SetlCup Tutorial

Jonas Eilers

February 4, 2016

Contents

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.

```
load("setlcup_load.stlx");
print(call_generate_ast('examples\math_expression_grammar_ast.g',
                    '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 adviced 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 by assigning the Regex to the Tokenname (see Figure 1.1, Page 3).

```
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 "$*, +, ?, |, \{, \}, (,), \cdots$" 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 seperated 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 a syntax which has many analogies to JavaCup and ANTLR (see Figure 1.2, Page 4).

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

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 bodies.

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 (see Figure 1.3, Page 5) can be defined using the syntax mentioned above. The input file consists of multiple lines with

```
%%%

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

%%%
arith_expr
 ::= expr_list:esl                {: result := ExprList(esl); :};

expr_list
 ::= expr_part:part expr_list:l {: result := [part] + l; :}
   |                            {: result := []; :}
   ;
expr_part
 ::= expr:e ';'             {: result := e; :} ;
expr
 ::=  expr:e '+'  prod:p   {: result := Plus(e , p);  :}
   |  expr:e '-'  prod:p   {: result := Minus(e , p); :}
   |  prod:p               {: result := p;     :}
   ;
prod
 ::=  prod:p '*'  fact:f   {: result := Times(p , f); :}
   |  prod:p DIVIDE fact:f {: result := Div(p , f); :}
   |  prod:p '%'    fact:f {: result := Mod(p , f); :}
   |  fact:f               {: result := f;     :}
   ;
fact
 ::=  '(' expr:e_part ')'   {: result :=  e_part ;   :}
   |  INTEGER:n             {: result := Integer(eval(n)); :}
   ;
```

Figure 1.3: Example arithmetic grammar

arithmetic expressions 1.4. The output AST(see Figure 1.5, Page 6) consists of the three expressions noted above.

