Übung zur Vorlesung Compiler 1: Grundlagen

Prof. Dr. Andreas Koch
Jens Huthmann, Julian Oppermann

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 14/15
Aufgabenblatt 1

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 23.11.2014 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 1.2 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 1.3 AST zu Triangle

Geben Sie für den abstrakten Syntaxbaum auf den Seiten 4, 5 und 6 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). *Hinweis:* Sie brauchen den `let`-Block nicht anzugeben.

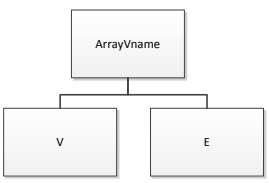


Abbildung 1: Regel für Arrays

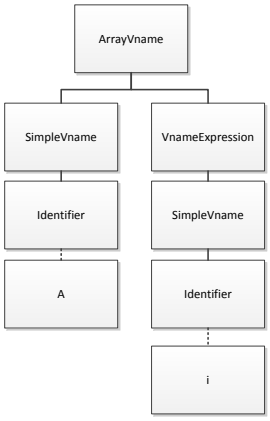


Abbildung 2: Beispiel

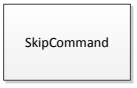


Abbildung 3: Leerer Befehl

[1] Programming Language Processors in Java

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 1.5 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 1.6 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.

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

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. Die Anwendung von Grammatiktransformationen ist nicht erlaubt.

