SetlCup Tutorial

Jonas Eilers

February 18, 2016

Contents

Chapter 1

Funktionalität

SetlCup ist ein LR-Parser-Generator, welcher angelehnt an den bereits vorhandenen JavaCup ist. Dabei ist die Idee wie folgt: Der Benutzer von SetlCup erstellt durch eine gegebene Scanner- und Parserdefinition eine neue Setlx-Datei. Diese beinhaltet einen kanonischen LR-Parser, welcher auf den gegebenen Definitionen beruht. Anschließend kann eine Eingabedatei von dem generierten Parser überprüft werden. Dabei wird ggf. angegebener Code bei der Reduzierung der entsprechenden Regeln durchgeführt.

Zunächst wird erklärt, wie die Komponente aufgerufen werden kann.

Chapter 2

Aufrufsmethoden

Es gibt mehrere Möglichkeiten SetlCup aufzurufen:

2.1 Aufruf über Kommandozeile

1. `setlx setlcup.stlx -p parser_scanner_datei.stlx`

   Mit diesem Aufruf wird ein Parser gemäß der in der Eingabedatei gegebenen Definitionen erstellt.

2. `setlx setlcup.stlx -p parser_scanner_datei.stlx -d`

   Um die Art und Weise, wie der Parser generiert wird Nachzuvollziehen, kann mit der Option "'-d"' das Debugging eingeschaltet werden. Dabei wird empfohlen die Ausgabe in eine Datei umzuleiten.

3. `setlx parser_datei.stlx -p eingabe_datei.txt`

   Der erstellte Parser kann mit dem o.g. Befehl aufgerufen werden und probiert die Eingabedatei nach den angegebenen Regeln zu überprüfen.

4. `setlx parser_datei.stlx -p eingabe_datei.txt -d`

   Analog zur Parsererstellung wird durch die Option "'-d"' das Debugging eingeschaltet.

5. `setlx setlcup.stlx -h`

   Dieser Aufruf zeigt die Hilfe an, wie SetlCup aufgerufen werden kann.

Chapter 3

Aufbau der Definitionen

Der Aufbau der Scanner- und Parserdefinitionen ist in drei Abschnitte zu unterteilen:

1. Kommentare
2. Scannerdefinition
3. Parserdefinition

3.1 Kommentare

Im obersten Bereich der Datei ist es möglich die Idee des Parsers zu beschreiben. Dieser Abschnitt endet mit dem Symbol "‘%%%"’.

3.2 Scannerdefinition

Der Scanner ist verantwortlich um zu Überprüfen, ob die Eingabedatei aus den angegebenen Tokens besteht. Die Syntax(see Figure 3.1, Page 4) wird im Folgenden erklärt.

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

Figure 3.1: Scanner Definition

- In Zeile 1 wird der Token "INTEGER" definiert. Tokens werden auf die folgende Weise deklariert:
  token_name := regex ;
  Dabei ist folgendes bei der Syntax des regulären Ausdrucks zu beachten:

  - Die Rückgabe der Capture-Gruppen z.B. "‘ab**(.*)**ab"’ werden nicht unterstützt. Jedoch sind Nicht-Captures möglich: "‘ab(**?:**.*)ab"’

- 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.

3.3 Parser-Part

In this part the grammar-rules are defined with the a syntax which has many analogies to JavaCup and ANTLR (see Figure 3.2, Page 5).

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

Figure 3.2: 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.

3.4 Example

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

3.4.1 Arithmetic grammar

The arithmetic grammar and scanner (see Figure 3.3, Page 6) 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 3.3: Example arithmetic grammar

arithmetic expressions 3.4. The output AST(see Figure 3.5, Page 7) consists of the three expressions noted above.

