Smacc: a Compiler-Compiler

John Brant, Jason Lecerf, Thierry Goubier, Stéphane Ducasse

The Pharo Booklet Collection
edited by S. Ducasse
May 1, 2017
master @ 7a20b0d*

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

About this booklet

This booklet describes SmaCC the Smalltalk Compiler-Compiler developed by John Brant.

1.1 Contents

It contains:

- A tutorial originally written by John Brant and Don Roberts (SmaCC[1]) and adapted to Pharo.
- Syntax to declare Syntax trees.
- Details about the directives.
- Scanner and Parser details.
- Support for transformations.
- Idioms: Often we have recurring patterns and it is nice to document them.

SmaCC was ported by Thierry Goubier to Pharo who is maintaining actively the Smacc Pharo port. SmaCC is used in production systems and support for example the automatic conversion from Delphi to C# and other large projects.

1.2 Obtaining SmaCC

If you haven't already done so, you will first need to load Smacc.

[1] http://www.refactory.com/Software/SmaCC

```
Metacello new
  baseline: 'SmaCC';
  repository: 'github://ThierryGoubier/SmaCC';
  load
```

Note that there is another version that appeared recently and that is now part of Moose http://moosetechnology.com the difference is that there are different tools to load the parsers/scanner and in addition there is a special debugger. This debugger may be integrated in the future with the version used in this book.

1.3 Basics

Scanning takes an input stream of characters and converts that into a stream of tokens. The tokens are then passed on to the parsing phase. Parsing converts the stream of tokens provided by the scanner into some object.

CHAPTER 2

A first SmaCC tutorial

This is a walk-through tutorial to demonstrate some features of SmaCC, the Smalltalk Compiler Compiler. This tutorial develop a simple calculator. This tutorial was originally developed by Don Roberts and John Brant and improved by T. Goubier, S. Ducasse and J. Lecerf.

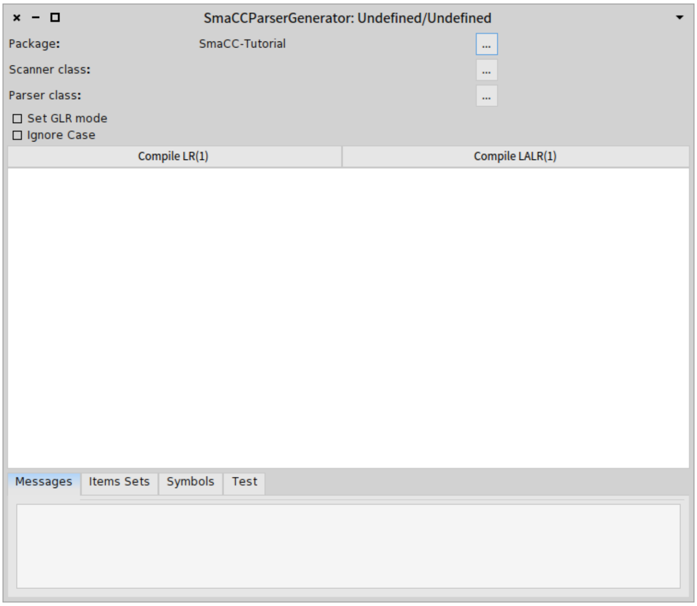
2.1 Opening the tools

Once you have loaded the code, you need to open the SmaCC Parser Generator (Figure 2-1). In Pharo, it is in the Tools submenu of the main menu.

Our first calculator is going to be relatively simple. It is going to take two numbers and add them together. To get started

- Edit the definition the pane below the compile LR(1) buttons.
- Once you are done:
  - Click on the Scanner class and type ExpressionScanner.
  - Click on the Parser class and type ExpressionParser.
- press either Compiler LR(1) or Compiled LALR(1) buttons.

Now you are ready to edit first your scanner and parser. Not that you edit everything in one file (edited using the tools) and once compiled the tools will generate two classes and fill them with adequate information as shown as Figure 2-2.



SmaCCParserGenerator: Undefined/Undefined

Package: SmaCC-Tutorial

Scanner class:

Parser class:

□ Set GLR mode

□ Ignore Case

Figure 2-1 SmaCC GUI Tool: The place to define the scanner and parser.

2.2 First the scanner

To start things off, we have to tell the scanner how to recognize a number. It starts with one or more digits, possibly followed by an decimal point with zero or more digits after it. The scanner definition for this token (called a token specification) is:

```
<number>        :       [0-9]+ (\. [0-9]*) ? ;
```

Let's go over each part:

`<number>` Names the token identified by the token specification. The name inside the <> must be a legal Pharo variable name.

`:` Separates the name of the token from the token's definition.

`[0-9]` Matches any single character in the range `'0'` to `'9'` (a digit). We could also use `\d` or `<isDigit>` as these also match digits.

`+` Matches the previous expression one or more times. In this case, we are matching one or more digits.

`( ... )` Groups subexpressions. In this case we are grouping the decimal point and the numbers following the decimal point.

`\.` Matches the '.' character (. has a special meaning in regular expressions, quotes it).

`*` Matches the previous expression zero or more times.

`?` Matches the previous expression zero or one time (i.e., it is optional).

`;` Terminates a token specification.

Handling whitespaces

We don’t want to have to worry about whitespace in our language, so we need to define what a whitespace is and to ignore it. To do this, enter the following token specification on the next line:

```
<whitespace>    :       \s+;
```

`\s` matches any whitespace character (space, tab, linefeed, etc.). So how do we tell the scanner to ignore it? If you look in the `SmaCCScanner` class (the superclass of all the scanners created by SmaCC), you will find a method named `whitespace`. If a scanner understands a method that has the same name as a token name, that method will get called whenever the scanner matches that kind of token. As you can see, the `SmaCCScanner>>whitespace` method eats whitespace.

```
SmaCCScanner >> whitespace
  "By default, eat the whitespace"

  self resetScanner.
  ^ self scanForToken
```

`SmaCCScanner` also defines a `comment` method that behaves similarly. However, the `SmaCCScanner>>comment` method also stores the interval in the source where the comment occurred in the comments instance variable.

```
SmaCCScanner >> comment
  comments add: (Array with: start + 1 with: matchEnd).
  ^ self whitespace
```

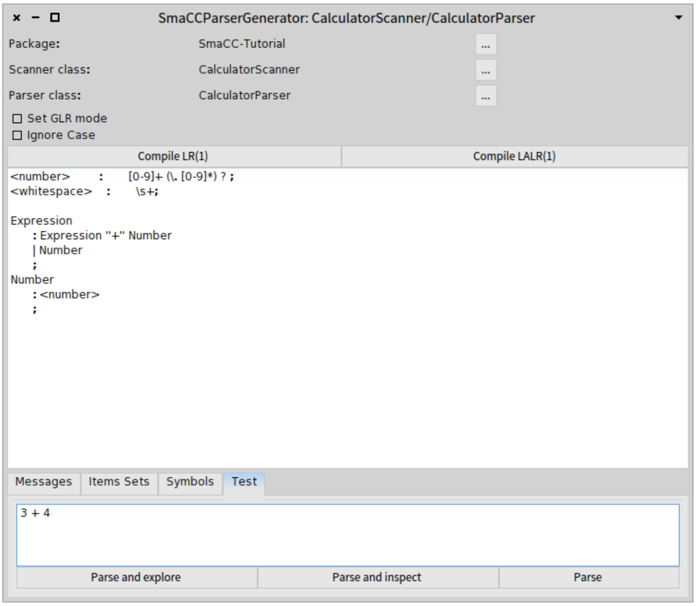
The only other token that will appear in our system would be the `+` token for addition. However, since this is token is always the same, we can enter it directly in our production rules.

