SetlCup Tutorial

Jonas Eilers

March 4, 2016

Contents

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.

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.

6. `setlx test_setlcup.stlx`

   Dieser Aufruf testet die Funktionen des Parsergenerators mit den verschiedenen Beispielen.

3 Dateistruktur

Der Parser-Generator SetlCup ist in mehrere Dateien aufgeteilt.

setlcup.stlx Diese Datei wird aufgerufen, falls man über die Kommandozeile mit SetlCup arbeiten möchte. Sie nimmt die Parserdefinition als Parameter an und gibt sie an die jeweiligen Programme weiter, damit die Generierung des Parsers gestartet werden kann.

setlcup_load.stlx Diese Datei lässt zunächst den Scanner und daraufhin den Parser generieren. Falls über Setlx mit SetlCup gearbeitet werden soll, ist es möglich diese Datei als Grundlage zu nutzen. Sie wird u.a. für den Test der Funktionalität benutzt (siehe "test_setlcup");

scanner_generator.stlx In dieser Datei wird die Eingabe analysiert. Es wird der Scanner aus den angegebenen Tokens generiert. Dieser wird daraufhin in einer Ausgabedatei abgelegt. Auch der Parser aus der Datei "_sr_parser_part.stlx" wird der Datei angehangen. Zusätzlich wird die Grammatikdefinition analysiert und in einer Token-Liste zurückgegeben.

parser_generator.stlx Der Parser Generator erstellt den kanonischen LR-Parser. Er erzeugt die notwendigen Action-, State- und Gototabellen. Diese werden zusammen mit den Regeln und dem vom User gewünschten Code auch in der o.g. Datei abgespeichert.

_sr_parser_part.stlx Der Shift-Reduce-Parser ist in dieser Datei umgesetzt. Er wird zusammen mit den benötigten Tabellen, sowie dem Scanner in eine neue Datei kopiert.

_Grammar.stlx In dieser Datei wird der erstelle Scanner und Parser abgelegt. Beim Aufruf wird die übergebene Datei geparsed. Dabei werden die o.g. Tabellen, sowie der vorher erstellte Scanner genutzt. Der Parser ist ein Shift-Reduce Parser. Beim Reducen wird der vom User angegebene Code ausgeführt. Der Wert der Variable "result" wird am Ende zurückgegeben.

test_setlcup.stlx Für einen Regressionstest kann mit dieser Datei überprüft werden ob alle Beispiele momentan geparsed werden können. Dabei wird zunächst für den arithmetischen Term der Parser erstellt. Dieser wird anschließend mit einer Eingabedatei aufgerufen und die Ergebnisse mit den vorher festgelegten verglichen. Das gleiche geschieht mit der Parserdefinition für eine einfache Programmiersprache.

4 Aufbau der Definitionen

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

1. Kommentare
2. Scannerdefinition
3. Parserdefinition

4.1 Kommentare

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

4.2 Scannerdefinition

Der Scanner ist verantwortlich um zu Überprüfen, ob die Eingabedatei aus den angegebenen Tokens besteht. Die Syntax(siehe Figur 1, Seite 5) wird im Folgenden erklärt.

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

Figure 1: Scanner Definition

1. 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:
   (a) Die Rückgabe der Capture-Gruppen z.B. "‘ab**(.*)**ab"’ werden nicht unterstützt. Jedoch sind Nicht-Captures möglich: "‘ab(**?:**.*)ab"’
   (b) Die Nutzung der geschweiften Klammer wurde überlagert, sodass die regulären Ausdrücke der großgeschriebene Wörter innerhalb der Klammern im Nachhinein ersetz werden. Siehe Zeile 4 - "‘{TOKEN-NAME}"’.
   (c) Ansonsten sind die bereits in SetlX vorhandenen Regex-Ausdrücke benutzbar.
2. Wie in SetlX (siehe Zeile 2 und 3) müssen auch in SetlCup vordefinierte Symbole wie "$*, +, ?, |, \{, \}, (,), \cdots$" escaped werden .
3. In Zeile 4 wird "SKIP"-Token genutzt. In manchen Fällen werden gewisse Tokens nicht benötigt. Diese können mithilfe des "‘SKIP"’-Tokens ignoriert werden. Dabei ist die Eingabe mit der o.g. Ersetzstrategie ("‘ {TOKENNAME}"’) oder die Nutzung eines regulären Ausdrucks möglich. Verschiedene Tokens müssen mit der Pipe "‘|"’ separiert werden.