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

Figure 3.4: Example arithmetic input

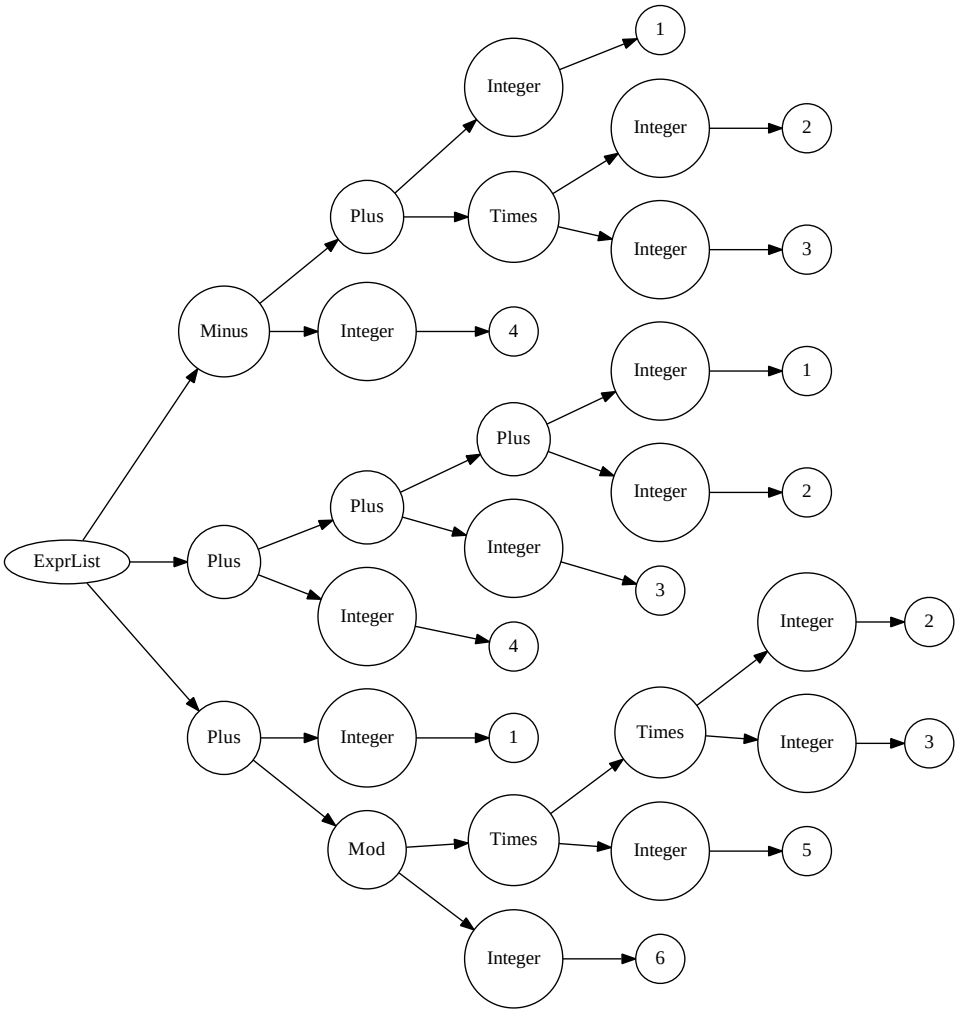

Figure 3.5: Arithexpr AST

3.4.2 Programming language grammar

The programming language grammar and scanner. The Scanner (see Figure 3.6, Page 8) 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 3.7, Page 9) describe the structure of the input file. It consists of

multiple statement and definitions. The Lists (see Figure 3.8, Page 10) describe how multiple arguments, expressions, definitions and statements are chained. The Expressions (see Figure 3.9, Page 11) describe boolean and arithmetic expressions. A sample input program (see Figure 3.10, Page 12) describes

```
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 3.6: Example interpreter scanner

the recursive function factorial. The output AST (see Figure 3.11, Page 12) 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 3.7: 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 3.8: 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 3.9: 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 3.10: Example interpreter input

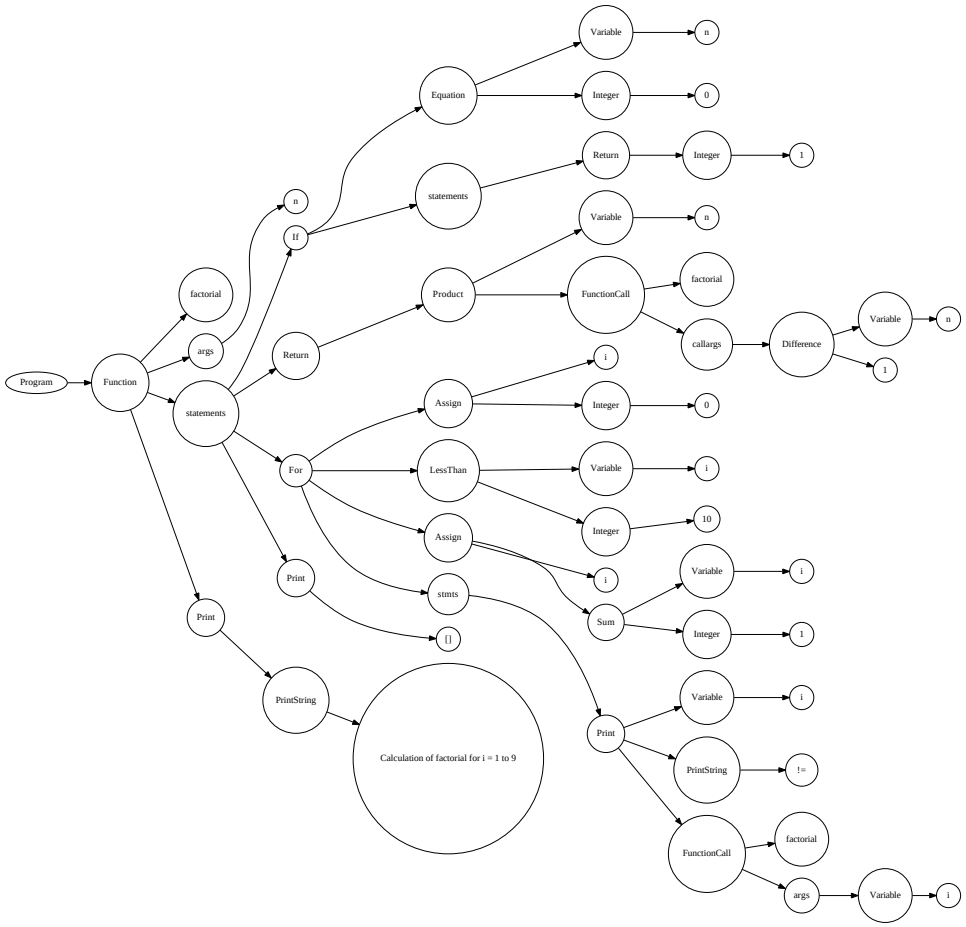

Figure 3.11: Interpreter AST