To do SD: I do not get the previous sentence.

2.3 Second the calculator grammar

Speaking of our grammar, let’s go ahead and define it. Enter the following specification below your two previous rules in the editor pane as shown in Figure 2-2.

```
Expression
  : Expression "+" Number
```



Package: SmaCC-Tutorial
Scanner class: CalculatorScanner
Parser class: CalculatorParser
□ Set GLR mode
□ Ignore Case

```
<number>       :      [0-9]+ (\. [0-9]*) ? ;
<whitespace>   :      \s+;

Expression
     : Expression "+" Number
     | Number
     ;
Number
     : <number>
     ;
```

```
3 + 4
```

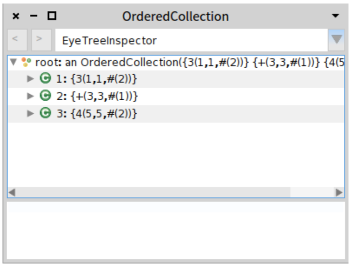
Figure 2-2 First grammar: the Scanner part followed by the Parser part.

```
   | Number
   ;
Number
   : <number>
   ;
```

This basically says that an expression is either a number or an expression added to a number. You should now have something that looks like Figure 2-2.

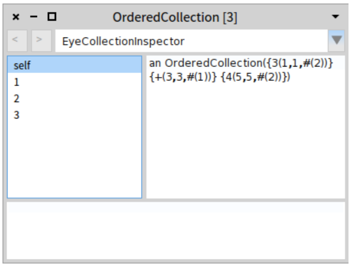
2.4 Compile the Scanner and the Parser

We should be able to compile a parser now. First, we need to specify the scanner and parser classes to create. These are entered from the `...` buttons for scanner class and parser class. Enter `CalculatorScanner` and `CalculatorParser` respectively. Once the class names are entered, we are ready to compile the parser. Press Compile LR(1) or Compile LALR(1). This will create new Pharo classes for the `CalculatorScanner` and `CalculatorParser` and compile several methods in those classes. All methods that SmaCC compiles will go into a ”generated” method protocol. You should not change those methods or add new methods to the ”generated” method protocols since they are replaced or deleted each time you compile.



```
OrderedCollection
EyeTreeInspector
root: an OrderedCollection({3(1,1,#(2))} {+(3,3,#(1))} {4(5
    1: {3(1,1,#(2))}
    2: {+(3,3,#(1))}
    3: {4(5,5,#(2))}
```

Figure 2-3 Explorer on 3 + 4



```
OrderedCollection [3]
EyeCollectionInspector
self    an OrderedCollection({3(1,1,#(2))}
1       {+(3,3,#(1))} {4(5,5,#(2))})
2
3
```

Figure 2-4 Inspector on 3 + 4

Whenever SmaCC creates new classes, they are compiled in the package (or package tag) named on the Package entry box. You may wish to select a different package by pressing '...'.

2.5 Testing our parser

Now we are ready to test our parser. Go to the "test" pane, enter `3 + 4`, and press "Parse"; you will see that the parser correctly parses it. If you press "Parse and explore" or "Parse and inspect" you will see an explorer (Figure 2-3) (respectively an inspector (Figure 2-4)) on an OrderedCollection that contains the parsed tokens. This is because we haven't specified what the parser is supposed to do when it parses. You can also enter incorrect items. For example, try to parse `3 + + 4` or `3 + a`. An error message should appear in the text.

If you are interested in the generated parser, you may wish to look at the output from compiling the parser in the Symbols or Item Sets tab.

- The Symbols tab lists all of the terminal and non-terminal symbols that were used in the parser. The number besides each is the internal id used by the parser.
- The Item Sets tab lists the LR item sets that were used in the parser.

These are printed in a format that is similar to the format used by many text books.

- The Messages tab is used to display any warnings generated while the parser was compiled. The most common warning is for ambiguous actions.

2.6 Defining actions

Now we need to define the actions that need to happen when we parse our expressions. Currently, our parser is just validating that the expression is a bunch of numbers added together. Generally, you will create some structure that represents what you've parsed (e.g., a parse tree). However, in this case, we are not concerned about the structure, but we are concerned about the result (the value of the expression). For our example, you need to modify the grammar definition to be:

```
Expression
  : Expression "+" Number {'1' + '3'}
  | Number {'1'}
  ;
Number
  : <number> {'1' value asNumber}
  ;
```

The text between the braces is Pharo code that gets evaluated when the rule is applied. Strings with a number get replaced with the corresponding expression in the production.

- In the first Expression rule, the `'1'` will get replaced by the object that matches Expression and the `'3'` gets replaced by the object that matches `Number`.
- The second item in the rule is the `"+"` token. Since we already know what it is, it is not interesting.

Compile the new parser. Now when you do a 'Parse and inspect' from the test pane containing `3 + 4`, you should see the result: `7`.

2.7 Named expressions

One problem with the previous example is that if you need to change a rule then you may also need to change the code for that rule. For example, suppose you inserted a new token at the beginning of a rule, then you would need to change all of your references in the Pharo code. We can alleviate this problem by using named expressions. After each part of a rule, we can specify its name. Names are specified with single quotes and must be legal Pharo variable names. Doing this for our grammar we get:

```
Expression
  : Expression 'expression' "+" Number 'number' {expression + number}
  | Number 'number' {number}
  ;
Number
  : <number> 'numberToken' {numberToken value asNumber}
  ;
```

While this will result in the same language being parsed, it makes it easier to maintain your parsers.

