Übung zur Vorlesung Compiler 1: Grundlagen

Prof. Dr. Andreas Koch
Jens Huthmann, Julian Oppermann

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 13/14
Aufgabenblatt 1 - Lösungsvorschlag

Abgabemodalitäten

Gruppenarbeit ist erlaubt und erwünscht. Bitte geben Sie dann nur eine Lösung pro Gruppe ab. Ihre Lösungen reichen Sie als PDF per E-Mail bis zum 22.11.2013 um 23:59 MET an der Adresse oc@esa.informatik.tu-darmstadt.de ein. Die E-Mail muss als Betreff "Compiler 1 Aufgabe 1" haben.

Aufgabe 1.1 Singlepass / Multipass Compiler

Beschreiben Sie den Unterschied zwischen Single-Pass- und Multi-Pass-Compilern. Geben Sie hierzu die jeweiligen Vor- und Nachteile der Techniken an.

Aufgabe 1.1 Lösung

Ein Pass ist ein kompletter Durchgang des Programms, das heißt, der Quelltext oder eine Zwischendarstellung wird vom Compiler eingelesen und verarbeitet. Single-Pass-Compiler führen alle Phasen der Übersetzung (Syntaxprüfung, Semantikprüfung und Codeerzeugung) in einem Pass durch. Beispielsweise wird direkt nach dem Lesen einer Anweisung der entsprechende Maschinencode erzeugt.

Multi-Pass-Compiler hingegenen nutzen Zwischendarstellungen, um das Programm von einer Phase an die nächste weiterzugeben.

Single-Pass-Compiler sind schneller und können für große Programme weniger Speicher verbrauchen, weil das Programm nur einmal gelesen wird und es keine Zwischendarstellungen gibt. In Multi-Pass-Compilern sind die einzelnen Phasen üblicherweise gekapselt, so dass der Compiler leichter wart- und erweiterbar ist. Für kleine Programme kann auch der Speicherverbrauch geringer sein, weil jeweils nur die Daten einer Phase statt aller Phasen im Speicher gehalten werden müssen. Ein großer Vorteil von Multi-Pass-Compilern ist zudem, dass die globale Optimierungen unterstützen, die zunächst eine komplette Prozedur analysieren und dann erst entscheiden, welche Transformationen angewendet werden sollen.

Bestimmte Eingabesprachen lassen sich ausschließlich mit Multi-Pass-Compilern übersetzen. Ein prominentes Beispiel ist Java, da dort Deklarationen textuell nicht vor Verwendungen von Bezeichnern erfolgen müssen.

Aufgabe 1.2 Triangle zu AST

Zeichnen Sie für das folgende Triangle Programm den abstrakten Syntaxbaum. Halten Sie sich dabei an den Regeln auf auf Seite 109 und 110 im Buch[1]. Zum Zeichnen können Sie zum Beispiel Visio verwenden, welches Sie im MSDNAA bekommen können.

```
if (x < 40) then
        x := x * 2
else
        x := x + 1
```

Aufgabe 1.2 Lösung

Siehe Abbildung 1.

