SetlCup Tutorial

Jonas Eilers

21. Mai 2016

Inhaltsverzeichnis

1 Überblick

1.1 Motivation

Zur Anwendung von formalen Sprachen in der Informatik gehören unter anderem Scanner und Parser[ASU86]. Programmgeneratoren, welche z.B. Parser generieren existieren bereits für Sprachen wie Java[Hud16], C[DS15] usw. In der Vorlesung „Formal Languages and their Applications"[Str16c] werden momentan verschiedene Tools genutzt um Parser zu generieren. Dazu gehören u.a. JavaCup zum Erstellen von LALR-Parsern, sowie ANTLR4[Par12] für die Erstellung eines Top-Down-Parsers mit Nutzung von EBNF-Grammatiken. Diese Programme ermöglichen es beim Parsen eines Programmes Aktionen durchzuführen. Diese Aktionen werden in der Programmiersprache Java definiert. Ziel dieser Studienarbeit ist es eine Alternative zur Nutzung von Java(in der Parser-Generierung) zu ermöglichen. Dies soll geschehen, indem ein Parser-Generator in der Programmiersprache SetlX geschrieben wird. SetlX ist eine mengenbasierte Programmiersprache. Sie kann mit dem SetlX-Interpreter genutzt werden. Dieser ist unter der folgenden Adresse verfügbar:
http://randoom.org/Software/SetlX
Diese Programmiersprache wird in den Vorlesungen der theoretischen Informatik 1 und 2(Logik[Str16b], Algorithms[Str16a]) bereits ausgiebig genutzt. Eine Ergänzung für die Vorlesung über die formalen Sprachen kann somit den Studenten helfen die Konzepte des Parsens zu verstehen, ohne in eine andere Programmiersprache „umzudenken". Da der Parser-Generator sehr stark an JavaCup orientiert ist, entstand der Name SetlCup. Ein großer Unterschied ist jedoch, dass SetlCup ein **LR**-Parser-Generator ist. Ein **LALR**-Parser-Generator kann weniger Grammatiken verarbeiten als ein **LR**-Parser. Zwar hat der **LALR**-Parser den Vorteil, dass die Parse-Tabellen kleiner sind als bei **LR**-Parsern, jedoch fällt dies bei den Heute verfügbaren Rechnern aufgrund der Größe moderner Hauptspeicher kaum noch ins Gewicht.

2 Funktionalität

2.1 Benutzung

Der Benutzer erstellt eine Grammatik, die mit Aktionen attributiert ist. Daraus erstellt SetlCup dann eine SetlX-Datei, die Parser und Scanner erhält. 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.

3 Aufrufsmethoden

Es gibt mehrere Möglichkeiten SetlCup aufzurufen:

3.1 Aufruf über Kommandozeile

1. ```
   setlx setlcup.stlx -p parser_scanner_datei
   setlx setlcup.stlx -p examples/math_simple_expression.g
   ```

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

2. ```
   setlx setlcup.stlx -p parser_scanner_datei -d
   setlx setlcup.stlx -p examples/math_simple_expression.g -d
   ```

   Um die Art und Weise, wie der Parser generiert wird nachvollziehen zu können, kann mit der Option „-d“ das Debugging eingeschaltet werden. Dabei wird empfohlen die Ausgabe in eine Datei umzuleiten.

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

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

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

   Analog zur Parsererstellung wird durch die Option „-d“ das Debugging eingeschaltet.

5. ```
   setlx setlcup.stlx -p -help
   ```

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

6. ```
   setlx test_setlcup.stlx
   ```

   Dieser Aufruf testet die Funktionen des Parsergenerators mit verschiedenen Beispielen. Dabei sollte das folgende Ergebnis erzeugt werden:

   ```
   Testing math_expression_grammar..
   parser generated succefully for math_expression_grammar_ast.g
   grammar: math_expression_grammar_ast.g input: math_expression_input.txt
   successfull: yes
   Testing interpreter_grammar..
   parser generated succefully for interpreter_grammar_ast.g
   grammar: interpreter_grammar_ast.g input: factorial.sl successfull: yes
   grammar: interpreter_grammar_ast.g input: solve.sl successfull: yes
   grammar: interpreter_grammar_ast.g input: sum.sl successfull: yes
   grammar: interpreter_grammar_ast.g input: sum-for.sl successfull: yes
   ```