2.8 Extending the language

Let's extend our language to add subtraction. Here's the new grammar:

```
Expression
  : Expression 'expression' "+" Number 'number' {expression + number}
  | Expression 'expression' "-" Number 'number' {expression - number}
  | Number 'number' {number}
  ;
Number
  : <number> 'numberToken' {numberToken value asNumber}
  ;
```

After you've compiled this, `'3 + 4 - 2'` should return `'5'`. Next, let's add multiplication and division:

```
Expression
  : Expression 'expression' "+" Number 'number' {expression + number}
  | Expression 'expression' "-" Number 'number' {expression - number}
  | Expression 'expression' "*" Number 'number' {expression * number}
  | Expression 'expression' "/" Number 'number' {expression / number}
  | Number 'number' {number}
  ;
Number
  : <number> 'numberToken' {numberToken value asNumber}
  ;
```

2.9 Handling priority

Here we run into a problem. If you evaluate `'2 + 3 * 4'` you end up with 20. The problem is that in standard mathematics, multiplication has a higher precedence than addition. Our grammar evaluates strictly left-to-right. The standard solution for this problem is to define additional non-terminals to force the sequence of evaluation. Our grammar with that solution would look like:

```
Expression
  : Term 'term' {term}
  | Expression 'expression' "+" Term 'term' {expression + term}
  | Expression 'expression' "-" Term 'term' {expression - term}
  ;
Term
  : Number 'number' {number}
  | Term 'term' "*" Number 'number' {term * number}
  | Term 'term' "/" Number 'number' {term / number}
  ;
Number
  : <number> 'numberToken' {numberToken value asNumber}
  ;
```

If you compile this grammar, you will see that `'2 + 3 * 4'` evaluates to `'14'` like we would expect.

2.10 Handling priority with directives

Now, as you can imagine, this gets pretty complicated as the number of precedence rules increases (e.g., C). We can use ambiguous grammars and precedence rules to simplify this situation. Here is the same grammar using precedence to enforce our evaluation order:

```
%left "+" "-";
%left "*" "/";

Expression
  : Expression 'exp1' "+" Expression 'exp2' {exp1 + exp2}
  | Expression 'exp1' "-" Expression 'exp2' {exp1 - exp2}
  | Expression 'exp1' "*" Expression 'exp2' {exp1 * exp2}
  | Expression 'exp1' "/" Expression 'exp2' {exp1 / exp2}
  | Number 'number' {number}
  ;
Number
  : <number> 'numberToken' {numberToken value asNumber}
  ;
```

Notice that we changed the grammar so that there are Expressions on both sides of the operator. The two lines that we added to the top of the grammar mean that `+` and `-` are evaluated left-to-right and have the same precedence, which is lower than `*` and `/`. Likewise, the second line means that `*` and `/` have equal precedence. Grammars in this form are usually much more intuitive, especially in cases with many precedence levels. Just as an example, let’s add exponentiation and parentheses. Here is our final grammar:

```
<number> : [0-9]+ (\. [0-9]*) ? ;
<whitespace> : \s+;
%left "+" "-";
%left "*" "/";
%right "^";
```

```
Expression
  : Expression 'exp1' "+" Expression 'exp2' {exp1 + exp2}
  | Expression 'exp1' "-" Expression 'exp2' {exp1 - exp2}
  | Expression 'exp1' "*" Expression 'exp2' {exp1 * exp2}
  | Expression 'exp1' "/" Expression 'exp2' {exp1 / exp2}
  | Expression 'exp1' "^" Expression 'exp2' {exp1 raisedTo: exp2}
  | "(" Expression 'expression' ")" {expression}
  | Number 'number' {number}
  ;
Number
  : <number> 'numberToken' {numberToken value asNumber}
  ;
```

Once you have compiled the grammar, you will be able to evaluate `3 + 4 * 5 ^ 2 ^ 2` to get 2503. Since the exponent operator is right associative, this expression is evaluated like `3 + (4 * (5 ^ (2 ^ 2)))`. We can also evaluate expressions with parentheses. For example, evaluating `(3 + 4) * (5 - 2) ^ 3` results in 189.

If you would like to extend the calculator to create abstract syntax trees, you can keep reading. If you want more information on other SmaCC options, you can visit the directives, scanner, and parser sections.

CHAPTER 3

SmaCC Scanner

Scanning takes an input stream of characters and converts that into a stream of tokens. The tokens are then passed on to the parsing phase.

The scanner is specified by a collection of token specifications. Each token is specified by:

```
TokenName    :    RegularExpression ;
```

TokenName is a valid variable name that is surrounded by `<>`. For example, `<token>` is a valid `TokenName`, but `<token name>` is not since `token name` isn't a valid variable name. The `RegularExpression` is a regular expression that matches a token. It should match one or more characters in the input stream. The colon character, `:`, is used to separate the TokenName and the `RegularExpression`, and the semicolon character, `;`, is used to terminate the token specification.

3.1 Regular Expression Syntax

While the rules are specified as regular expressions, there are many different syntaxes for regular expressions. We choose a relatively simple syntax that is specified below. If you wish to have a more rich syntax, you can modify the scanner's parser: `SmaCCDefinitionScanner` and `SmaCCDefinitionParser`. These classes were created using SmaCC.

`\character` Matches a special character. The character immediately following the backslash is matched exactly, unless it is a letter. Backslash-letter combinations have other meanings and are specified below.

`\cLetter` Matches a control character. Control characters are the first 26 characters (e.g., \cA equals `Character value: 0`). The letter that follows the `\c` must be an uppercase letter.

\d Matches a digit, `0-9`.

\D Matches anything that is not a digit.

\f Matches a form-feed character, `Character value: 12`.

\n Matches a newline character, `Character value: 10`.

\r Matches a carriage return character, `Character value: 13`.

\s Matches any whitespace character, [\f\n\r\t\v].

\S Matches any non-whitespace character.

\t Matches a tab, `Character value: 9`.

\v Matches a vertical tab, `Character value: 11`.

\w Matches any letter, number or underscore, `[A-Za-z0-9_]`.

\W Matches anything that is not a letter, number or underscore.

\xHexNumber Matches a character specified by the hex number following the `\x`. The hex number must be at least one character long and no more than four characters for Unicode characters and two characters for non-Unicode characters. For example, `\x20` matches the space character (`Character value: 16r20`), and `\x1FFF` matches `Character value: 16r1FFF`.

<token> Copies the definition of <token> into the current regular expression. For example, if we have `<hexdigit> : \d | [A-F] ;`, we can use `<hexdigit>` in a later rule: `<hexnumber> : <hexdigit> + ;`.

<isMethod> Copies the characters where `Character>>isMethod` returns true into the current regular expression. For example, instead of using `\d`, we could use `<isDigit>` since `Character>>isDigit` returns true for digits.

[characters] Matches one of the characters inside the []. This is a shortcut for the `|` operator. In addition to single characters, you can also specify character ranges with the `-` character. For example, `[a-z]` matches any lower case letter.

[^characters] Matches any character not listed in the characters block. `[^a]` matches anything except for `a`.