[1] Programming Language Processors in Java

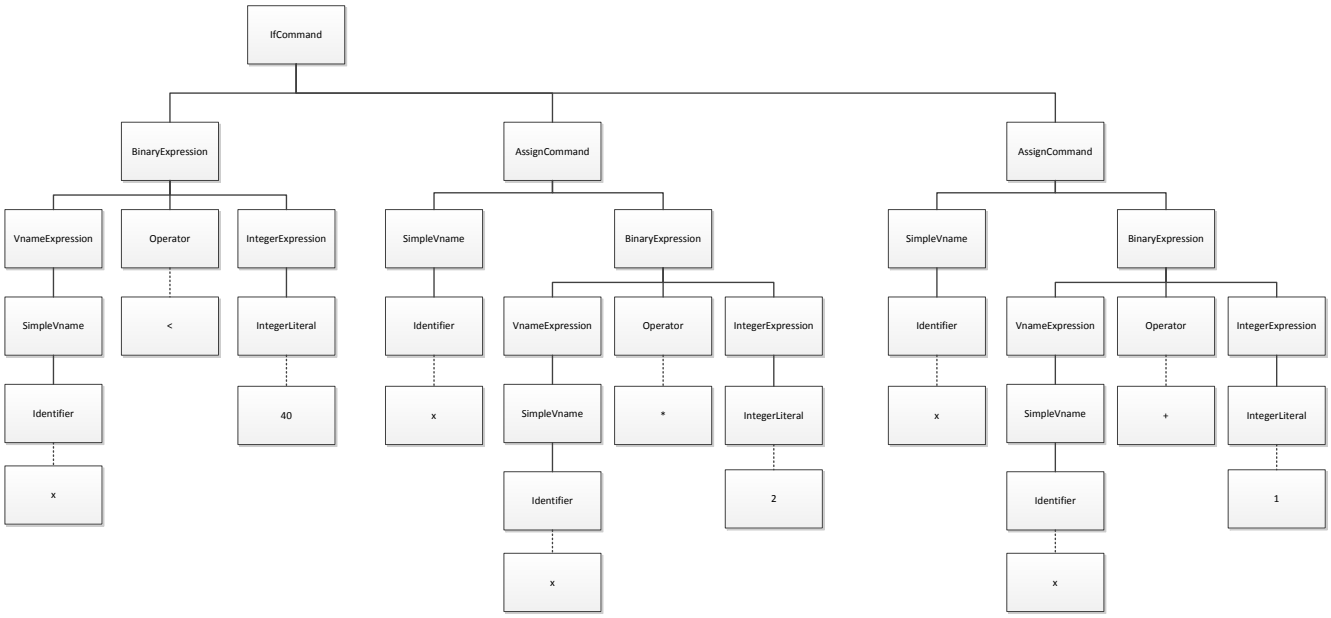


Abbildung 1: Lösung der Aufgabe 1.2

Aufgabe 1.3 AST zu Triangle

Geben Sie für den abstrakten Syntaxbaum auf den Seiten 8, 9 und 10 das zugehörige Triangle Programm an. Die AST Regeln auf Seite 109 und 110 im Buch wurden um eine Regel für Arrays erweitert. Diese Regel gehört in die Gruppe der V-name ASTs. Zusätzlich gibt es noch einen SkipCommand welcher einem leeren Befehl entspricht. Ein Beispiel hierzu gibt es auf Seite 21 des Buches (Example 1.9).

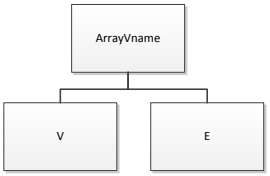


Abbildung 2: Regel für Arrays

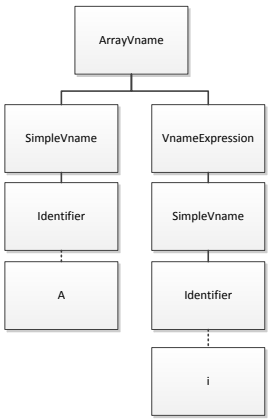


Abbildung 3: Beispiel

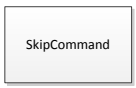


Abbildung 4: Leerer Befehl

Aufgabe 1.3 Lösung

```
Die Deklaration der Variablen ist nicht notwendig für die Lösung

let
  var i: Integer;
  var j: Integer;
  var buf : Integer;
  var x : Integer
in begin
  i := 0;
```

```
  while i < n do begin
    x := n - i - 2;
    j := 0;
    while j <= x do begin
      if val[j] > val[j+1] then
        begin
          buf := val[j];
          val[j] := val[j+1];
          val[j+1] := buf
        end
      else;
      j := j + 1
    end;
    i := i + 1
  end

end
```

Aufgabe 1.4 LL(1) Grammatiken

Welche der gegebenen Grammatiken sind durch einen LL(1) Parser zu verarbeiten? Falls eine Grammatik dies nicht ist, so formen Sie diese entweder um, so dass sie verarbeitet werden kann, oder begründen Sie, warum eine Umformung gegebenenfalls nicht möglich ist.

