

SetlCup Tutorial

Jonas Eilers

February 2, 2016

Contents

1	Functionality	2
1.1	Using SetlCup	2
1.1.1	Calling via comment prompt	2
1.1.2	Calling via Setlx	3
1.2	Comment-Part	3
1.3	Scanner-Part	3
1.4	Parser-Part	4
1.5	Example	5
1.5.1	Arithmetic grammar	5
1.5.2	Programming language grammar	6

Chapter 1

Functionality

The Setlx-addition SetlCup is a LR-Parser-Generator based on JavaCup. The idea is to use a user given scanner- and parser-definition and create an AST out of a given input using the definitions.

In this document the needed syntax of the definitions is examined and the given output is evaluated.

A sample input file is divided into three Sections:

1. Commentpart
2. Scanner-Part
3. Parser-Part

At first the correct call of the program is discussed.

1.1 Using SetlCup

SetlCup has multiple different variants in which it can be called:

1.1.1 Calling via comment prompt

1. `setlx setlcup.stlx -p parser_scanner_file.stlx file_to_be_read.txt`

With this call there will be no output for the user.

2. `setlx setlcup.stlx -p parser_scanner_file.stlx file_to_be_read.txt -d`

With this call debugging is possible. It shows the different tables and states and the whole parsing progress. HINT: It is recommended to pipe the output into a file if you are using the "-d" option.

3. `setlx setlcup.stlx -h`

With this call a little help will be showed, on how to call SetlCup correctly.

1.1.2 Calling via Setlx

SetlCup can also be called in Setlx itself. If this case is used, you need to load the program "setlcup_load.stlx". Afterwards Setlcup can be used via the method call

```
call_generate_ast(input_grammar, file_to_parse, silent_mode);
```

e.g.

```
1      load("setlcup_load.stlx");
2      print(call_generate_ast('examples\math_expression_grammar_ast.g',
3                             'examples\math_expression_input.txt', true));
```

1.2 Comment-Part

In the comment-part everything which is written will not be used by the Program itself. It is advised to comment your idea behind the parser and scanner structure in this section. The section is ended with the "%%%" symbol.

1.3 Scanner-Part

The scanner is responsible for checking whether the input file consists of the defined tokens. It can be written like this:

```
1      INTEGER      := 0|[1-9][0-9]*      ;
2      ASTERISK     := \*                  ;
3      WHITESPACE   := [\t\v\r\s]         ;
4      SKIP         := {WHITESPACE} | \n ;
```

Figure 1.1: Scanner Definition

- In line 1 the Token "INTEGER" is defined. Tokens are defined in the following way:
token_name := regex ;
- In line 2 it is shown, that predefined tokens in Regular Expressions like "*", "+", "?", "|", "{", "}", "(", ")", "..." need to be escaped.
- In line 3 the "Whitespace" symbols are demonstrated.
- In Line 4 the "SKIP"-Token is shown. In some contexts tokens like Whitespaces are not needed. They can be skipped by defining the "SKIP"-Token with "{TOKENNAME}" of the respective tokens, which shall be skipped. Multiple tokens need to be separated by a pipe "|". It is also possible to skip by inserting a regex itself.

1.4 Parser-Part

In this part the grammar-rules are defined with the following syntax:

```
1      expr ::=
2          expr:e MINUS prod:p {: result := Minus(e,p);:}
3      |   expr:e '+' prod:p   {: result := Plus(e,p); :}
4      |
5      ;
```

Figure 1.2: Example grammar rule

rule_head The rule_head is the name of the rule i.e. "expr". It is possible to reference defined rules via their rule_name

body_list The rule can consist of multiple bodys.

rule_body The body can contain multiple elements.

rule_element A rule element is either a :

1. Token (defined in the scanner) e.g. "MINUS"
2. Token in ' ' e.g. '+' as a literal
3. other rule_heads e.g. "prod"

The Tokens defined in the scanner, as well as the rule_heads can have an id. This can be used in the action_code.

action_code The action_code is an optional part in a body. It needs to be at the end of the body it self. Each rule_element can have an action_code. In this action_code Setlx Code can be written. By using the variable "result" it is possible to pass values between rules. The id of the elements in the respective rule can be referred to by using its name.

| The pipe separates the different bodys.

1.5 Example

The first example shows a simple arithmetic grammar. The second example shows how a simple programming language can be parsed using SetlCup.

