

# SetlCup Tutorial

Jonas Eilers

February 26, 2016

# Contents

<b>1</b>	<b>Funktionalität</b>	<b>2</b>
<b>2</b>	<b>Aufrufsmethoden</b>	<b>3</b>
2.1	Aufruf über Kommandozeile . . . . .	3
<b>3</b>	<b>Aufbau der Definitionen</b>	<b>4</b>
3.1	Kommentare . . . . .	4
3.2	Scannerdefinition . . . . .	4
3.3	Parserdefinition . . . . .	5
3.4	Beispiele . . . . .	6
3.4.1	Arithmetische Ausdrücke . . . . .	6
3.4.2	Programmiersprachen Parser . . . . .	6

# 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(siehe Figur 3.1, Seite 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"
  - Die Nutzung der geschweiften Klammer wurde überlagert, sodass die regulären Ausdrücke der großgeschriebene Wörter innerhalb der Klammern im Nachhinein ersetzt werden. Siehe Zeile 4 - "{TOKEN-NAME}"
  - Ansonsten sind die bereits in SetlX vorhandenen Regex-Ausdrücke benutzbar.
- Wie in SetlX (siehe Zeile 2 und 3) müssen auch in SetlCup vordefinierte Symbole wie "\*", "+", "?", "|", "{", "}", "(", ")", "... " escaped werden .
  - 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 ( "{TOKEN-NAME}" ) oder die Nutzung eines regulären Ausdrucks möglich. Verschiedene Tokens müssen mit der Pipe "|" separiert werden.

### 3.3 Parserdefinition

Die Definition der Grammatik für den Parser benutzt Konzepte aus JavaCup und ANTLR (siehe Figur 3.2, Seite 6).

**RULE\_HEAD** **RULE\_HEAD** beschreibt den Namen einer Regel z.B. "body". Somit können Regeln referenziert werden.

**TOKEN\_NAME** **TOKEN\_NAME** bezieht sich auf einen Token aus der Scanner Definition.

**LITERAL** **LITERAL** beschreibt wortwörtliche Ausdrücke, welche genutzt werden können.

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

**action\_code** Der **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.

---

```

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']*;
```

---

Figure 3.2: Beispiel parser

## 3.4 Beispiele

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

### 3.4.1 Arithmetische Ausdrücke

Der arithmetische Parser (siehe Figur 3.3, Seite 7) kann mit der oben beschriebenen Syntax definiert werden. Eine beispielhafte Eingabe besteht aus drei verschiedenen mathematischen Termen 3.4. Die Ausgabe des AST(siehe Figur 3.5, Seite 8) beschreibt die gezeigte Eingabe.

### 3.4.2 Programmiersprachen Parser

Der Scanner (siehe Figur 3.6, Seite 9) besteht aus den Tokens für Strings, Dezimalzahlen, Ganzzahlen usw. Die Anweisungen und Definitionen (siehe Figur 3.7, Seite 10) beschreiben den Aufbau der Eingabedatei. Sie besteht aus Definitionen und Anweisungen. describe the structure of the input file. It consists of multiple statement and definitions. Listen (siehe Figur 3.8, Seite 11) sind zur

---

```

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 3.3: Parserdefinition für arithmetische Ausdrücke

---