a) $G = (\{S, A, B\}, \{x, y, z, 0, 1\}, P, S), P:$
$S ::= A \mid B$
$A ::= x\ A\ y \mid 0$
$B ::= x\ B\ z \mid 1$

b) $G = (\{A, B, C, D\}, \{\text{if}(,), \text{else}, \text{fi}, \text{true}, \text{false}, a, b, c\}, P, A), P:$
$A ::= \text{if}\ (\ B\ )\ A\ C \mid \text{D}$
$B ::= \text{true} \mid \text{false}$
$C ::= \text{else}\ A\ \text{fi} \mid \text{fi}$
$D ::= a \mid b \mid c$

c) $G = (\{S, X, Y\}, \{0, 1, 2, a, b\}, P, S), P:$
$S ::= X\ 0 \mid Y\ 1 \mid 2 \mid S\ S$
$X ::= Y$
$Y ::= a \mid b$

Aufgabe 1.4 Lösung

a) Diese Grammatik ist durch einen LL(1) Parser nicht zu verarbeiten. Linksausklammern hilft nicht, da die Anzahl der führenden 'x' variabel ist, und die Anzahl der 'y' bzw. 'z' von der Anzahl der 'x' abhängt.

b) Die vorliegende Grammatik kann durch einen LL(1) Parser verabeitet werden.

c) Diese Grammatik ist in folgendem, umgeformten Zustand LL(1) konform.
$G = (\{S, Y\}, \{0, 1, 2, a, b\}, P, S), P:$
$S ::= (Y\ (0 \mid 1) \mid 2)+$
$Y ::= a|b$

Aufgabe 1.5 LL(k) Grammatiken

Bestimmen Sie unter Verwendung der Zerlegungsregeln von Foliensatz 2, Folie 50f, ob die gegebene Grammatik LL(1) ist.

$G = (\{S, B\}, \{a, b\}, P, S), P:$
$S ::= aBab \mid bBbb$
$B ::= a \mid \epsilon$

Aufgabe 1.5 Lösung

Zum Lösen dieser Aufgabe ist ein Test auf die Anwendbarkeit der Zerlegungsregeln von Foliensatz 2, Folie 50 auf alle Produktionen notwendig.

Wir prüfen zunächst Produktion $S ::= aBab \mid bBbb$. Die rechte Seite enthält eine Alternative, wir müssen also die Regeln anwenden mit $X = aBab$ und $Y = bBbb$. Weder X noch Y lassen sich zu ϵ ableiten, daher testen wir die erste Regel.

$$
\begin{aligned}
&X = aBab \\
&Y = bBbb \\
&\text{starters}_1[[aBab]] = \{a\} \\
&\text{starters}_1[[bBbb]] = \{b\} \\
&\text{starters}_1[[aBab]] \cap \text{starters}_1[[bBbb]] = \emptyset
\end{aligned}
$$

Die erste Regel ist also für die S-Produktion erfüllt.

Betrachten wir nun die B-Produktion. Auch sie enthält eine Alternative, so dass wir die Regeln mit $X = a$ und $Y = \epsilon$ anwenden. Y kann zu ϵ abgeleitet werden, weswegen wir nun die zweite Regel testen müssen. Es ist also notwendig zu überprüfen ob $\text{starters}[[X]] \cap (\text{starters}[[Y]] \cup \text{follow}[[X|Y]]) = \emptyset$ verletzt wird.

$$
\begin{aligned}
&X = a \\
&Y = \epsilon \\
&\text{starters}_1[[a]] = \{a\} \\
&\text{starters}_1[[\epsilon]] = \emptyset \\
&\text{follow}_1[[a|\epsilon]] = \{a, b\} \\
&\text{starters}_1[[\epsilon]] \cup \text{follow}[[a|\epsilon]] = \{a, b\} \\
&\text{starters}_1[[a]] \cap \{a, b\} = \{a\} \neq \emptyset
\end{aligned}
$$

Die Regel wird also für $k = 1$ verletzt.