# comment Creates a comment that is ignored by SmaCC. Everything from the # to the end of the line is ignored.

exp1 | exp2 Matches either exp1 or exp2.

exp1 exp2 Matches exp1 followed by exp2. `\d \d` matches two digits.

exp* Matches exp zero or more times. `0*` matches `''` and `000`.

exp? Matches exp zero or one time. `0?` matches only `''` or `0`.

exp+ Matches exp one or more times. `0+` matches `0` and `000`, but not `''`.

exp{min,max} Matches exp at least min times but no more than max times. `0{1,2}` matches only `0` or `00`. It does not match `''` or `000`.

(exp) Groups exp for precedence. For example, `(a b)*` matches `ababab`. Without the parentheses, `a b *` would match `abbbb` but not `ababab`.

Since there are multiple ways to combine expressions, we need precedence rules for their combination. The or operator, `|`, has the lowest precedence and the `*`, `?`, `+`, and `{,}` operators have the highest precedence. For example, `a | b c *` matches `a` or `bcccc`, but not `accc` or `bcbcbc`. If you wish to match `a` or `b` followed by any number of c's, you need to use `(a | b) c *`.

3.2 Overlapping Tokens

SmaCC can handle overlapping tokens without any problems. For example, the following is a legal SmaCC scanner definition:

```
<variable>  : [a-zA-Z] \w* ;
<any_character> : . ;
```

This definition will match a variable or a single character. A variable can also be a single character `[a-zA-Z]`, so the two tokens overlap. SmaCC handles overlapping tokens by preferring the longest matching token. If multiple tokens match the longest possible token, then the parser uses the first token specified by the grammar unless you override the `SmaCCParser>>tryAllTokens` method. For example, an `a` could be a `<variable>` or an `<any_character>` token, but since `<variable>` is specified first, SmaCC will use it.

3.3 Matching Methods

If your scanner has a method name that matches the name of the token, (e.g. method `whitespace`), that method will get called upon a match of that type. The `SmaCCScanner` superclass already has a default implementation of `whitespace` and `comment`. These methods ignore those tokens by default. If you want to store comments, then you should override the SmaCCScanner»comment method.

Matching methods can also be used to handle overlapping token classes. For example, in the C grammar, a type definition is the same as an identifier. The only way that they can be disambiguated is by looking up the name in the type table. In our example C parser, we have an IDENTIFIER method that is used to determine whether the token is really an IDENTIFIER or whether it is a TYPE_NAME.

3.4 Unreferenced Tokens

If a token is not referenced from a grammar specification, it will not be included in the generated scanner, unless the token’s name is also a name of a method (see previous section). This, coupled with the ability to do substitutions, allows you to have the equivalent of macros within your scanner specification. However, be aware that if you are simply trying to generate a scanner, you will have to make sure that you create a dummy parser specification that references all of the tokens that you want in the final scanner.

3.5 Unicode Characters

SmaCC compiles the scanner into a bunch of conditional tests on characters. Normally, it assumes that characters have values between 0 and 255, and it can make some optimizations based on this fact. With the ”Allow Unicode Characters” option checked, it will assume that characters have values between 0 and 65535.

CHAPTER 4

SmaCC Parser

Parsing converts the stream of tokens provided by the scanner into some object. Normally, this object will be a parse tree, but it does not have to be a parse tree. For example, the SmaCC tutorial shows a calculator. This calculator does not produce a parse tree; it produces the result, a number.

4.1 Production Rules

The production rules contains the grammar for the parser. The first production rule is considered to be the starting rule for the parser. Each production rule consists of a non-terminal symbol name followed by a ":" separator which is followed by a list of possible productions separated by vertical bar, "|", and finally terminated by a semicolon, ";".

```
Expression
  : Expression 'left' "+" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "-" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "*" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "/" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "^" 'operator' Expression 'right' {{Binary}}
  | "(" Expression ")" {{}}
  | Number
  ;
Number
  : <number> {{Number}}
  ;
```

Each production consists of a sequence of non-terminal symbols, tokens, or keywords followed by some optional Smalltalk code enclosed in curly brackets, {} or a AST node definition enclosed in two curly brackets, {{}}. Non-terminal symbols are valid Smalltalk variable names and must be defined somewhere in the parser definition. Forward references are valid. Tokens are enclosed in angle brackets as they are defined in the scanner (e.g., <token>) and keywords are enclosed in double-quotes (e.g., ”then”). Keywords that contain double-quotes need to have two double-quotes per each double-quote in the keyword. For example, if you need a keyword for one double-quote character, you would need to enter ”””” (four double-quote characters).

The Smalltalk code is evaluated whenever that production is matched. If the code is a zero or a one argument symbol, then that method is performed. For a one argument symbol, the argument is an OrderedCollection that contains one element for each item in the production. If the code isn’t a zero or one argument symbol, then the code is executed and whatever is returned by the code is the result of the production. If no Smalltalk code is specified, then the default action is to execute the #reduceFor: method (unless you are producing an AST parser). This method converts all items into an OrderedCollection. If one of the items is another OrderedCollection, then all of its elements are added to the new collection.

Inside the Smalltalk code you can refer to the values of each production item by using literal strings. The literal string, ’1’, refers the to value of the first production item. The values for tokens and keywords will be SmaCCToken objects. The value for all non-terminal symbols will be whatever the Smalltalk code evaluates to for that non-terminal symbol.

4.2 Named Symbols

When entering the Smalltalk code, you can get the value for a symbol by using the literal strings (e.g., ’2’). However, this creates difficulties when modifying a grammar. If you insert some symbol at the beginning of a production, then you will need to modify your Smalltalk code changing all literal string numbers. Instead you can name each symbol in the production and then refer to the name in the Smalltalk code. To name a symbol (non-terminal, token, or keyword), you need to add a quoted variable name after the symbol in the grammar. For example, ”MySymbol : Expression ’expr’ ”+” <number> ’num’ {expr + num} ;” creates two named variables. One for the non-terminal Expression and one for the <number> token. These variables are then used in the Smalltalk code.

4.3 Error Recovery

Normally, when the parser encounters an error, it raises the SmaCCParserError exception and parsing is immediately stopped. However, there are times when you may wish to try to parse more of the input. For example, if you are highlighting code, you do not want to stop highlighting at the first syntax error. Instead you may wish to attempt to recover after the statement separator – the period ".". SmaCC uses the error symbol to specify where error recovery should be attempted. For example, we may have the following rule to specify a list of Smalltalk statements:

```
Statements : Expression | Statements "." Expression ;
```

If we wish to attempt recovery from a syntax error when we encounter a period, we can change our rule to be:

```
Statements : Expression | Statements "." Expression | error "."
    Expression ;
```

While the error recovery allows you to proceed parsing after a syntax error, it will not allow you to return a parse tree from the input. Once the input has been parsed with errors, it will raise a non-resumable SmaCCParserError.