1.5.1 Arithmetic grammar

The arithmetic grammar and scanner is the following 1.3. A sample input file1.4.

```
1  %%%
2
3  INTEGER      := 0|[1-9][0-9]* ;
4  WHITESPACE   := [\t\v\r\s] ;
5  SKIP         := {WHITESPACE} | \n ;
6
7  %%%
8  arith_expr
9  ::= expr_list:esl          {: result := ExprList(esl); :};
10
11 expr_list
12 ::= expr_part:part expr_list:l {: result := [part] + l; :}
13    |                               {: result := []; :}
14    ;
15 expr_part
16 ::= expr:e ';'              {: result := e; :} ;
17 expr
18 ::=  expr:e '+' prod:p      {: result := Plus(e , p); :}
19    |  expr:e '-' prod:p     {: result := Minus(e , p); :}
20    |  prod:p                {: result := p; :}
21    ;
22 prod
23 ::=  prod:p '*' fact:f      {: result := Times(p , f); :}
24    |  prod:p DIVIDE fact:f  {: result := Div(p , f); :}
25    |  prod:p '%' fact:f     {: result := Mod(p , f); :}
26    |  fact:f               {: result := f; :}
27    ;
28 fact
29 ::= '(' expr:e_part ')'     {: result := e_part ; :}
30    |  INTEGER:n             {: result := Integer(eval(n)); :}
31    ;
```

Figure 1.3: Example arithmetic grammar

The output AST:

```

1 1 + 2 * 3 - 4;
2 1 + 2 + 3 + 4;
3 1 + ( 2 * 3 ) * 5 % 6;

```

Figure 1.4: Example arithmetic input

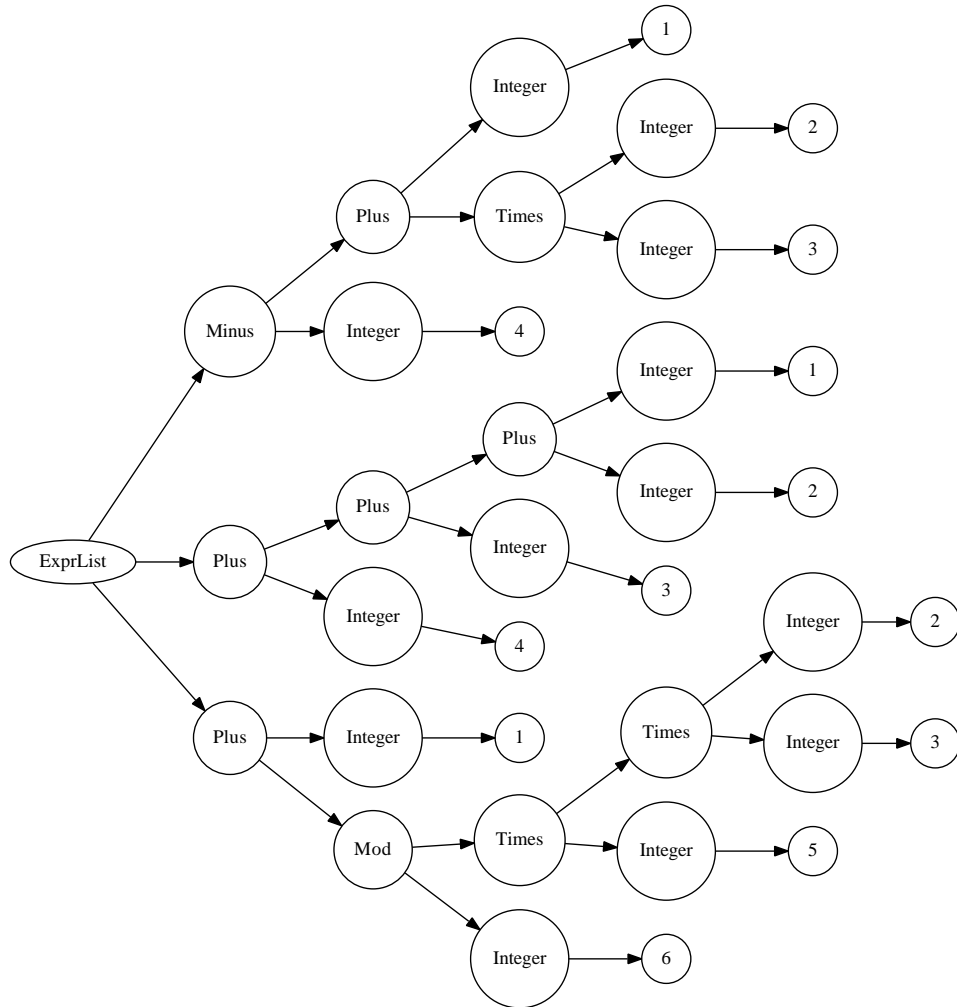


Figure 1.5: Arithexpr AST

1.5.2 Programming language grammar

The programming language grammar and scanner. The Scanner 1.6. The Statements 1.7. The Lists 1.8. The Expressions 1.9. A sample input file 1.10. The output AST:

```

1  %%%
2  FUNCTION      := function ;
3  RETURN       := return ;
4  IF           := if ;
5  ELSE         := else ;
6  WHILE        := while ;
7  FOR          := for ;
8  PRINT        := print ;
9  QUIT         := exit ;
10 STRING       := \"(?:\\.|[^\"])*\" ;
11 WHITESPACE    := [ \t\v\r\s] ;
12 SKIP         := {WHITESPACE}|\n|//[^\n]* ;
13 INTEGER       := 0|[1-9][0-9]* ;
14 DECIMAL       := 0\.[0-9]+|[1-9][0-9]*\.[0-9]+ ;
15 ZID          := [a-zA-Z_][a-zA-Z0-9_]* ;
16 %%%