Anmerkungen

- Der Zusammenhang zwischen den Argumenten der starters und follows-Mengen ist, dass es sich um eine der beiden Alternativen, oder eben die gesamte rechte Seite der Produktion handelt, die sie gerade auf die LL-Eigenschaft testen wollen.
- Wenn Sie eine Produktion mit mehr als zwei Alternativen testen wollen, müssen Sie die Regeln paarweise anwenden. Sei bspw. $A ::= B \mid C \mid D$, so müssen Sie die Regeln mit $X = B, Y = C$ und $X = C, Y = D$ und $X = B, Y = D$ prüfen.

Aufgabe 1.6 LL(k) Grammatik-Transformation

Transformieren Sie die folgende LL(k) Grammatik mit k > 1 so um, dass diese anschließend LL(1) entsprechen. Verwenden Sie hierzu die Transformationsregeln, welche Sie in der Vorlesung kennen gelernt haben.

$$
\begin{aligned}
&G = (\{S, X, Y\}, \{a, b, c, d, e, f\}, P, S), P : \\
&S ::= abXS \mid acYS \mid d \\
&X ::= e \mid f \\
&Y ::= a \mid b
\end{aligned}
$$

Aufgabe 1.6 Lösung

Man sieht, dass die gegebene Grammatik LL(2) ist, da für die Unterscheidung der ersten und zweiten Alternative der S-Produktion Kenntnis von zwei Zeichen erforderlich ist.

Linksausklammern der S-Produktion liefert:

$$
\begin{aligned}
&G = (\{S, X, Y, R\}, \{a, b, c, d, e, f\}, P, S), P : \\
&S ::= aR \mid d \\
&R ::= bXS \mid cYS \\
&X ::= e \mid f \\
&Y ::= a \mid b
\end{aligned}
$$

Aufgabe 1.7 Parser

Geben Sie die Namen der Java Klassen des AST für die gegebene Grammatik an. Unterstreichen sie hierbei die Klassen welche die Schreibweise des Tokens benötigen. Schreiben Sie dann die benötigten parseN-Methoden. Halten Sie sich bei Ihrer Umsetzung an das Beispiel in den Folien bzw. des Buches (Seite 95, Schritt 2). Jedes Terminalsymbol außer denen aus der Produktion ID sind als Schlüsselwort zu betrachten.

Es ist nicht notwendig die AST Klassen zu implementieren. Substitution ist nicht erlaubt.

$G = (\{S, E, B, ID\}, \{\text{while}(,), \text{end}, \text{true}, \text{false}, a, b, c\}, P, S), P:$
$S ::= \text{while} \; (\; B \;) \; S \; E \mid \text{ID}$
$E ::= \text{end}$
$B ::= \text{true} \mid \text{false}$
$ID ::= a \mid b \mid c$

Aufgabe 1.7 Lösung

AST Klassen: S, while, B, true, false, end, E, ID

```
private void parseS () {
  switch (currentToken.kind) {

  case Token.While:
    {
      acceptIt();
      // optional, falls man nicht "while (" als ein Token betrachtet.
      acceptToken(Token.openBrace);
      parseB();
      acceptToken(Token.closeBrace);
      parseS();
      parseE();
    }
    break;

  case Token.ID:
    {
      acceptIt();
    }
    break;
  }
}

private void parseE() {
  switch (currentToken.kind) {

  case Token.end:
    {
      acceptIt();
    }
    break;
  }
}

private void parseB() {
  switch (currentToken.kind) {

  case Token.true:
    {
      acceptIt();
    }
```

```
      break;

    case Token.false:
      {
        acceptIt();
      }
      break;
    }
}
```

Anmerkungen:

- In der Aufgabenstellung sollten eigentlich `while` und `(` separate Tokens sein. Je nachdem, wie Sie die Aufgabenstellung interpretiert haben, ist die markierte Stelle im Quelltext optional.
- Achten Sie (auch im Hinblick auf die Klausur) genau auf die Aufgabenstellung. Es gibt viele Möglichkeiten, einen funktionieren, rekursiven Abstiegsparser zu bauen. In dieser Aufgabe sollten Sie jedoch den Parser ohne die Anwendung von Substitutionen konstruieren.