CHAPTER 5

SmaCC Directives

SmaCC has several directives that can change how the scanner and parser is generated. Each directive begins with a % character and the directive keyword. Depending on the directive, there may be a set of arguments. Finally, the directive is terminated with a semicolon character, ; as shown below:

```
%left "+" "-";
%left "*" "/";
%right "^";
%annotate_tokens;
%root Expression;
%prefix AST;
%suffix Node;
%ignore_variables leftParenToken rightParenToken;
```

5.1 Ambiguous Grammars and Precedence

SmaCC can handle ambiguous grammars. Given an ambiguous grammar, SmaCC will produce some parser. However, it may not parse correctly. For an LR parser, there are two basic types of ambiguities, reduce/reduce conflicts and shift/reduce conflicts. Reduce/reduce conflicts are bad. SmaCC has no directives to handle them and just picks one of the choices. These conflicts normally require a rewrite of your grammar or switch to GLR parsing.

On the other hand, shift/reduce conflicts can be handled by directives. When SmaCC encounters a shift/reduce conflict it will perform the shift action by default. However, you can control this action with the `%left`, `%right`, and `%nonassoc` directives. If a token has been declared in a `%left` directive, it means that the token is left-associative. Therefore, the parser will perform a reduce operation. However, if it has been declared as right-associative, it will perform a shift operation. A token defined as %nonassoc will produce an error if that is encountered during parsing. For example, you may want to specify that the equal operator, ”=”, is non-associative, so a = b = c is not parsed as a valid expression. All three directives are followed by a list of tokens.

Additionally, the %left, %right, and %nonassoc directives allow precedence to be specified. The order of the directives specifies the precedence of the tokens. The higher precedence tokens appear on the higher line numbers. For example, the following directive section gives the precedence for the simple calculator in our tutorial:

```
%left "+" "-";
%left "*" "/";
%right "^";
```

The + and - symbols appear on the first line and have the lowest precedence. They are also left-associative so 1 + 2 +3 will be evaluated as (1 + 2) + 3. On the next line are the * and / symbols. Since they appear on a higher line number, they have higher precedence than the + and -. Finally, on line three we have the ”^” symbol. It has the highest precedence. Combining all the rules allows us to parse 1 + 2 * 3 / 4 ^ 2 ^ 3 as 1 + ((2 * 3) / (4 ^ (2 ^ 3))).

5.2 Start Symbols

By default, the left-hand side of the first grammar rule is the start symbol. If you want to multiple start symbols, then you can specify them by using the %start directive followed by the nonterminals that are additional start symbols. This is useful for creating two parsers with two grammars that are similar but slightly different. For example, consider a Pharo parser. You can parse methods, and you can parse expressions. These are two different operations, but have very similar grammars. Instead of creating two different parsers for parsing methods and expressions, we can specify one grammar that parses methods and also specify another starting position for parsing expressions.

The StParser in the SmaCC Example Parsers package has an example of this. The StParser class>>parseMethod: uses the startingStateForMethod position to parse methods and the StParser class>>parseExpression: uses the startingStateForSequenceNode position to parse expressions.

For example if you add the following to an hypothetical grammar

```
%start file expression statement declaration;
```

SmaCC will generate the following class methods on the parseur: startingStateForfile, startingStateForexpression, startingStateForstatement and startingStateFordeclaration.

Then you can parse a subpart as follows:

```
YourParser >> parseStatement: aString
  "Parse an statement."

  ^ (self on: (ReadStream on: aString))
    setStartingState: self startingStateForstatement;
    parse
```

This is useful when you incrementally build your grammar and want to test it at different points.

5.3 Id Methods

Internally, the various token types are represented as integers. However, there are times that you need to reference the various token types. For example, in the CScanner and CParser classes, the `TYPE_NAME` token is identical to the IDENTIFIER token. The IDENTIFIER matching method does a lookup in the type table and if it finds a type definition with the same name as the current IDENTIFIER, it returns the TYPE_NAME token type. To determine what integer this is, the parser was created with an `%id` directive for <IDENTIFIER> and <TYPE_NAME>. This generates the IDENTIFIERId and TYPE_NAMEId methods on the scanner. These methods simply return the number representing that token type. See the `C` sample scanner and parser for an example of how this is used.

5.4 Case Insensitive Scanning

You can specify that the scanner should ignore case differences by using the `%ignorecase;` directive. If you have a language that is case insensitive and has several keywords, this can be a handy feature to have. For example, if you have `THEN` as a keyword in a case insensitive language, you would need to specify a token for then as `<then> : [tT] [hH] [eE] [nN] ;`. This is a pain to enter correctly. When the ignorecase directive is used, SmaCC will automatically convert `THEN` into `[tT][hH][eE][nN]`.

5.5 GLR Parsing

SmaCC allows you to parse ambiguous grammars using a GLR parser. The `%glr;` directive changes the type of parser that SmaCC generates. Instead of your generated parser being a subclass of `SmaCCParser`, when you specify the `%glr;` directive, your parser will be a subclass of SmaCCGLRParser.

If you parse a string that has multiple representations, SmaCC will throw a `SmaCCAmbiguousResultNotification` exception that can be handled by user code. This exception has the potential parses. The value that it is resumed with will be selected as the definitive parse value. If the exception is not handled, then it will pick one as the definitive parse value.

5.6 AST Directives

There are several directives that are used when creating AST's.

- The `%root` directive is used to specify the root class in the AST hierarchy. The `%root` directive has a single argument that is the name that will be used to create the root class in the AST. This class will be created as a subclass of `SmaCCParseNode`.
- The `%prefix` and `%suffix` directives tell SmaCC the prefix and suffix to add to every AST node's class name. These are automatically added to every AST node including the `%root` node. For example, the following will create a `RBProgramNode` class that is a subclass of `SmaCCParseNode` and is the root of all AST nodes defined by this parser.

```
%root Program;
%prefix RB;
%suffix Node;
```

By default all nodes created by SmaCC will be direct subclass of your `%root class`. However, you can specify the hierarchy by using the `%hierarchy` directive. The syntax of the `%hierarchy` is `%hierarchy SuperclassName ( SubclassName );`. If you have multiple subclasses, you can list all of them inside the parenthesis separated by whitespace:

```
%hierarchy Program (Expression Statement);
```

Two final AST directives deal with the generated classes' instance variables.

- One directive allows you to add some unused instance variables to your classes so you can later extend the generated classes to use those variables. To add an instance variable to your class, you can use the `%attributes` directive. The first argument to the directive is the class name, and the second argument is a list of variable names. For example, we could add a variable named `cachedValue` to our `Expression` class with the following `%attributes Expression (cachedValue);`.
- The other instance variable directive is `%ignore_variables`. When SmaCC creates the AST nodes it automatically creates appropriate `=` and `hash` methods. By default, these methods use all variables when comparing equality. The `%ignore_variables` directive allows you to specify certain variables to ignore when comparing. For example, you may wish to ignore parentheses when you compare expressions. If you named your `(` token 'leftParen' and your `)` token 'rightParen', then you can use `%ignore_variables leftParen rightParen;`.

