far beyond the recognition of regular languages. A detailed presentation is out of the scope of this handbook. The example collection that comes with **Jaccie** contains a few scanner examples which demonstrate the use and the power of these features.

Here, we briefly introduce these elements. A token definition may optionally be followed by up to three **blocks**. A block starts with **Start:** or with **End:** or with **Action:**, respectively. After that comes Java code (a statement sequence for an action, otherwise a boolean expression). The character sequence `[\\]` terminates a block.

A hypothetical example demonstrates uses of all three kinds of blocks:

```
<surnameToken> := <namePattern>
Start:  firstNameRecognized [\]
End:    surnameKnown()  [\]
Action: surnameRecognized  = true;
        firstNameRecognized = false; [\]
```

This example requires `firstNameRecognized` and `surnameRecognized` to be boolean variables and `surnameKnown()` to be a Java method that returns a boolean value. Typically, such variables and methods will be used in blocks belonging to different token definitions, i.e., within the scanner component they must be globally known. There are two ways to make declarations globally available: Either you place them in a Java package which you import, or you make them attributes and methods of the generated Scanner class (see *Import and Global declarations* below).

In blocks three special methods are available for accessing the input text of a token, and its position (line, column) within the input file:

```
scannerService.getString(),
scannerService.getLineNumber() and
scannerService.getColumnNumber().
```

Import and Global declarations

Entries into both **Import** and **Global declarations** text fields will be copied into the source code of the generated scanner class.

Verbatim tokens

Verbatim tokens allow you to ‘work’ with tokens that you have not yet provided a definition for. Essentially, you enclose the token (and its text) into left and right **bounds** which are defined as follows (left and right bound need not be different and there can be more than one pair of bounds):

```
Verbatim-Token:
$< $>
```

The scanner will then transform the input

```
<name "alpha"> <addOp "+"> <number "565767">
```

into the (desired) output

```
name "alpha"  
addOp "+"  
number "565767"
```

Comments

Jaccie allows you to insert **comments** into scanner input that explains the input to the human reader but does not influence processing. As in programming languages there are two kinds of comments:

End-of-line comments start with some **special character sequence** and extend to the end of the line. E.g. the definition **Line Comment: "--"** lets every character sequence -- start an end-of-line comment.

Comments begin with a **start character sequence** and end with an **end character sequence**. They may extend across several lines. E.g. the definition **Comment: "/*" "*/"** defines /* and */ to be the the start and end character sequences of a comment, respectively.

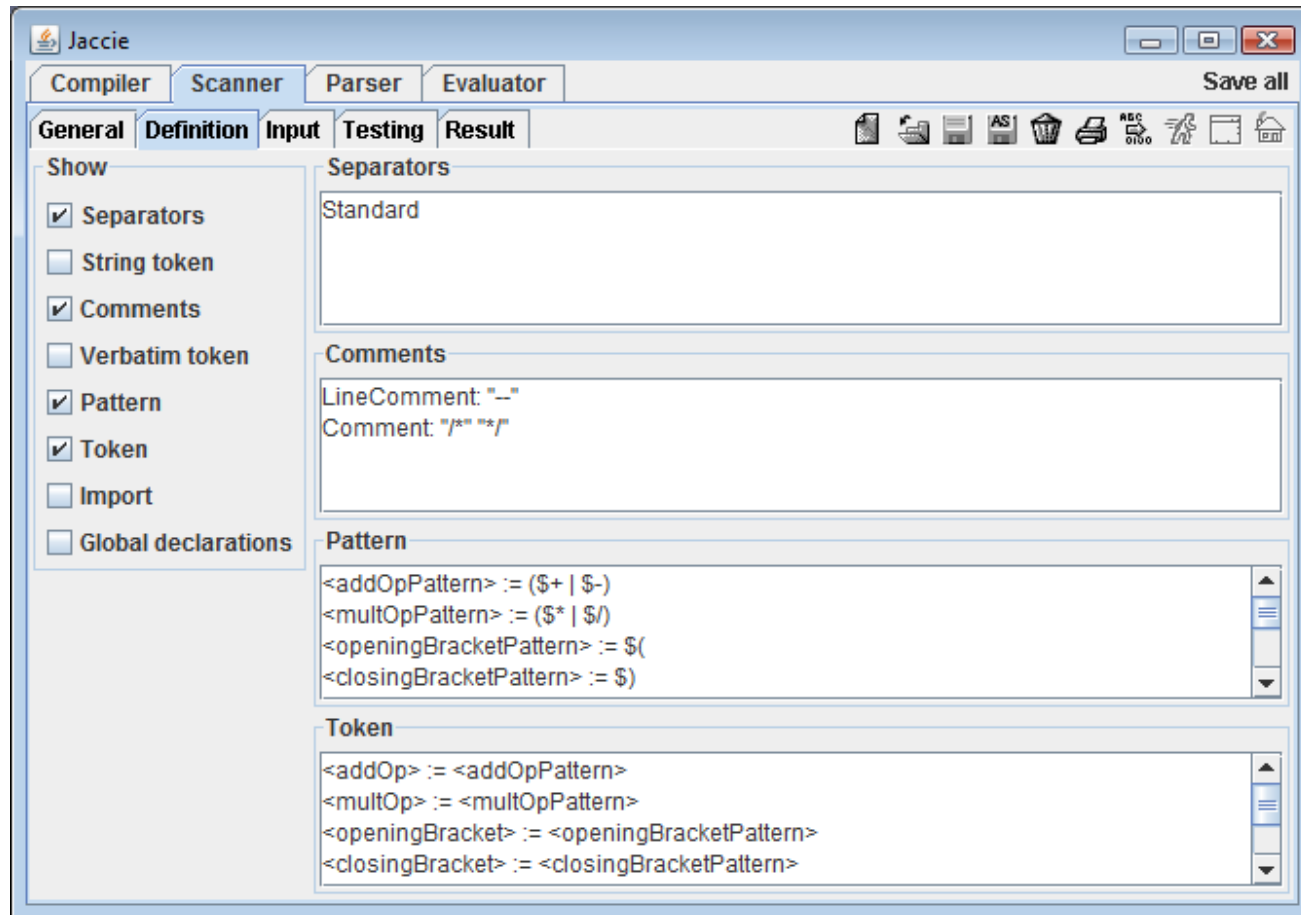
Separators

Separators are those 'whitespace' characters that separate tokens but themselves do not belong to any token. In the current version of **Jaccie**, only the definition Standard may be used: By this definition, NewLine, Tab, Return, FormFeed, and Space are the only separator characters. In future versions, some of these special characters may be defined to be 'normal' characters, and vice versa.

String token

This is a deprecated feature to be removed from future system versions.

Now the purpose of every panel in the scanner editor shown below should be obvious.

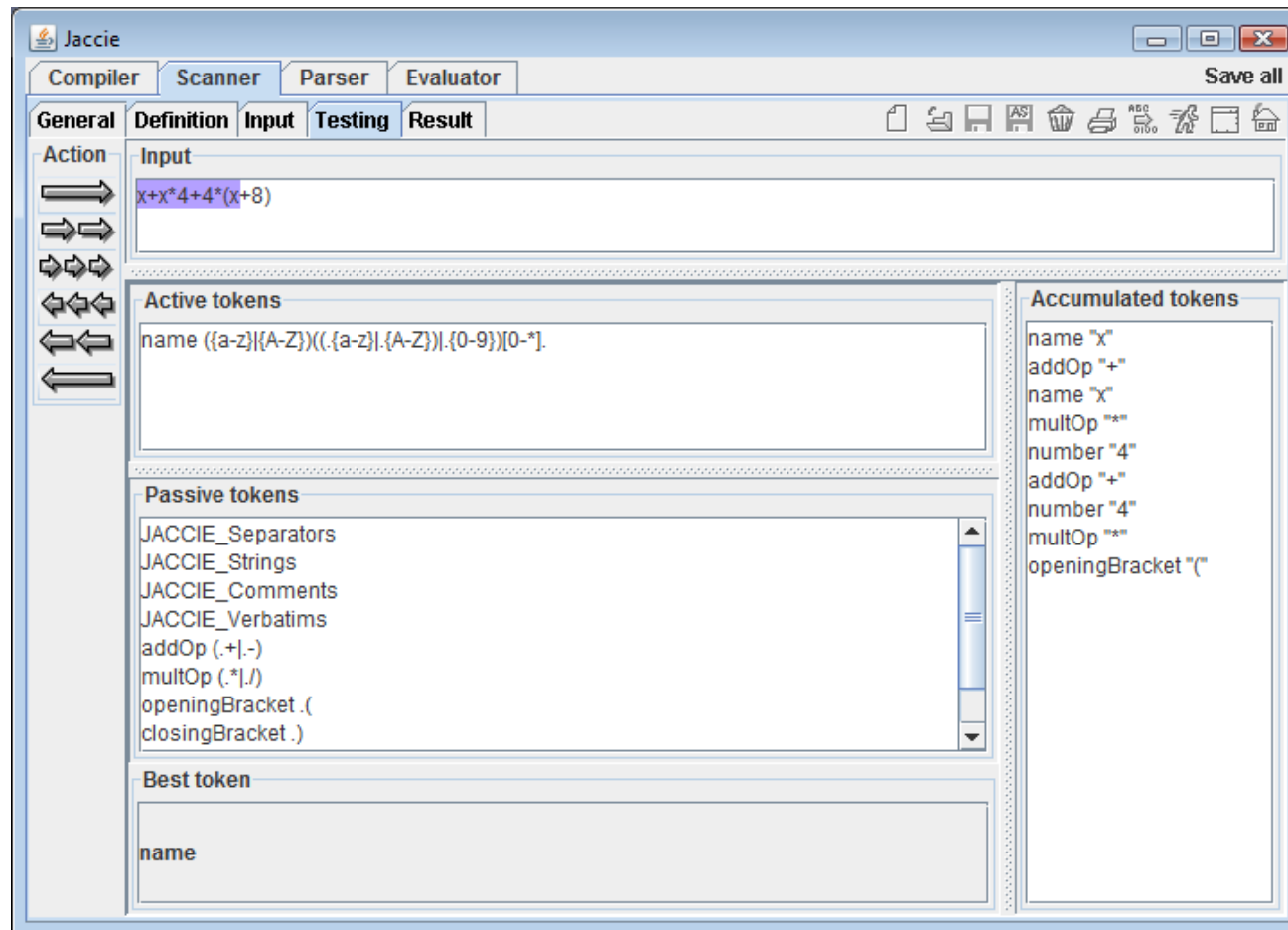



Using the check boxes in the *Show* panel you can dynamically ‘switch on and off’ those panels you want to work with. Java imports and declarations to be referenced by **Java code** from the *Start / End / Action* blocks in token definitions are entered in the *Import* and *Global declarations* panels (switched off in the screenshot).

As mentioned before, in the *General* subarea you will find a *Generate Java* button for producing the scanner’s Java source code. In case there are errors in the generated code – or rather, in the code snippets provided by the user –, the error will be highlighted in the editor and the detailed error message produced by the Java compiler will be displayed in a Dialog window. Otherwise, the generated Java source code will be shown in the *Result* subarea.

Debugging scanners

The description of the scanner debugging facility in [Tour, S. 7-9](#) is almost complete. In the screenshot below, an input without whitespace characters was used to show that no `Jaccie_Separators` are reported in that case.



In the *Passive tokens* subwindow the currently inactive tokens are listed. The token sequence displayed in the *Result* subarea is automatically passed on as parser input. **Unfortunately, however, the current Jaccie version does not pass the token sequence from the Testing subarea to the Result subarea.** Thus, a result token sequence can only be produced (and passed on to the parser) by using the run () icon in the scanner *Input* subarea!

4. Grammars and Parsing

Syntax analysis is done by the **Parser**. The definition of a parser essentially consists of a **contextfree grammar** which defines both the language to be recognized and the structure of recognized sentences in the form of **syntax trees**. Section [Describing syntax](#) of the theory paper briefly discusses the pros and cons of two grammars for arithmetic expressions. Section [Syntax analysis](#) of the same papers in some detail describes the construction and workings of the different kinds of parsers that are used in practice (LL(1), LR(0), SLR(1), LALR(1), and LR(1)). In the [Jaccie](#) tour we have briefly seen a [Jaccie](#) parser in action.

Parser definition

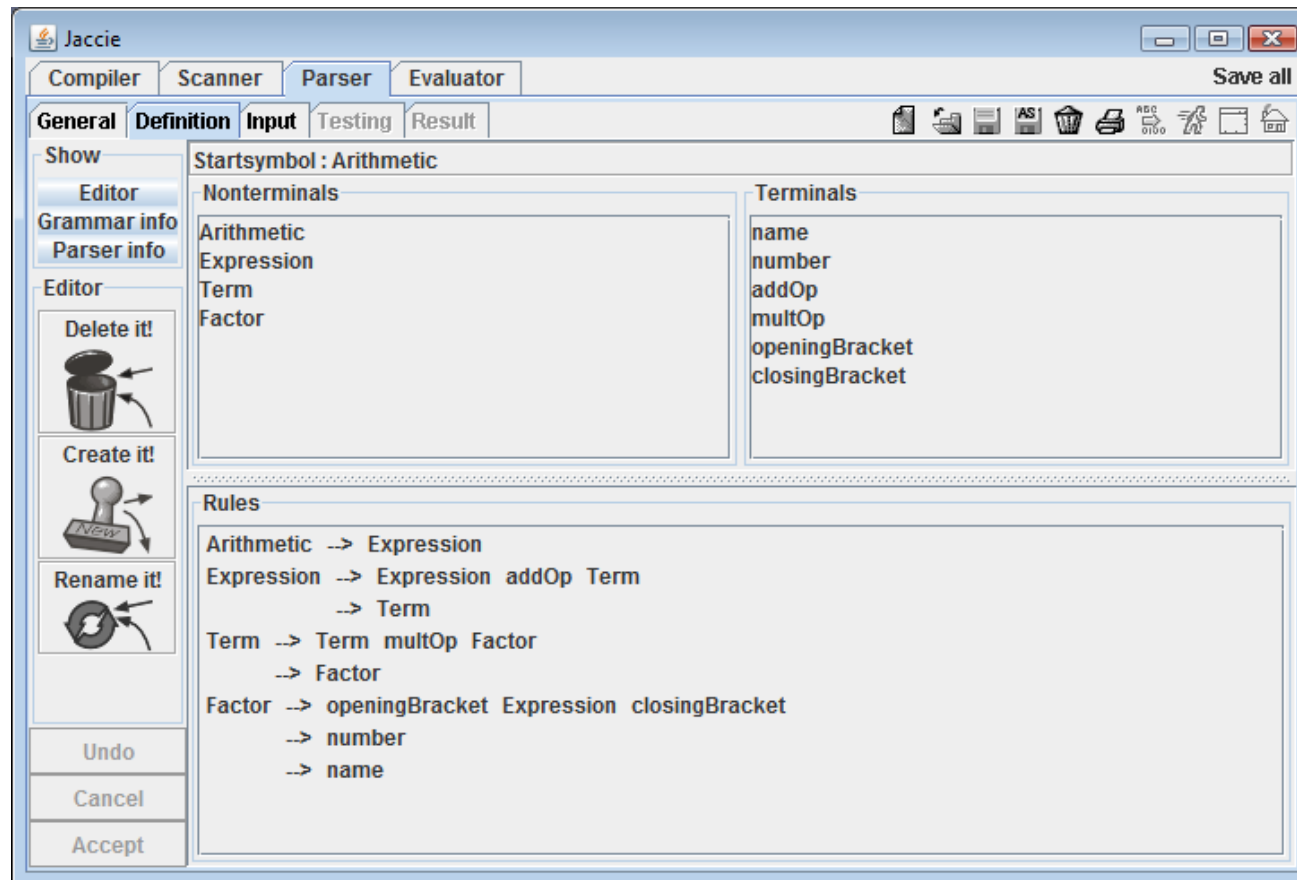
When compared with the [Jaccie](#) notation for regular expressions, the description of contextfree grammars appears to be rather straightforward and simple – and this in spite of contextfree grammars being the more powerful mechanism: Formal language theory tells us that every language you can describe with a regular expression can also be defined using a contextfree grammar, while the converse is not true.

There are several reasons for [Jaccie](#) scanner definitions to be more complex than [Jaccie](#) parser definitions:

- Notational frameworks for regular expressions – like those used by Unix filter programmes – for reasons of usability extend ‘pure’ regular expressions by features like escape symbols, ranges, and different character codes (text, ASCII, Unicode).
- A scanner definition does not consist of one regular expression only: rather, there is a set of competing regular expressions (requiring disambiguating rules for processing) – one expression for each token.
- As mentioned in the previous section, some features of [Jaccie](#) scanners (notably, the different kinds of Java code blocks) extend their power far beyond the recognition of regular tokens. In a smaller context, these Java code blocks offer means similar to the attribute evaluation rules in an attribute grammar (see next section). For simple translation tasks, [Jaccie](#) scanners may suffice.

The symbols of a **contextfree grammar** fall into two disjoint subsets: the **terminal symbols** that correspond exactly to the tokens recognized by the scanner, and the **nonterminal symbols**. Each of the grammar’s **production rules** has exactly one nonterminal symbol on its *left hand side*, and any sequence of terminal and nonterminal symbols on its *right hand side*; this sequence may be empty, i.e. have length 0. Production rules with the same nonterminal A on their left hand sides are called **alternatives** for A.

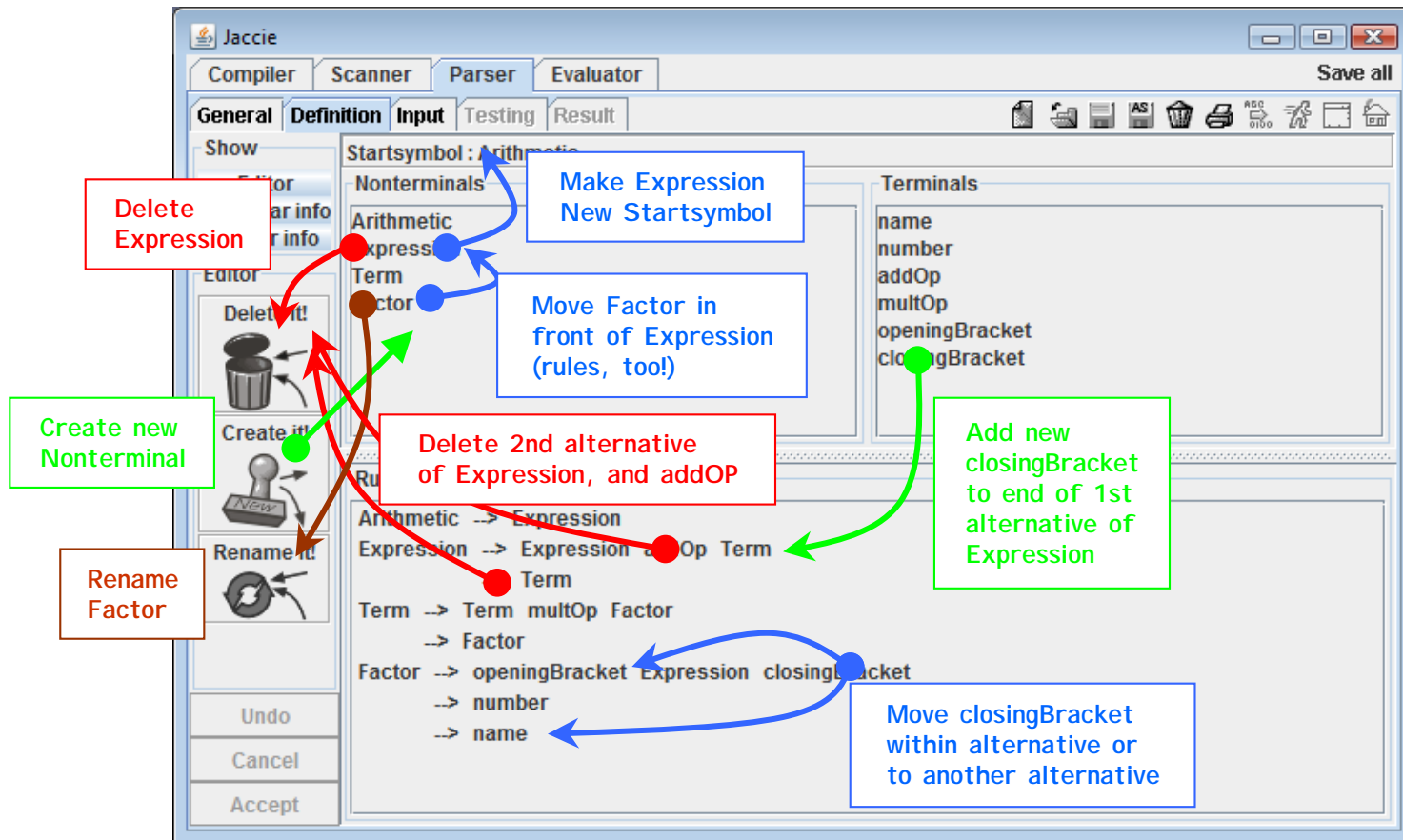
In the *Parser – Definition* subarea the special editor for contextfree grammars in the screenshot below is available. Using the buttons in the *Show* panel you can switch between the editor and two windows showing information automatically derived from the grammar and parsing automata, respectively (we will come back to that later).



The nonterminal symbols, the terminal symbols, and the grammar's production rules are shown in one list panel each. Note, that the alternatives are grouped together (the left hand side is shown only for the first alternative) and that these blocks have the same order as the nonterminals in their panel. Nonterminals and terminals can be reordered with the mouse within their panels using **drag and drop**: Place the mouse pointer on the symbol, press (and keep pressed) the left mouse button, and push the symbol to the position you wish; then let go the mouse button. If you reorder the nonterminals the production rules are reordered accordingly. In a similar way, the alternatives for the same left hand side can be reordered: move an alternative around using the arrow as its handle.

In the same simple and suggestive manner you can use drag and drop to rename symbols and to create and delete both symbols and alternatives: The **stamp** creates such objects, in the **trash bin** they disappear. The **circular arrow** symbolizes renaming (which is automatically applied to all occurrences of this symbol in every rule).

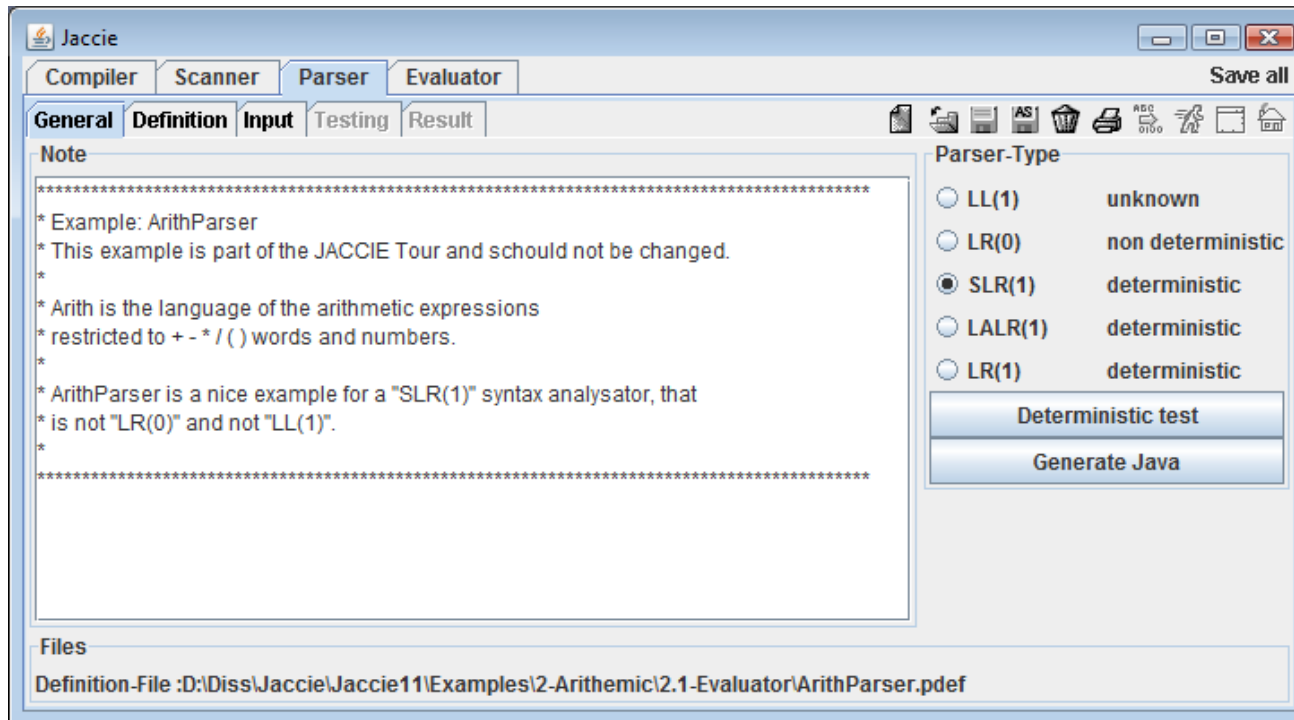
Some of the many opportunities for direct manipulation are illustrated in the screenshot below:




Every editing operation can be undone using the *Undo* button. Before leaving the **editing mode** (entered with the first editing operation) you either can make all changes permanent (pushing the *Accept* button) or dismiss all changes (pushing the *Cancel* button). Jaccie will not let you leave the editing mode unless you push one of these two buttons.

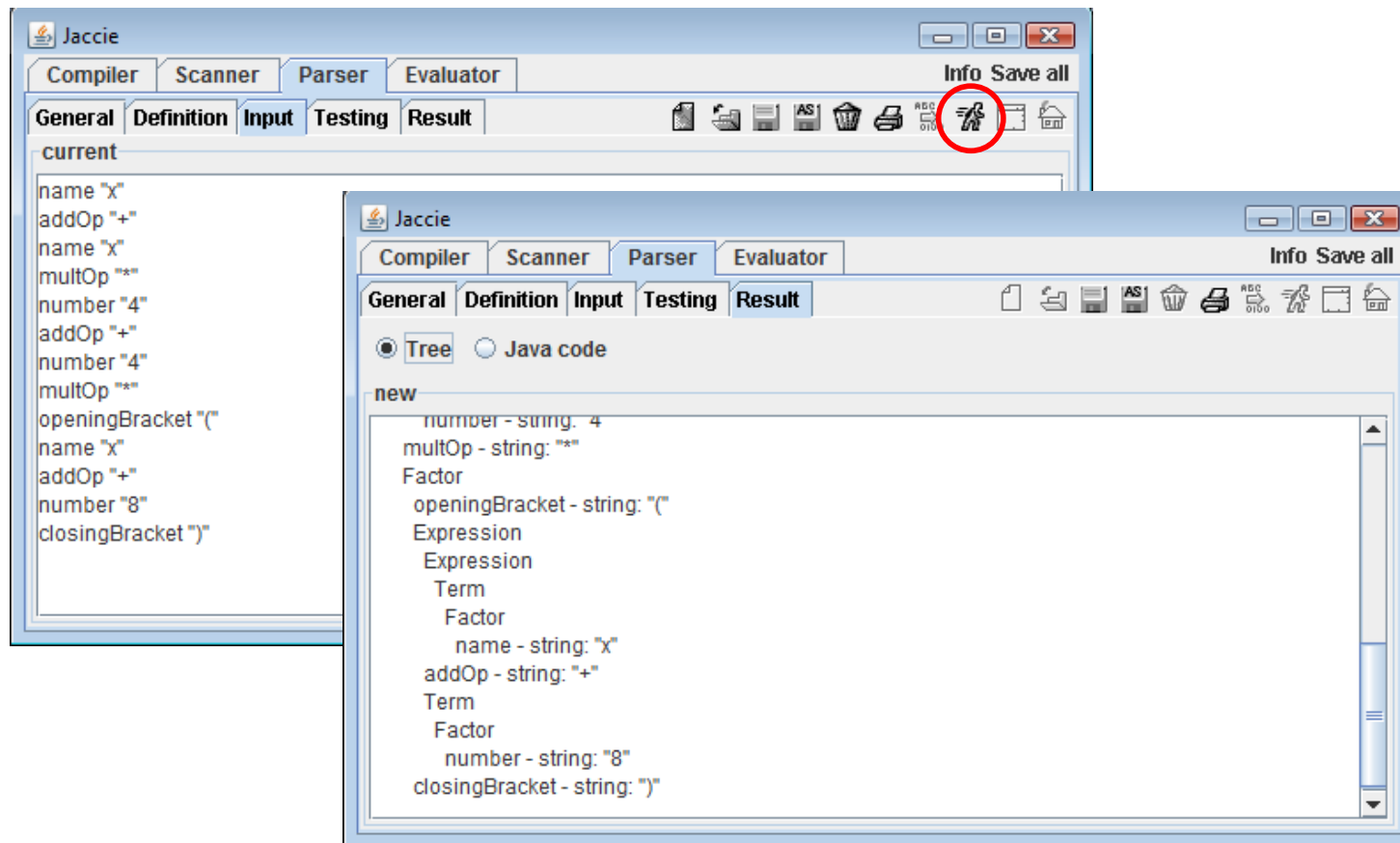
Debugging parsers

Jaccie has only one scanner generator, but a number of parser generators – one for each parsing algorithm in common use (LL(1), LR(0), SLR(1), LALR(1), and LR(1)). So the first step in debugging a grammar is to decide which parsing algorithm should be used. For this purpose, you can use the radio buttons in subarea *Parser-General*:



When in doubt, first ask **Jaccie** (using button *Deterministic test*) which parsers are deterministic, i.e. can be executed without backtracking or conflict resolution provided 'manually' by users. Test results will appear next to the radio buttons. Let us now assume that the chosen parser is deterministic for the grammar under consideration. Handling of nondeterministic **Jaccie** parsers will be described later.