4.3 Parserdefinition

Die Definition der Grammatik für den Parser benutzt Konzepte aus JavaCup und ANTLR (siehe Figur 5, Seite 6).
RULE_HEAD beschreibt den Namen einer Regel z.B. "body". Somit können Regeln referenziert werden.

```
1       grammar :=  definition_list;
2 definition_list := rule_definition  definition_list
3                  | ;
4       rule_definition := RULEHEAD '::=' body_list ';' ;
5       body_list := body '|' neBody_list
6                                                  | body
7           | ;
8 neBody_List := body body_list;
9       body := element_list action_code;
10 action_code := '{:' CODE ':}'
11               | ;
12 element_list := element element_list
13             | ;
14 element := token;
15 token := LITERAL id
16        | TOKEN_NAME id
17        | RULE_HEAD id
18        ;
19 id := ':'ID_NAME
20     |;
21 RULE_HEAD := [a-z][a-zA-Z_0-9]*;
22 TOKEN_NAME := [A-Z][A-Z_0-9]*;
23 LITERAL := ''[^'']*'';
24 ID_NAME := [a-z][a-zA-Z_0-9]*;
25 CODE := [.\n]*;
```

5 Beispiel parser

TOKEN_NAME bezieht sich auf einen Token aus der Scanner Definition.

LITERAL beschreibt wortwörtliche Ausdrücke, welche genutzt werden können.

ID_NAME ermöglicht es eine ID für den Rückgabewert einer Regel, bzw. eines Regex zu benutzen.

action_code ist ein optionaler Teil einer Regel. Er kann am Ende einer Regel hinzugefügt werden. Der CODE innerhalb der Klammern wird beim Reduzieren der Regel ausgeführt. Durch die Nutzung der Variable "'result"' können Ergebnisse zwischen den Regeln transferiert werden. Die IDs der Elemente der jeweiligen Regel können im Code benutzt werden. Der Code selber darf keine Anführungszeichen enthalten. Wenn sie doch notwendig sind, wird empfohlen sie zu escapen mit "\\" oder den Literalstring z.B. 'a+b = 15' zu nutzen. Außerdem sollte **generell** auf die Nutzung von Dollarzeichen verzichtet werden. Diese führen zu Komplikationen bei der Serialisierung des Parsers, da sie in SetlX eine Sonderfunktion einnehmen.

5.1 Beispiele

Das erste Beispiel beschreibt eine einfache arithmetische Grammatik. Das zweite Beispiel beschreibt eine simple Programmiersprache.

5.1.1 Arithmetische Ausdrücke

Der arithmetische Parser (siehe Figur 2, Seite 7) kann mit der oben beschriebenen Syntax definiert werden. Eine beispielhafte Eingabe besteht aus drei ver-

```
1  %%%
2
3  INTEGER       := 0|[1-9][0-9]* ;
4  WHITESPACE    := [ \t\v\r\s] ;
5
6
7  SKIP          := {WHITESPACE} | \n ;
8
9  %%%
10 arith_expr
11  ::= expr_list:esl               {: result := ExprList(esl); :};
12
13 expr_list
14  ::= expr_part:part expr_list:l {: result := [part] + l; :}
15    |                            {: result := []; :}
16    ;
17 expr_part
18  ::= expr:e ';'            {: result := e; :} ;
19 expr
20  ::=  expr:e '+'  prod:p   {: result := Plus(e , p);  :}
21    |  expr:e '-'  prod:p   {: result := Minus(e , p); :}
22    |  prod:p               {: result := p;     :}
23    ;
24 prod
25  ::=  prod:p '*'  fact:f   {: result := Times(p , f); :}
26    |  prod:p DIVIDE fact:f {: result := Div(p , f); :}
27    |  prod:p '%'    fact:f {: result := Mod(p , f); :}
28    |  fact:f               {: result := f;     :}
29    ;
30 fact
31  ::=  '(' expr:e_part ')'   {: result :=  e_part ;   :}
32    |  INTEGER:n             {: result := Integer(eval(n)); :}
33    ;
```

Figure 2: Parserdefinition für arithmetische Ausdrücke

schiedenen mathematischen Termen 3. Die Ausgabe des AST(siehe Figur 4, Seite 8) beschreibt die gezeigte Eingabe.