```

Figure 1.6: Example interpreter scanner

```

1 program
2 ::= dfnStmntList:d {: result := Program(d); :}
3 ;
4 dfnStmntList
5 ::= definition:d dfnStmntList:d1      {: result := [d] + d1; :}
6   | statement:stmts dfnStmntList:dsl  {: result := [stmts] + dsl; :}
7   |                                  {: result := []; :}
8   ;
9 definition
10 ::= FUNCTION ZID:function_name '(' paramList:param_list ')'
11      '{' stmtList:statement_list '}'
12      {: result := Function(function_name, param_list, statement_list);:}
13 ;
14 stmtList
15 ::= statement:s stmtList:sl {: result := [s] + sl ; :}
16   | {: result := []; :}
17   ;
18 statement
19 ::= assignment:a ';'      {: result := Assign(a); :}
20   | PRINT '(' printExprList:printexpr_list ')' ';'
21      {: result := Print(printexpr_list); :}
22   | IF '(' boolExpr:b ')' '{' stmtList:st_list1 '}'
23      {: result := If(b, st_list1); :}
24   | WHILE '(' boolExpr:b ')' '{' stmtList:st_list2 '}'
25      {: result := While(b, st_list2); :}
26   | FOR '(' assignment:i_a ';' boolExpr:b ';' assignment:e_a ')'
27      '{' stmtList:st_list3 '}'
28      {: result := For(i_a, b, e_a, st_list3); :}
29   | RETURN expr:e ';'     {: result := Return(e); :}
30   | RETURN ';'           {: result := Return(); :}
31   | expr:e ';'           {: result := Expr(e); :}
32   | QUIT ';'            {: result := Exit(); :}
33 ;

```

Figure 1.7: Example interpreter statements

```

1 printExprList
2 ::= printExpr:p ',' nePrintExprList:np {: result := [p] + np ; :}
3   | printExpr:p                               {: result := [p]; :}
4   |                               {: result := []; :}
5   ;
6 nePrintExprList
7 ::= printExpr:p                               {: result := [p]; :}
8   | printExpr:p ',' nePrintExprList:np {: result := [p] + np ; :}
9   ;
10 printExpr
11 ::= STRING:string {: result := PrintString(string); :}
12   | expr:e         {: result := e; :}
13   ;
14 assignment
15 ::= ZID:id '=' expr:e {: result := Assign(id, e); :}
16   ;
17 paramList
18 ::= ZID:id ',' neIDList:nid {: result := [id] + nid ; :}
19   | ZID:id                               {: result := [id] ; :}
20   |                               {: result := []; :}
21   ;
22 neIDList
23 ::= ZID:id ',' neIDList:nid {: result := [id] + nid ; :}
24   | ZID:id                               {: result := [id] ; :}
25   ;
26 exprList
27 ::= expr:e ',' neExprList:el {: result := [e] + el; :}
28   | expr:e                               {: result := [e]; :}
29   |                               {: result := []; :}
30   ;
31 neExprList
32 ::= expr:e ',' neExprList:el {: result := [e] + el; :}
33   | expr:e                               {: result := [e]; :}
34   ;