Aufgabe 1.8 Scanner

Nehmen Sie an, dass die lexikalische Grammatik von Mini-Triangle um hexadezimale Literale erweitert wird. Geben Sie hierzu analog zu den Verfahren im Kapitel 4.5 ab Seite 118 im Buch einen passenden Scanner an. Hierbei reicht es aus die ”scanToken” Methode zu schreiben. Die Fälle für einzelnen Buchstaben oder Ziffern die keine Sonderfälle betreffen können sie mit case ’a’ ... case ’z’ zusammenfassen. Hinweis: Es kann passieren, dass Sie von dem Vefahren für einige Regeln leicht abweichen müssen.

Token ::= Identifier | Operator | IntegerLiteral | XLiteral | ...
Identifier ::= Letter (Letter | Digit)∗
IntegerLiteral ::= (Digit)+
XLiteral ::= 0x(Digit)+

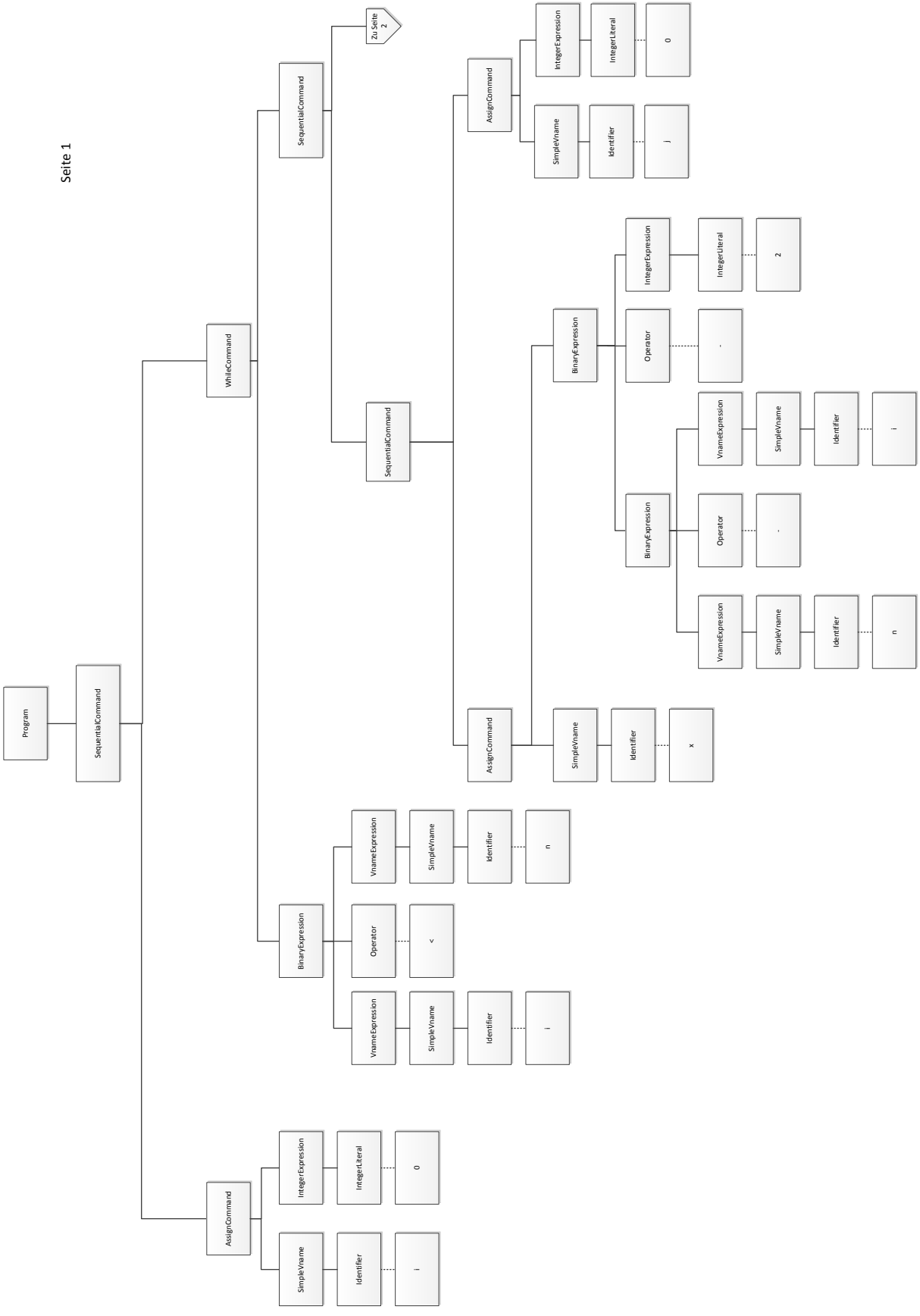
Nennen Sie anschliesend das wesentliche Problem dieser Grammatik welches Sie zur Abweichung von dem Verfahren gezwungen hat.