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

Figure 3: Beispiel für arithmetische Terme

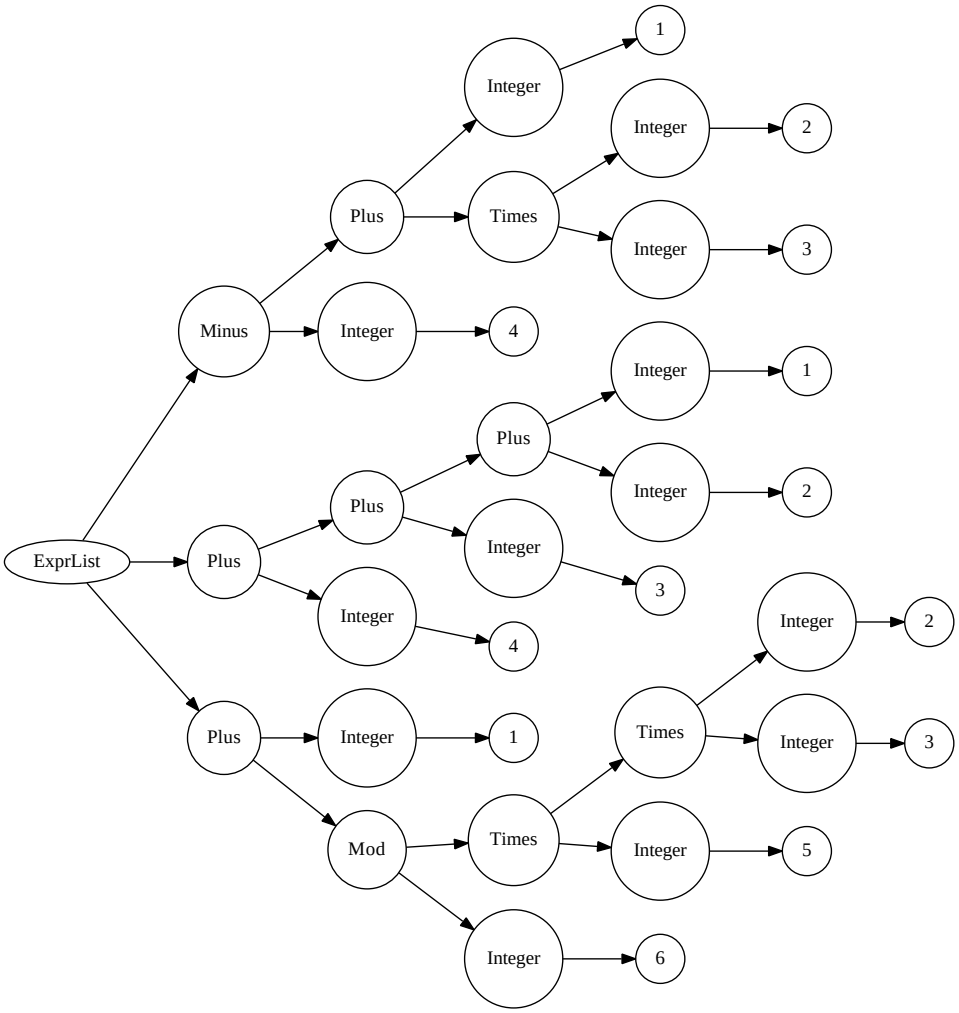

Figure 4: Arithexpr AST

5.1.2 Programmiersprachen Parser

Der Scanner (siehe Figur 5, Seite 9) besteht aus den Tokens für Strings, Dezimalzahlen, Ganzzahlen usw. Die Anwwisungen und Definitionen (siehe Figur 6, Seite 10) beschreiben den Aufbau der Eingabedatei. Sie besteht aus Definitionen und Anweisungen. Listen (siehe Figur 7, Seite 11) sind zur Aneinanderreihung