```

Figure 1.8: Example interpreter Lists

```

1 boolExpr
2 ::= expr:lhs '==' expr:rhs           {: result := Equation(lhs,rhs); :}
3   | expr:lhs '!=' expr:rhs           {: result := Inequation(lhs,rhs); :}
4   | disjunction:lhs '==' disjunction:rhs {: result := Equation(lhs,rhs); :}
5   | disjunction:lhs '!=' disjunction:rhs {: result := Inequation(lhs,rhs); :}
6   | expr:lhs '<=' expr:rhs           {: result := LessOrEqual(lhs,rhs); :}
7   | expr:lhs '>=' expr:rhs           {: result := GreaterOrEqual(lhs,rhs); :}
8   | expr:lhs '<' expr:rhs            {: result := LessThan(lhs,rhs); :}
9   | expr:lhs '>' expr:rhs            {: result := GreaterThan(lhs,rhs); :}
10  | disjunction:d                    {: result := d; :}
11  ;
12 disjunction
13 ::= disjunction:d '||' conjunction:c {: result := Disjunction(d,c); :}
14   | conjunction:c                    {: result := c; :}
15   ;
16 conjunction
17 ::= conjunction:c '&&' boolFactor:f {: result := Conjunction(c,f); :}
18   | boolFactor:f                     {: result := f; :}
19   ;
20 boolFactor
21 ::= '(' boolExpr:be_par ')' {: result := be_par; :}
22   | '!' boolExpr:e           {: result := Negation(e); :}
23   ;
24 expr
25 ::= expr:e '+' prod:p {: result := Sum(e,p); :}
26   | expr:e '-' prod:p {: result := Difference(e,p); :}
27   | prod:p             {: result := p; :}
28   ;
29 prod
30 ::= prod:p '*' fact:f   {: result := Product(p,f); :}
31   | prod:p '\' fact:f   {: result := Quotient(p,f); :}
32   | prod:p '%' fact:f   {: result := Mod(p,f); :}
33   | fact:f              {: result := f; :}
34   ;
35 fact
36 ::= '(' expr:e_par ')'   {: result := e_par; :}
37   | INTEGER:n            {: result := Integer(eval(n)); :}
38   | DECIMAL:d            {: result := Decimal(eval(d)); :}
39   | ZID:id_1 '(' exprList:el ')' {: result := FunctionCall(id_1,el); :}
40   | ZID:id_2              {: result := Variable(id_2); :}
41   ;

```

Figure 1.9: Example interpreter expressions

```

1 function factorial(n) {
2     if (n == 0) {
3         return 1;
4     }
5     return n * factorial(n - 1);
6 }
7 print("Calculation of factorial for i = 1 to 9");
8 for (i = 0; i < 10; i = i + 1) {
9     print(i, "! = ", factorial(i));
10 }
11 print();

```

Figure 1.10: Example interpreter input

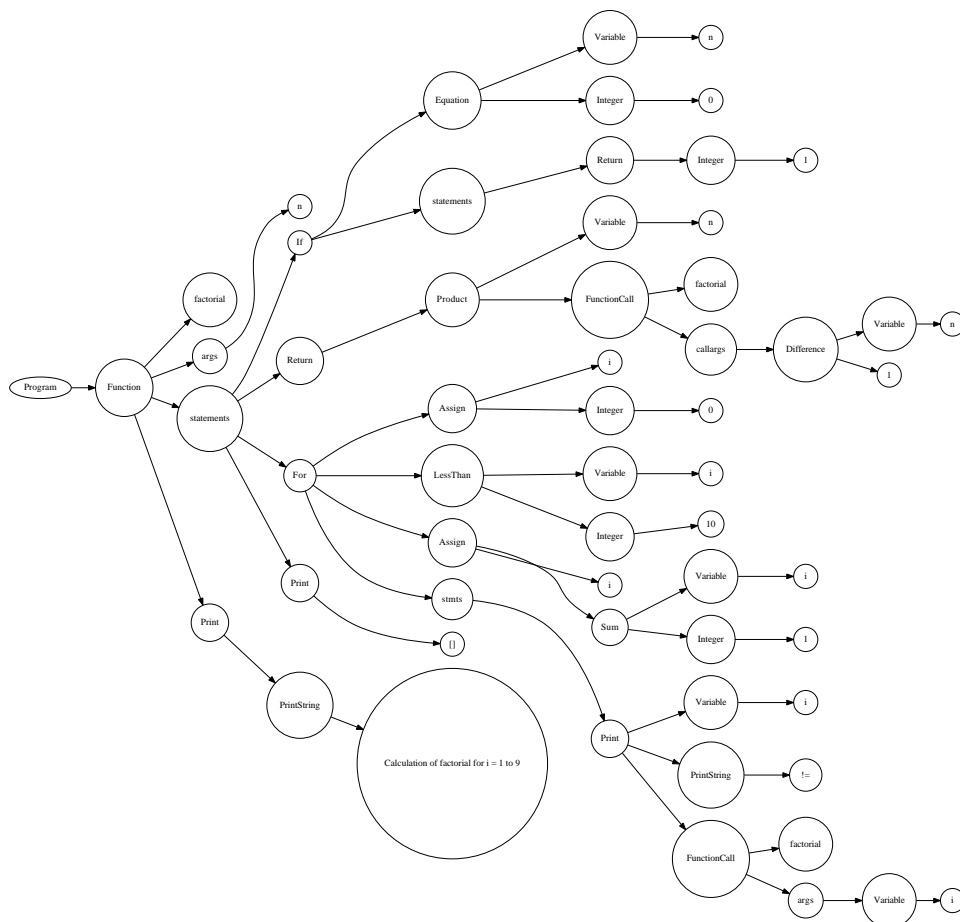


Figure 1.11: Interpreter AST