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

Figure 1.4: Example arithmetic input

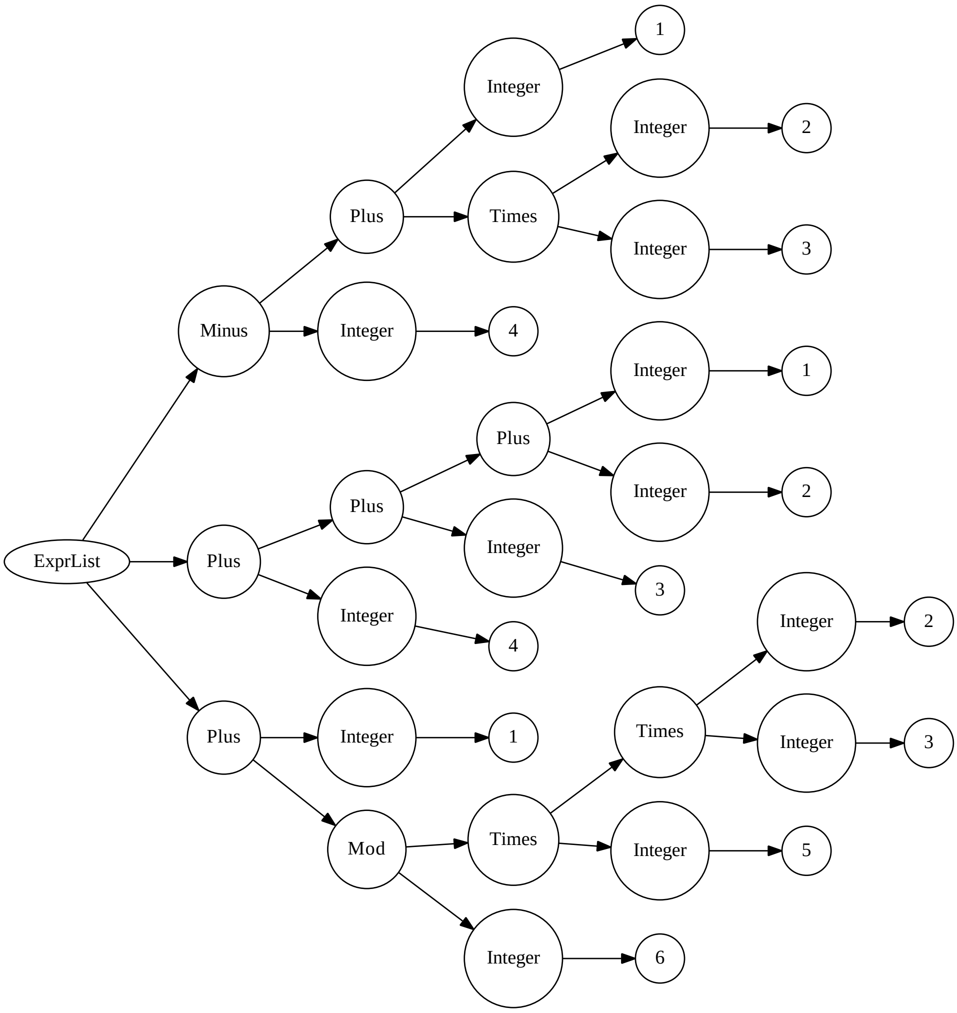

Figure 1.5: Arithexpr AST

1.5.2 Programming language grammar

The programming language grammar and scanner. The Scanner (see Figure 1.6, Page 7) consists of the string literals for keywords and the definition of the syntax for a variable or function name, integers and decimals. The Statements (see Figure 1.7, Page 8) describe the structure of the input file. It consists of multiple statement and definitions. The Lists (see Figure 1.8, Page 9) describe how multiple arguments, expressions, definitions and statements are chained. The Expressions (see Figure 1.9, Page 10) describe boolean and arithmetic expressions. A sample input program (see Figure 1.10, Page 11) describes

```
%%%
FUNCTION     := function ;
RETURN       := return ;
IF           := if ;
ELSE         := else ;
WHILE        := while ;
FOR          := for ;
PRINT        := print ;
QUIT         := exit ;
STRING       := \"(?:\\.|[^\"])*\" ;
WHITESPACE   := [ \t\v\r\s] ;
SKIP         := {WHITESPACE}|\n|//[^\n]* ;
INTEGER      := 0|[1-9][0-9]* ;
DECIMAL      := 0\.[0-9]+|[1-9][0-9]*\.[0-9]+ ;
ZID          := [a-zA-Z_][a-zA-Z0-9_]* ;
%%%
```

Figure 1.6: Example interpreter scanner

the recursive function factorial. The output AST (see Figure 1.11, Page 11) represents the syntactic structure of the factorial program.

```
program
 ::= dfnStmntList:d {: result := Program(d); :}
   ;
dfnStmntList
 ::= definition:d dfnStmntList:dl       {: result := [d] + dl; :}
   |  statement:stmts  dfnStmntList:dsl {: result := [stmts] + dsl; :}
   |                                    {: result := []; :}
   ;
definition
 ::= FUNCTION ZID:function_name '(' paramList:param_list ')'
                      '{' stmntList:statement_list '}'
     {: result := Function(function_name, param_list, statement_list);:}
   ;
stmntList
 ::= statement:s stmntList:sl {: result := [s] + sl ; :}
   |  {: result := []; :}
   ;
statement
 ::= assignment:a ';'   {: result := Assign(a); :}
   |  PRINT '(' printExprList:printexpr_list ')' ';'
                        {: result := Print(printexpr_list); :}
   |  IF '(' boolExpr:b ')' '{' stmntList:st_list1 '}'
                        {: result := If(b, st_list1); :}
   |  WHILE '(' boolExpr:b ')' '{' stmntList:st_list2 '}'
                        {: result := While(b, st_list2); :}
   |  FOR '(' assignment:i_a ';' boolExpr:b ';' assignment:e_a ')'
                  '{' stmntList:st_list3 '}'
                        {: result := For(i_a, b, e_a, st_list3);  :}
   |  RETURN expr:e ';' {: result := Return(e); :}
   |  RETURN ';'        {: result := Return();  :}
   |  expr:e ';'        {: result := Expr(e);   :}
   |  QUIT ';'          {: result := Exit();    :}
   ;
```

Figure 1.7: Example interpreter statements

```
printExprList
 ::= printExpr:p ’,’ nePrintExprList:np {: result := [p] + np ; :}
   |  printExpr:p                      {: result := [p]; :}
   |                                   {: result := [];  :}
   ;
nePrintExprList
 ::= printExpr:p                        {: result := [p]; :}
   |  printExpr:p ’,’ nePrintExprList:np {: result := [p] + np ; :}
   ;
printExpr
 ::= STRING:string {: result := PrintString(string); :}
   |  expr:e       {: result := e; :}
   ;
assignment
 ::= ZID:id ’=’ expr:e {: result := Assign(id, e); :}
   ;
paramList
 ::= ZID:id ’,’ neIDList:nid {: result := [id] + nid ; :}
   |  ZID:id                 {: result := [id] ; :}
   |                         {: result := []; :}
   ;
neIDList
 ::= ZID:id ’,’ neIDList:nid {: result := [id] + nid ; :}
   |  ZID:id                 {: result := [id] ; :}
   ;
exprList
 ::= expr:e ’,’ neExprList:el {: result := [e] + el; :}
   |  expr:e                  {: result := [e]; :}
   |                          {: result := [];  :}
   ;
neExprList
 ::= expr:e ’,’ neExprList:el {: result := [e] + el; :}
   |  expr:e                  {: result := [e]; :}
   ;
```