```

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

```

---

Figure 3.4: Beispiel für arithmetische Terme

Aneinanderreihung verschiedener Anweisungen, Deditionen, Ausdrücke etc. notwendig. describe how multiple arguments, expressions, definitions and statements are chained. Die Ausdrücke sind für boolesche, sowie arithmetische Terme



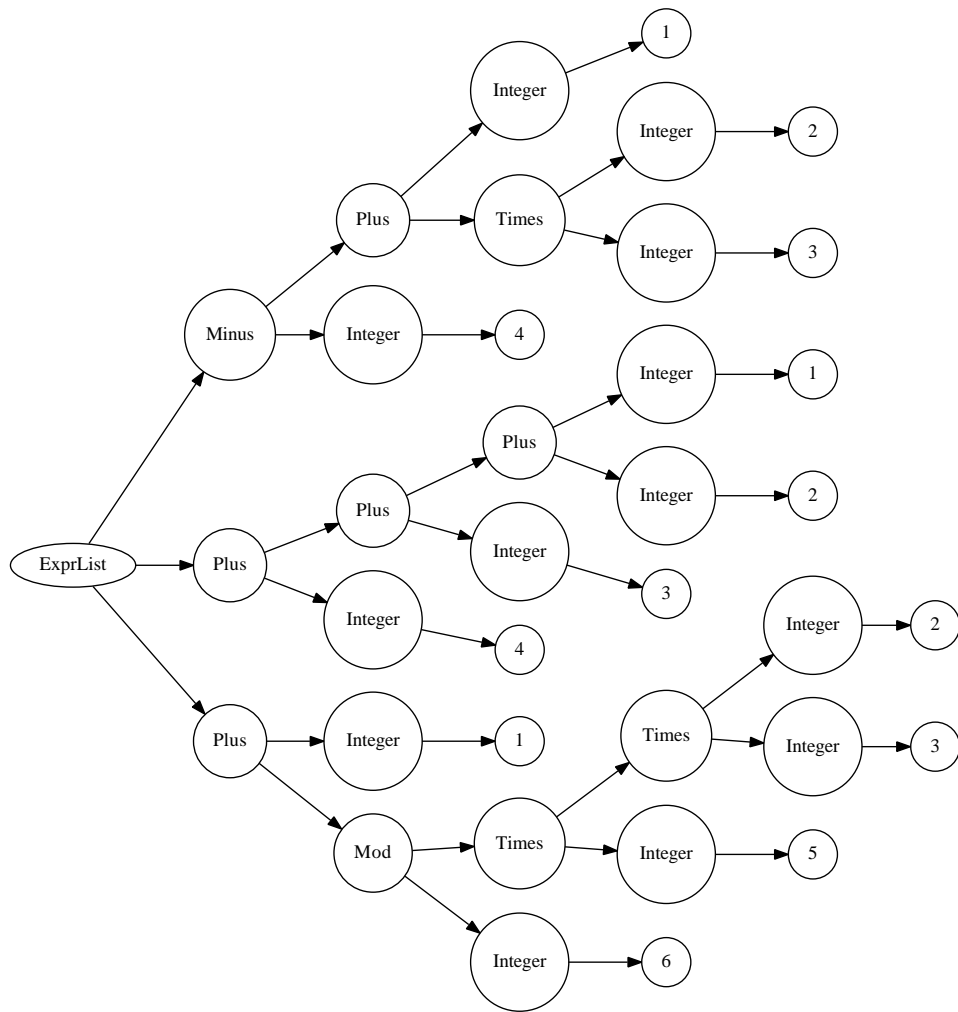


Figure 3.5: Arithexpr AST

notwendig (siehe Figur 3.9, Seite 12) describe boolean and arithmetic expressions. Ein Beispiel Programm (siehe Figur 3.10, Seite 13) zum Berechnen der Fakultät wird durch den Syntaxbaum (siehe Figur 3.11, Seite 13) abgebildet.

---

```
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 3.6: Scannerdefinition für Programmiersprache

---

```

1 program
2   ::= dfnStmntList:d {: result := Program(d); :}
3   ;
4
5 dfnStmntList
6   ::= definition:d dfnStmntList:d1      {: result := [d] + d1; :}
7   | statement:stmts dfnStmntList:dsl  {: result := [stmts] + dsl; :}
8   |                                     {: result := []; :}
9   ;
10
11 definition
12   ::= 'function' ZID:function_name '(' paramList:param_list ')'
13   '{' stmntList:statement_list '}'
14   {: result := Function(function_name, param_list, statement_list);:}
15   ;
16 stmntList
17   ::= statement:s stmntList:sl {: result := [s] + sl ; :}
18   | {: result := []; :}
19   ;
20 statement
21   ::= assignment:a ';' {: result := Assign(a); :}
22   | 'print' '(' printExprList:printexpr_list ')' ';'
23   {: result := Print(printexpr_list); :}
24   | 'if' '(' boolExpr:b ')' '{' stmntList:st_list1 '}'
25   {: result := If(b, st_list1); :}
26   | 'while' '(' boolExpr:b ')' '{' stmntList:st_list2 '}'
27   {: result := While(b, st_list2); :}
28   | 'for' '(' assignment:i_a ';' boolExpr:b ';' assignment:e_a ')'
29   '{' stmntList:st_list3 '}'
30   {: result := For(i_a, b, e_a, st_list3); :}
31   | 'return' expr:e ';' {: result := Return(e); :}
32   | 'return' ';'       {: result := Return(); :}
33   | expr:e ';'        {: result := Expr(e); :}
34   | 'quit' ';'        {: result := Exit(); :}
35   ;