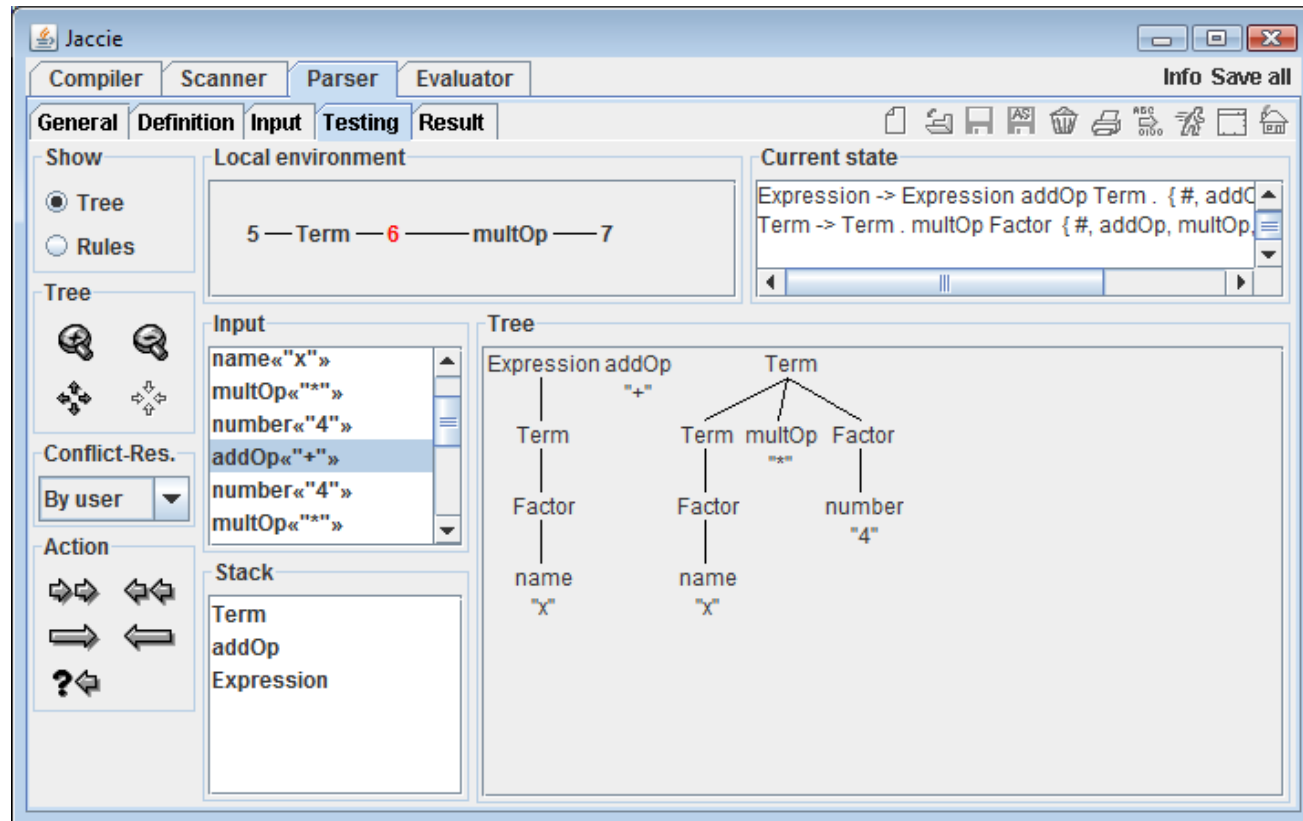
The simplest way to run a deterministic parser is to first move to subarea *Parser-Input*, load some input (in general, there should be some scanner output available), and then press the *run* icon (). In the same way we have started the scanner. The parser's result – a syntax tree in text representation – will appear in subarea *Parser-Result*:



In the text representation, each node of the syntax tree is shown on a line of its own. The tree structure is visualised by nesting and indentation: Immediately below any node, its (left to right) subtrees are shown (top to bottom). Indentation corresponds to the level of nesting; e.g., the first child of a node will appear in the next line, on the next level of indentation. To the right of any terminal node (corresponding to a lexical token) the original input string is shown, e.g., *number* is followed by – *string: "8"*.

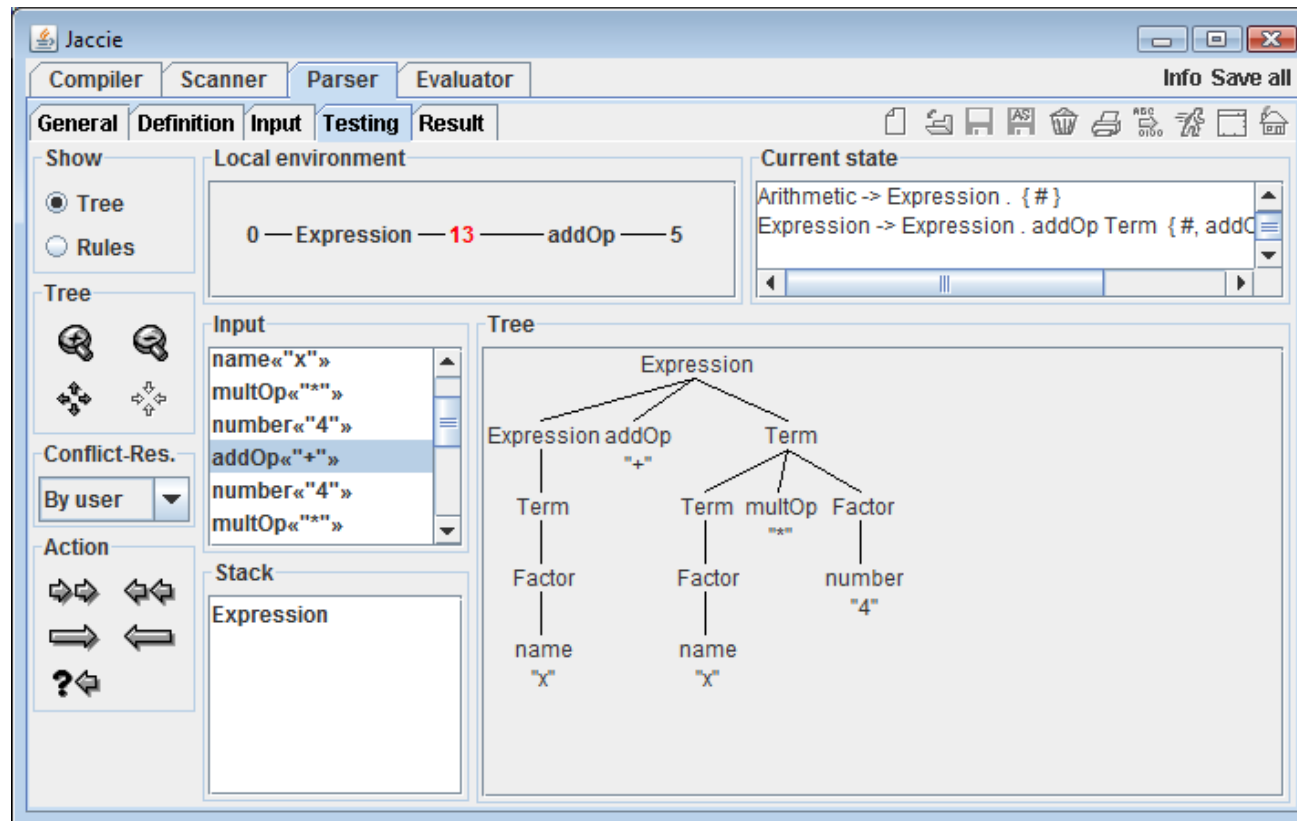
Since in this mode only the input token sequence and the output syntax tree are shown, we now proceed to subarea *Parser-Testing* where we can control and view the parsing process in detail.

The controls on the left hand side of the **parser debugger** window allow you to direct the parsing process and to determine the amount of information shown during that process. Main control elements are the arrows on the *Action* panel which let you direct the parser forward or backward, either in single steps (two short arrows) or to the end (long arrow). Alternatively, you can click with the mouse on a token in the *Input* panel to directly specify a new parsing position.

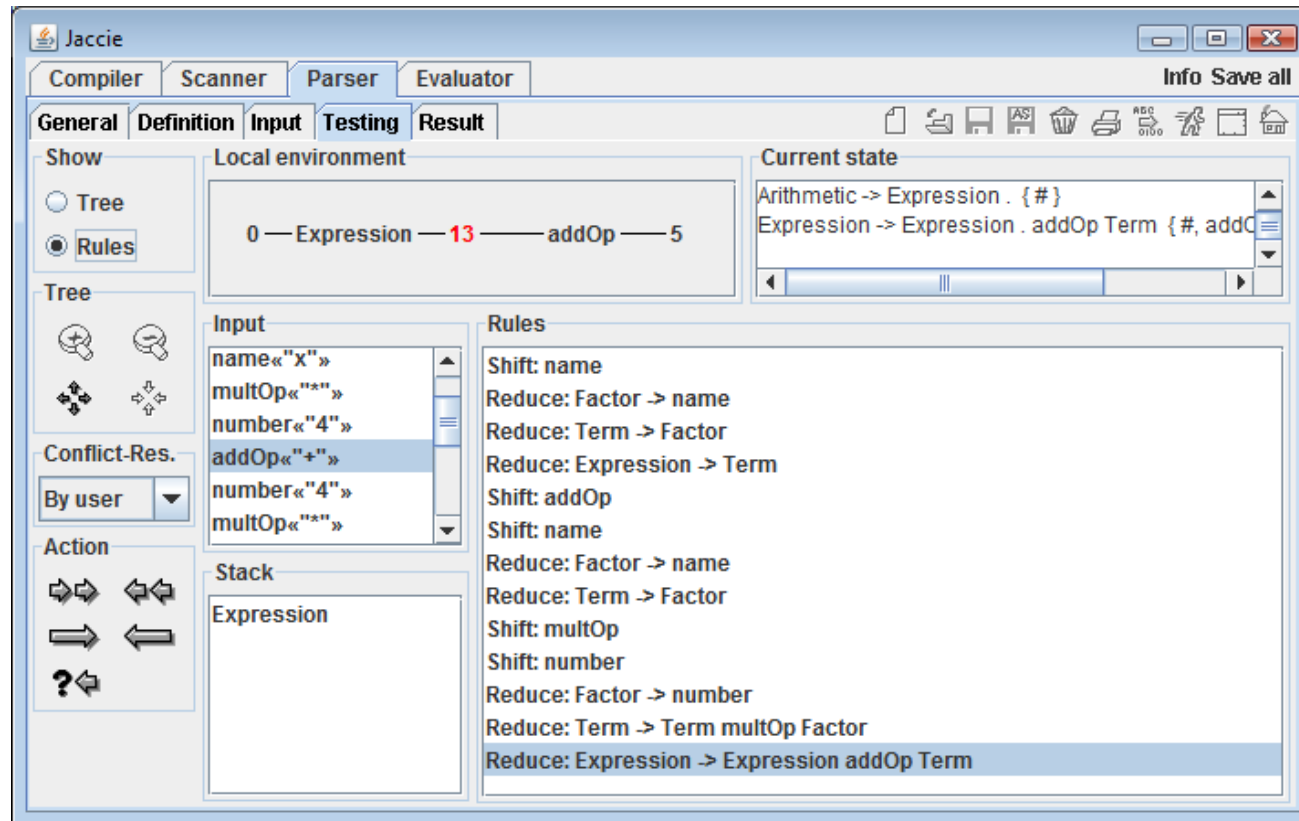


Those parts of the syntax tree under construction that have been recognized so far are displayed graphically in the main *Tree* panel. In the screenshot, a **bottom up parser** is shown which constructs the tree from its leaves towards the root. The *Local environment* and *Current state* panels in the upper part of the window exhibit relevant portions

of the **parsing automaton** which determine the parser's next step. Here, the reduction item in the first line of the *Current state* panel states that the three tree roots *Expression addOp Term* may be reduced ('combined') to one tree with root *Expression*, provided the next input token is an *addOp*. Highlighting in the *Input* panel indicates that *addOp* indeed is the next input token in the current situation. So when taking one forward move we obtain:



The shift item in the second line of the *Current state* panel indicates that in the next step the *addOp* input token may be read, etc. **Bottom up syntax analysis** essentially is a sequence of **reductions** and **shift steps**. This becomes apparent if we change from the visual representation of the syntax tree to its textual representation (i.e., its leftmost reduction sequence) by choosing the radio button *Rules* in the *Show* panel. The screenshot on the next page shows the textual representation of the parsing situation from the previous page:

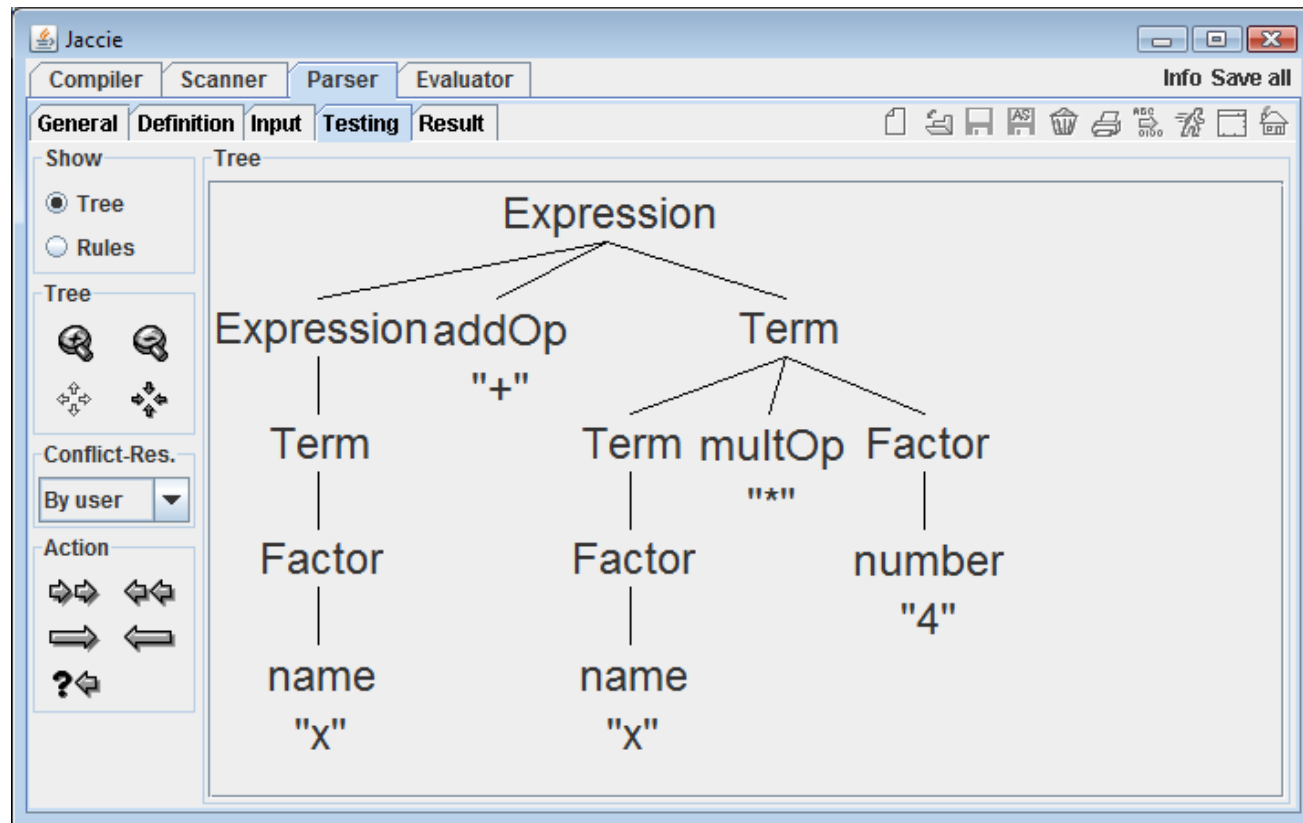


The **stack** of a bottom up parser always contains the sequence of root labels of all subtrees recognized so far. The symbol on top of the stack is the root label recognized last. In the above situation, a reduction according to the reduction item in the first line of the *Current state* panel will replace the stack contents *Expression addOp Term* by *Expression*.

Top down syntax analysis builds the syntax tree proceeding from the root towards the leaves. Its **shift steps** correspond exactly to those from bottom up analysis. Instead of reductions top down analysis uses **expansions**, where a tree node *A* according to some production rule $A \rightarrow B C D$ is expanded by child nodes *B C D*.

Details about the different parsing algorithms are described in section **Syntax analysis** of the theory paper.

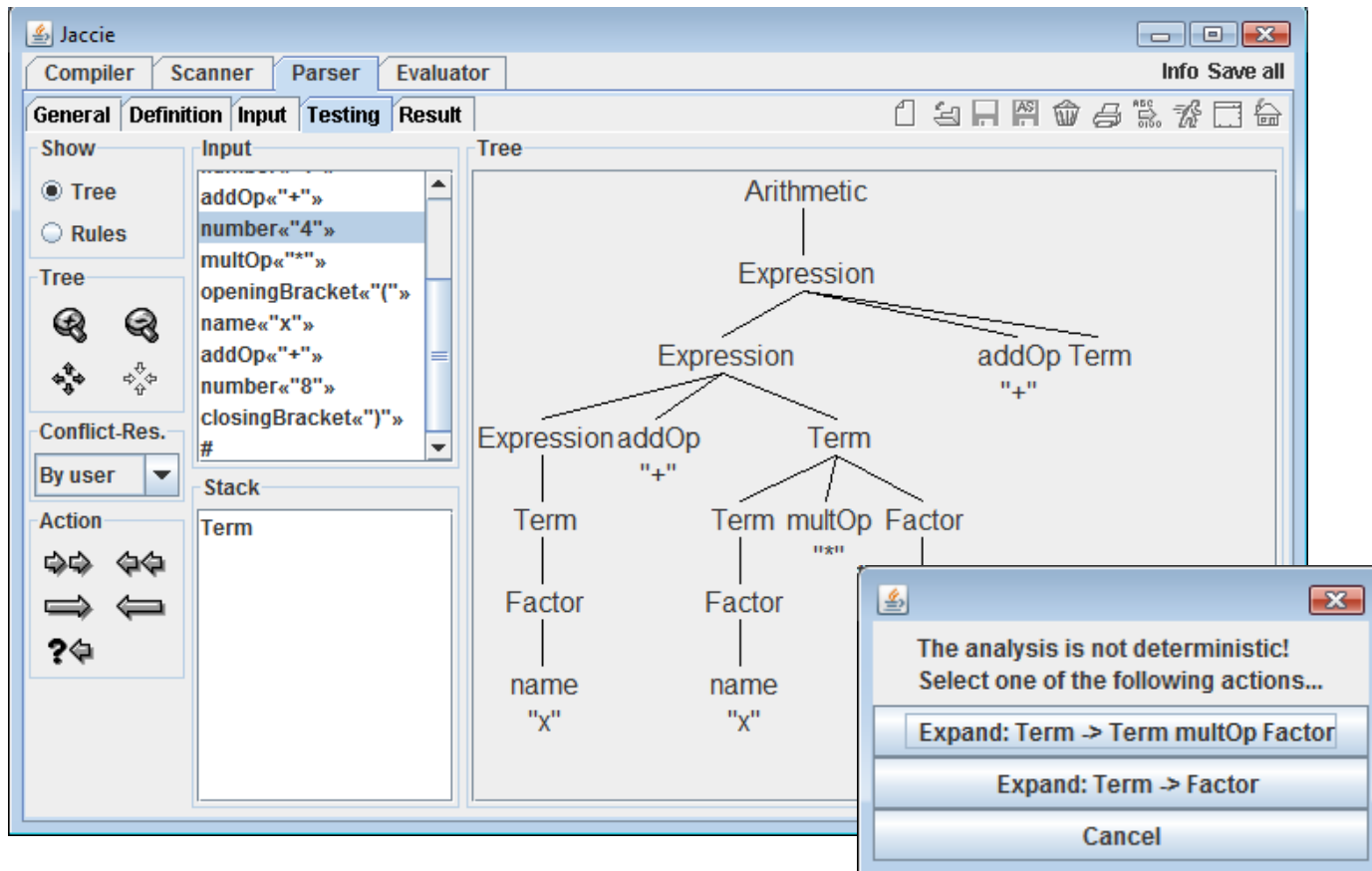
In the screenshot below, again the same parsing situation is shown as on the previous page (and two pages before that). The tree size (including its font size) has been increased using the *zoom in* function (magnifying glass marked with a plus sign). The panels showing the current state of the parsing automaton, the current input and stack have been switched off using the four *outward arrows* (below the magnifying glass). The inverse functions are available from the four *inward arrows* and the *zoom in* function (magnifying glass marked with a minus sign).




The other controls (drop down list *Conflict Res(olution)*. and ? left arrow) are used for nondeterministic parsing, our next topic.

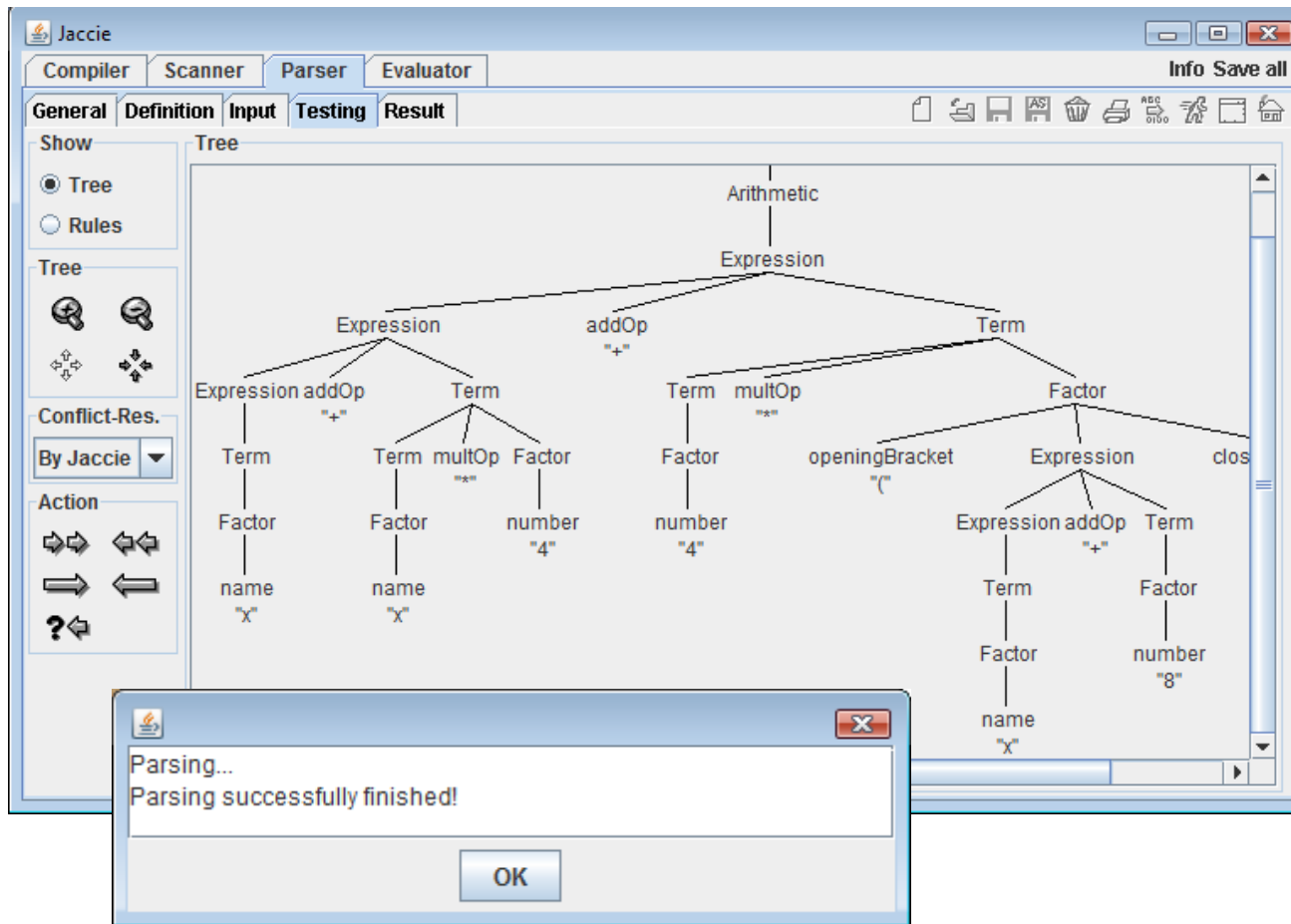
Nondeterministic parsing

During LL(1) parsing the stack contains those nonterminal symbols that label tips of the incomplete syntax tree, i.e., those symbols that still have to be expanded to complete the tree. The topmost stack symbol belongs to the node to be expanded next. In some situations (typically in the context of left recursive rules like `Term -> Term multOp Factor`) the LL(1) parser has to decide between different possible expansions without having enough information for such a decision. If the conflict resolution strategy *By user* has been chosen Jaccie will simply present all possible alternatives and leave the decision to the user:



In the above situation the user ought to choose the first alternative shown. Otherwise, parsing will not end successfully. The `? left arrow` control allows you to return to the last decision point and there to try out some other alternative.