3.2 Aufruf in SetlX

Ein Aufruf von SetlCup ist auch in SetlX selbst möglich. Dazu ist es notwendig die Funktionalität im eigenen Quellcode zu laden:

```
load("setlcup_load.stlx");
generate_parser('examples\math_expression_grammar_ast.g', true);
load('examples\math_expression_grammar_astGrammar.stlx');
result := test_parser_from_file('examples\math_expression_input.txt', true);
```

4 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 Definition der Grammatik 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 erstellte 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 für mehrere Grammatiken der Parser erstellt. Diese werden anschließend mit einer Eingabedatei aufgerufen und die Ergebnisse mit den vorher festgelegten verglichen.

5 Aufbau der Definitionen

Der Aufbau der Scanner- und Parserdefinitionen ist in drei, durch “%%%“ getrennte, Abschnitte zu unterteilen:

1. Kommentare

2. Scannerdefinition

3. Parserdefinition

5.1 Kommentare

Die Kommentar-sektion wird 1:1 in den späteren Parser reinkopiert. Das heißt, dass Kommentare über die Grammatik mit „//“ auskommentiert werden müssen. Jedoch bietet dies zusätzlich die Möglichkeit Funktionsdefinitionen abzuspeichern, welche im Action-Code genutzt werden können. Ein Beispiel folgt im Abschnitt 6.3. Dieser Abschnitt endet mit dem Symbol „%%%“.

5.2 Scannerdefinition

Die Aufgabe des Scanners ist es, die Eingabe in eine Liste von Tokens zu zerlegen. Die Syntax(siehe Figur 1, Seite 6) wird im Folgenden erklärt.

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

Abbildung 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 von Capture-Gruppen[1] z.B. „ab**(.*)**ab**(.*)**ab**(.*)**“ wird konkateniert zurückgegeben (es werden alle einzelnen Unterergebnisse aneinander gehangen). Ein Beispiel zeigt die Rückgabe des o.g. regulären Ausdrucks:

   ```
   => test_regex := 'ab(.*)ab(.*)ab(.*)';
   input := 'abBeiabSpielabText';
   output := matches(input, test_regex, true);
   ~< Result: ["abBeiabSpielabText", "Bei", "Spiel", "Text"] >~
   => scup_output := join(output[2..#output], "");
   ~< Result: "BeiSpielText" >~
   ```

   (b) Die Nutzung der geschweiften Klammer wurde überlagert, sodass die regulären Ausdrücke der großgeschriebene Wörter innerhalb der Klammern im Nachhinein ersetzt werden. Siehe Zeile 4 - „{WHITESPACE}“.

   (c) Ansonsten sind die bereits in SetlX bzw. Java vorhandenen Regex-Ausdrücke benutzbar.

[1] Gruppierungen in regulären Ausdrücken - https://de.wikipedia.org/wiki/Regulärer_Ausdruck#Gruppierungen_und_R.C3.BCckw.C3.A4rtsreferenzen

2. Vordefinierte Symbole für reguläre Ausdrücke, also:
”∗”, ”+”, ”?”, ”, ”{”, ”}”, ”(”, ”)”, ”.”, ”[”, ”]”, ”\”
müssen escaped werden, wenn diese wörtlich gematcht werden sollen (siehe Zeile 2 und 3).