```

---

Figure 3.7: Statements für Programmiersprachenparser

---

```

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 3.8: Listen für Programmiersprachenparser

---

```

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 3.9: Ausdrücke für Programmiersprachenparser

---

```

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 3.10: Example interpreter input

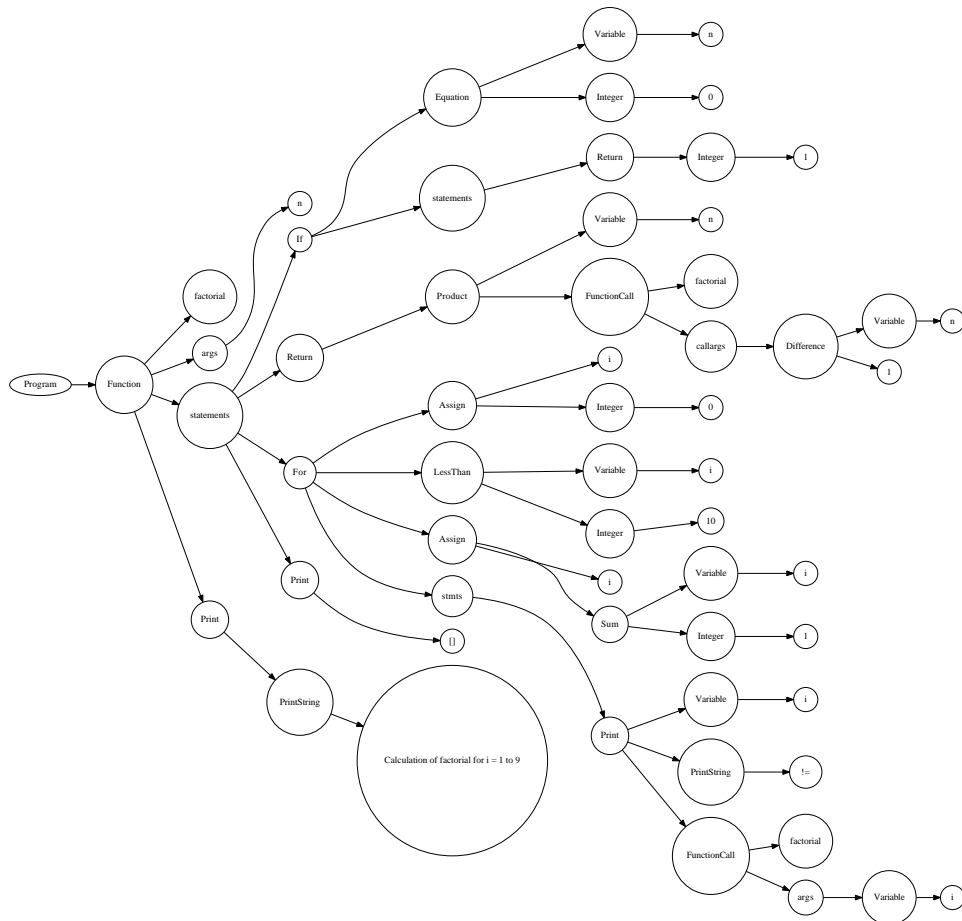


Figure 3.11: Interpreter AST