For the grammar of arithmetic expressions all bottom up parsers except for the LR(0) parser are deterministic. Therefore, let us choose LR(0) and leave resolution of conflicts to [Jaccie](#)! If you now press button  [Jaccie](#) may possibly finish successfully without any user interaction:



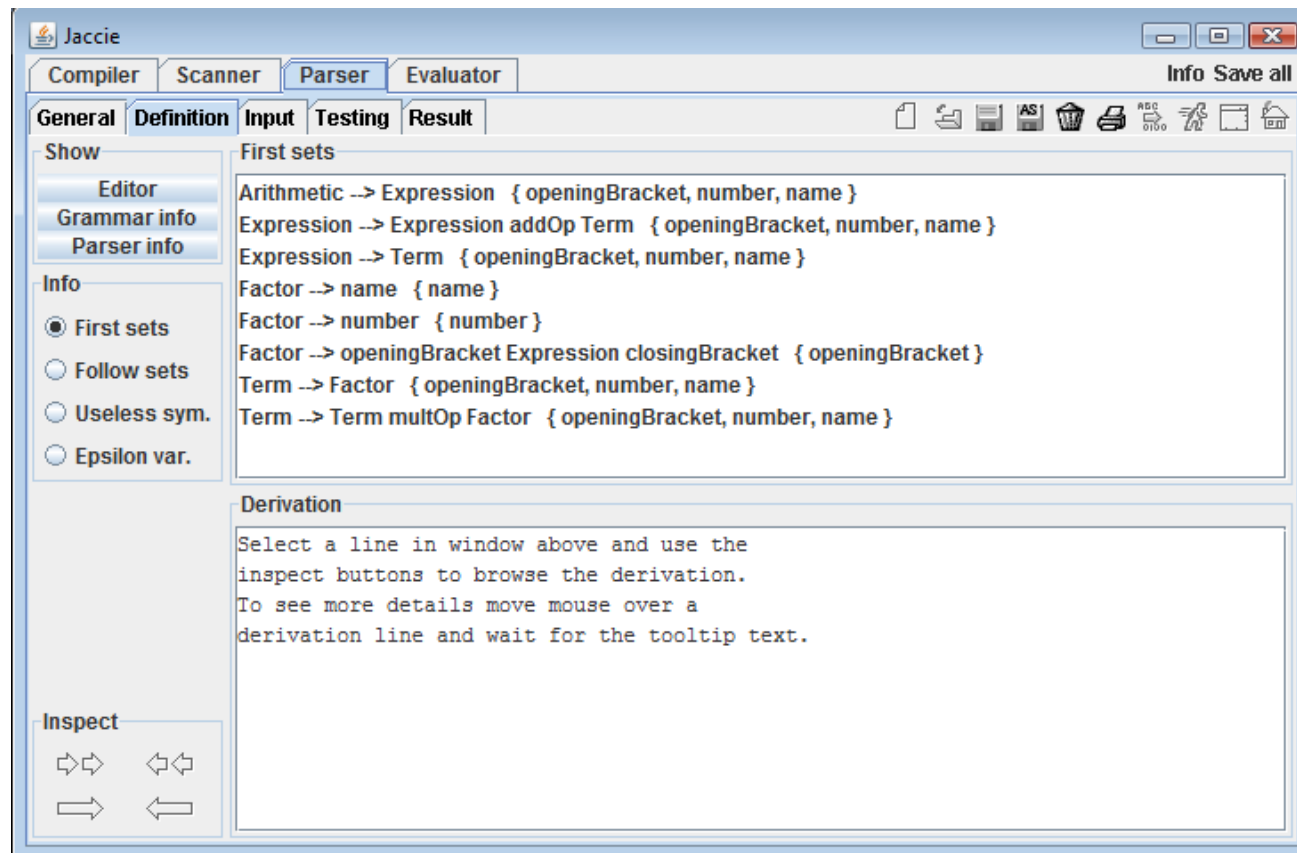
On its way, [Jaccie](#) has been able to undo all erroneous decisions using systematic backtracking. This is not always possible: If some parser state is reached for the second time after the last shift [Jaccie](#) suspects a cycle and stops backtracking. You can then either continue parsing or switch to conflict resolution *By user*.

Grammar info

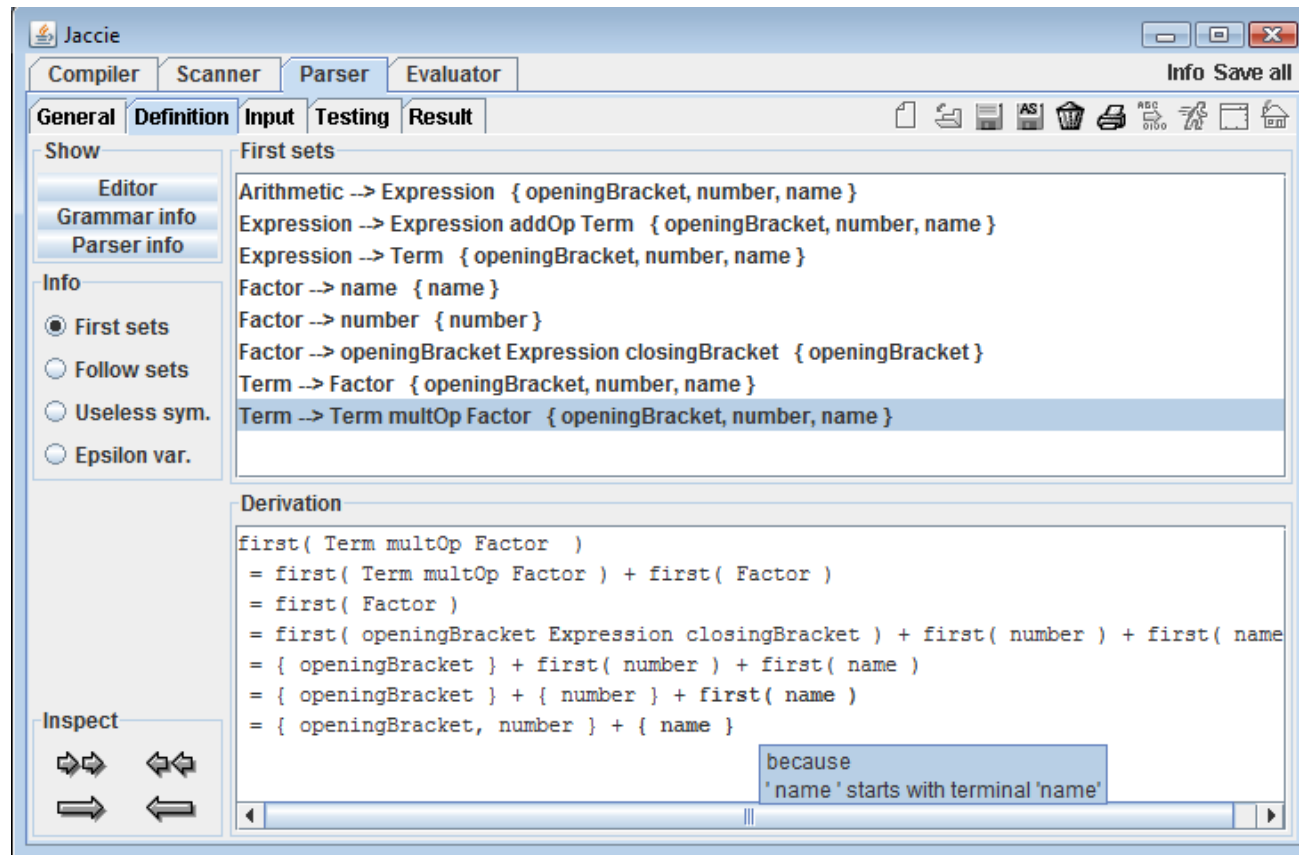
If in subarea *Parser-Definition* you choose *Grammar Info* from the *Show* panel **Jaccie** will show information sets that have been derived automatically from the user defined grammar and are relevant for the parsing process. This includes

- $first(\alpha)$: for string α the set of terminal symbols a word derived from α can start with
- $follow(A)$: for nonterminal symbol A the set of all those terminal symbols that follow A (i.e. are immediately to the right of A) in any sentential form

When being entered the information window displays a help text on its use in the lower panel:



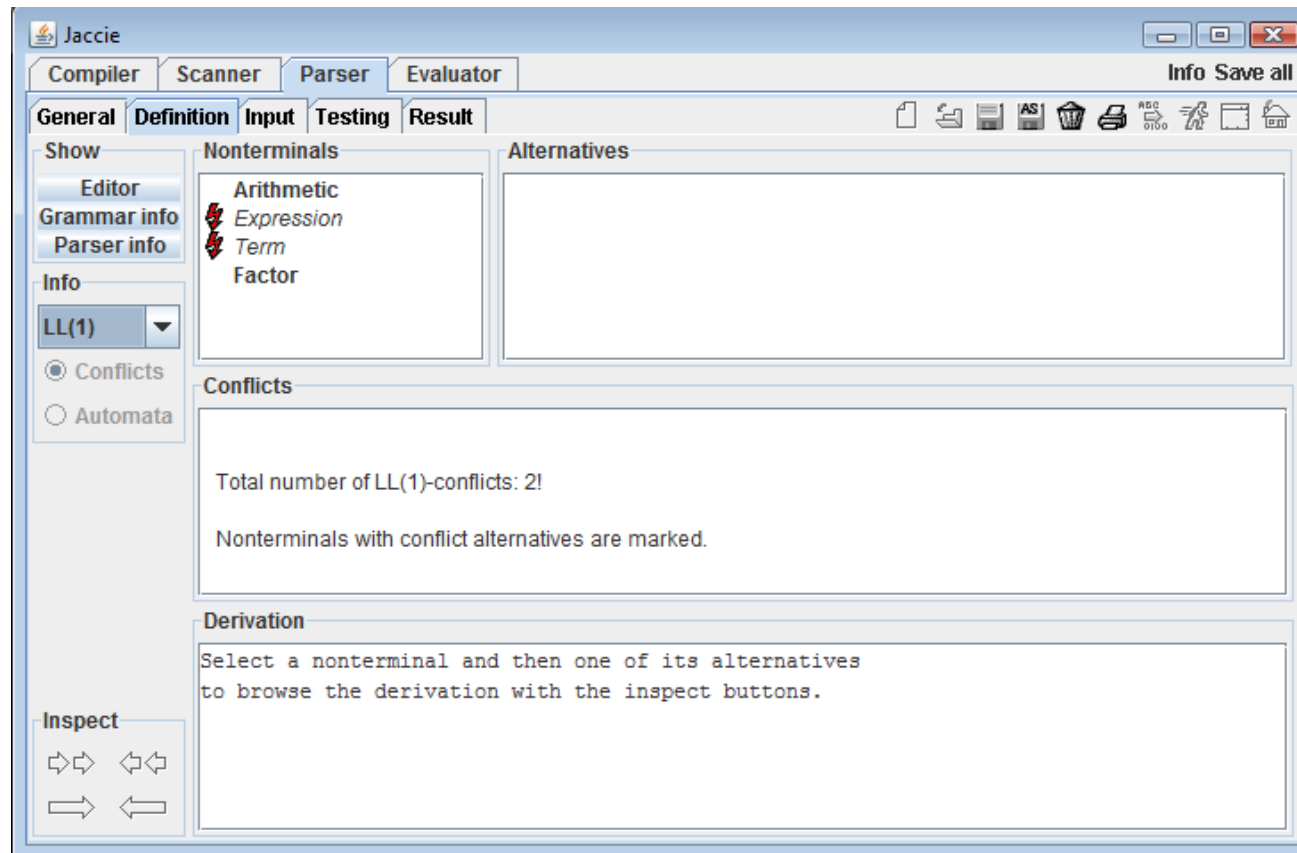
We proceed as suggested to learn about the set $first(\text{Term multOp Factor})$ which contains the terminal symbols a word derived from the right hand side of rule $\text{Term} \rightarrow \text{Term multOp Factor}$ can start with:



The set we are looking for ($\{\text{openingParenth}, \text{number}, \text{name}\}$) is already shown in the upper panel. The lower panel presents its stepwise derivation: Since Term is no ϵ symbol $first(\text{Term multOp Factor})$ equals $first(\text{Term})$. This in turn is equal to the union of $first(\text{Term multOp Factor})$ and $first(\text{Factor})$ – the first sets for both alternatives of Term. Of these only the nonrecursive alternative $first(\text{Factor})$ directly contributes to $first(\text{Term multOp Factor})$. $first(\text{Factor})$ is the union of the first sets for the three alternatives for Factor, i.e. of $first(\text{number})$, $first(\text{name})$ and $first(\text{openingParenth Expression closingParenth})$. The final result follows in three trivial steps (the last one is explained by the tool tip in the bottom right hand corner).

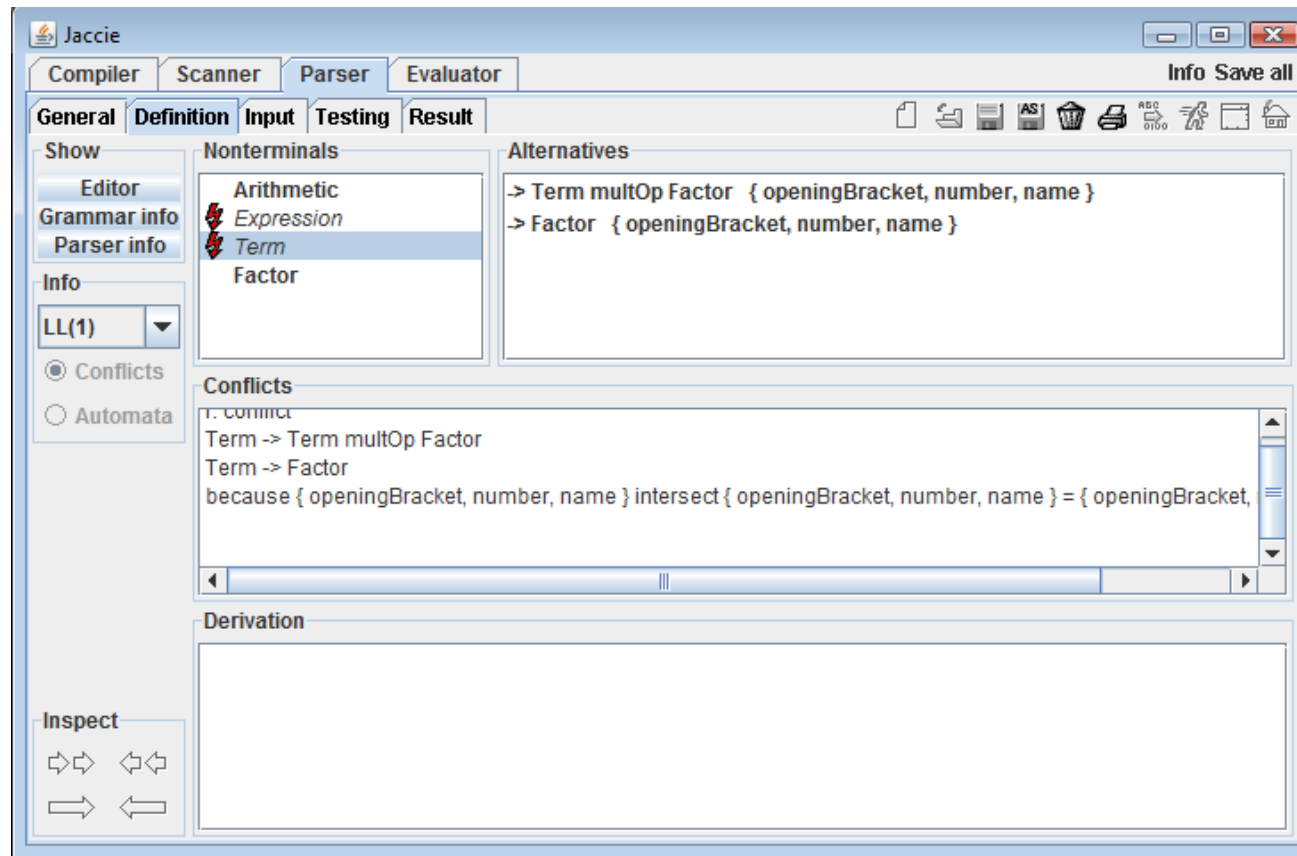
LL(1) Parser info

If in subarea *Parser-Definition* you choose *Parser info* from the *Show* panel and select LL(1) from the *Show* panel's drop down list, **Jaccie** will display the **LL(1) decision sets** that are used by LL(1) parsers to choose the correct alternative for expansion. In the absence of ϵ symbols the first sets of all alternatives must be pairwise disjoint (the criterion for ϵ symbols is slightly more complicated). In case two alternatives of the same nonterminal have overlapping decision sets, we cannot decide which one to use for expansion; this is called an **LL(1) conflict** – indicated by a red lightening arrow symbol:



In order to view the conflict for nonterminal **Term** we proceed as suggested by the help text in the *Derivation* panel.

The LL(1) decision sets for both alternatives belonging to **Term** are equal, i.e. not disjoint at all!



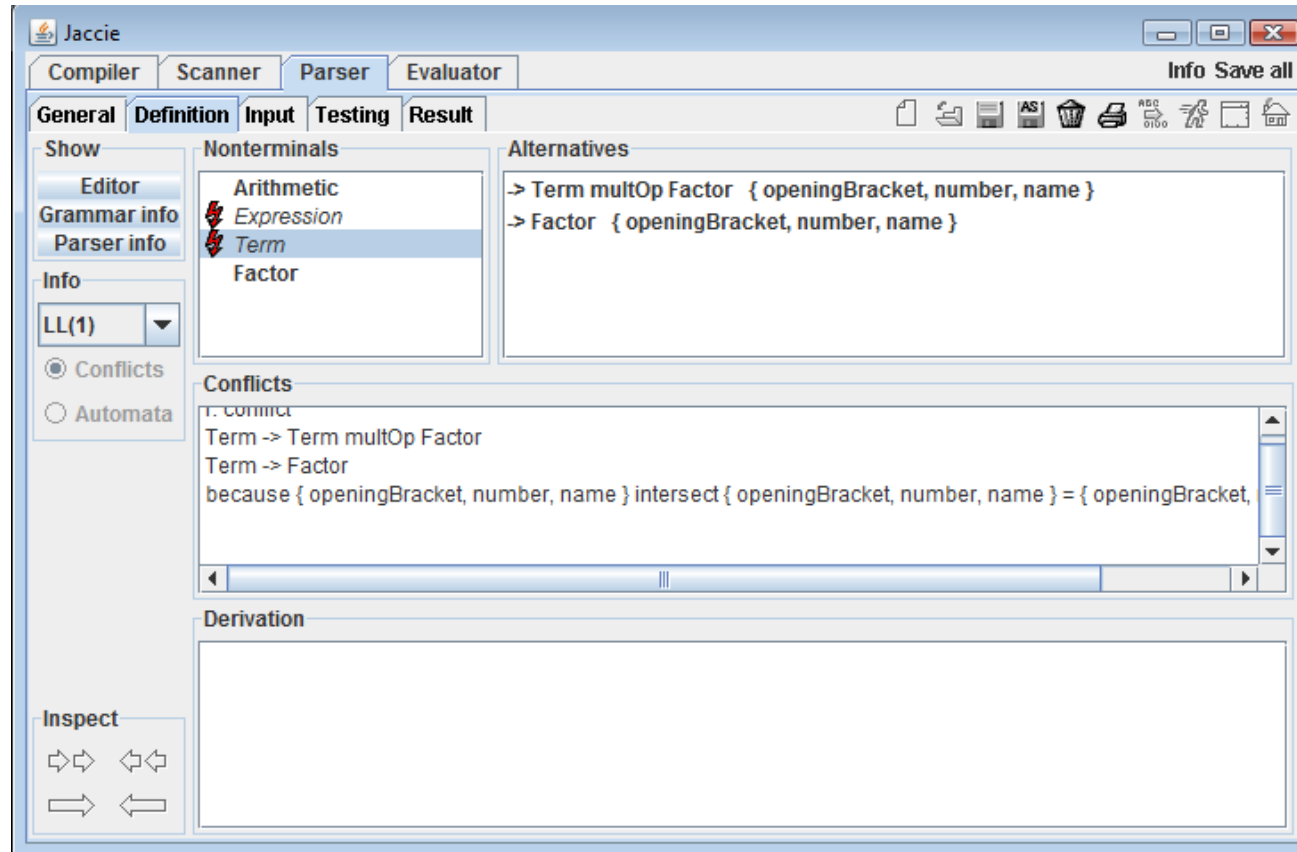
What to do in a situation like this?

In principle, there are two different approaches:

1. One is to transform the grammar in such a way that the same language is described without left recursive rules, thus eliminating LL(1) conflicts. Systematic transformations to this effect are described in the literature. Unfortunately, however, they produce rather unnatural grammars.
2. The other is to apply another, more powerful parsing algorithm to the original grammar. There is a good chance that one of the bottom up parsing algorithms can be applied deterministically to the grammar. (If not, your grammar probably tries to differentiate between entities that cannot be distinguished on the syntax level.)

LR Parser infos

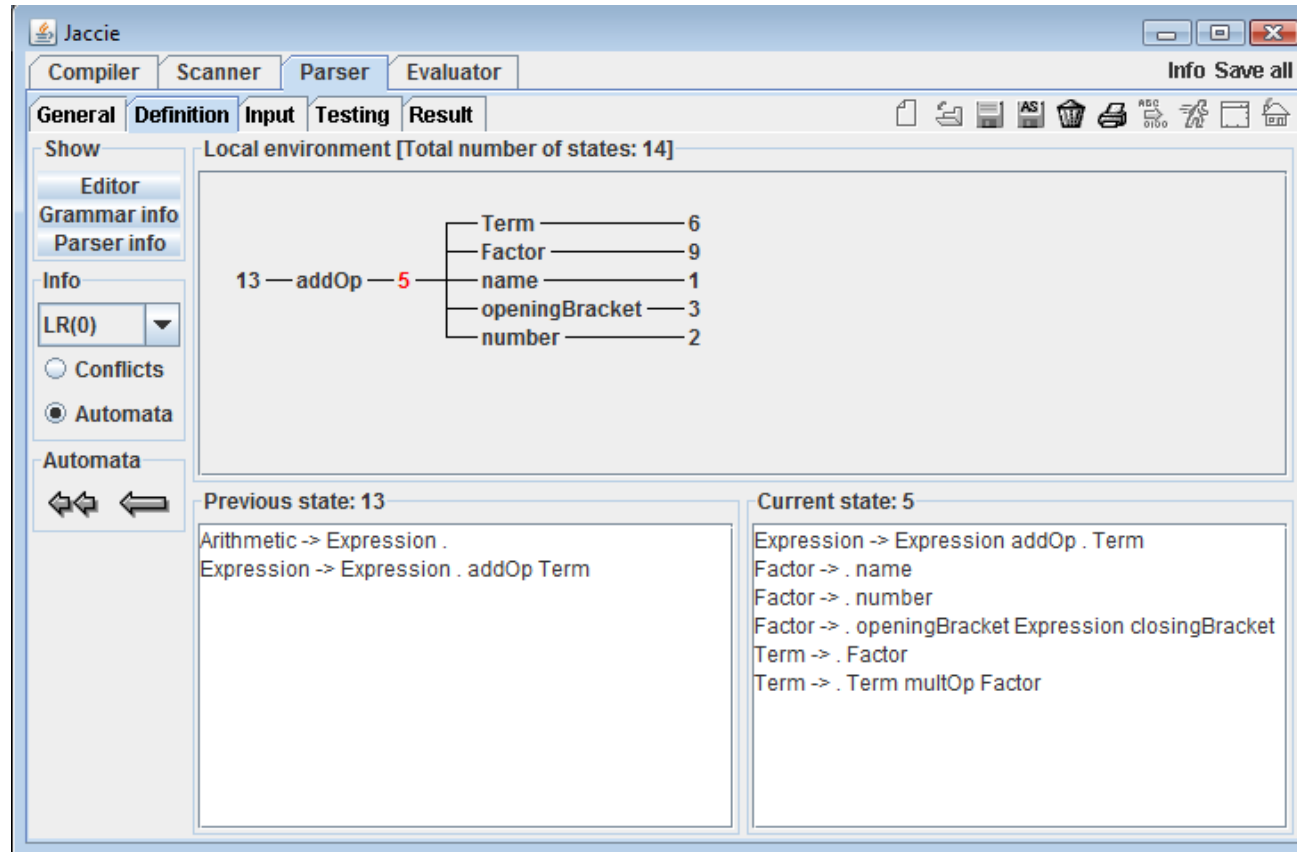
The LR family of bottom up parsing algorithms (LR(0), SLR(1), LALR(1), LR(1)) forms a hierarchy: each member is strictly more powerful than its predecessor (i.e., is deterministic for more grammars). Also, LR parsers tend to be more powerful than LL parsers. For an explanation, see section [Syntax analysis](#) of the theory paper. The screenshot below shows that the LR(0) parser for the grammar of arithmetic expressions has conflicts:



All three conflicts are **shift reduce conflicts**, i.e. parser states that define both a shift and a reduce action for the same input symbol. The upper panels display the details of a conflict in state 13 for input symbol `addOp`.

So why not use the most powerful parser (LR(1)) all the time?

The reason is *efficiency*: parsing automata for more powerful algorithms are harder to compute. Also, for practical grammars, there may be a combinatorial explosion in the number of parser states. Thus, given a grammar G it pays to find the least powerful parser which is deterministic for G . Unfortunately, there are languages for which no deterministic LR parser can be found (e.g., the language of palindromes over $\{a, b\}$).



If you select the radio button Automata (like in the screenshot above) the states of the parsing automaton can be explored dynamically in order to gain a better understanding. The bottom right panel shows the current state (initially the start state, here state 5). Its predecessor (here: state 13) is displayed in the bottom left panel. The top panel contains a portion of the automaton's state transition graph (specifically, the current state 5, its predecessor state 13 and its successor states 6, 9, 1, 3, and 2).

5. Attribute Evaluation

The translation proper, or synthesis, is done by the **attribute evaluator**. In the [Jaccie-Tour](#) we have had a first look at translation by attribute evaluation (starting on [page 15](#)). Section [Synthesis using attributes](#) of the theory paper explains the basic notions of attribute grammars and demonstrates their use giving two examples (how to translate arithmetic expressions into machine code and into Kantorovic tree representations, respectively).

For the rest of this section, we assume that you know all the basics about attribute grammars and how to use them for translation as described in the theory paper!

As a running example we shall use the **evaluation of binary constants** (floating point). The inventor of attribute grammars, Stanford professor Donald E. Knuth, used the same example when he introduced attribute grammars in his seminal paper. For completeness, we provide a [Jaccie](#) scanner by defining:

```
<zero>   := $0
<one>    := $1
<point>  := $.
```

According to this definition, the binary constant [110.011](#) has seven tokens. Essentially, we want to compute its value [6.375](#).

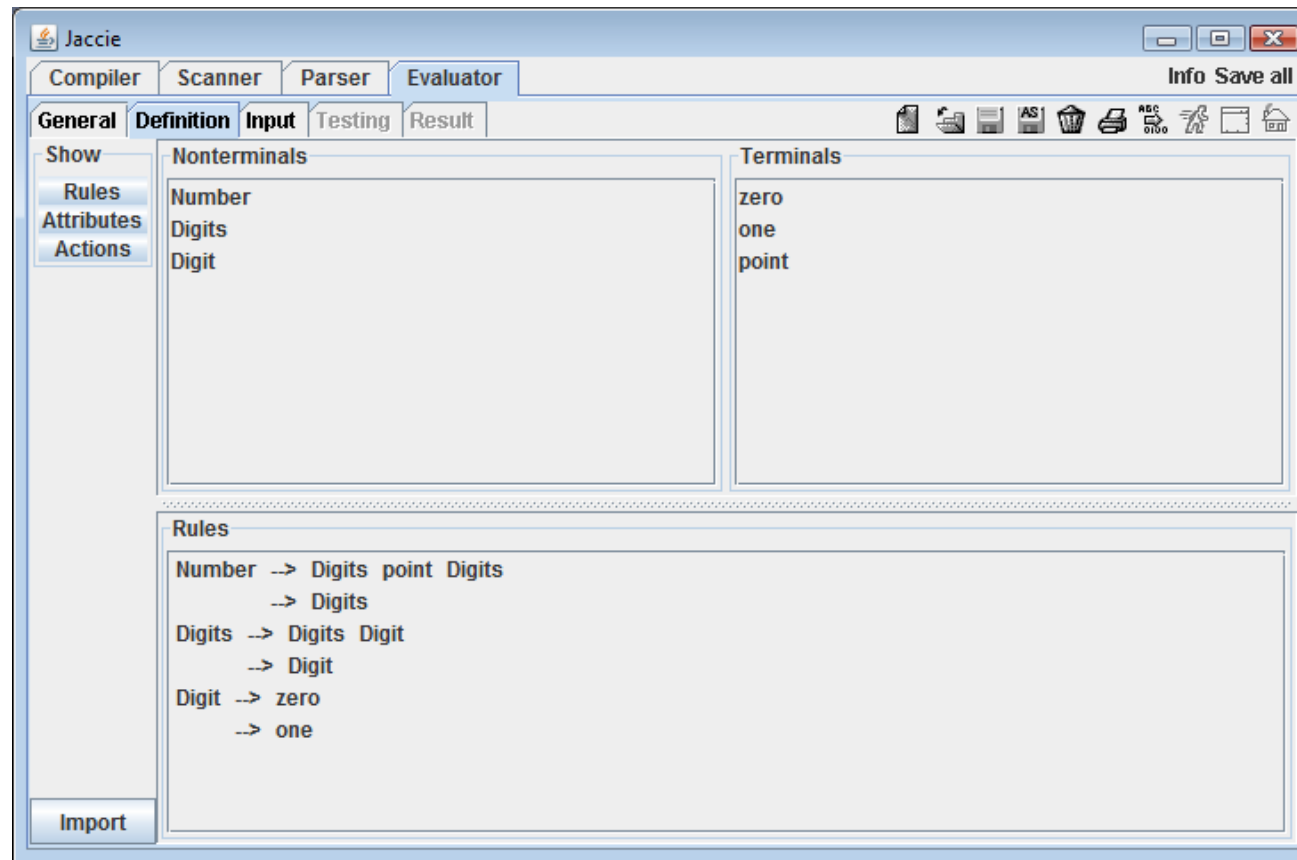
The screenshot of the attribute grammar editor on the next page shows a contextfree grammar defining binary floating point constants.