verschiedener Anweisungen, Definitionen, Ausdrücke etc. notwendig. Die Ausdrücke sind für boolsche, sowie arithmetische Terme notwendig (siehe Figur 8, Seite 12). Ein Beispiel Programm (siehe Figur 9, Seite 13) zum Berechnen

```
1 %%%
2 STRING       := \"(?:\\.|[^\"])*\" ;
3 WHITESPACE   := [ \t\v\r\s] ;
4 INTEGER      := 0|[1-9][0-9]* ;
5 DECIMAL      := 0\.[0-9]+|[1-9][0-9]*\.[0-9]+ ;
6 ZID          := [a-zA-Z_][a-zA-Z0-9_]* ;
7
8 SKIP         := {WHITESPACE}|\n|//[^\n]* ;
9 %%%
```

Figure 5: Scannerdefinition für Programmiersprache

der Fakultät wird durch den Syntaxbaum (siehe Figur 10, Seite 13) abgebildet.

```
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 6: Statements für Progammiersprachenparser

```
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 7: Listen für Progammiersprachenparser

```
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 8: Ausdrücke für Progammiersprachenparser

```
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 9: Example interpreter input

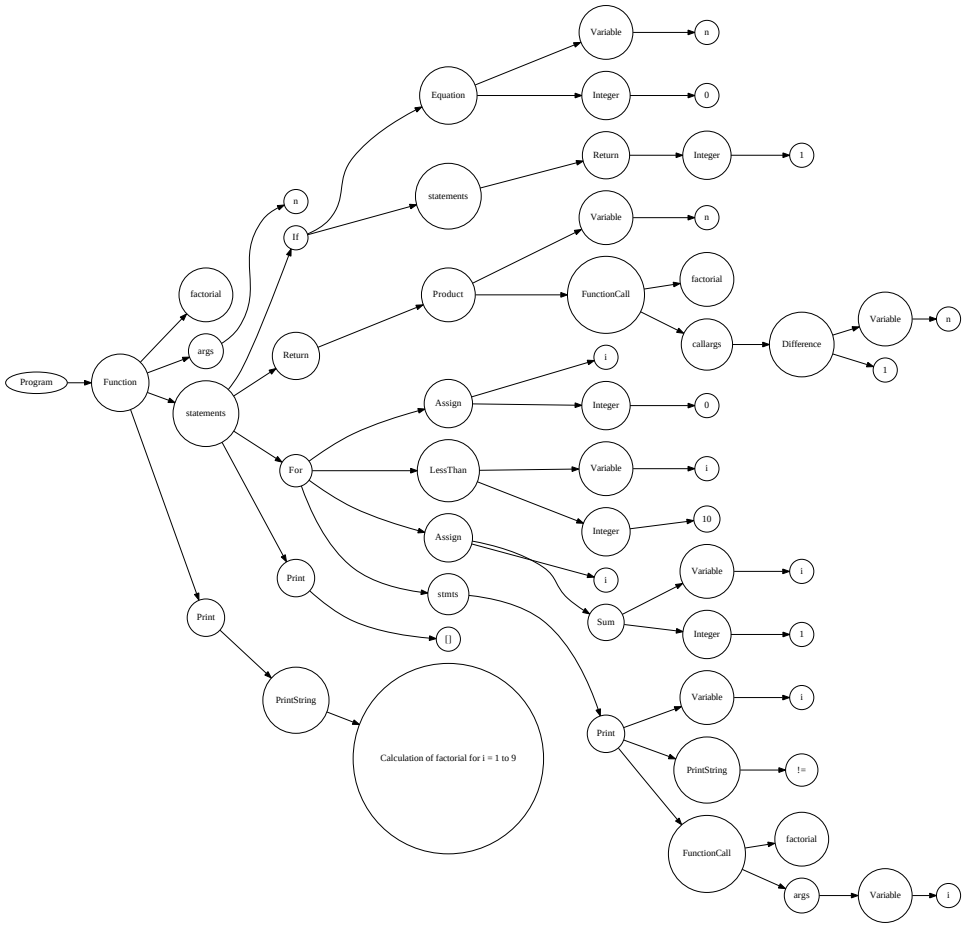

Figure 10: Interpreter AST