CHAPTER 6

SmaCC Abstract Syntax Trees

SmaCC can generate abstract syntax trees from an annotated grammar. In addition to the node classes to represent the trees, SmaCC also generates a generic visitor for the tree classes. This is handy and boost your productivity especially since we can decide to change the AST form afterwards and get a new one in no time.

6.1 Restarting

To create an AST, you need to annotate your grammar. Let's start with the grammar of our simple expression parser from the tutorial. Since we want to build an AST, we've removed the code that evaluates the expression.

```
<number> : [0-9]+ (\. [0-9]*) ? ;
<whitespace> : \s+;

%left "+" "-";
%left "*" "/";
%right "^";

Expression
    : Expression "+" Expression
    | Expression "-" Expression
    | Expression "*" Expression
    | Expression "/" Expression
    | Expression "^" Expression
    | "(" Expression ")"
    | Number
    ;
Number
```

```
  : <number>
  ;
```

6.2 Building nodes

Building an AST parser works similarly to the normal parser. Instead of inserting Pharo code after each production rule inside braces, {}, we insert the class name inside of double braces, {{}}. Also, instead of naming a variable for use in the Pharo code, we name a variable so that it will be included as an instance variable in the class we are defining.

Let's start with annotating the grammar for the AST node classes that we wish to parse. We need to tell SmaCC where the AST node should be created and the name of the node's class to create. In our example, we'll start by creating three node classes: Expression, Binary, and Number.

```
<number> : [0-9]+ (\. [0-9]*) ? ;
<whitespace> : \s+;

%left "+" "-";
%left "*" "/";
%right "^";

Expression
  : Expression "+" Expression {{Binary}}
  | Expression "-" Expression {{Binary}}
  | Expression "*" Expression {{Binary}}
  | Expression "/" Expression {{Binary}}
  | Expression "^" Expression {{Binary}}
  | "(" Expression ")" {{}}
  | Number
  ;
Number
  : <number> {{Number}}
  ;
```

If you compile this grammar, SmaCC will complain that we need to define a root node. Since the root hasn't been defined SmaCC compiles the grammar as if the {{...}} expressions where not there and generates the same parser as above.

- Notice that for the parenthesized expression, we are using {{}}. This is a shortcut for {{Expression}} (the name of our production's symbol).

- Notice that we didn't annotate the last production in the Expression definition. Since it only contains a single item, Number, SmaCC will pull up its value which in this case will be a Number AST node.

6.3 Variables and unnamed entities

Now, let's add variable names to our rules:

```
<number> : [0-9]+ (\. [0-9]*) ? ;
<whitespace> : \s+;

%left "+" "-";
%left "*" "/";
%right "^";
%annotate_tokens;

Expression
  : Expression 'left' "+" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "-" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "*" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "/" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "^" 'operator' Expression 'right' {{Binary}}
  | "(" Expression ")" {{}}
  | Number
  ;
Number
  : <number> {{Number}}
  ;
```

The first thing to notice is that we added the `%annotate_tokens;` directive. This directive tells SmaCC to automatically create an instance variable for every unnamed token and keyword in the grammar. An unamed token is a <> not followed by a variable (defined with `'aVariable'`) and an unnamed keyword is delimited by double quotes as in `"("`.

In our example above, we have

- one unnamed token, `<number>`, and
- two unnamed keywords, `(` and `)`.

When SmaCC sees an unnamed token or keyword, it adds a variable that is named based on the item and appends Token to the name. For example, in our example above, SmaCC will use

- leftParenToken for `(`,
- rightParenToken for `)`, and
- `numberToken` for `<number>`.

The method `SmaCCGrammar class>>tokenNameMap` contains the mapping to convert the keyword characters into valid Pharo variable names. You can modify this dictionary if you wish to change the default names.

6.4 Unnamed symbols

Notice that we did not name Expression in the `(` Expression `)` production rule. When you don't name a symbol in a production, SmaCC tries to figure out what you want to do. In this case, SmaCC determines that the Expression symbol produces either a Binary or Number node. Since both of these are subclasses of the Expression, SmaCC will pull up the value of Expression and add the parentheses to that node. So, if you parse `(3 + 4)`, you'll get a Binary node instead of an Expression node.

6.5 Generating the AST

Now we are ready to generate our AST. We need to add directives that tell SmaCC our root AST class node and the prefix and suffix of our classes.

```
<number> : [0-9]+ (\. [0-9]*) ? ;
<whitespace> : \s+;

%left "+" "-";
%left "*" "/";
%right "^";
%annotate_tokens;
%root Expression;
%prefix AST;
%suffix Node;

Expression
  : Expression 'left' "+" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "-" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "*" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "/" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "^" 'operator' Expression 'right' {{Binary}}
  | "(" Expression ")" {{}}
  | Number
  ;
Number
  : <number> {{Number}}
  ;
```

When you compile this grammar, in addition to the normal parser and scanner classes, SmaCC will create `ASTExpressionNode`, `ASTBinaryNode`, and `ASTNumberNode` node classes and an `ASTExpressionNodeVisitor` class that implements the visitor pattern for the tree classes.

The `ASTExpressionNode` class will define two instance variables, `leftParenTokens` and `rightParenTokens`, that will hold the `(` and `)` tokens. Notice that these variables hold a collection of tokens instead of a single parenthesis token. SmaCC figured out that each expression node could contain multiple parentheses and made their variables hold a collection. Also, it pluralized the leftParentToken variable name to leftParenTokens. You can customize how it pluralizes names in the SmaCCVariableDefinition class (See pluralNameBlock and pluralNames).

The ASTBinaryNode will be a subclass of ASTExpressionNode and will define three variables: left, operator, and right.

- The left and right instance variables will hold other ASTExpressionNodes and
- the operator instance variable will hold a token for the operator.

Finally, the ASTNumberNode will be a subclass of ASTExpressionNode and will define a single instance variable, number, that holds the token for the number.

Now, if we inspect the result of parsing 3 + 4, we'll get an Inspector on an ASTBinaryNode.