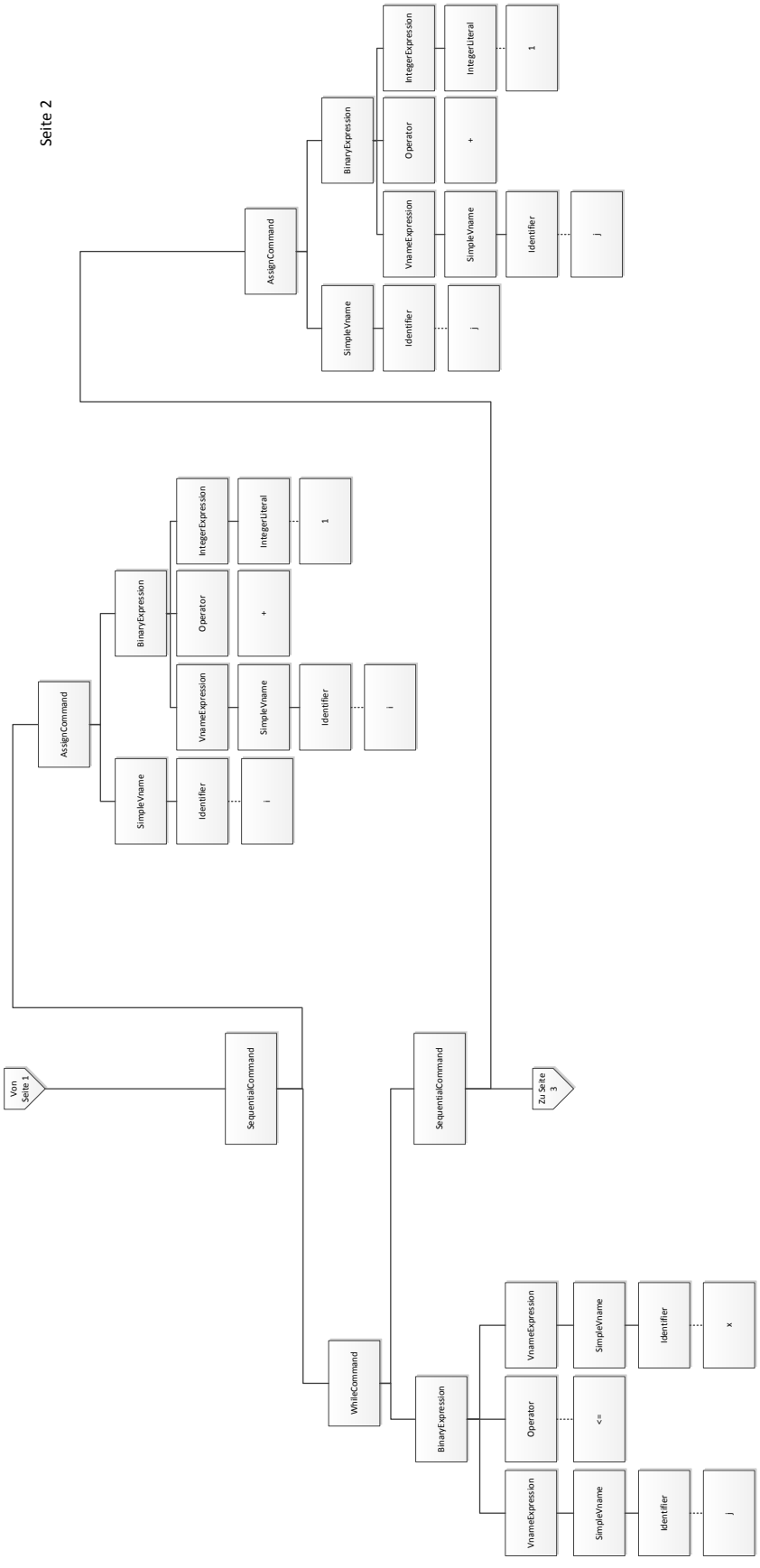
Aufgabe 1.8 Lösung

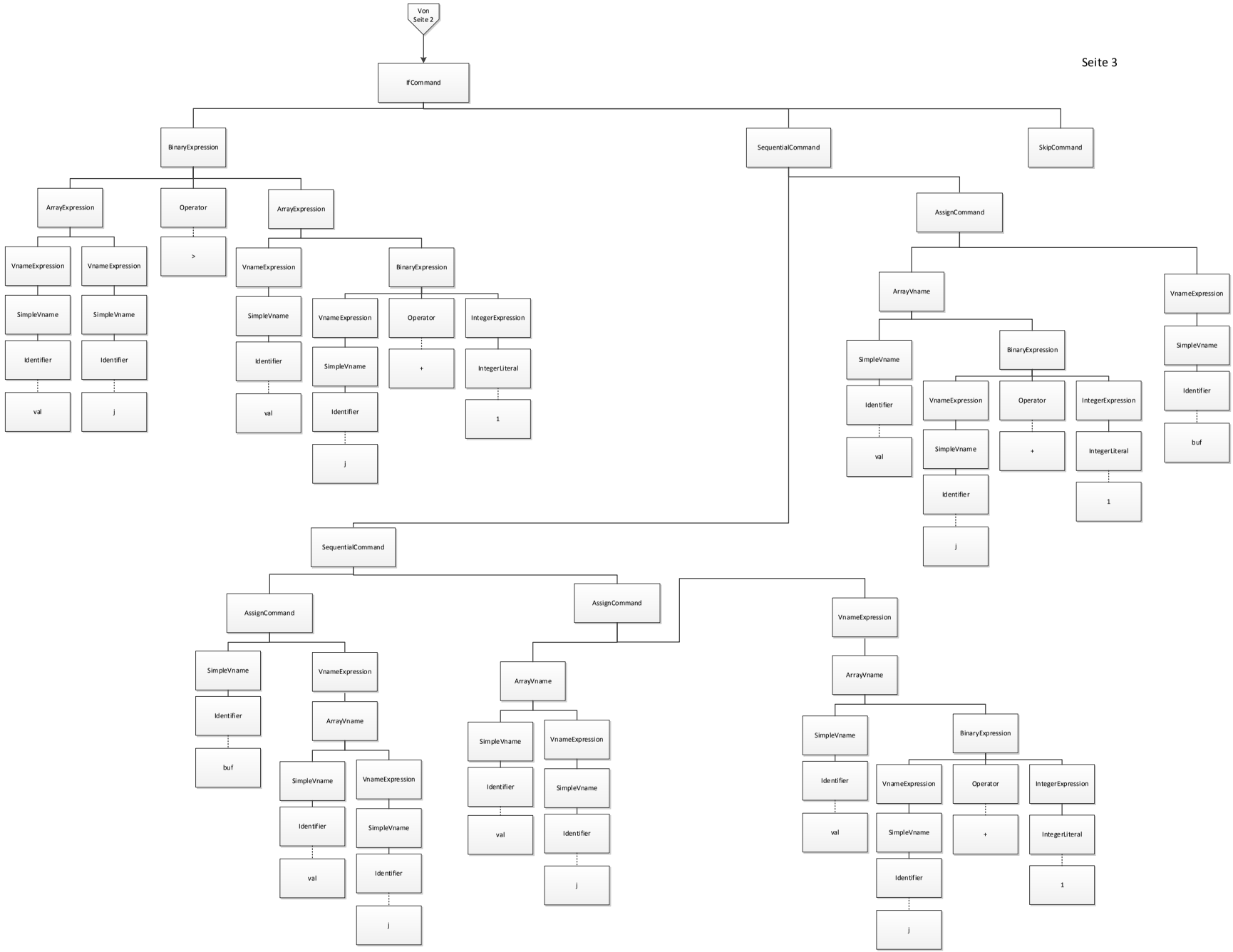
```
private byte scanToken() {
  switch (currentChar) {

  // ...
  // hier nur die entscheidende Stelle:
  case '0' ... case '9':
    if(currentChar == '0') {
      takeIt();
      if(currentChar == 'x') {
        takeIt();
        while(isDigit(currentChar))
          takeIt();
        return Token.XLiteral;
      }
      else {
        while(isDigit(currentChar))
          takeIt();
        return Token.IntegerLiteral;
      }
    }
    else {
      while(isDigit(currentChar))
```

```
        takeIt();
      return Token.IntegerLiteral;
    }
  }
}
```