3. In Zeile 4 wird das „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.

5.3 Parserdefinition

Die Definition der Grammatik für den Parser benutzt Konzepte aus JavaCup und ANTLR (siehe Figur 2, Seite 7).

```
1  grammar         ::=  definition_list;
2  definition_list ::=  rule_definition  definition_list
3                        |
4                  ;
5  rule_definition ::=  VARIABLE ’::=’ body_list ’;’ ;
6  body_list       ::=  body ’|’ neBody_list
7                        | body
8                        |
9                  ;
10 neBody_List     ::=  body ’|’ nebody_list
11                 | body
12                 ;
13 body            ::=  element_list action_code;
14 action_code     ::=  ’{:’ CODE ’:}’
15                       |
16                 ;
17 element_list    ::=  element element_list
18                       |
19                 ;
20 element         ::=  token;
21 token           ::=  LITERAL id
22                       | TOKEN_NAME id
23                       | VARIABLE id ;
24 id              ::=  ’:’VARIABLE
25                       |
26                 ;
27 VARIABLE        ::=  [a-z][a-zA-Z_0-9]*;
28 TOKEN_NAME      ::=  [A-Z][A-Z_0-9]*;
29 LITERAL         ::=  ’’[^’’]*’’;
30 CODE            ::=  ^(?!(.|\n)*:\})(.|\n)*;
```

Abbildung 2: Grammatik zum Aufbau der Parserdefinition

VARIABLE beschreibt den Namen einer Regel z.B. „body“. Somit können Regeln referenziert werden. Außerdem haben auch die IDs einer Variable den gleichen Aufbau. Dies ermöglicht es den Rückgabewert einer Regel bzw. eines regulären Ausdrucks zu nutzen.

TOKEN_NAME bezieht sich auf einen Token aus der Scanner Definition.

LITERAL beschreibt wortwörtliche Ausdrücke, welche genutzt werden können (z.B. '+', '.', '*', 'print').

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. Eine Alternative bietet der Literalstring z.B. 'a+b = 15'. 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.

6 Beispiele

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

6.1 Arithmetische Ausdrücke

Der arithmetische Parser (siehe Figur 3, Seite 9) kann mit der oben beschriebenen Syntax definiert werden. Die Generierung des Parsers kann nun über

```
1  %%%
2  INTEGER       := 0|[1-9][0-9]* ;
3  WHITESPACE    := [ \t\v\r\s] ;
4  SKIP          := {WHITESPACE} | \n ;
5  %%%
6  term
7   ::=  expr:e               {: print(e);          :}
8      ;
9  expr
10  ::=  expr:e '+'  prod:p   {: result := e+p;     :}
11    |  prod:p               {: result := p;       :}
12    ;
13 prod
14  ::=  prod:p '*'  fact:f   {: result := p*f;     :}
15    |  fact:f               {: result := f;       :}
16    ;
17 fact
18  ::=  INTEGER:n            {: result := eval(n); :}
19    ;
```

Abbildung 3: Parserdefinition für einen simplen arithmetischen Ausdruck

```
setlx setlcup.stlx -p examples\math_simple_expression.g
```

gestartet werden. Mit einem Treiber Programm(siehe Figur 4, Seite 10) können eingegebene Ausdrücke geparsed werden. Der Treiber kann mit dem folgenden Kommando aufgerufen werden:

```
setlx math_simple_expression_driver.stlx
```

Beispielausgabe:

```
Give an arithmetic Expression: (q for end)2 + 3 * 4 * 5 * 6 * 7 + 8
2530
Give an arithmetic Expression: (q for end)7 * 8 + 3 * 4 + 5 + 6 + 9 * 2
97
```

```
load("math_simple_expressionGrammar.stlx");
query := "Give an arithmetic Expression: (q for end)";
while(true){
  arith_expr :=  read(query);
  if(!("q" in arith_expr)){
    test_parser_from_string(arith_expr, true);
  }
  else{
    break;
   }
}
```

Abbildung 4: Beispiel für Treiber

6.2 Programmiersprachen Parser

Eine Erweiterung der arithmethischen Ausdrücken um Ausgabe-, sowie Zuweisungsfunktionen spiegelt diese simple Programmiersprache wieder. Mit dieser simplen Programmiersprache ist es möglich einer Variablen einen Term zuzuweisen („Assignment“). Auch eine Variable selbst kann Teil des Terms sein. Außerdem können Terme mit dem „print“-Befehl ausgegeben werden. Der Parser gibt für ein Programm, welches aus den o.g. Elementen besteht, den AST[2] zurück. Der Scanner (siehe Figur 5, Seite 10) besteht aus den Tokens für Dezimalzahlen, Variablen, Kommentaren und Leerzeichen o.ä. Die Anweisungen und

```
WHITESPACE  := [ \t\v\r\s] ;
INTEGER     := 0|[1-9][0-9]* ;
ZID         := [a-zA-Z_][a-zA-Z0-9_]* ;
SKIP        := {WHITESPACE}|\n|//[^\n]* ;
```