6.6 AST comparison

SmaCC also generates the comparison methods for each AST node. Let's add function evaluation to our expression grammar to illustrate this point.

```
<number> : [0-9]+ (\. [0-9]*) ? ;
<name> : [a-zA-Z]\w*;
<whitespace> : \s+;

%left "+" "-";
%left "*" "/";
%right "^";
%annotate_tokens;
%root Expression;
%prefix AST;
%suffix Node;

Expression
  : Expression 'left' "+" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "-" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "*" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "/" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "^" 'operator' Expression 'right' {{Binary}}
  | "(" Expression ")" {{}}
  | Number
  | Function
  ;
Number
  : <number> {{Number}}
  ;
```

```
Function
  : <name> "(" 'leftParen' _Arguments ")" 'rightParen' {{}}
  ;
_Arguments
  :
  | Arguments
  ;
Arguments
  : Expression 'argument'
  | Arguments "," Expression 'argument'
  ;
```

Now, if we inspect `Add(3, 4)`, we'll get something that looks like an `ASTFunctionNode`.

In addition to the generating the classes, SmaCC also generates the comparison methods for each AST node. For example, we can compare two parse nodes: `(CalculatorParser parse: '3 + 4') = (CalculatorParser parse: '3+4')`. This returns true as whitespace is ignored. However, if we compare `(CalculatorParser parse: '(3 + 4)') = (CalculatorParser parse: '3+4')`, we get false, since the first expression has parentheses. We can tell SmaCC to ignore these by adding the `%ignore_variables directive`:

```
<number> : [0-9]+ (\. [0-9]*) ? ;
<name> : [a-zA-Z]\w*;
<whitespace> : \s+;

%left "+" "-";
%left "*" "/";
%right "^";
%annotate_tokens;
%root Expression;
%prefix AST;
%suffix Node;
%ignore_variables leftParenToken rightParenToken;

Expression
  : Expression 'left' "+" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "-" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "*" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "/" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "^" 'operator' Expression 'right' {{Binary}}
  | "(" Expression ")" {{}}
  | Number
  | Function
  ;
Number
  : <number> {{Number}}
  ;
```

```
Function
  : <name> "(" 'leftParen' _Arguments ")" 'rightParen' {{}}
  ;
_Arguments
  :
  | Arguments
  ;
Arguments
  : Expression 'argument'
  | Arguments "," Expression 'argument'
  ;
```

Now, we get true when we compare `(CalculatorParser parse: '(3 + 4)') = (CalculatorParser parse: '3+4')`.

6.7 Extending the visitor

Finally, let's subclass the generated visitor to create a visitor that evaluates the expressions. Here's the code for Pharo:

```
ASTExpressionNodeVisitor subclass: #ASTExpressionEvaluator
  instanceVariableNames: 'functions'
  classVariableNames: ''
  package: 'SmaCC-Tutorial'.
```

```
ASTExpressionEvaluator >> functions
  ^functions
    ifNil:
      [functions := (Dictionary new)
            at: ''Add'' put: [:a :b | a + b];
            yourself ]' classified: 'private'.
```

```
ASTExpressionEvaluator >> visitBinary: aBinary
  | left right operation |
  left := self acceptNode: aBinary left.
  right := self acceptNode: aBinary right.
  operation := aBinary operator value.
  operation = ''^'' ifTrue: [ ^left ** right ].
  ^left perform: operation asSymbol with: right' classified:
    'visiting'.
```

```
ASTExpressionEvaluator >> visitFunction: aFunction
  | function arguments |
  function := self functions at: aFunction nameToken value
        ifAbsent:
          [self error: ''Function '' ,
            aFunction nameToken value ,
            '' is not defined'' ].
  arguments := aFunction arguments collect: [ :each | self
    acceptNode: each ].
  ^function valueWithArguments: arguments asArray' classified:
    'visiting'.
```

```
ASTExpressionEvaluator >> visitNumber: aNumber
  ^ aNumber numberToken value asNumber' classified: 'visiting'.
```

Now we can evaluate `ASTExpressionEvaluator new accept: (CalculatorParser parse: 'Add(3,4) * 12 / 2 ^ (3 - 1) + 10')` and get 31.

CHAPTER 7

SmaCC Transformations

Once you have generated your AST using SmaCC, you can use SmaCC's built in transformation support.

7.1 Transforming

Let's add support for transforming our simple expressions generated from our AST example.

The first thing we need to do is to extend our grammar by adding two lines.

- The first line we need to add is the definition of a pattern for our language. When your grammar defines the `<patternToken>`, SmaCC uses this as the definition of a pattern for your language. For most languages, patterns are simply anything delimited by ` characters (e.g., `` `pattern` ``).
- The other line we need to add is the line to tell SmaCC to generate a GLR parser (`%glr;`). This allows SmaCC to parse all possible representations of a pattern expression. Here is our grammar with those two additional lines:

```
  <number> : [0-9]+ (\. [0-9]*) ? ;
  <name> : [a-zA-Z]\w*;
  <whitespace> : \s+;

+ <patternToken> : \` [^\`]* \` ;
+ %glr;

  %left "+" "-";
  %left "*" "/";
  %right "^";
```

```
%annotate_tokens;
%root Expression;
%prefix AST;
%suffix Node;
%ignore_variables leftParenToken rightParenToken;

Expression
  : Expression 'left' "+" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "-" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "*" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "/" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "^" 'operator' Expression 'right' {{Binary}}
  | "(" Expression ")" {{}}
  | Number
  | Function;
Number : <number> {{Number}};
Function
  : <name> "(" 'leftParen' _Arguments ")" 'rightParen' {{}};
_Arguments
  :
  | Arguments;
Arguments
  : Expression 'argument'
  | Arguments "," Expression 'argument';
```

7.2 Pattern matching expressions

These changes modify our grammar to support parsing pattern matching expressions. Pattern matching expressions look like normal expressions, but may include pattern's that are surrounded by the back quote, `, character.

For example, `` `a` + 1`` is a simple pattern matching expression that matches any expression `+ 1`.

Once the pattern has been matched, we can supply a replacement expression that uses the pattern variables from our match. Replacement expressions are strings that can contain back quoted variables. These back quoted variables are replaced with their source from their corresponding matched node.

For example, if we are searching for `` `a` + 1``, we can supply a replacement expression like `` 1 + `a` ``. This pattern will match `(3 + 4) + 1`. When we perform the replacement we take the literal `1 + ` part of the string and concatenate the value of the node that matched `` `a` ``. In this case, we would concatenate `(3 + 4)` to give us `1 + (3 + 4)`.