Defining attribute grammars

An **attribute grammar** consist of

1. an underlying contextfree grammar, G
2. definitions of synthesized and inherited attribute sets including type definitions; every nonterminal symbol of G is assigned a subset of the available attributes
3. attribute evaluation rules plus imports, global variables and subprograms (all these as Java code snippets)

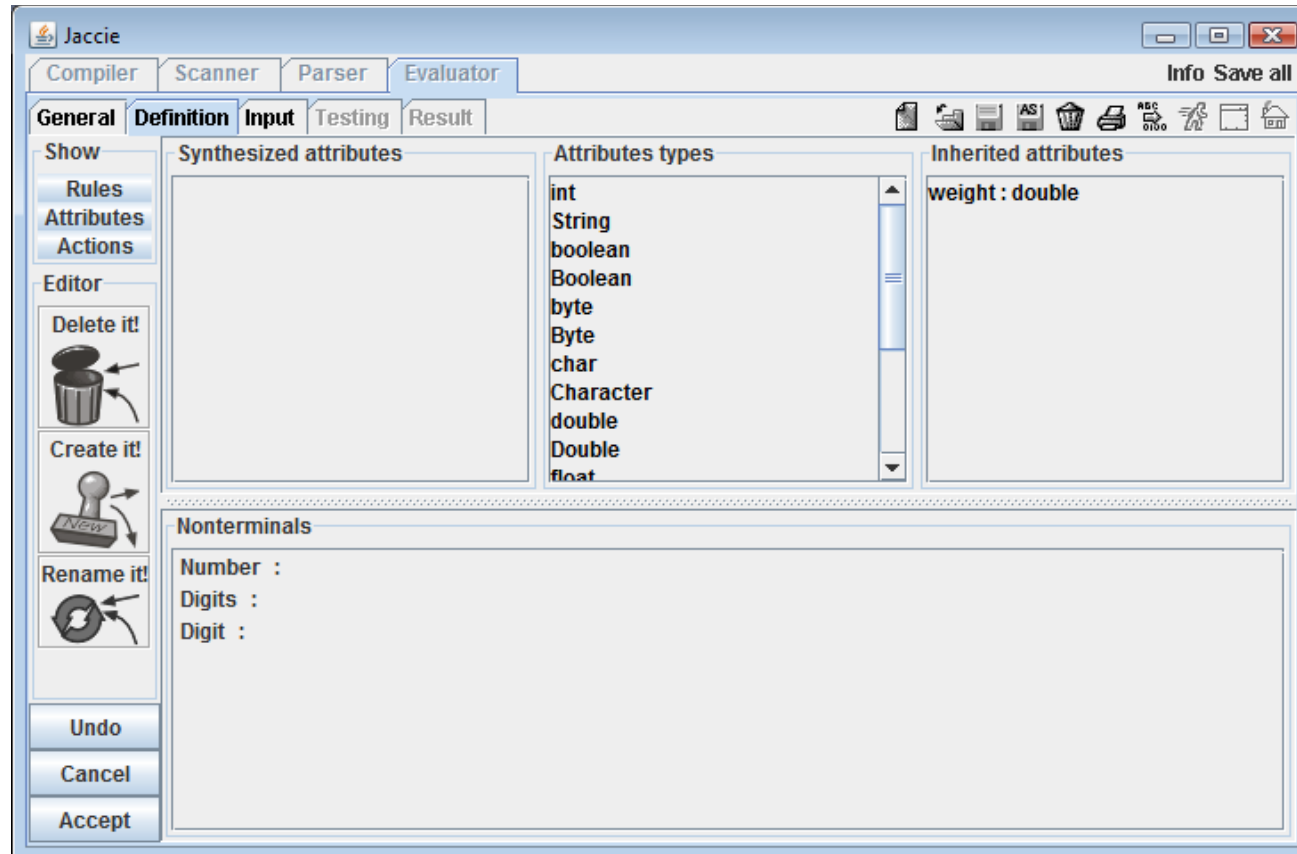
In its initial **mode**, the **attribute grammar editor** looks like the grammar editor used for parser definition:



The difference is that there are no controls for creating, deleting or renaming of elements. In fact, this mode of the *Evaluator-Definition* subarea is meant only for *viewing* the underlying contextfree grammar, not for editing. Using the *Import* button (bottom left) you can import an existing contextfree grammar.

According to the grammar shown above, a binary constant **Number** either consists of one digit sequence **Digits** only (integer constant) or of two digit sequences separated by a point (floating point constant). A digit sequence either consists of just one digit **Digit** or of a digit sequence followed by a digit. Since we are considering binary numbers, available digits are **zero** and **one**.

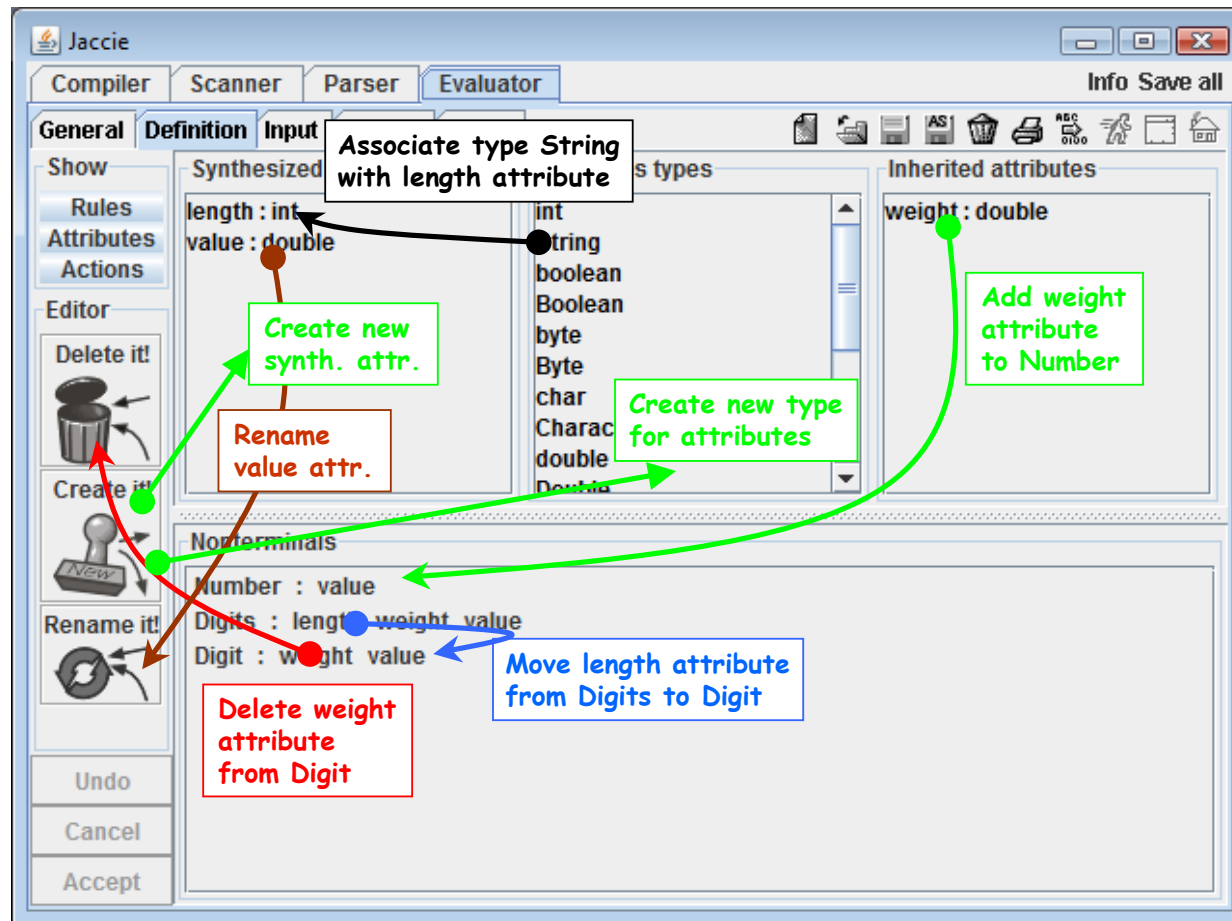
Using *Show-Attributes* we enter the second **mode** of the attribute grammar editor, where attributes are defined and assigned to the grammar's nonterminals:



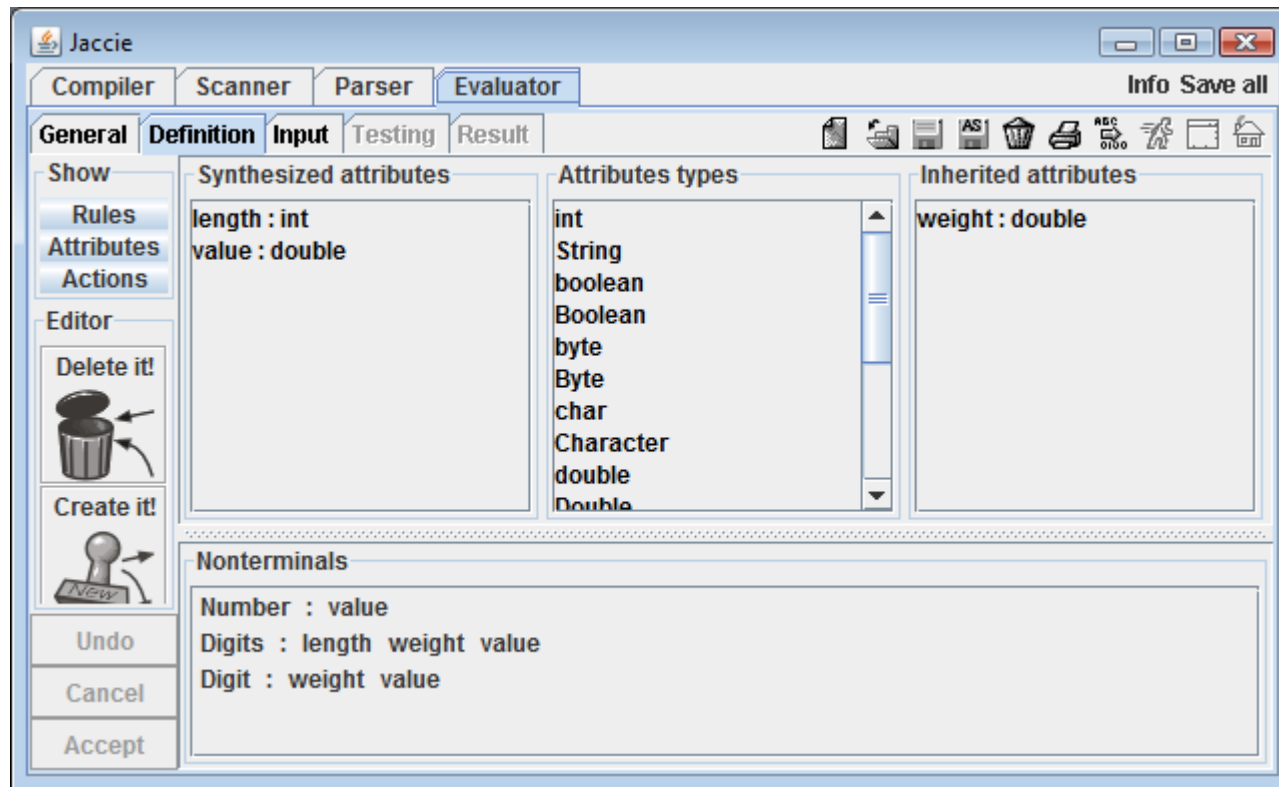
The controls on the left hand side resemble (in form and function) those of the *Parser-Definition* subarea. Again, direct manipulation via *drag-and-drop* is used for editing. The screenshot on the next page graphically indicates some of the possibilities for direct manipulation. Alternatively, in the three upper panel *context menus* are available. We will come back to that later.

Window layout: the top left hand and top right hand panels are for defining the sets of synthesized and inherited attributes, respectively. Available attribute types (Java) are displayed in the top middle panel; you can add new types as you wish. By dragging types to attributes types are assigned to attributes. Likewise, by dragging attributes to nonterminals (in the bottom panel) attributes are assigned to nonterminals.

Using the *Undo* button editing operations can be undone step by step. Before leaving the editing mode, you either have to accept the effect of all modifications (since editing began) using the *Accept* button or cancel them all using the *Cancel* button. Only then *Jaccie* will let us leave this editing mode.



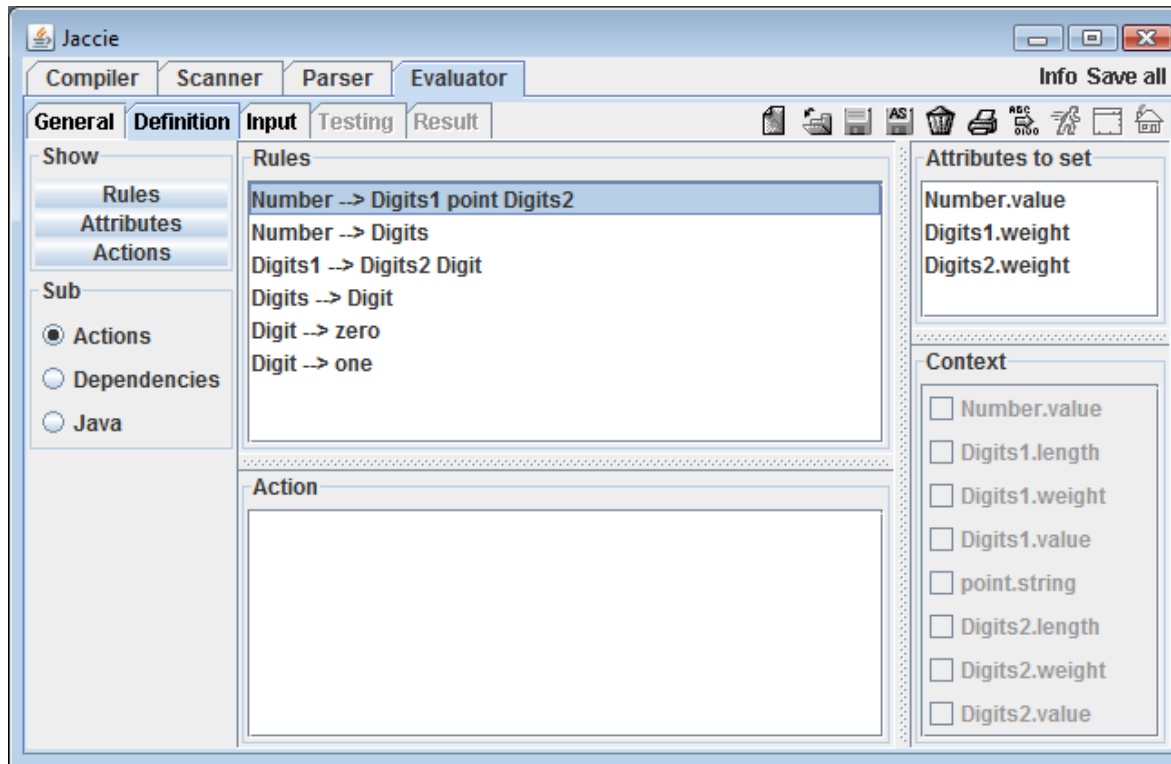
Using direct manipulation we define two synthesized attributes (`length` and `value`) and an inherited attribute (`weight`), assign types to the attributes (`int` and `double`, respectively) and assign attributes to nonterminal symbols, resulting in the situation shown on the next page.



Why have we chosen attributes and attribute types that way?

- We aim to compute the value **value** of a binary number constant. Since floating point constants are considered, too, **double** is a suitable Java type for **value**. The value of a number results from the value(s) of the digit sequence(s) it contains. In turn, the value of a digit sequence results from the value of the digits it contains. Since values are computed proceeding from the leaves of the syntax tree towards its root, **value** is a *synthesized* attribute.
- The value of a digit depends on the digit's weight. Weight information will be stored in **weight** attributes. The weight of a digit depends on the position of the digit within its digit sequence and whether that sequence is the integral or the fractional part of the number. Since weights are computed proceeding from the root of the syntax tree towards its leaves, **weight** is an *inherited* attribute.
- For computing the weight of the rightmost digit within the fractional part we need the **length** of this digit sequence. **length** like **value** is a *synthesized* attribute.

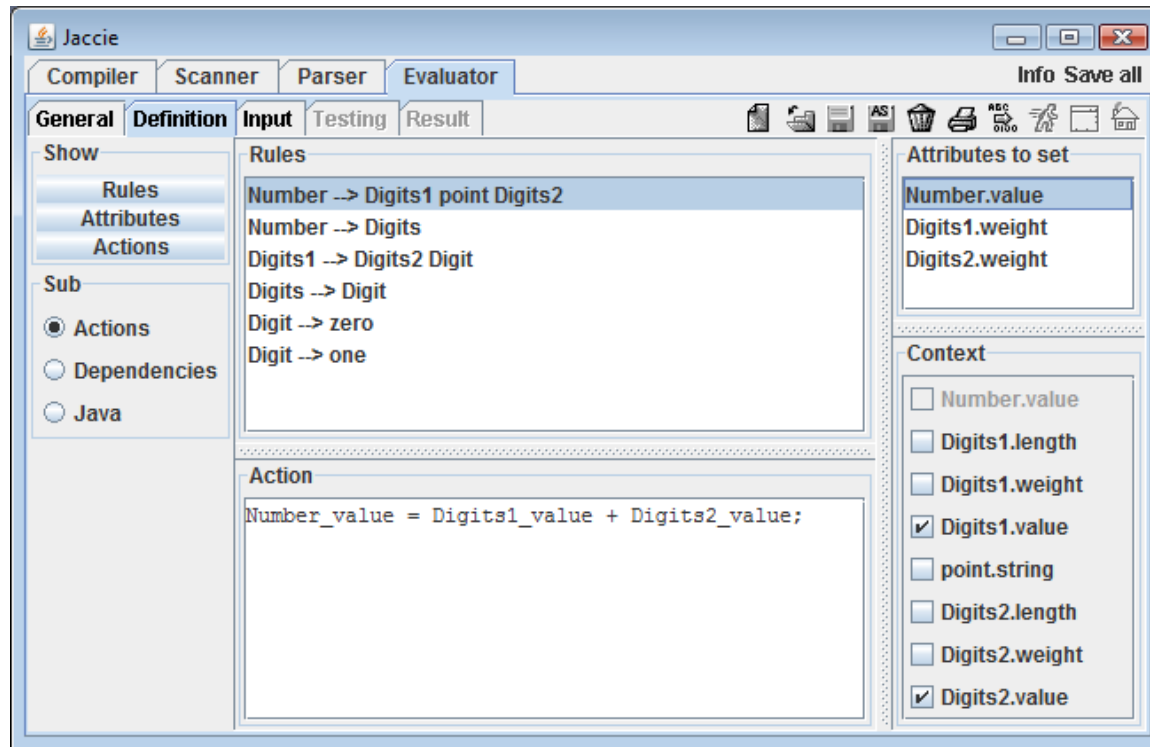
From the attributes and their semantics that we have defined on the previous page the attribute evaluation rules for computing attribute values follow almost unavoidably: *Attribute definition is the creative part of writing an attribute grammar!* Jaccie supports writing attribute evaluation rules by providing bookkeeping and dependency management functionality. Choosing *Show-Actions* we enter that **mode** of the attribute editor where attribute dependencies are defined and evaluation rules are written in Java:



In this mode there are four panels. In the top left hand panel we see the grammar's production rules. Every attribute evaluation rule is defined in the context of a production rule. It describes how in this context to compute an attribute value either directly or using other attribute values from the same context. Once a production rule has been selected, the attribute to be computed in this context appear in the top right hand panel. There, you select the attribute for which you want to write an evaluation rule next. Before actually writing the Java code of the evaluation rule into the bottom left hand text panel, be sure to choose (from the bottom right hand panel) all attributes this attribute will depend on - if you forget to do so, required attribute values will not be available in the evaluation rule.

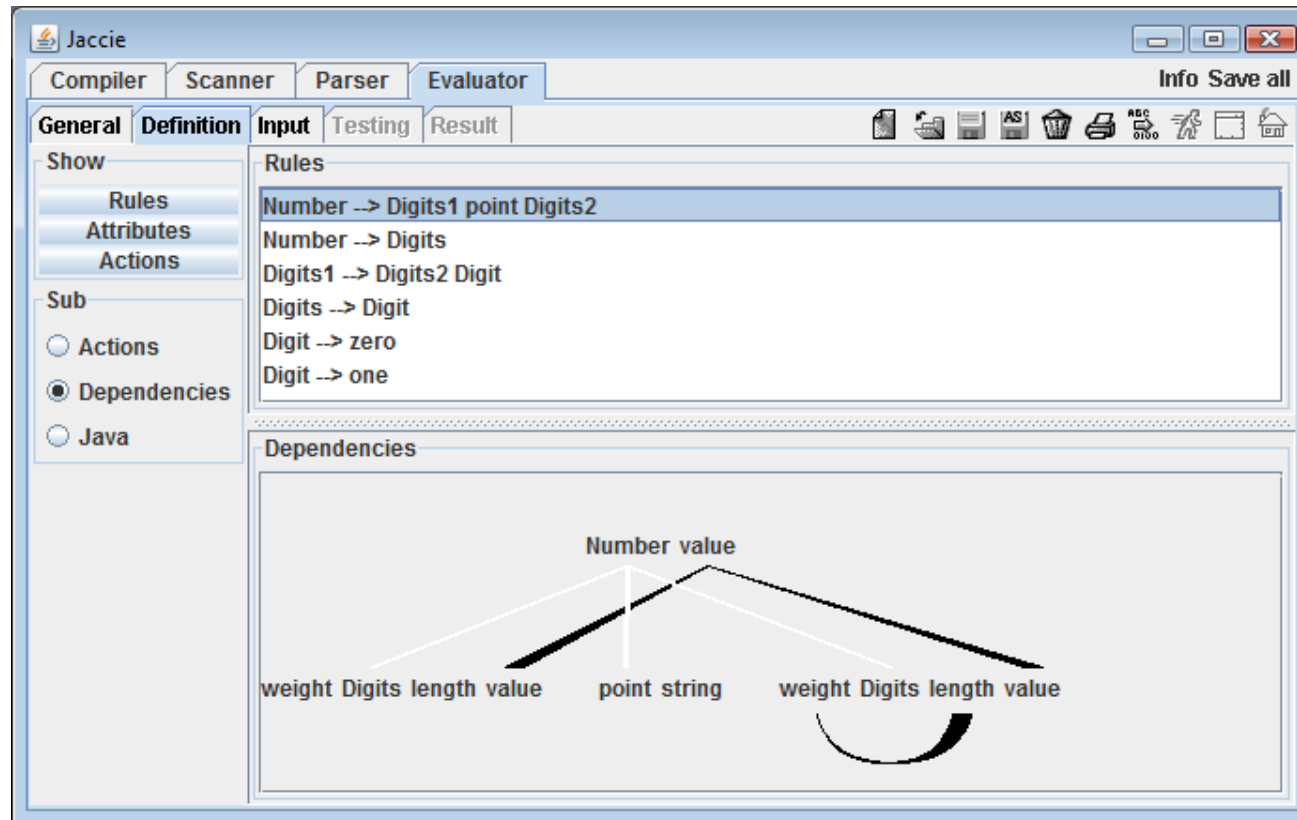
Here, we have selected the production rule `Number -> Digits point Digits`. In this context we have chosen to define the evaluation rule for attribute `Number.value`. Two minor but important notational details are:

1. Jaccie automatically numbers different occurrences of the same symbol within one production rule from left to right. This helps to uniquely address the different symbol occurrences in the attribute evaluation rule.
2. In the Java text of attribute evaluation rules points between nonterminal name and attribute name are replaced by underscores, e.g. `Number_value` instead of `Number.value`.



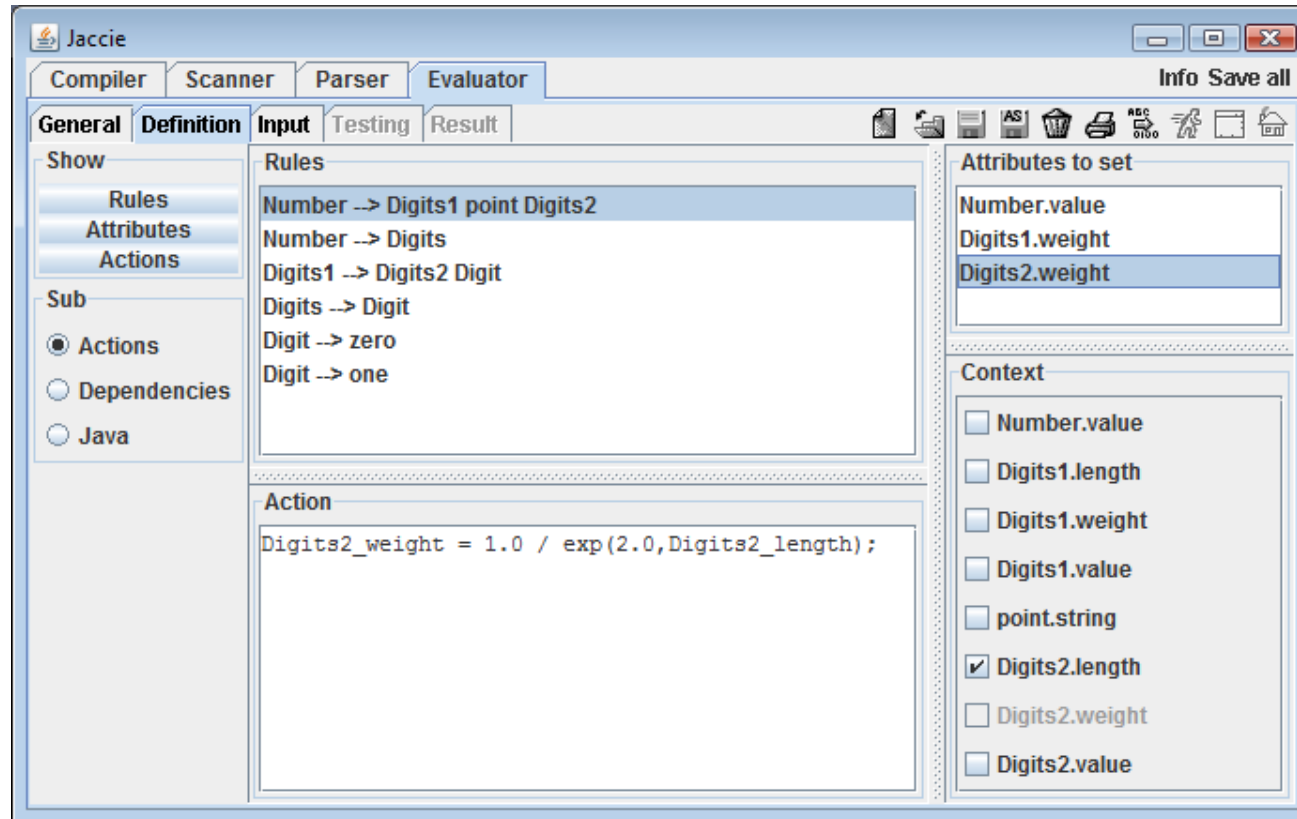
According to the attribute evaluation rule for `Number.value` (shown in the Action panel) the attribute `Digits1.value` is required for computing `Number.value` (`Digits2.value` as well). Therefore, `Digits1.value` has been checked in the Context panel. As a consequence, Jaccie generates a local variable `Digits1_value` that is initialized with the value of `Digits1.value` taken from the attributed syntax tree. Conversely, attribute values computed in local variables like `Number_value` are stored back into the tree. User-provided evaluation rules are embedded by Jaccie in a statement sequence that transfers attribute values to the rule and back into the tree.