Abbildung 5: Scannerdefinition für eine sehr einfache Programmiersprache

Definitionen (siehe Figur 6, Seite 11) beschreiben den Aufbau der Eingabedatei. Sie besteht aus Zuweisungen oder Ausgaben. Diese können wieder aus arithmetischen Ausdrücken bestehen. Ein Beispiel Programm (siehe Figur 7, Seite 11) wird durch den Syntaxbaum (siehe Figur 8, Seite 12) abgebildet. Die Ausgabe des Programms könnte dabei für die Erstellung eines Interpreters genutzt werden.

[2] AST - abstract syntax tree, siehe `https://de.wikipedia.org/wiki/Abstrakter_Syntaxbaum`

```
program
  ::= stmntList:d              {: result := Program(d); :}
  ;
stmntList
  ::= statement:s stmntList:sl {: result := [s] + sl ; :}
   |                           {: result := []; :}
   ;
statement
 ::= ZID:id '=' expr:e ';'     {: result := Assignment(id, e); :}
   | 'print' '(' expr:e ')' ';'{: result := PrintExpr(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); :}
    | fact:f                   {: result := f; :}
    ;
fact
  ::= '(' expr:e ')'           {: result := e; :}
    |  INTEGER:n               {: result := Integer(eval(n)); :}
    |  ZID:id_1                {: result := Variable(id_1); :}
    ;
```

Abbildung 6: Grammatik für eine sehr einfache Programmiersprache

```
a := 5;
b := 7;
c := b*a +a/b*(7+6);
print(c/2);
```