Die gegebene Grammatik ist nicht eindeutig. Wenn im Zeichenstrom beispielsweise das Zeichen *0* vorliegt und mit einem neuen Token begonnen wird, so ist unklar, ob hier ein Integer Literal oder ein X Literal beginnt. Dieses Problem kann nur auf kompliziertem Weg mittels Präzedenzregeln o. a. gelöst werden.


Program
SequentialCommand
AssignCommand
SimpleVname
Identifier
i
IntegerExpression
IntegerLiteral
0
WhileCommand
BinaryExpression
VnameExpression
SimpleVname
Identifier
i
Operator
<
VnameExpression
SimpleVname
Identifier
n
SequentialCommand
AssignCommand
SimpleVname
Identifier
x
BinaryExpression
VnameExpression
SimpleVname
Identifier
i
Operator
-
VnameExpression
SimpleVname
Identifier
n
Operator
-
IntegerExpression
IntegerLiteral
2
SequentialCommand
AssignCommand
SimpleVname
Identifier
j
IntegerExpression
IntegerLiteral
0
Zu Seite 2


Seite 2
Von Seite 1
SequentialCommand
WhileCommand
BinaryExpression
VnameExpression
SimpleVname
Identifier
x
Operator
<=
VnameExpression
SimpleVname
Identifier
j
SequentialCommand
Zu Seite 3
AssignCommand
SimpleVname
Identifier
i
BinaryExpression
Operator
+
VnameExpression
SimpleVname
Identifier
i
IntegerExpression
IntegerLiteral
1
AssignCommand
SimpleVname
Identifier
j
BinaryExpression
Operator
+
VnameExpression
SimpleVname
Identifier
j
IntegerExpression
IntegerLiteral
1


Von Seite 2
IfCommand
BinaryExpression
ArrayExpression
VnameExpression
SimpleVname
Identifier
val
VnameExpression
SimpleVname
Identifier
j
Operator
>
ArrayExpression
VnameExpression
SimpleVname
Identifier
val
BinaryExpression
VnameExpression
SimpleVname
Identifier
j
Operator
+
IntegerExpression
IntegerLiteral
1
SequentialCommand
SkipCommand
AssignCommand
ArrayVname
SimpleVname
Identifier
val
BinaryExpression
VnameExpression
SimpleVname
Identifier
j
Operator
+
IntegerExpression
IntegerLiteral
1
VnameExpression
SimpleVname
Identifier
buf
SequentialCommand
AssignCommand
SimpleVname
Identifier
buf
VnameExpression
ArrayVname
SimpleVname
Identifier
val
VnameExpression
SimpleVname
Identifier
j
AssignCommand
ArrayVname
SimpleVname
Identifier
val
VnameExpression
SimpleVname
Identifier
j
VnameExpression
ArrayVname
SimpleVname
Identifier
val
BinaryExpression
VnameExpression
SimpleVname
Identifier
j
Operator
+
IntegerExpression
IntegerLiteral
1