Selecting the radio button *Sub-Dependencies* we enter a **submode** where all attribute dependencies from the current context (i.e., selected production rule) are visualized in the lower panel:



White edges connect the left hand side of the production rule (on top) with the symbols on the production's right hand side (bottom row).

Each black, pointed edge represents an attribute dependency. Two straight edges show that the attribute `Number.value` depends on both `Digits1.value` and `Digits2.value`. The 'half circle' edge indicates that `Digits2.weight` depends on `Digits2.length`. On the next page we shall see the corresponding attribute evaluation rule.

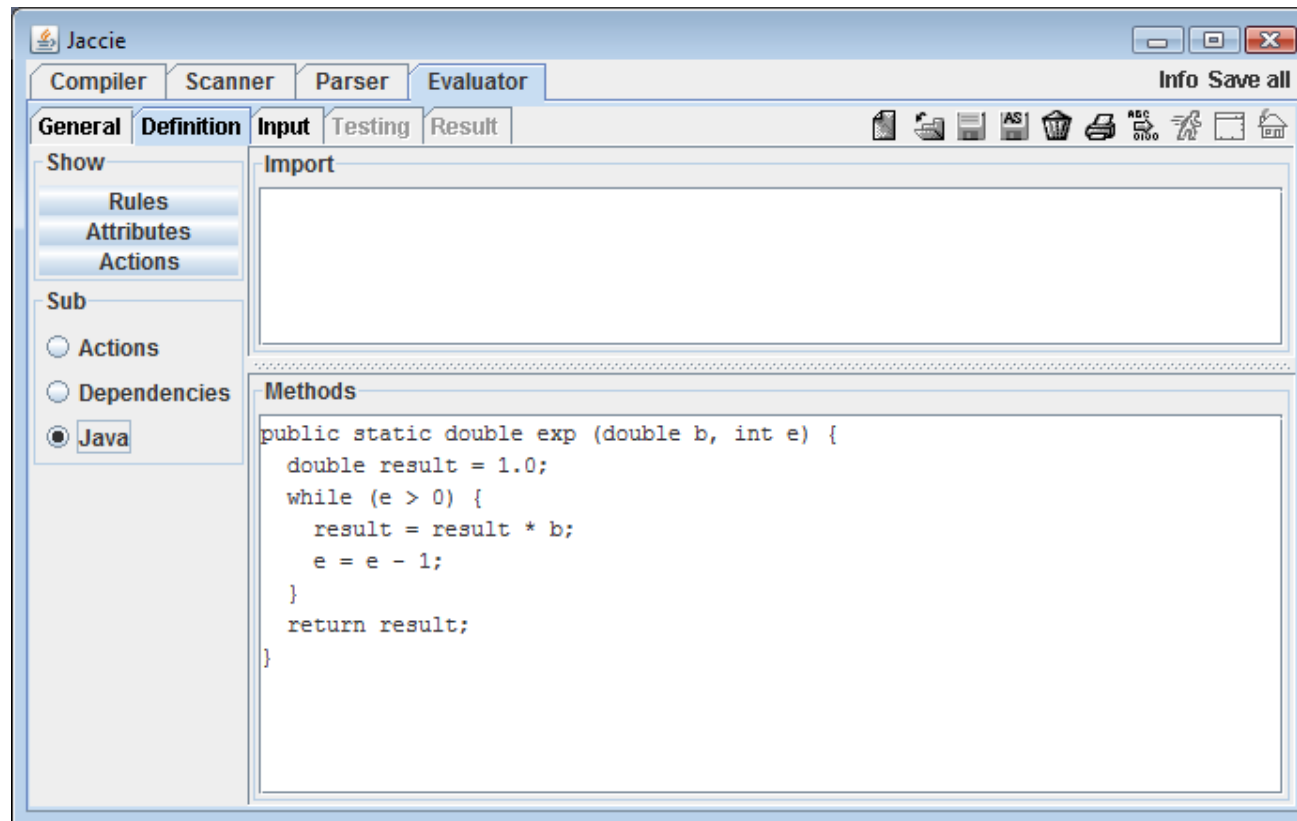


The weight `Digits2.weight` of the rightmost digit within the fractional part `Digits2` is computed from the fractional part's length `Digits2.length` using the formula:

$$\text{Digits2.weight} = \frac{1.0}{2.0^{\text{Digits2.length}}}$$

Unfortunately, standard Java does not provide an exponentiation operator. It is not difficult to implement a Java exponentiation method. The problem is rather: Where to put the text of auxiliary methods so that they can be used in Java attribute evaluation rules?

For this kind of material there is another **submode** (entered via radio button *Sub-Java*). All kinds of auxiliary Java definitions written into the *Methods* text panel will be integrated into the scanner's declaration part:




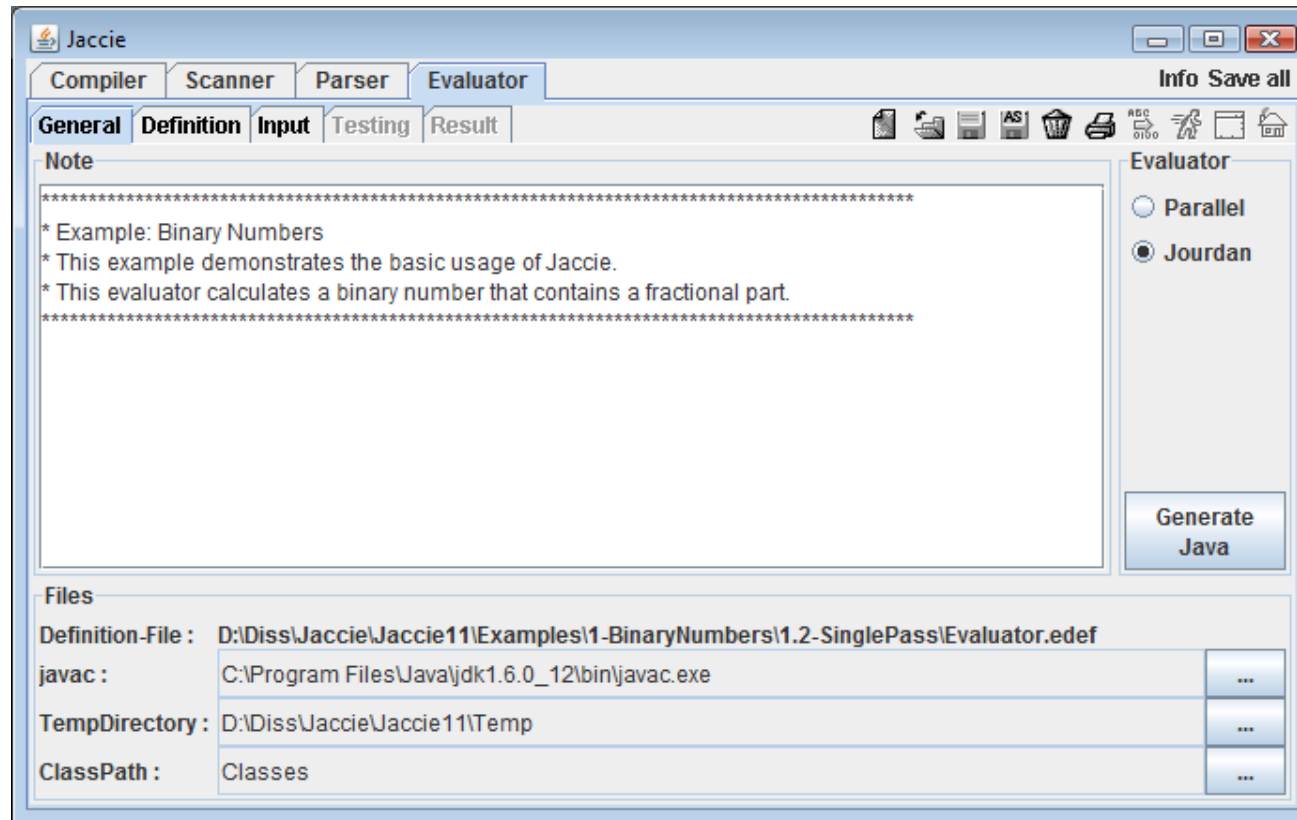
The method text is straightforward. As the `static` modifier indicates, this is not a method addressed to evaluator objects, but rather a 'classical' utility subroutine.

More comprehensive components would better be implemented in separate auxiliary packages and be integrated via import statements (written into the *Import* text panel).


These mechanisms are already known from the scanner definition.

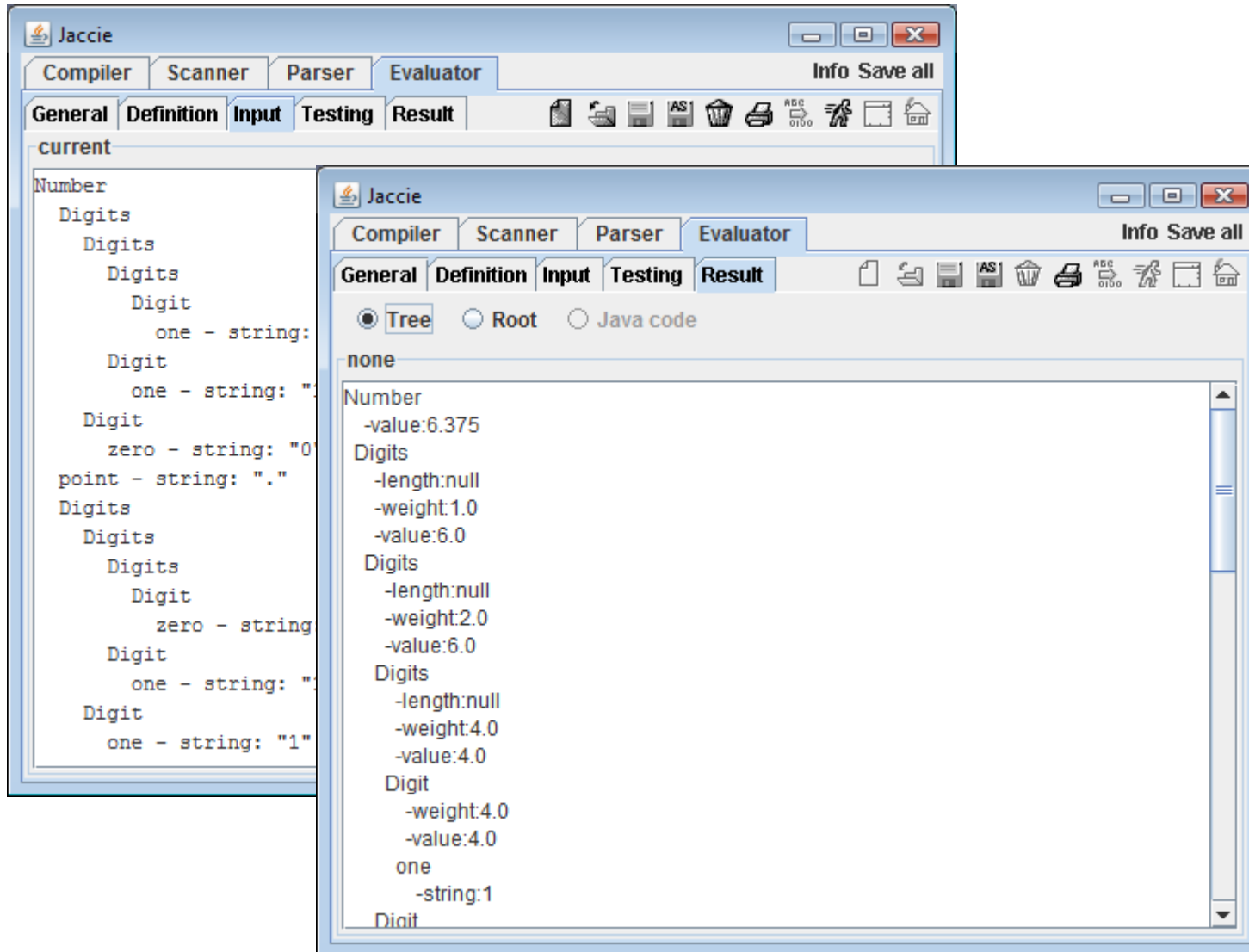
Testing attribute evaluators

We first save the attribute evaluator definition in the *Evaluator-General* subarea using the *SaveAs* button (). The file name will appear as *Definition-File* (see screenshot). Here, you can also define the attribute evaluation strategy to be applied. Currently, there are two *dynamic* evaluation strategies to choose from: The *Parallel* ([Jaccie](#) name for **data driven evaluator**) and the *Jourdan* ([Jaccie](#) name for **demand driven evaluator**) strategies.



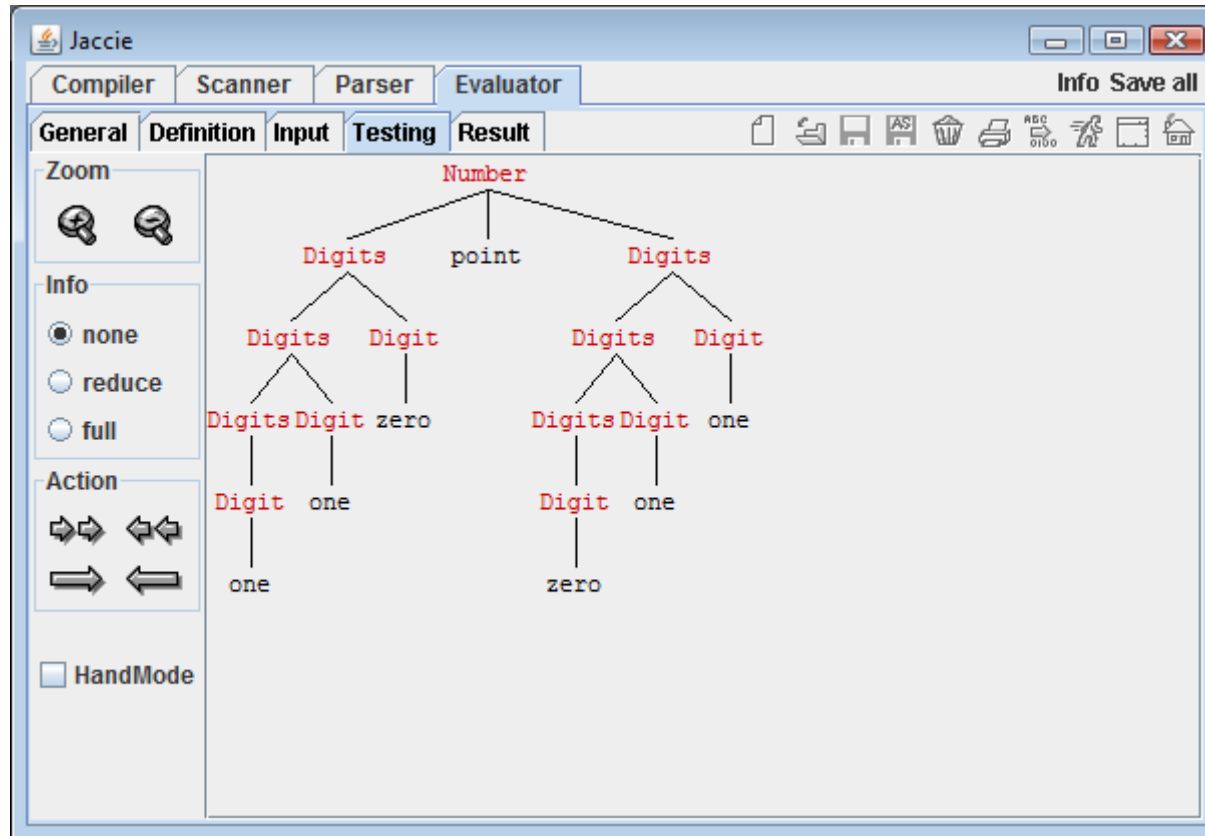
*Please note: In the absence of cyclic definitions (and other errors), the Parallel evaluation strategy will compute **all** attribute instances in the syntax tree. In contrast, the Jourdan evaluation strategy will compute only those attribute instances that are (directly or indirectly) required for computing the root's attribute instances. If there are no root attribute instances, nothing will be computed!*

Like the scanner and the parser the attribute evaluator can be started from the *Evaluator-Input* subarea using the *Run* button (). The resulting attributed tree will appear in the *Evaluator-Result* subarea:



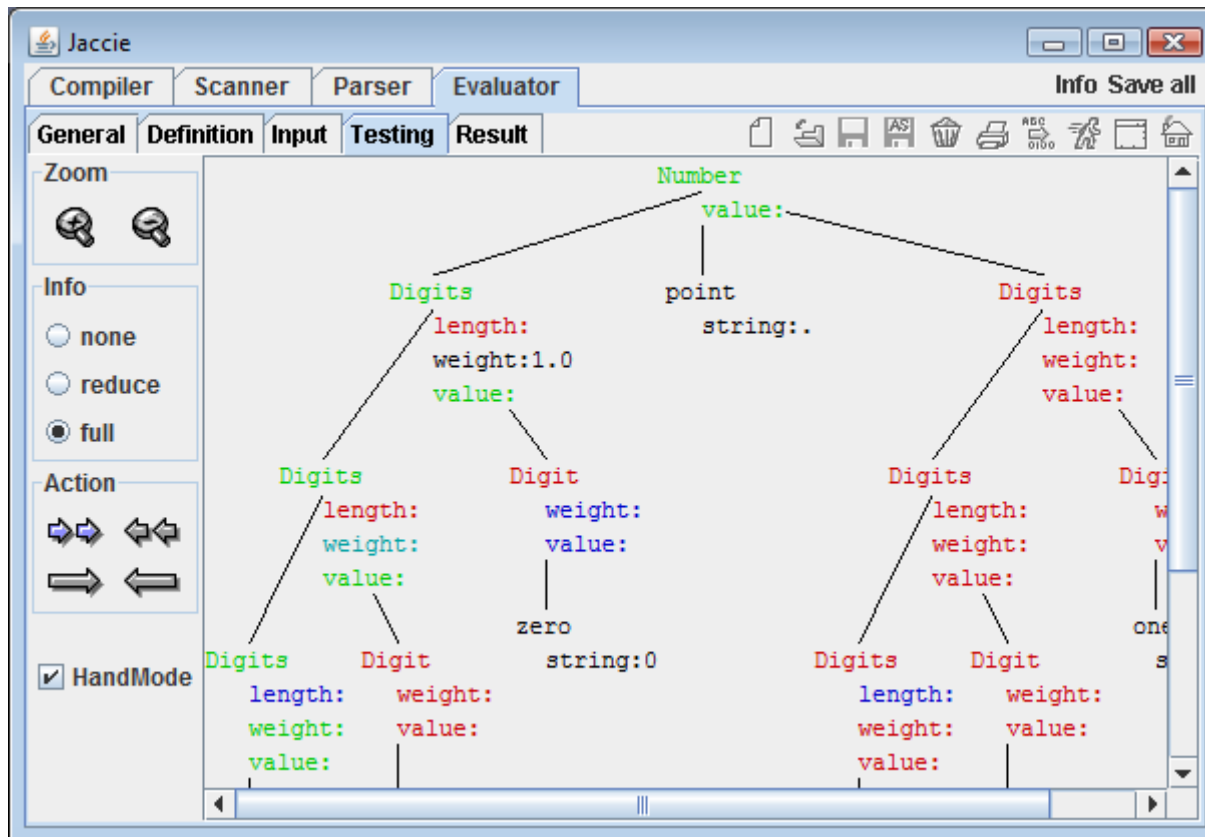
Note that evaluation of the binary number constant `110.011` results in the decimal value `6.375`, as expected!

In order to watch the attribute evaluation process in detail, we move on to the *Evaluator-Testing* subarea where (like in the preceding phases) there is a **visual debugger**:



The control elements in the *Zoom* and *Action* panels are analogous to those of the **parser debugger**. The order of attribute evaluation is determined by the attribute evaluation strategy chosen. Using the short arrows you can proceed in single steps either forwards or backwards. The long arrows will bring you directly to either end of the evaluation sequence.

Using the *Info* radio buttons you can control the amount of attribute information displayed: In mode *Info-none* (as in the screenshot above) only the syntax tree is shown. Nodes in red colour contain attribute instances to be evaluated. In mode *Info-reduce* additionally the attribute names are shown and in mode *Info-full* we see the attribute values as well. Let us now choose this latter mode.



The *Colours* are to be interpreted as follows:

Red are nodes (attribute instances, respectively) whose evaluation has not been started yet.

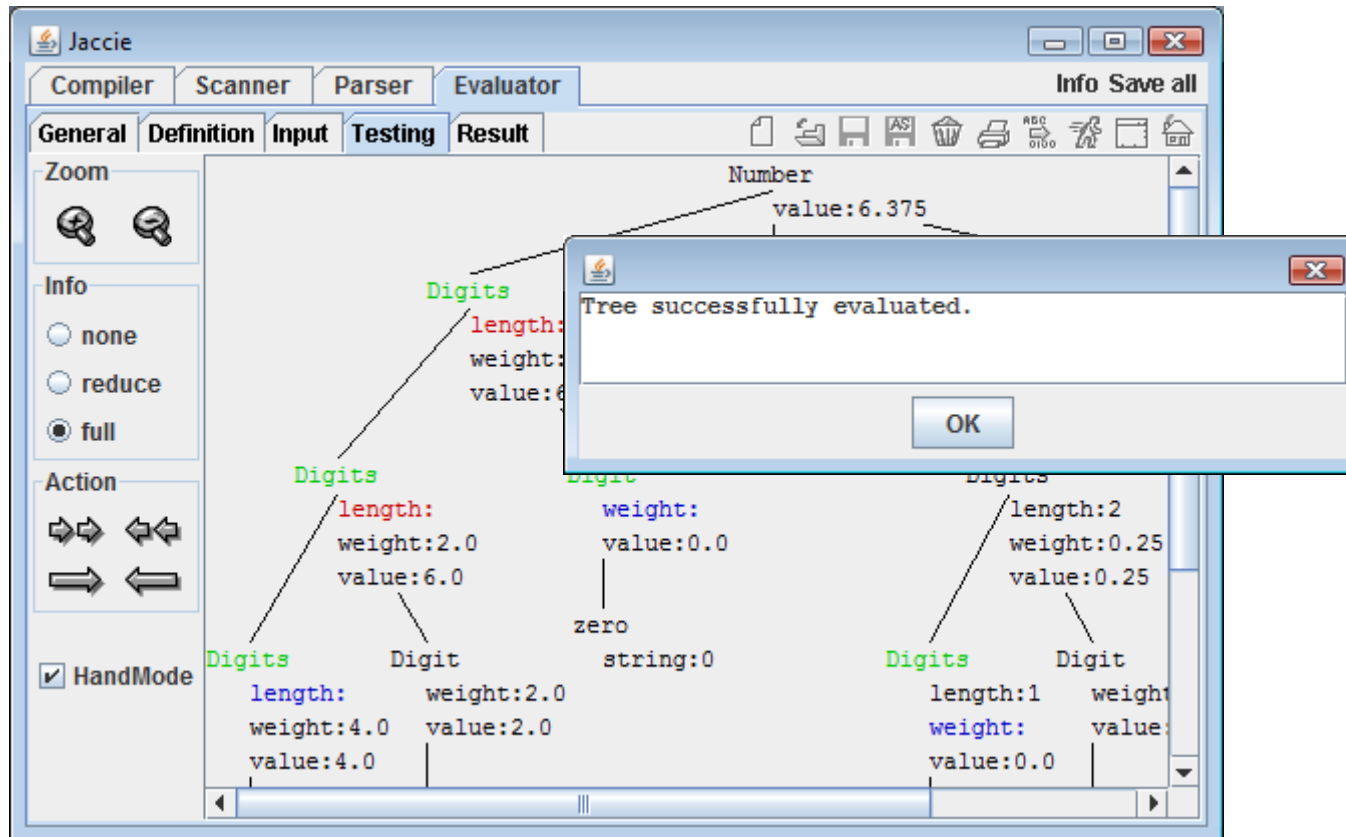
Green are nodes (attribute instances, respectively) whose evaluation has been started but not yet been completed.

Blue are attribute instances ready for evaluation in HandMode since all attributes they depend on are available.

Black are nodes (attribute instances, respectively) that have been evaluated completely.

(Watching closely, you will detect blue-green attributes combining the properties of blue and green attributes.)

Using the long forward arrow to move to the end of the evaluation sequence we obtain the situation shown below:

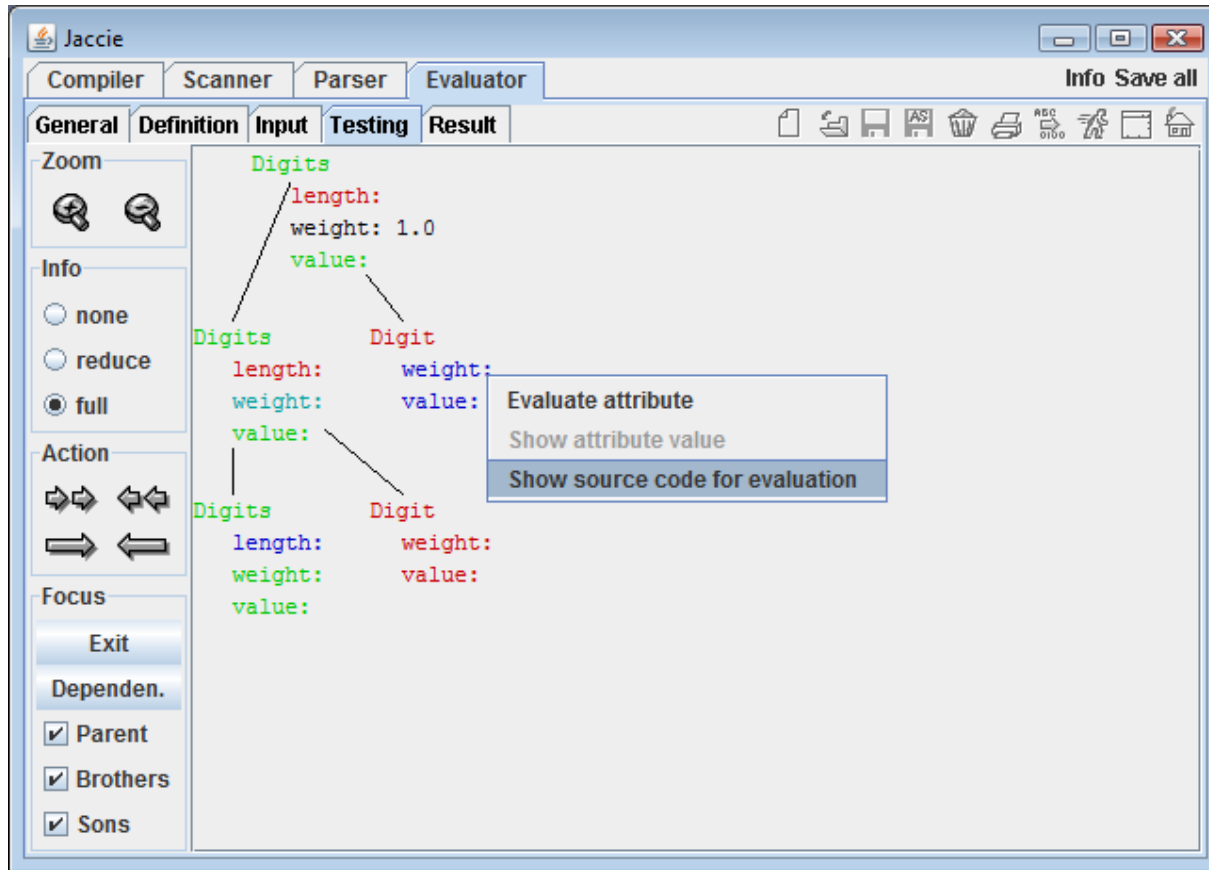


The decimal value 6.375 of the binary number appears as the value of the black attribute *value* of the black root node *Number*. You might wonder why at the end of the evaluation sequence not all nodes and attributes are black. The reason is that some attribute instances are not needed to compute the root attributes:

- The value of a binary digit 0 does not depend on the digit's weight. (For binary digits 1 this does obviously not hold.)
- The length of a digit sequence is relevant only in the number's fractional part, not in its integral part. Thus the *length* attributes in the left hand part of the syntax tree are not evaluated - in HandMode, however, you can explicitly trigger their evaluation.