Figure 1.8: Example interpreter Lists

```
boolExpr
 ::= expr:lhs '==' expr:rhs                {: result := Equation(lhs,rhs); :}
   |  expr:lhs '!=' expr:rhs               {: result := Inequation(lhs,rhs); :}
   |  disjunction:lhs '==' disjunction:rhs {: result := Equation(lhs,rhs); :}
   |  disjunction:lhs '!=' disjunction:rhs {: result := Inequation(lhs,rhs); :}
   |  expr:lhs '<=' expr:rhs               {: result := LessOrEqual(lhs,rhs);    :}
   |  expr:lhs '>=' expr:rhs               {: result := GreaterOrEqual(lhs,rhs); :}
   |  expr:lhs '<' expr:rhs                {: result := LessThan(lhs,rhs); :}
   |  expr:lhs '>' expr:rhs                {: result := GreaterThan(lhs,rhs); :}
   |  disjunction:d                        {: result := d; :}
   ;
disjunction
 ::= disjunction:d '||' conjunction:c {: result := Disjunction(d,c); :}
   |  conjunction:c                   {: result := c; :}
   ;
conjunction
 ::= conjunction:c '&&' boolFactor:f {:result := Conjunction(c,f); :}
   | boolFactor:f                    {: result := f; :}
   ;
boolFactor
 ::= '(' boolExpr:be_par ')' {:  result := be_par; :}
   | '!' boolExpr:e          {: result := Negation(e); :}
   ;
expr
 ::= expr:e '+'   prod:p {: result := Sum(e,p); :}
   |  expr:e '-'  prod:p {: result := Difference(e,p); :}
   |  prod:p             {: result := p;     :}
   ;
prod
 ::= prod:p '*'  fact:f    {: result := Product(p,f); :}
   |  prod:p '\' fact:f    {: result := Quotient(p,f); :}
   |  prod:p '%' fact:f    {: result := Mod(p,f); :}
   |  fact:f               {: result := f;     :}
   ;
fact
 ::= '(' expr:e_par ')'            {: result := e_par;   :}
   |  INTEGER:n                    {: result := Integer(eval(n));   :}
   |  DECIMAL:d                    {: result := Decimal(eval(d)); :}
   |  ZID:id_1 '(' exprList:el ')' {: result := FunctionCall(id_1,el); :}
   |  ZID:id_2                     {: result := Variable(id_2); :}
   ;
```

Figure 1.9: Example interpreter expressions

```
function factorial(n) {
    if (n == 0) {
       return 1;
    }
    return n * factorial(n - 1);
}
print("Calculation of factorial for i = 1 to 9");
for (i = 0; i < 10; i = i + 1) {
    print(i, "! = ", factorial(i));
}
print();
```

Figure 1.10: Example interpreter input

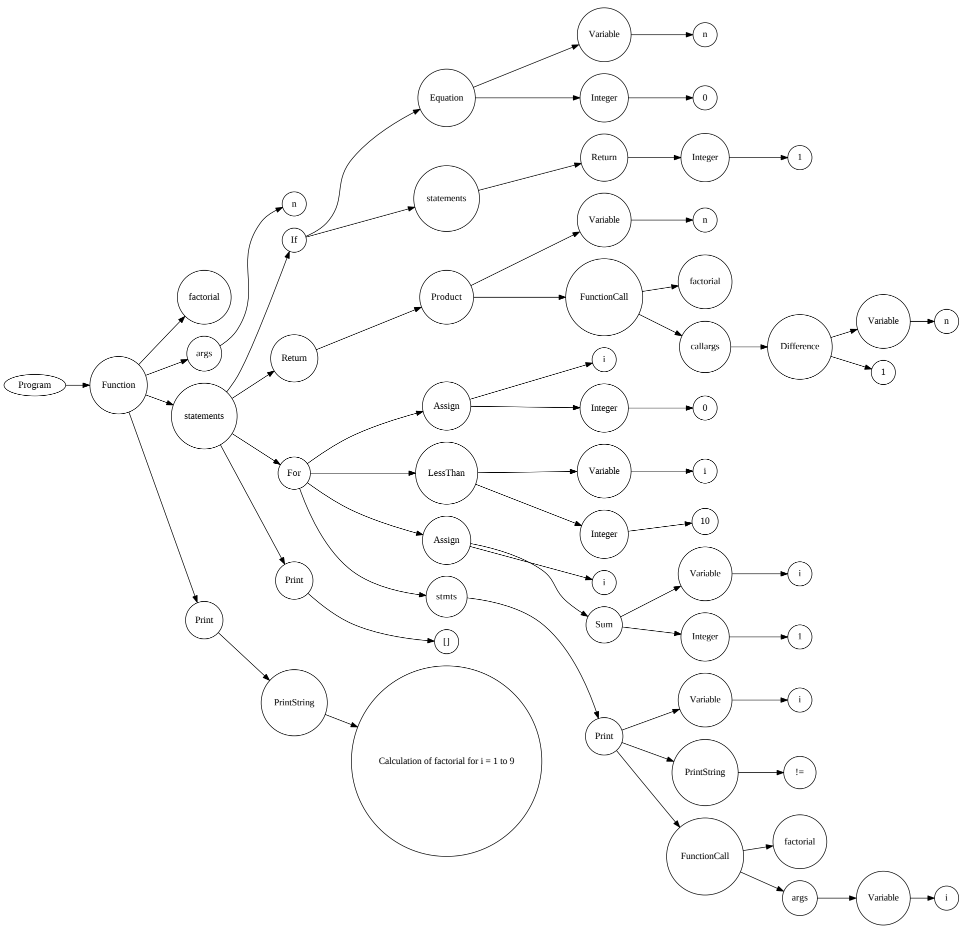

Figure 1.11: Interpreter AST