Abbildung 7: Beispiel-Eingabe

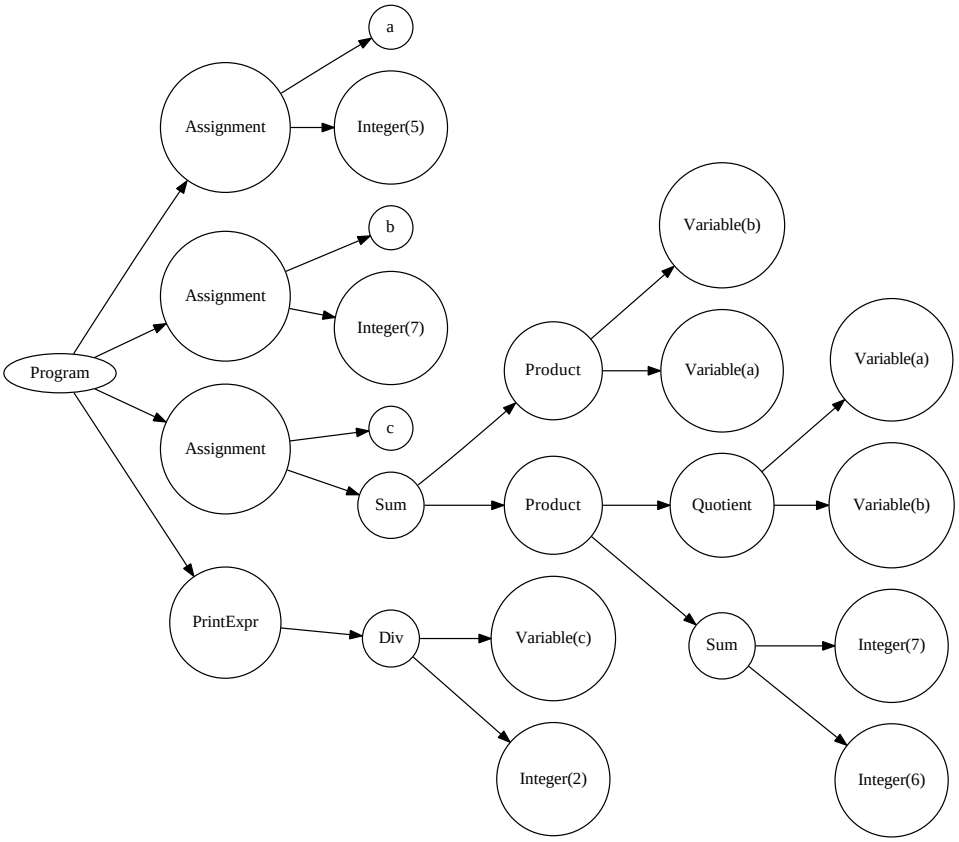

Abbildung 8: AST der Beispieleingabe

6.3 Nutzung von Definitionen im Kommentarbereich

Eine Ergänzung ist durch die Nutzung der Kommentarsektion(siehe Figur 9, Seite 14) möglich. Zu finden unter:

```
examples/interpreter_with_comment_action.g
```

Dabei wird die einfache Interpreter Grammatik(siehe Figur 6, Seite 11) ein wenig verändert. Die neue Grammatik(siehe Figur 10, Seite 15) wird nämlich um ein neues Startsymbol („file“) ergänzt(Zeile 1 bis 3). Von diesem wird als Rückgabewert die in der Kommentarsektion definierte Prozedur „parseElement“ aufgerufen. Die Übergabeparameter sind das Programm „p“, sowie eine leere Menge, welche in der Funktion als Dictionary benutzt wird, übergeben. Der Rest der Grammatik bleibt identisch zu dem Original(siehe Figur 6, Seite 11). Wenn für diese angepasste Definition ein Parser erzeugt wird, kann dieser die Beispieleingabe(siehe Figur 7, Seite 11) interpretieren.

```
setlx setlcup.stlx -p examples/interpreter_with_comment_action.g
cd examples
setlx interpreter_with_comment_actionGrammar.stlx -p simple_interpreter_statements.txt
```

Dabei wird der korrekte Rückgabewert geliefert:

```
155/7
```

```
parseElement := closure(element, rw values){
  match(element){
    case Program(d):
      parseElement(d, values);
    case [h|t]:
      parseElement(h, values);
      parseElement(t, values);
    case Assignment(id, e_1):
      values["$id$"] := parseElement(e_1,values);
    case PrintExpr(e_2):
      print(parseElement(e_2, values));
    case Sum(e,p):
      return parseElement(e, values) + parseElement(p, values);
    case Difference(e,p):
      return parseElement(e, values) - parseElement(p, values);
    case Product(p,f):
      return parseElement(p, values) * parseElement(f, values);
    case Quotient(p,f):
      return parseElement(p, values) / parseElement(f, values);
    case Integer(n):
      return n;
    case Variable(id_1):
      return values[id_1];
  }
};
```

Abbildung 9: Beispiel-Kommentar

```
file
  ::= program:p          {: parseElement(p, {});  :}
  ;
program
  ::= stmntList:d        {: result := Program(d); :}
  ;
stmntList
  ::= statement:s stmntList:sl {: result := [s] + sl ; :}
   |                           {: result := []; :}
   ;
statement
 ::= ZID:id '=' expr:e ';'     {: result := Assignment(id, e); :}
   | 'print' '(' expr:e ')' ';'{: result := PrintExpr(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); :}
    | fact:f                   {: result := f; :}
    ;
fact
  ::= '(' expr:e ')'           {: result := e; :}
    |  INTEGER:n               {: result := Integer(eval(n)); :}
    |  ZID:id_1                {: result := Variable(id_1); :}
    ;
```

Abbildung 10: Abgeänderte Grammatikzeilen für Interpreter

Literatur

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[DS15] Charles Donnelly and Richard Stallman. *Bison 3.04: the Yacc-compatible parser generator*, 2015.

[Hud16] Scott E. Hudson. CUP - LALR parser generator for *Java*, 2016. Available at `http://www2.cs.tum.edu/projects/cup/`.

[Par12] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2012.

[Str16a] Karl Stroetmann. Algorithms, 2016. Available at `https://github.com/karlstroetmann/Algorithms/`.

[Str16b] Karl Stroetmann. Computer science `I`: Logic and set theory, 2016. Available at `https://github.com/karlstroetmann/Logik/`.

[Str16c] Karl Stroetmann. Formal languages and their applications, 2016. Available at `https://github.com/karlstroetmann/Formal-Languages/`.