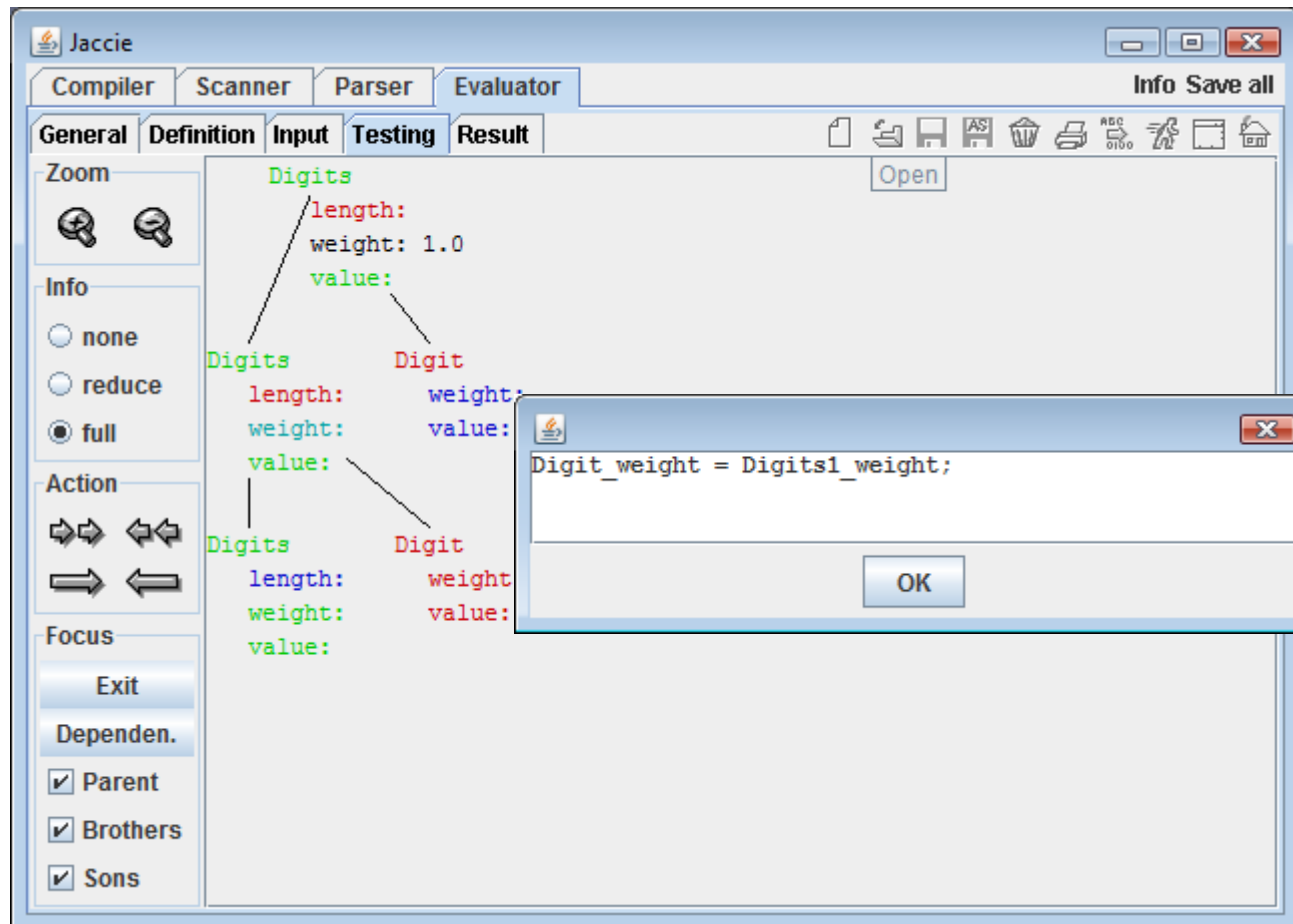
Focus on evaluation details

Attributed trees tend to be large and, hence, confusing: You can easily loose track of what is going on. By clicking on a tree node you can enter a 'focused' view showing the details of the chosen node and its immediate neighbours. In the screenshot below, *focus* is on the *Digits* node with the blue-green *weight* attribute:



Using *context menus* you can obtain more information about attributes or trigger their evaluation. Here, the blue *weight* attribute of the centre *Digit* node has been clicked with the right hand mouse button. (This works in full tree mode as well.) Now selecting the last menu item will result in the situation shown on the next page.

Here, you see the attribute evaluation rule for computing the inherited `weight` attribute of `Digit` - in this case the value `1.0` of the `weight` attribute from the `Digits` node would be propagated:

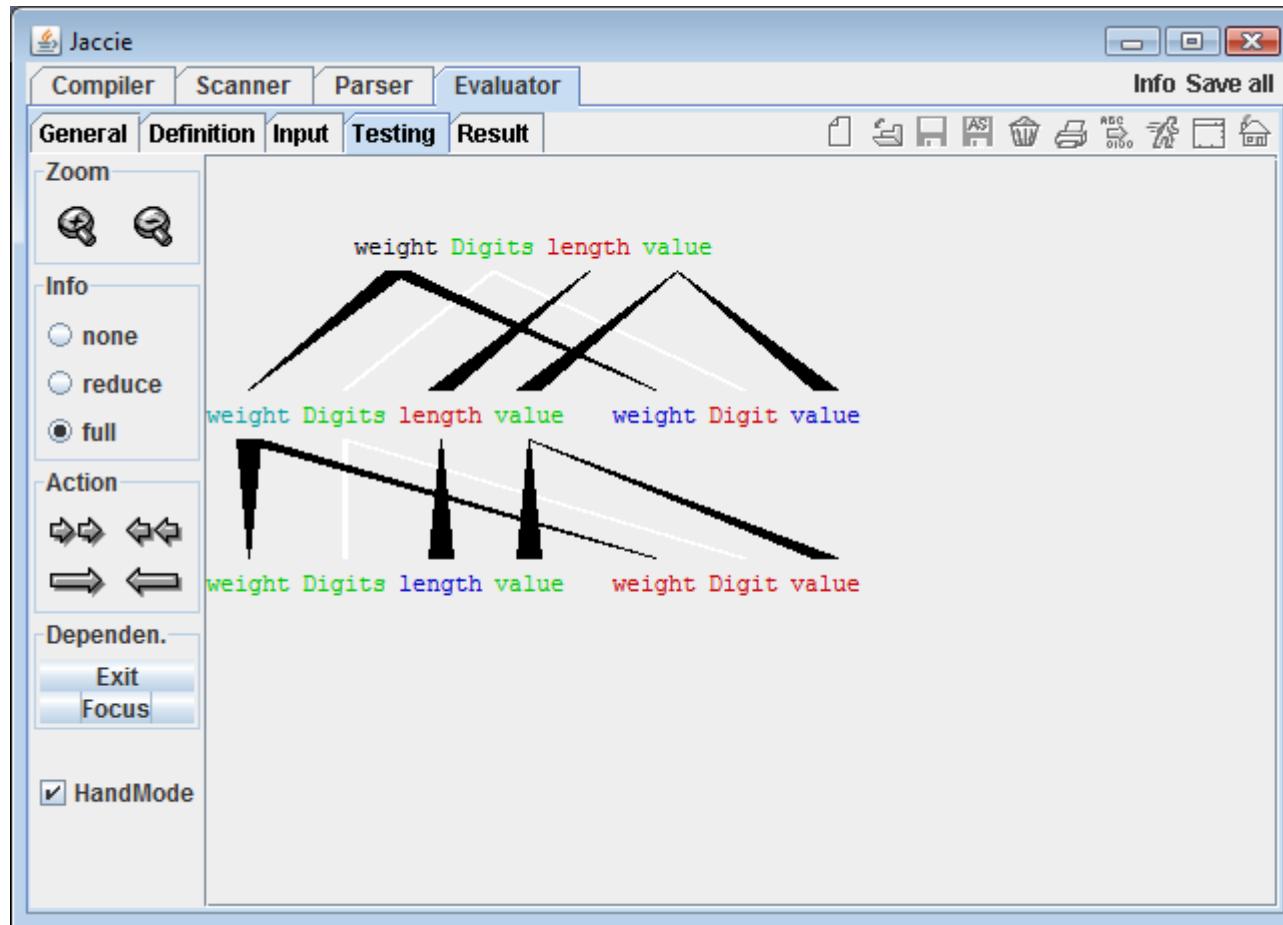


The controls in the *Focus* panel allow you

- to return to the full tree view (*Exit*),
- to switch on / off neighbours of the focused node (*Parent*, *Brothers*, *Sons*)
- or to show all attribute dependencies within this part of the syntax tree (*Dependen.*).

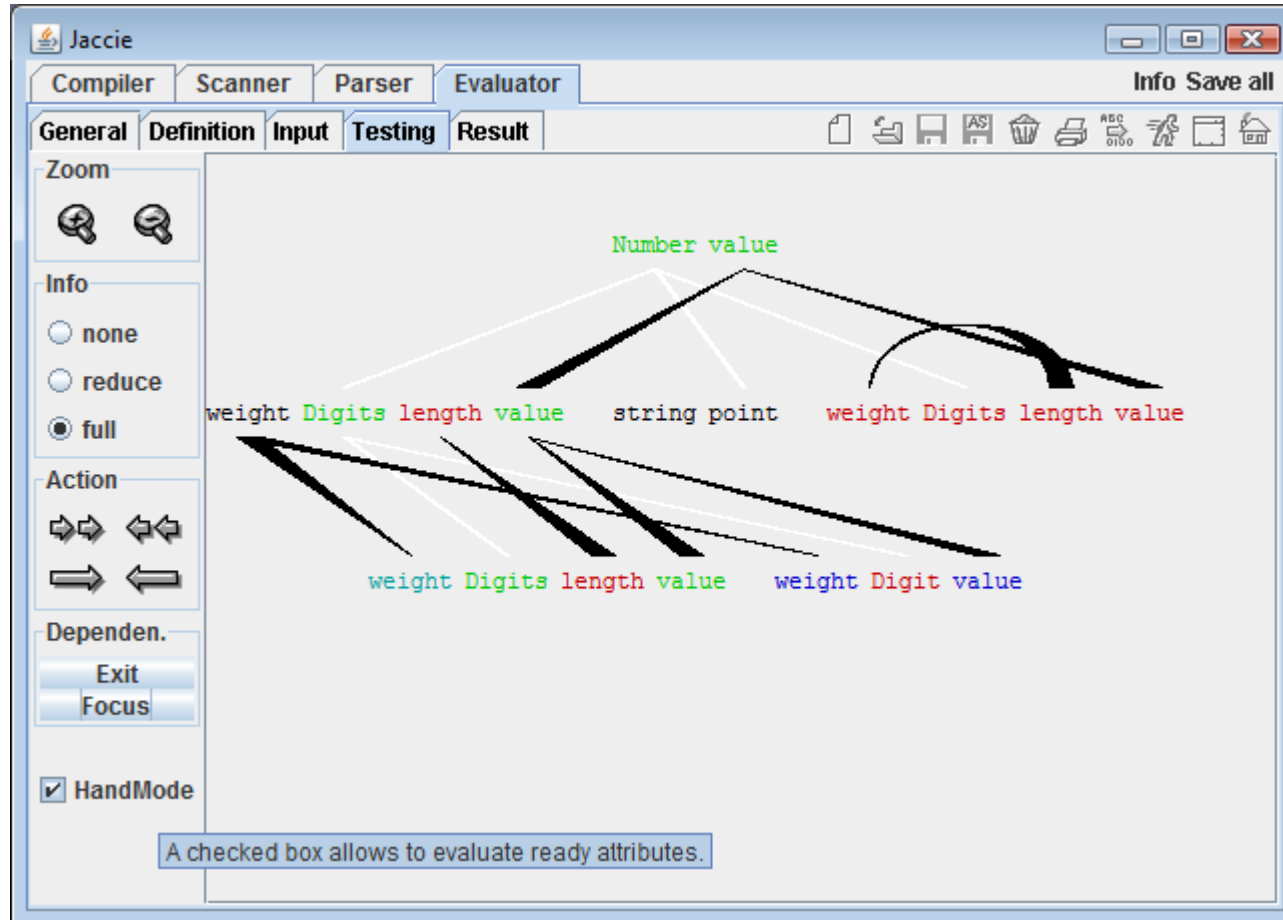
We switch on attribute dependencies to obtain the situation shown on the next page.

Except for the root and the leaves, all nodes of the syntax tree are part of exactly two production rule contexts. The focused node belongs to both contexts. In the screenshot below, the focused view (consisting of these two contexts) is overlapped by all direct attribute dependencies from both contexts.



This view is *traversable*: If you click on any node shown, this node will become the new focused node. Its environment adapts accordingly. That way the whole syntax tree can be traversed, always proceeding from a node to one of its neighbours (i.e., parent, brother, or son). We click on the parent of the focused node to obtain the situation shown on the next page.

Here, we have arrived at the left hand son **Digits** of the root **Number** of the syntax tree.



The controls in the *Dependen.* panel allow you

- to return to the full tree view (*Exit*)
- or to 'normal' focus view (*Focus*).

In the bottom part you see one of the *tool tips* **Jaccie** uses to explain itself to users. As usual, a tool tip appears if the mouse cursor (not visible) hovers for some time over some control element.

Splitting evaluators into passes

Even for small examples single production rule contexts may contain quite a number of attribute instances and attribute evaluation rules. E.g. in our running example, the binary numbers, for the production rule

`Digits -> Digits Digit`

there are eight attribute instances and four attribute evaluation rules.

For real world grammars the sheer bulk of attribute instances and attribute evaluation rules involved will require suitable modularization in order to avoid errors.

An obvious approach is to split an attribute grammar into smaller modules that can be dealt with and tested separately. Jaccie supports one such modularization mechanism where the whole evaluation process is split into a number of so-called 'passes' which can be executed sequentially and will together yield the desired overall result.

Each **pass** P corresponds to a subset of the set of all attributes. When splitting an attribute grammar into a sequence $P_i, 1 \leq i \leq n$, of disjoint passes, there is one *modularization rule* to be obeyed: Attribute evaluation rules in pass P_i may depend only on attribute values that have been (or are) computed in passes $P_j, 1 \leq j \leq i$.

For our running example we find: All `length` attributes can be computed without using any other attributes. The `weight` attributes depend on `length` attributes and other `weight` attributes only. Finally, for computing `value` attributes we need all kinds of attributes. This suggests the following passes

$$P_1 = \{\text{length}\}$$

$$P_2 = \{\text{weight}\}$$

$$P_3 = \{\text{value}\}$$

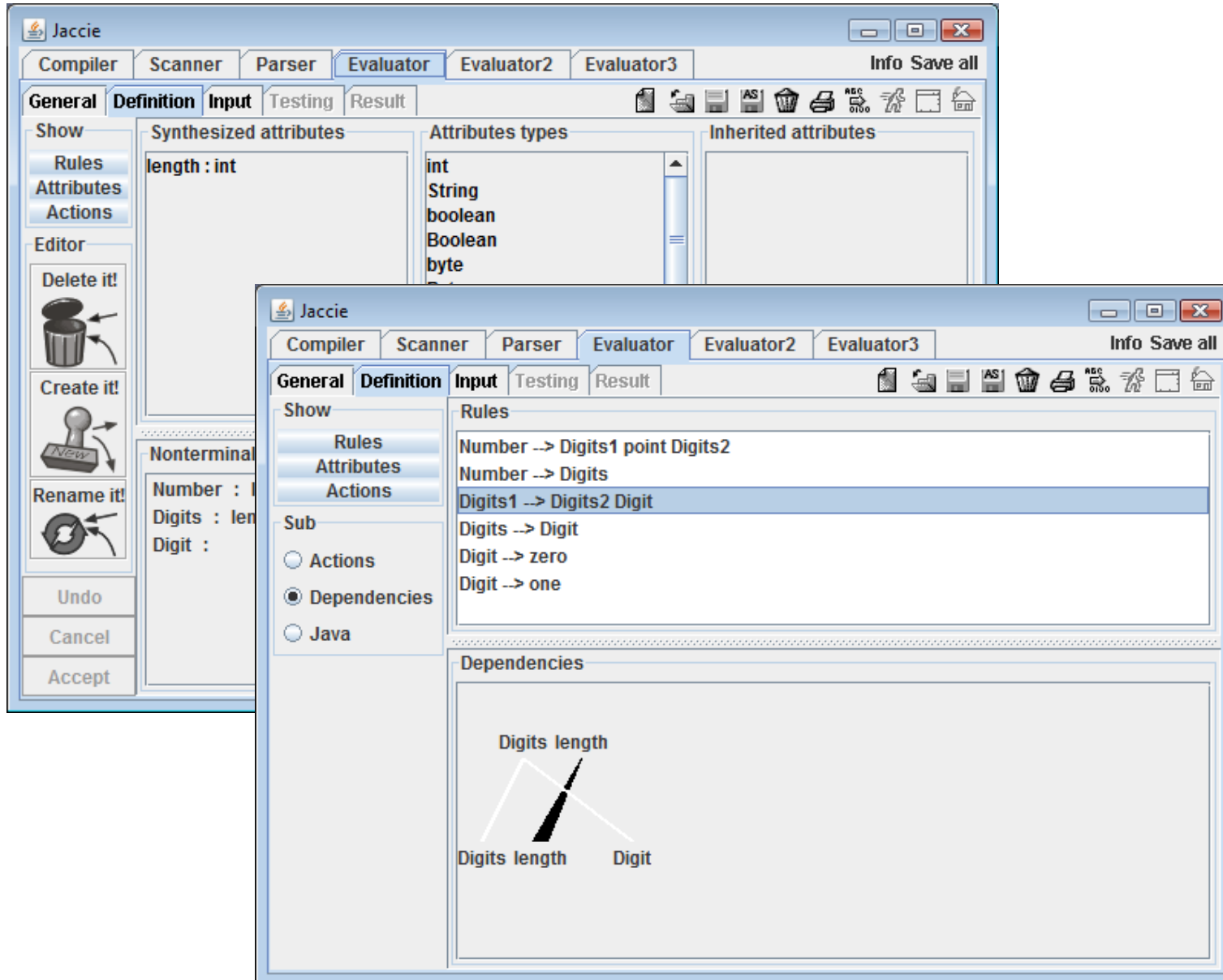
which we are going to implement subsequently. By the way: The modularization

$$P_1 = \{\text{length}, \text{weight}\}$$

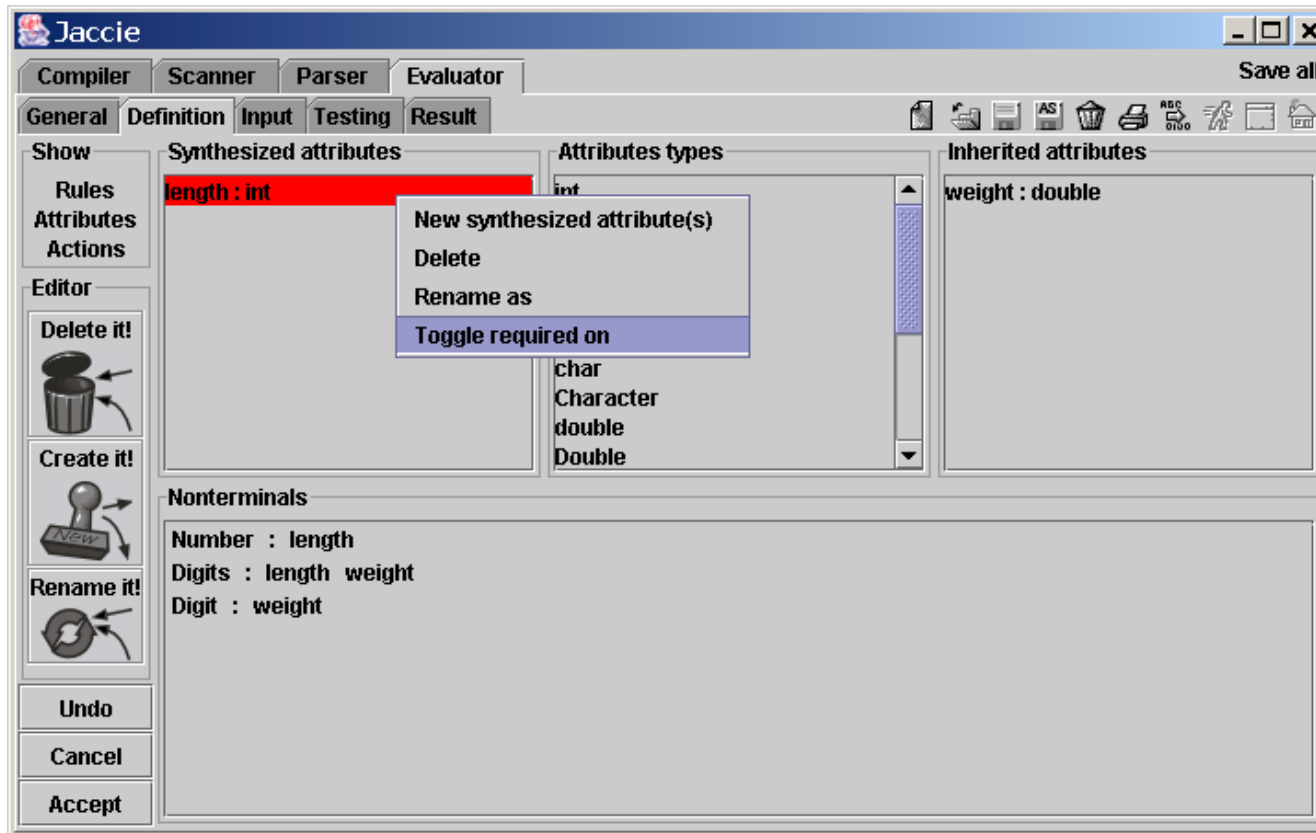
$$P_2 = \{\text{value}\}$$

would have been admissible as well – as are all modularizations obtained from valid modularizations by merging adjacent passes into one. For demonstrating the technique, we use the highest degree of modularization possible.

In the **first pass** $P_1 = \{\text{length}\}$ there are only two attribute evaluation rules, one of which is represented by the black upward arrow between length attributes (within context `Digits -> Digits Digit`) in the screenshot below:

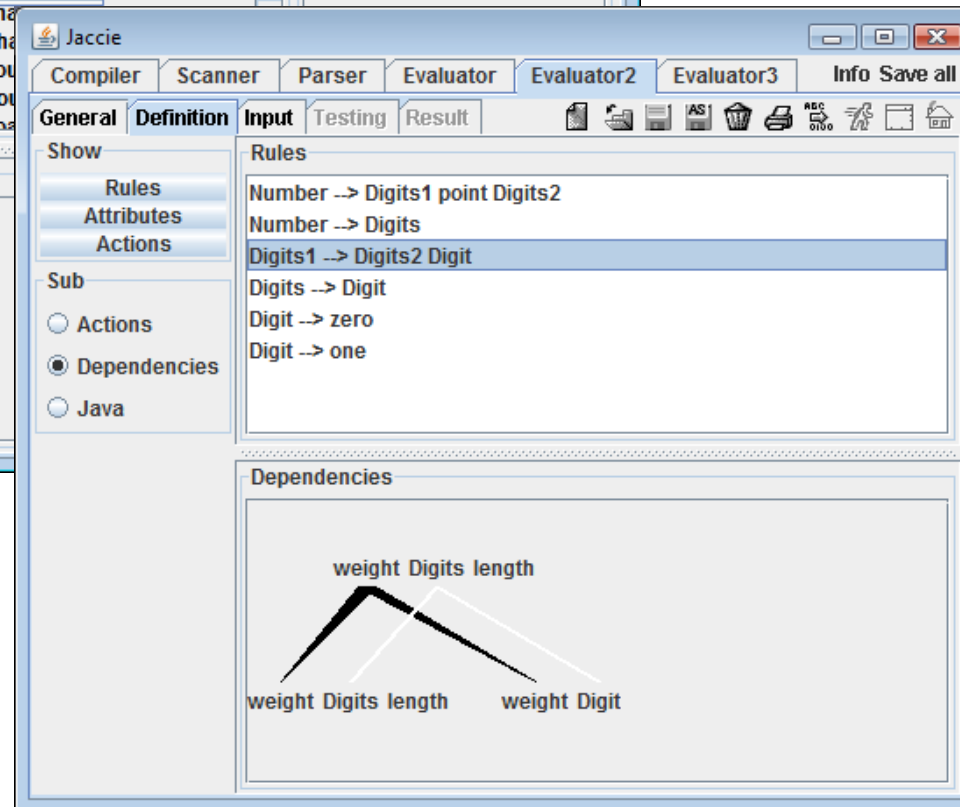
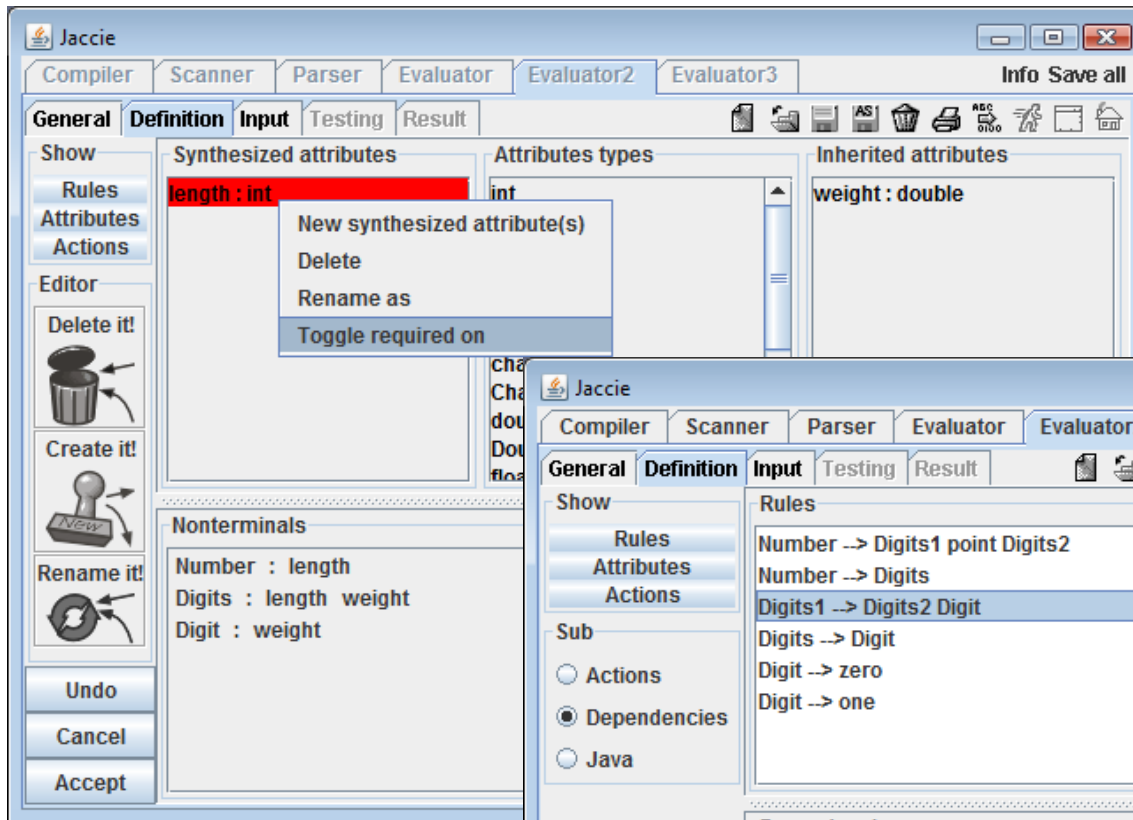


In the **second pass** the inherited `weight` attributes are computed. Hence, in a second evaluator definition, we assign the new `weight` attribute to the nonterminals `Digits` and `Digit`:



The more interesting aspect is how attribute `length` is dealt with in this pass: As in the first pass, we define this attribute and assign it to nonterminals `Digits` and `Number`, but this time provide no attribute evaluation rules.

Instead, we declare `length` attributes to have been *computed previously* and at the same time `required` for this pass. This is achieved by toggling on the `required` mark in the context menu (see screenshot). Thus, `required` attributes are attributes whose values are available in the syntax tree (i.e., have been computed in previous passes). As you can observe on the next page, `required` attributes are shown in gray, other attributes in black.