7.3 Example

As an example, let's rewrite addition expressions using reverse Polish notation. Our search pattern is `` `a` + `b` `` and our replacement expression is `` `a` `b` + ``.

```
| rewriter compositeRewrite rewrite matcher transformation |
compositeRewrite := SmaCCRewriteFile new.
compositeRewrite parserClass: CalculatorParser.
matcher := SmaCCRewriteTreeMatch new.
matcher source: '`a` + `b`'.
transformation := SmaCCRewriteStringTransformation new.
transformation string: '`a` `b` +'.
rewrite := SmaCCRewrite
  comment: 'Postfix rewriter'
  match: matcher
  transformation: transformation.
compositeRewrite addTransformation: rewrite.
rewriter := SmaCCRewriteEngine new.
rewriter rewriteRule: compositeRewrite.
rewriter rewriteTree: (CalculatorParser parse: '(3 + 4) + (4 + 3)')
```

This code rewrites `(3 + 4) + (4 + 3)` in RPN format and returns `3 4 + 4 3 + +`. The first match that this finds is `` `a` `` = `(3 + 4)` and `` `b` `` = `(4 + 3)`. Inside our replacement expression, we refer to `` `a` `` and `` `b` ``, so we first process those expression for more transformations. Since both contain other addition expressions, we rewrite both expressions to get `` `a` `` = `3 4 +` and `` `b` `` = `4 3 +`.

Here's the same example, using SmaCC special rewrite syntax.

```
| rewriter rewriteExpression |
rewriteExpression :=
  'Parser: CalculatorParser
  >>>`a` + `b`<<<
  ->
  >>>`a` `b` +<<<'.
rewriter := SmaCCRewriteEngine new.
rewriter rewriteRule: (SmaCCRewriteRuleFileParser parse:
   rewriteExpression).
rewriter rewriteTree: (CalculatorParser parse: '(3 + 4) + (4 + 3)')
```

7.4 Parametrizing transformations

Let's extend our RPN rewriter to support other expressions besides addition. Now we could do that by providing rewrites for all possible operators (+, -, *, /, ^), but it would be better if we could do it with patterns. We may wish to do use `` `a` `op` `b` ``, but the pattern `` `op` `` will only match AST nodes and not a token `(+)`. We can tell SmaCC to match tokens by using `` `a` `op{beToken}` `b` ``. Here's the rewrite expression that works for all expressions:

```
Parser: CalculatorParser
\>\>\>`a` `op{beToken}` `b`\<\<\<
->
\>\>\>`a` `b` `op`\<\<\<
```

If we transform `(3 + 4) * (5 - 2) ^ 3`, we'll get `3 4 + 5 2 - 3 ^ *`.

CHAPTER 8

Grammar idiomatic patterns

In this part, we want to share some coding grammar idioms. Imagine that we have a description using the traditional * (for 0 or more), interrogation mark (? for 0 or 1) and + (for 1 or more). The question then is how can we express this in SmaCC.

8.1 Managing List

Smacc automatically determines if, in the production rules, there is a recursion that represents a list. In such case, it adds an s to the generated instance variable and manage it as a list.

Let us take an example

```
<a> : a;
<whitespace> : \s+;

%root Line;
%prefix SmaccTutorial;

Line
  : <a> 'line' {{}}
  | Line <a> 'line' {{}}
  ;
```

Here we see that Line is recursive. Smacc will generate a class SmaccTutorialLine with an instance variable lines initialized as an ordered collection.

Pay attention, if the production is empty, the generation does not see the list.

```
Line
  :
  | Line <a> 'line' {{}}
  ;
```

In such a case you should write it as follows:

```
Line
  : {{}}
  | Line <a> 'line' {{}}
  ;
```

8.2 Expressing optional repetition

Here is a typical expression mixing

```
TypeNameList = '(' (TypeName (',' TypeName )* )? ')'
```

Here is how we can express it.

```
ParenthesizedTypeNameList
  : "(" TypeNameList_Opt ")"
  ;

TypeNameList_Opt
  :
  | TypeNameList
  ;

TypeNameList
  : TypeName 'typename' {{}}
  | TypeNameList "," TypeName 'typename' {{}}
  ;
```

Not that in the following

```
TypeNameList_Opt
  :
  | TypeNameList
  ;
```

will return nil when empty. If you want to get the node you should use {{}}

CHAPTER 9

Conclusion

SmaCC is a really strong and stable library that is used in production since many years. It is an essential assets for dealing with languages. While PetitParser (See Deep into Pharo http://books.pharo.org) is useful for composing and reusing fragments of parsers, Smacc offers speed and more traditional parsing technology.

CHAPTER 10

Vocabulary

10.1 Reference example

Let us take the following grammar.

```
<number> : [0-9]+ (\. [0-9]*) ? ;
<whitespace> : \s+;

%left "+" "-";
%left "*" "/";
%right "^";
%annotate_tokens;
%root Expression;
%prefix AST;
%suffix Node;

Expression
  : Expression 'left' "+" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "-" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "*" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "/" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "^" 'operator' Expression 'right' {{Binary}}
  | "(" Expression ")" {{}}
  | Number
  ;
Number
  : <number> {{Number}}
  ;
```

10.2 Grammar structure

It is composed of

- Scanner part: all rules starting with <>
- Directive part: all lines starting with `%`
- Parser part: the rest

Elements

Production rule

The following expressions define two production rules

```
Expression
  : Expression 'left' "+" 'operator' Expression 'right' {{Binary}}
  | Expression 'left' "-" 'operator' Expression 'right' {{Binary}}
  ;

Number
  : <number> {{Number}}
  ;
```

A rule group is defined by several production rules.

- Here the first production rule has two production rules.
- While the seconde production rule has only one.

A production rule can be composed of

- non terminal often starting with uppercase
- scanner token
- keywords (delimited by `"`)
- variables (delimited by `'`)
- and action (delimited by `{}`)

Tokens

Tokens are identified by the scanner. A token specification is composed of a token name and a token regular expressions.

```
<TokenName>    :    RegularExpression ;
```

The following token specification describes a number: It starts with one or more digits, possibly followed by an decimal point with zero or more digits after it. The scanner definition for this token is:

```
<number>        :       [0-9]+ (\. [0-9]*) ? ;
```

Let’s go over each part:

`<number>` Names the token identified by the expression. The name inside the <> must be a legal Pharo variable name.

`:` Separates the name of the token from the token’s definition.

`[0-9]` Matches any single character in the range `'0'` to `'9'` (a digit). We could also use `\d` or `<isDigit>` as these also match digits.

`+` Matches the previous expression one or more times. In this case, we are matching one or more digits.

`( ... )` Groups subexpressions. In this case we are grouping the decimal point and the numbers following the decimal point.

`\.` Matches the ’.’ character (. has a special meaning in regular expressions, quotes it).

`*` Matches the previous expression zero or more times.

`?` Matches the previous expression zero or one time (i.e., it is optional).

`;` Terminates a token specification.

Keywords

Keywords are defined in the production and delimited by `"`. In the following

Non Terminal

In the production rule `Expression 'left' "+" 'operator' Expression 'right'`, Expression is a non terminal.

Variables

Variables give name to one element of a production. For example

```
Expression 'left' "^" 'operator' Expression 'right'
```

- ’left’ and ’right’ denote the expression matched by the rules
- ’operator’ denotes the caret token.