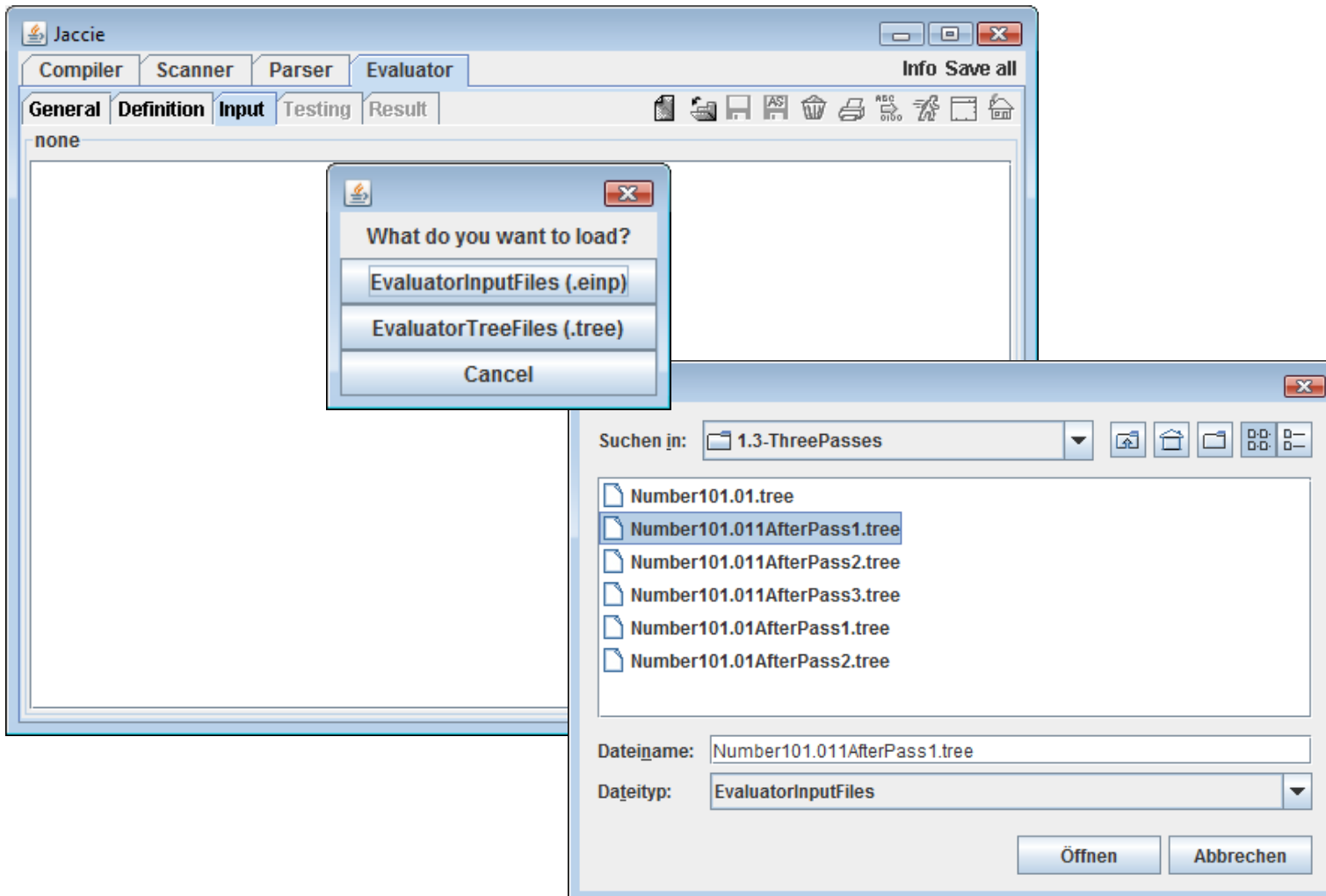


The **weight** attributes are evaluated in the context **Digits -> Digits Digit** (see above) and in other contexts.

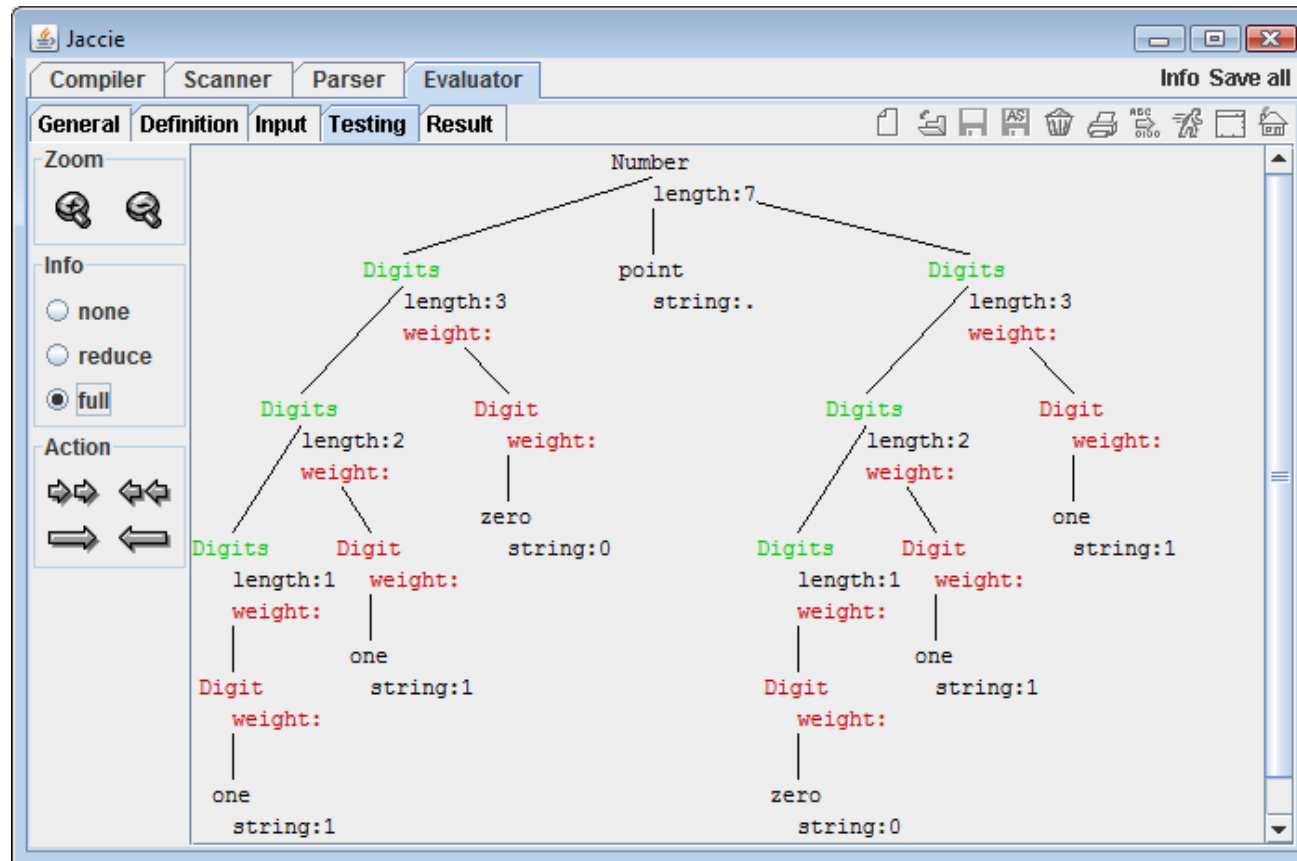
The **third pass** with evaluation rules for attribute **value** will be discussed later.

When starting the pass 2 evaluator as usual, subarea *Evaluator-Input* contains the parsing result, i.e. the syntax tree. For pass 2, however, the **length** attribute values from pass 1 are required. We can make them available by following a somewhat complicated procedure:

- At the end of pass 1 in subarea *Evaluator-Result* save the (partially attributed) syntax tree in a .tree file.
- In subarea *Evaluator-General* (or *Evaluator-Definition*) load the definition of pass 2.
- Before starting pass 2 load the saved .tree file in subarea *Evaluator-Input* as indicated below.



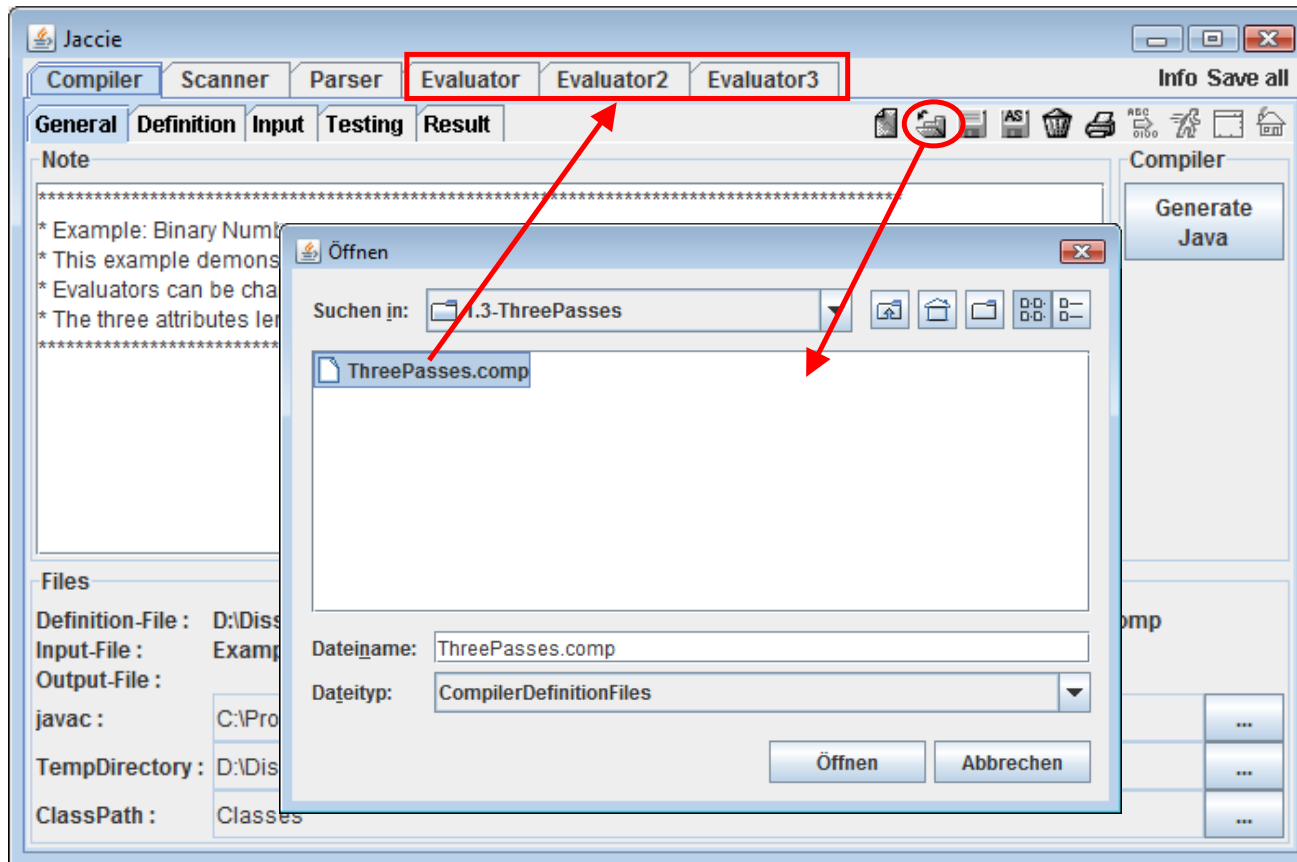
Now in the debugging subarea the **length** attributes from pass 1 are shown in black, i.e., their values are available. At the same time all the **weight** attributes are shown in red, indicating that they still have to be computed:



At the end of pass 2 essentially the same procedure is repeated:

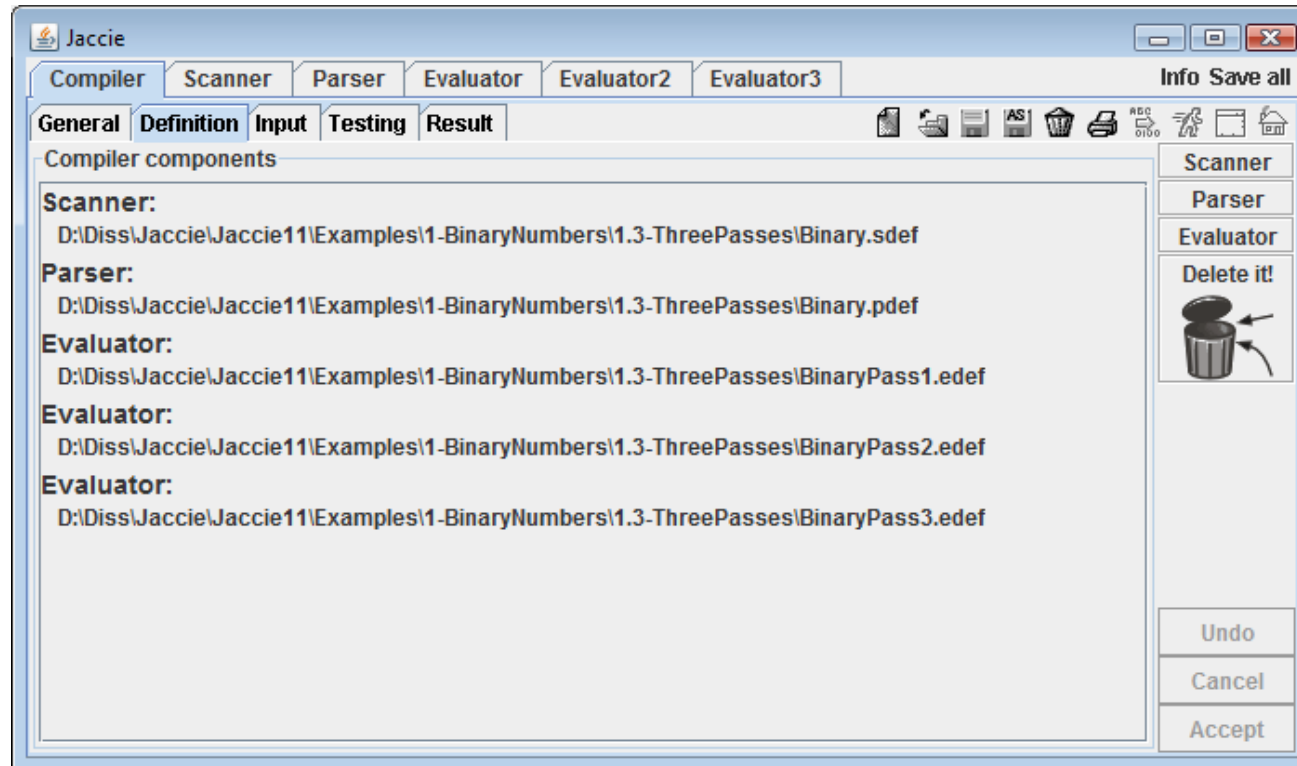
- in subarea *Evaluator-Result* save the result (syntax tree with evaluated **length** and **weight** attribute values) in a .tree file;
- load the definition of pass 3.
- in subarea *Evaluator-Input* load the .tree file saved in pass 2 and execute pass 3.

Obviously, this approach is rather awkward. In its *Compiler* area Jaccie therefore offers a more comfortable solution: You can define a **compiler** to be composed of a scanner, a parser, and a **sequence of evaluators**. In order to demonstrate this feature, we move to subarea *Compiler-General* and there load a 3 pass compiler for the binary numbers examples from the file `Binary3Passes.comp`:



After (!) loading this file *three* evaluator tabs (highlighted in red) will appear, each corresponding to one pass.

On the next page we learn how to create a multipass compiler definitions like this one. In subarea *Compiler-Definition* a **compiler editor** is available which allows you to compose a compiler from existing components via *drag-and-drop*:

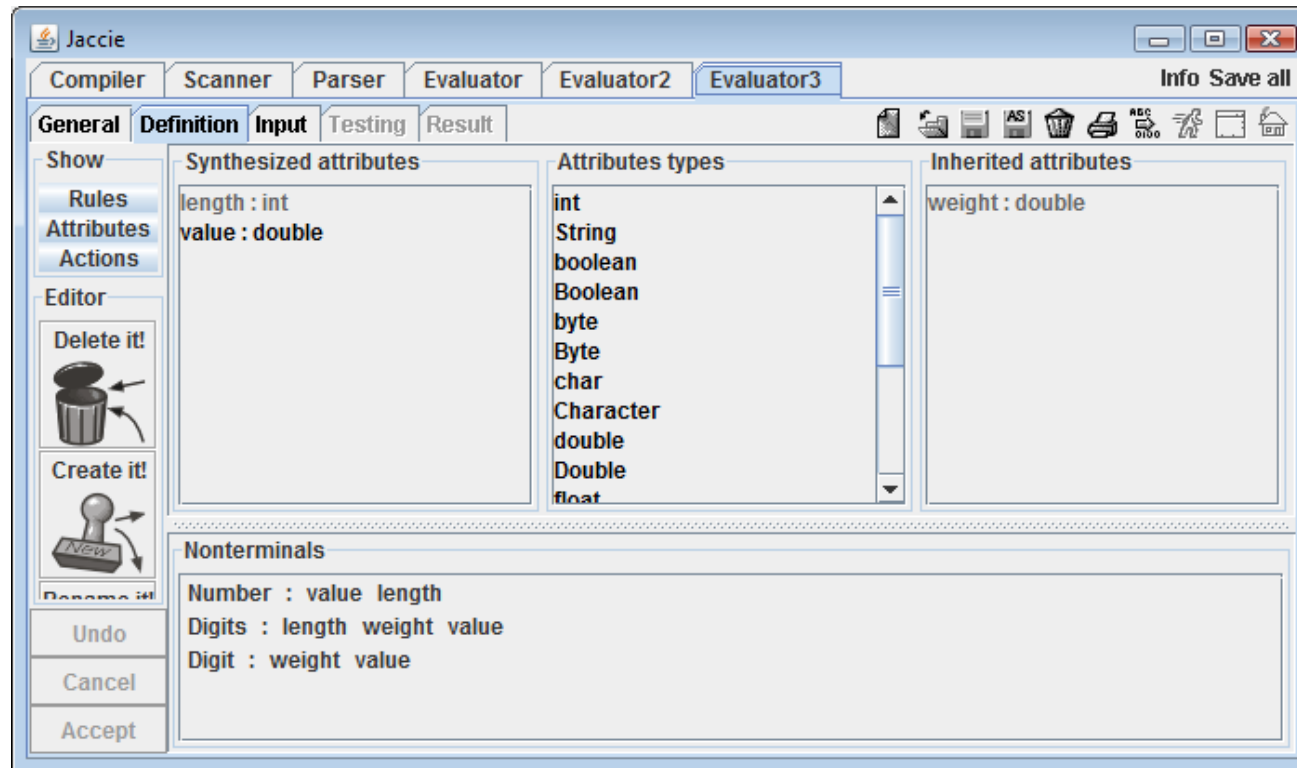


Components are created by pressing the left hand mouse button on the corresponding control (i.e., button labelled *Scanner*, *Parser*, or *Evaluator*), keeping it pressed while dragging the new component to the main panel, where at last the mouse button is released. When creating, say, a scanner component, a file selection dialog will appear showing you available scanners to choose from. In exactly the same way, other kinds of components are created.

Using this style of direct manipulation ('grabbing' with the left hand mouse button which you keep pressed while moving an object) you can *reorder the sequence of components* or *delete components* that are not needed any more by disposing of them in the trash bin.

All components of a compiler are *optional*: There is at most one scanner, at most one parser, and any number of evaluators (occurring, if present, in this order); between a scanner and an evaluator there must be a parser. These rules are consequences of a **general requirement**: *output from any component must be valid input for the next*.



Now, each pass is available for inspection, modification, and use in its own area. We take a look at the attribute assignment in pass 3:

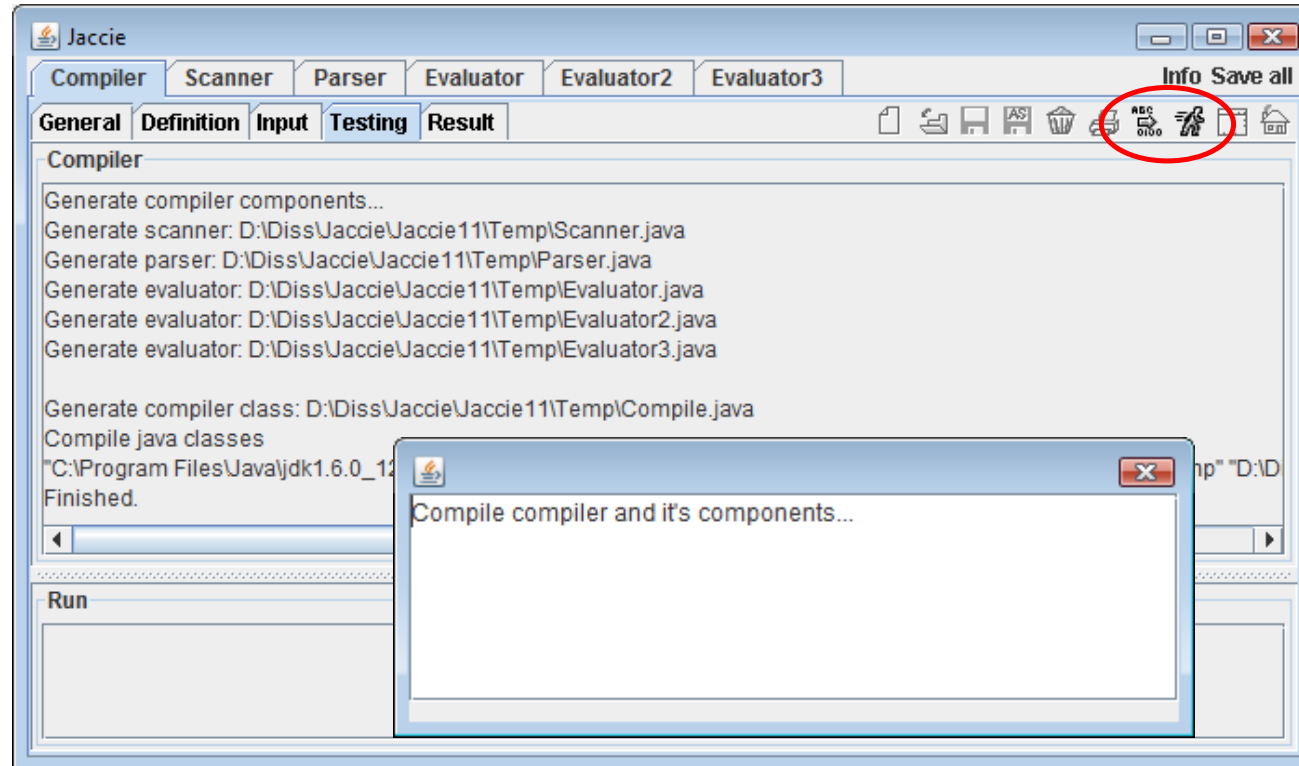


The attribute to be evaluated in this pass, **value**, appears in black in the *Synthesized attributes* panel. For this attribute only, evaluation rules must be defined in the action mode of the attribute editor.

The other attributes (**length** and **weight**) must be available from previous passes. Accordingly, they have been marked as **required** in this pass and are written in gray in the top left and right hand panels.

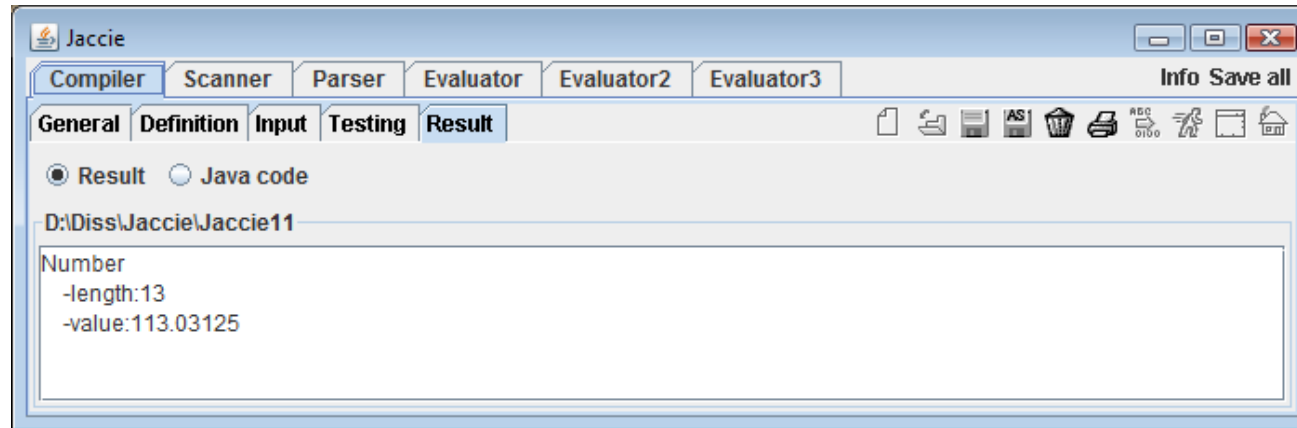
In the bottom panel, the assignments to the grammar's nonterminals are shown for *all* attributes, i.e., not only for the attributes evaluated in this pass, but also for those attributes they depend on. (Jaccie needs this information when extracting the required attribute values from partially evaluated trees.)

Jaccie offers visual debuggers for all kind of components. Once all components have been tested you can **integrate** them in one compiler as described above. In the *Compiler-Testing* subarea the controls for compiling () and executing () a complete compiler are active (i.e., available for selection).



During compilation **Jaccie** displays messages describing its activities in the top (*Compiler*) panel. Output from the Java compiler that is used to compile the generated components is shown in a separate dialog window.

As soon as all components have been compiled successfully, the complete compiler can be executed, i.e., applied to the input to its first component (typically, this will be a scanner, as above). The result is displayed in subarea *Compiler-Result*. Intermediate output from the generated compiler is shown in the bottom (*Run*) panel.



Again, the decimal value of the binary number constant `110.011` has been computed to be `6.375`.

The completely evaluated attribute tree can be inspected in subarea *Evaluator3-Result* – more generally, in the *Result* subarea belonging to the last evaluator component.

So far, we have not explained what the *Source* (*Java code*, respectively) radio buttons (as in the screenshot above) and the corresponding *Generate Java* buttons (available in every *General* subarea) will do exactly. In short: These controls are used for generating the Java source code for your own compilers (see section [Building Compilers](#)).

6. Getting Started

The current **Jaccie** version (**Jaccie04.zip**) is available for download from the Internet. This distribution archive contains four directories and one Jar file:

```
Classes      Config      Examples      Temp
Jaccie04.jar
```

The **Config** directory contains the file **Compiler.cfg** that was described at the end of section **Tool architecture**. In the **Classes** directory (and in its subdirectories) you will find all classes that are used by compiler components generated by **Jaccie**. The subdirectories of **Examples** contain introductory examples like „Arith“. More substantial example applications are included in the **Projects** subdirectory. The **Temp** directory initially is empty. At run time it will be used by **Jaccie** when debugging components.

Installing Jaccie

- The **Jaccie** system is written in Java. Also, it generates (and compiles at run time) compiler components in Java source code. Therefore, **Jaccie** requires a Java „Software Development Kit“ (SDK) to be installed on your computer. The current SDK is available from the SUN web site (<http://java.sun.com/j2se/index.jsp>).
- Create a **Jaccie** directory anywhere on your hard disk, e.g., **C:\Jaccie**.
- Unpack the contents of the **Jaccie** distribution archive **Jaccie04.zip** to the **Jaccie** directory **C:\Jaccie**.

Now in most cases a double click on **C:\Jaccie\Jaccie04.jar** should suffice to start **Jaccie**.

If this does not work (as might be experienced on old Windows versions), create (using some ASCII text editor) in directory **C:\Jaccie** a batch file **Start.bat** containing one line like (you will probably have to adapt the path leading to **javaw.exe**):

```
C:\j2sdk1.4.0\bin\javaw.exe -jar Jaccie.jar
```

Now **Jaccie** can be started by executing **Start.bat**. For convenience, you might want to create a desktop icon for **C:\Jaccie\Jaccie04.jar** (or **C:\Jaccie\Start.bat**, respectively).

General Settings

Immediately after starting *Jaccie* for the first time you have to provide the general settings. Essentially, this is path information *Jaccie* needs in order to access required resources like, e.g., a standard Java compiler. All these settings must be supplied in subarea *Compiler-General*:

javac: The path leading to the directory where the SDK Java compiler is installed on your system. Without this information, *Jaccie* cannot compile user-supplied classes or generated compiler components (which also may contain user-supplied Java code snippets like scanner actions or attribute evaluation rules).

TempDirectory: The path leading to the working directory where *Jaccie* stores generated Java source code and the class files the Java compiler produces .

ClassPath: The path leading to the directory where precompiled *Jaccie* components can be found. Examples are fixed parser components and attribute evaluation strategies. These together with the generated components make up the compilers produced by *Jaccie*.

When starting *Jaccie* for the first time, the latter two settings are initialized with values that will probably do: path information pointing to the *Jaccie Temp* and *Classes* directories. Therefore, the only setting you have to provide manually is the path leading to the directory where the SDK Java compiler is installed on your system. For changing the value of this setting (or other settings), push the dotted button adjacent to the setting's text field. Then a standard Dialog window will pop up which allows you to browse for file *javac.exe* in the SDK's *bin* subdirectory.

The settings have to be provided only once. They are stored in the *Jaccie* configuration file and will be loaded automatically in all subsequent starts of the *Jaccie* system.

Testing Jaccie

In order to check that *Jaccie* is installed properly (and also to get acquainted with it) let us use one of the simple *Arithmetic* examples: the one in subdirectory *Examples\2-Arithmetic\2.4-Infix2MathML* which transforms 'normal' infix arithmetic expressions into their MathML equivalents.

MathML is an XML dialect for describing mathematical formulae.

Jaccie- Tour

Tool architecture

Token Recognition

Grammars and Parsing

Attribute Evaluation

Getting Started

Building Compilers

Title Page

◀◀

▶▶

◀

▶

Page 85 of 99

Back

Full Screen on/off

Close

Quit

E.g., for the arithmetic expression

$$\frac{123 * x}{\alpha + 565767}$$


the MathML description would be

```
<apply>
  <divide/>
  <apply>
    <times/>
    <cn>123</cn>
    <ci>x</ci>
  </apply>
  <apply>
    <plus/>
    <ci>alpha</ci>
    <cn>565767</cn>
  </apply>
</apply>
```

Obviously, **number constants** are enclosed in tag pairs `<cn>...</cn>`, **variable names** in tag pairs `<ci>...</ci>`. Each subexpression consisting of a binary operator and its two operands is represented in the form:

```
<apply>
  <operator/>
  <operand1>
  <operand2>
</apply>
```

Levels of nesting are represented by indentation.

In order to load the **scanner** definition, go to subarea *Scanner-Definition* and click at the  icon. Now, a FileChooser Dialog appears which allows you to navigate to the `Examples\2-Arithmetic\2.4-Infix2MathML` subdirectory and select the (only) scanner definition file shown there. This will load the scanner described at the beginning of the **Jaccie-Tour**.

Jaccie-Tour

Tool architecture

Token Recognition

Grammars and Parsing

Attribute Evaluation

Getting Started

Building Compilers

Title Page

◀◀

▶▶

◀

▶



Page 86 of 99

Back

Full Screen on/off

Close

Quit

Also, in subarea *Scanner-Input* click the  icon to create a new scanner input. Type `123 * x / (alpha + 565767)` into the text pane and click the  icon in order to run the scanner on this input. The result will be shown in the *Scanner-Result* subarea:

```
number "123"
multOp "*"
name "x"
multOp "/"
openingBracket "("
name "alpha"
addOp "+"
number "565767"
closingBracket ")"
```

Analogously, load and run the **parser** and load and run the **evaluator**. Finally, in the *Evaluator-Result* subarea you should see the MathML code shown above (with some minor modifications: an additional preamble and a slightly different indentation). Now, on your own test [Jaccie](#)'s debugging features (as described in the preceding sections).

In the same way explore some of the other examples until you feel confident enough with [Jaccie](#) to undertake the next step, i.e., *create a new example application*. We suggest you to try and build a compiler that **transforms prefix arithmetic expressions into MathML**. Essentially, this is a combination of examples [2.3](#) and [2.4](#). So in case there are any problems, you can find all the required know-how there. You probably will also have to consult the preceding sections of this manual a few times - but this is the whole point of this exercise!

7. Building Compilers

The reason for developing a compiler is to either use it as a standalone transformation program or to integrate it as an auxiliary module into another program. Here, we only show how to create a standalone compiler. Auxiliary compiler modules are produced in essentially the same way.

Let us develop a compiler for one of the simple *Arithmetic* examples: the one which transforms a given arithmetic expressions into its (textual) Kantorovic tree representation (to be found in [Examples\2-Arithmetic\2.6-Kantorovic](#)). We expect this compiler to transform an input expression like, say,

```
alpha + 42 * centauri
```

into its output Kantorovic tree representation

```
+
|\
| *
| |\
| | centauri
| |
| |
| 42
|
alpha
```

Using [Jaccie's](#) *Generate Java* buttons we can produce the compiler's four main component classes:

[Scanner.java](#) contains the Java representation of the [Atrith2Kantorovic.sdef](#) scanner definition; together with the fixed parts derived from its superclass [ScannerScanner.java](#) it forms the executable scanner.

[Parser.java](#) contains the Java representation of the [Atrith2Kantorovic.pdef](#) parser definition; together with the fixed parts derived from its superclass [SICParser.java](#) it forms the executable parser.

[Evaluator.java](#) contains the Java representation of the [Atrith2Kantorovic.edef](#) scanner definition; together with the fixed parts derived from its superclass [EvaluatorJourdan.java](#) it forms the executable evaluator.

[Compile.java](#) contains the compiler's frame that calls the other components (the more obvious name, [Compiler.java](#), could not be used since there already exists a class with that).

[Jaccie-Tour](#)

[Tool architecture](#)

[Token Recognition](#)

[Grammars and Parsing](#)

[Attribute Evaluation](#)

[Getting Started](#)

[Building Compilers](#)

[Title Page](#)

◀◀

▶▶

◀

▶

Page 88 of 99

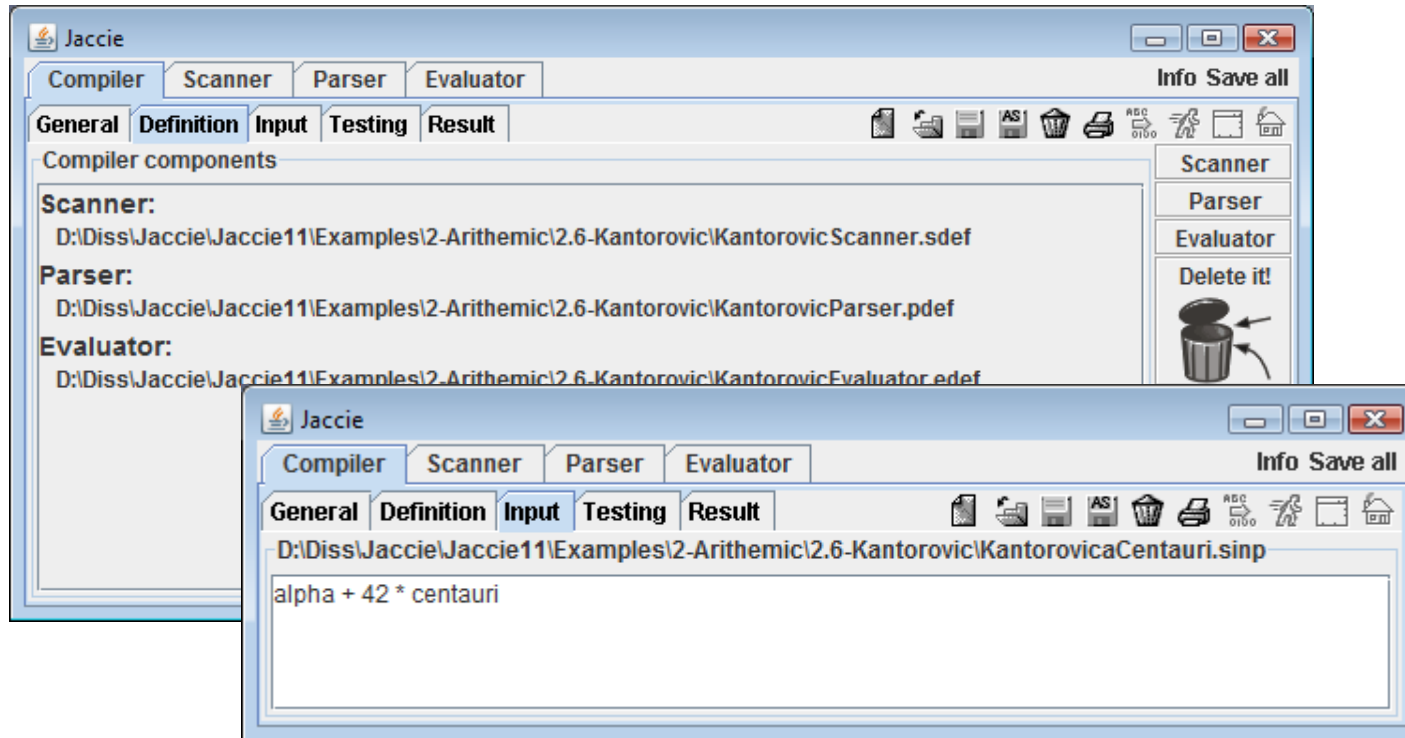
[Back](#)

[Full Screen on/off](#)



[Close](#)

[Quit](#)

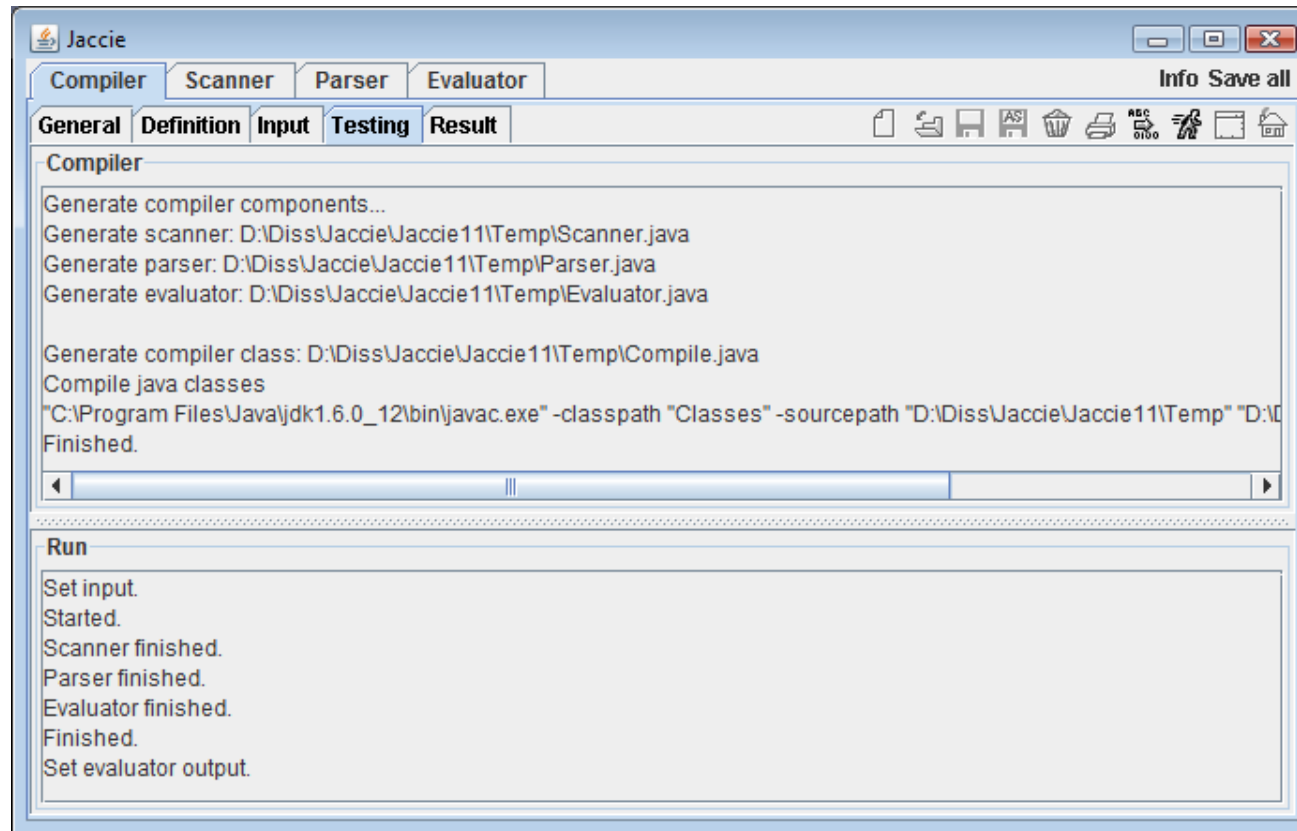
Code snippets from these components will be shown and explained at the end of this section. An alternative way of generating (and compiling) the Java program components is available via the [Jaccie Compiler](#) area. In the *Compiler-Definition* subarea we define the constituents of the compiler to be built and in the *Compiler-Input* we choose some arithmetic expression to be transformed:



Now, in the *Compiler-Testing* subarea we can

- generate the Java source code of the four compiler components above and also compile it by clicking on the compile () icon;
- run the resulting compiler on the chosen input by clicking on the run () icon.

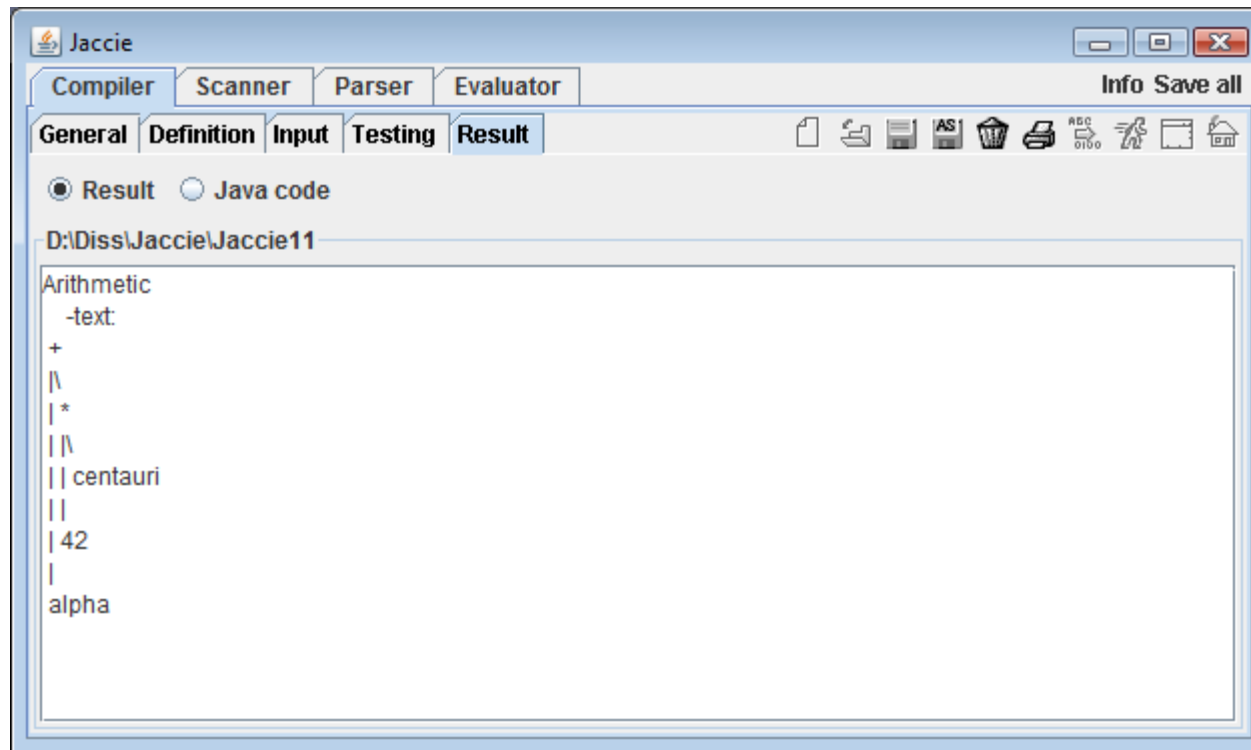
This is shown in the screenshot on the next page.



Notice the **options** the Java compiler is called with:

- `-classpath "C:\Jaccie\Classes"` points to the directory where (in the subdirectory `Data`) the fixed parts of the generated compiler are stored in the package `Data` as precompiled Java `.class` files. The very same class path is used by `Jaccie` when the generated and compiled target compiler is executed.
- `-sourcepath "C:\Jaccie\Temp"` points to the directory where the source code of the generated compiler components is stored by `Jaccie`. In this directory we will also find the `.class` files (result of `Jaccie` compile action) of the four generated compiler components.

Finally, the expected translation result is shown in the *Compiler-Result* subarea (screenshot on the next page).



Still, the generated compiler is executed within the [Jaccie](#) environment. This is not what we planned to do: We wanted to create a **standalone compiler**, i.e., a Java program that can be executed without [Jaccie](#).

What we need to finish our task is an application program with a main method which prompts the user for some input string, applies the generated compiler to this string, and presents the result to the user. Such an application is the [Arith2KantorovicWindow](#) class shown on the next page. The file [Arith2KantorovicWindow.java](#) should be placed in the directory where the files [Scanner.class](#), [Parser.class](#), [Evaluator.class](#), and [Compile.class](#) reside.

Given the directory structure introduced above we can compile this class using the command line

```
javac -classpath ".;c:\Jaccie\Classes" Arith2Kantorovic.java
```

Note that the classpath includes both the current directory and the [\Jaccie\Classes](#) directory where the [Data](#) package resides.

```

import javax.swing.*;
import Data.*;

public class Arith2KantorovicWindow {

    public static void main(String[] args) {

        String inputValue = JOptionPane.showInputDialog("Please input an arithmetic expression:");

        Compile myCompiler = new Compile();
        myCompiler.setInput(inputValue);
        myCompiler.run();

        EvaluatorTree result = (EvaluatorTree) myCompiler.getResult();

        String kantorovic = (String) result.getValueOf("text");

        JTextArea treeRep = new JTextArea();
        treeRep.setText(kantorovic);

        JOptionPane.showMessageDialog
            (null, treeRep, "Kantorovic tree representation", JOptionPane.INFORMATION_MESSAGE);
        System.exit(0);
    }
}

```

The Java Swing package is imported because we use `JOptionPane Dialog` windows for user interaction. The `Compile` object `myCompiler` is run on the user input. From the resulting attribute tree, the root `text` attribute – a String – is extracted using the method calls

```

EvaluatorTree result = (EvaluatorTree) myCompiler.getResult();
String kantorovic = (String) result.getValueOf("text");

```

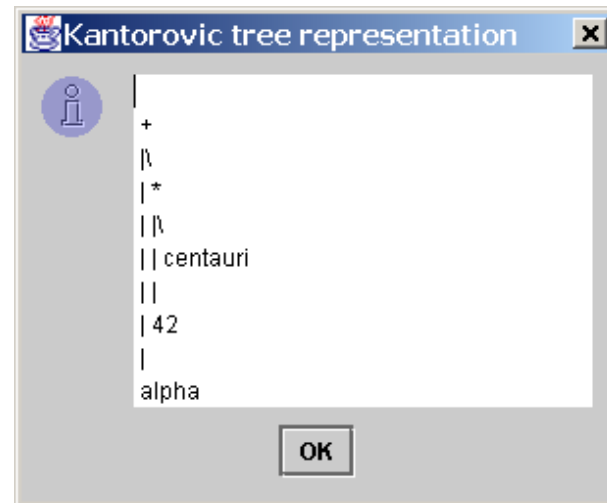
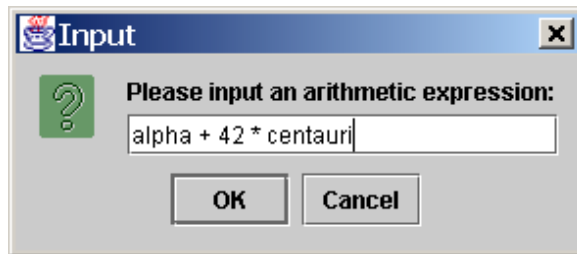
Class `EvaluatorTree` with the extraction method `getValueOf("<attributeName>")` is defined in the `Data` package which, therefore, must be imported.

Finally, we can run our application using the command line

```
java -classpath ".;c:\Jaccie\Classes" Arith2Kantorovic.java
```

(Note that the classpath again must include both the current directory and the `\Jaccie\Classes` directory where the `Data` package resides).

Below, two screenshots from a run of our standalone compiler are shown.



We conclude this section by taking a look at the Java source code of the generated compiler components. This helps to understand

- the components and their interaction
- how more than one compiler can be integrated into an application program
- how to run a compiler in a thread of its own

The **scanner component** essentially builds one finite automaton for each regular token definition (recall that most of the scanner's functionality is inherited from superclass `ScannerScanner.java`):

```

    automat = new Automat("addOp");           // build automaton for addOp token:
    state = new State(0,"(.+|-).",false);    // start state 0, text with
    state.add(new Transition("43,45",1));    // transition with ASCII codes for +,-
    automat.add(state);                      // add state to automaton
    state = new State(1,"(+|-).",true);      // new state 1 is final state
    automat.add(state);                      // add state to automaton
    add(automat);                            // add automaton to scanner

```

Likewise, the **parser component** builds up data representations of the grammar rules (again, the parser's functionality is inherited from superclass `SICParser.java`):

```

    // e.g., for rule Expression -> Expression addOp Term

    String[] rule2 = {"Expression","Expression","addOp","Term"};

```

and of the LR parser's states (combined *action* and *goto table*):

```

    // e.g., for the states 1 and 2:

    actionTable(1,"Term",SHIFT,12);
    actionTable(1,"Factor",SHIFT,10);
    actionTable(1,"name",SHIFT,2);
    actionTable(1,"openingBracket",SHIFT,4);
    actionTable(1,"Arithmetic",SHIFT,13);
    actionTable(1,"number",SHIFT,3);
    actionTable(1,"Expression",SHIFT,14);
    actionTable(2,"#",REDUCE,8);

    actionTable(2,"addOp",REDUCE,8);
    actionTable(2,"multOp",REDUCE,8);
    actionTable(2,"closingBracket",REDUCE,8);

```

The **evaluator component** essentially contains attribute definitions, method `checkIfPossible("<attributeName>")` for checking whether an attribute instance can be evaluated, and method `calculate("<attributeName>")` for actually evaluating attributes (recall that from its superclass `EvaluatorJourdan.java` most of the evaluator's functionality is inherited). The following code snippet shows how an attribute evaluation rule provided by the developer is embedded into the evaluation method:

```
// current context:
if (activeRule.equals("Expression1 Expression2 addOp Term")) {

    // attribute to be evaluated:
    if (attr.equals("Expression1.text")) {

        // extract required attributes from attribute tree:
        String Expression1_text = null;
        String Expression1_prefix = (String) get("Expression1.prefix");
        String Expression2_text = (String) get("Expression2.text");
        String addOp_string = (String) get("addOp.string");
        String Term_text = (String) get("Term.text");

        // compute attribute (rule as provided by developer):
        Expression1_text =
            Expression1_prefix + addOp_string + "\n" +
            Expression1_prefix + "|" + "\\" + "\n" +
            Term_text + "\n" +
            Expression1_prefix + "|" + "\n" +
            Expression2_text;

        // store this attribute in the attribute tree:
        set("Expression1.text",Expression1_text);
    }
    ... // other attribute evaluation rules for this context
}
... // other contexts
```

Below, the full text of class `Compile.java` is shown. This is where the compiler components are assembled into one sequence of actions.


```

import Data.*;
import java.util.*;

public class Compile extends Observable implements CompilerAction, Runnable {

    private Observer observer = null;

    public void setObserver(Observer o) { observer = o; }

    private void update(String message) {
        if (observer != null) observer.update(this,message);
    }

    private String input = "";
    private EvaluatorTree inputTree = null;
    private String einput = "";

    public void setInput(Object o) {
        inputTree = null;
        input = (String)o;
    }

    public Object getResult() { return inputTree; }

    public void run() {
        // shown on next page
    }
}

```

Notice that class `Compile.java` is both an `Observable` and `Runnable`. Thus, in the application program it can be started in a thread of its own. This will be demonstrated on the next page but one.

`Compile` is started with an input of type `String` and returns a result of type `EvaluatorTree`. Processing takes place in the method `run` which is shown on the next page.

Jaccie- Tour

Tool architecture

Token Recognition

Grammars and Parsing

Attribute Evaluation

Getting Started

Building Compilers

Title Page

◀◀

▶▶

◀

▶

Page 96 of 99

Back

Full Screen on/off

Close

Quit

```

public void run() {
    // scan ...
    Scanner scanner = new Scanner();
    scanner.getScannerInput().setInput(input);
    try { scanner.resetScanner(); scanner.scanAll(); }
    catch (Exception e) { update("Error while scanning: "+ e.toString()); }
    update("Scanner finished.");

    // ... parse scanner result (token sequence)...
    SICParserInput pinput = new SICParserInput();
    Vector scannerResult = scanner.getResult();
    for (int i=0; i<scannerResult.size(); i++)
        pinput.addScannerOutput((String) scannerResult.elementAt(i));
    Parser parser = new Parser();
    parser.start(pinput);
    update("Parser finished.");

    // ... and evaluate parsing tree
    einput = parser.getResultAsString();
    Evaluator evaluator = new Evaluator();
    if (inputTree == null) evaluator.setTree(einput);
        else evaluator.setTree(inputTree);
    evaluator.start();
    inputTree = evaluator.getTree();
    update("Evaluator finished.");
    update("Finished.");
}

```

If you want to use *two* compilers in your application, say, compiler1 and compiler2, you have to

- consistently rename the component Java classes into `Scanner1`, `Parser1`, `Evaluator1`, `Compile1` and `Scanner2`, `Parser2`, `Evaluator2`, `Compile2`;
- rename all references to these classes in the above `run` method accordingly.

```

import Data.*;
import javax.swing.*;
import java.util.*;

public class KantorovicObserver implements Observer {

    private Compile myCompiler;

    public KantorovicObserver () {
        String inputValue = JOptionPane.showInputDialog
            ("Please input an arithmetic expression:");
        myCompiler = new Compile();
        myCompiler.setInput(inputValue);
        myCompiler.setObserver(this);
        Thread compilerThread = new Thread(myCompiler);
        compilerThread.start();
    }

    public void update (Observable obs, Object arg) {
        if (arg.equals("Finished.")) {
            String kantorovicTree =
                (String)((EvaluatorTree) myCompiler.getResult()).getValueOf("text");
            JTextArea treeRepresentation = new JTextArea();
            treeRepresentation.setText(kantorovicTree);
            JOptionPane.showMessageDialog (null, treeRepresentation,
                "Kantorovic tree representation", JOptionPane.INFORMATION_MESSAGE);
            System.exit(0);
        }
    }

    public static void main(String[] args) {
        KantorovicObserver kantorovic = new KantorovicObserver();
    }
}

```

Jaccie- Tour

Tool architecture

Token Recognition

Grammars and Parsing

Attribute Evaluation

Getting Started

Building Compilers

Title Page

◀◀

▶▶

◀

▶

Page 98 of 99

Back

Full Screen on/off

Close

Quit

When run, class `KantorovicObserver` behaves like class `Arith2KantorovicWindow`. Most part of the application program code is identical in both classes. The technical organization of both classes, however, differs greatly.

This is due to the following reasons:

- Since the Observer pattern is used, we need to import package `java.util`.
- The `Compile` object `myCompiler` is observed by a `KantorovicObserver` object. Therefore, a *constructor* is required for class `KantorovicObserver`. In the constructor the user is prompted for input. Then the connection between Observable and Observer is established. Finally, the Observable `Compile` object – recall that `Compile` objects are runnable – is started in a Thread of its own.
- As an Observer, class `KantorovicObserver` must implement an `update()` method. If the observed `Compile` object signals that it is finished, the Observer gets the resulting attribute tree, extracts the `text` root attribute and displays this text to the user.
- Method `main()` starts this process by creating a `KantorovicObserver` object.