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

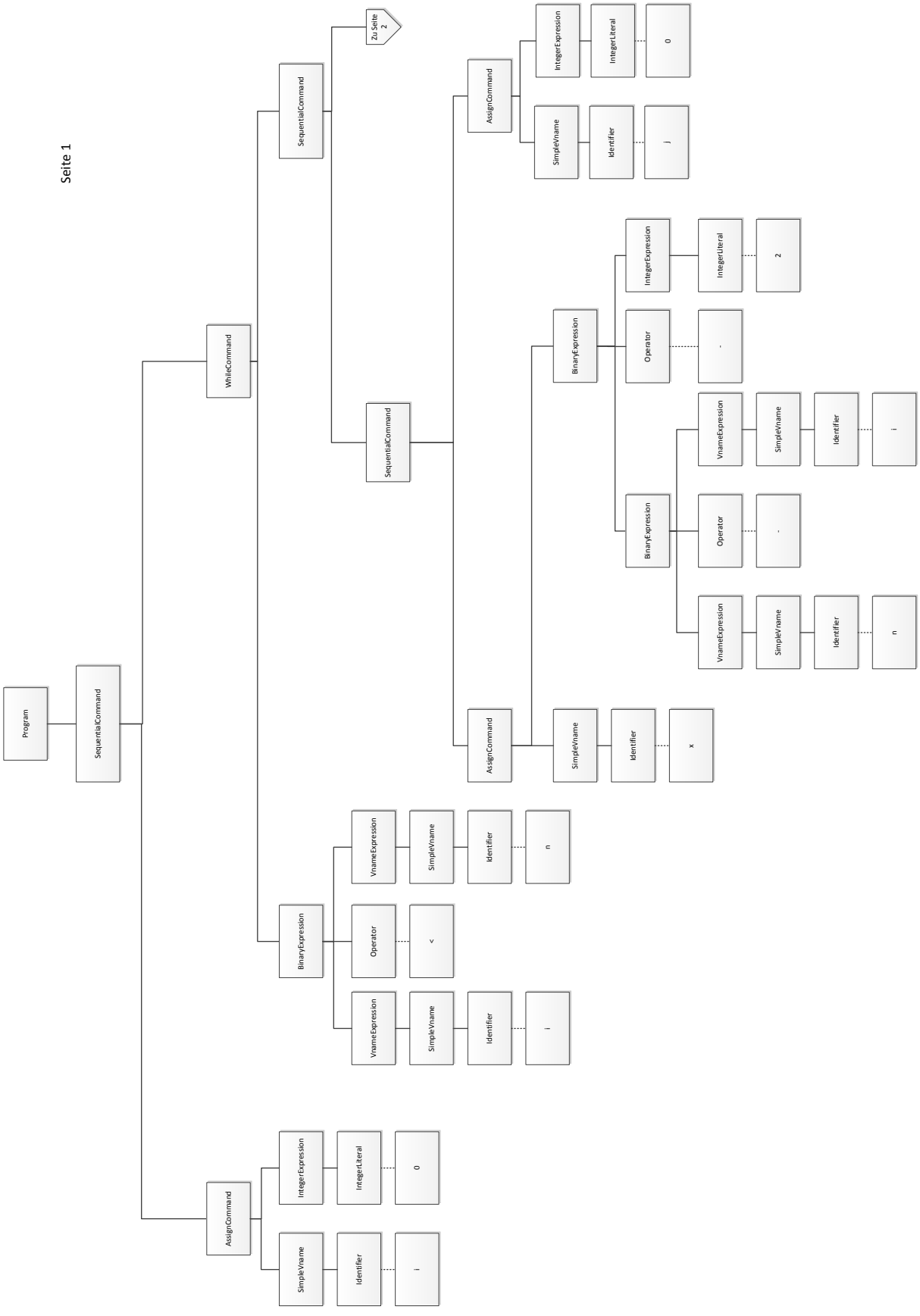
Aufgabe 1.8 Scanner

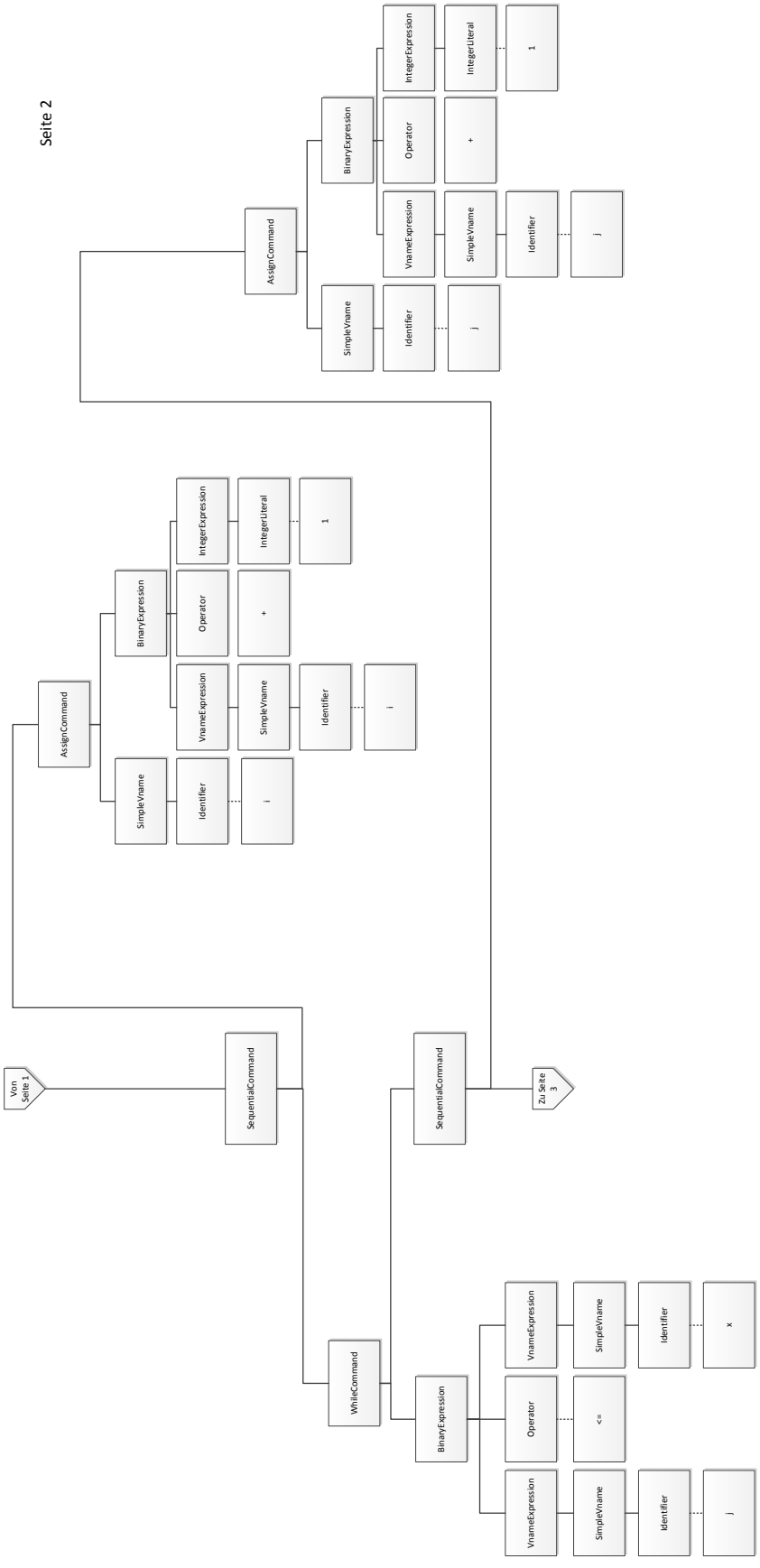
Nehmen Sie an, dass die lexikalische Grammatik von Mini-Triangle um ein spezielles X-Literal erweitert wird. Geben Sie hierzu analog zu den Verfahren im Kapitel 4.5 ab Seite 118 im Buch einen passenden Scanner an. Hierbei reicht es aus die ”scanToken” Methode zu schreiben. Die Fälle für einzelnen Buchstaben 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.

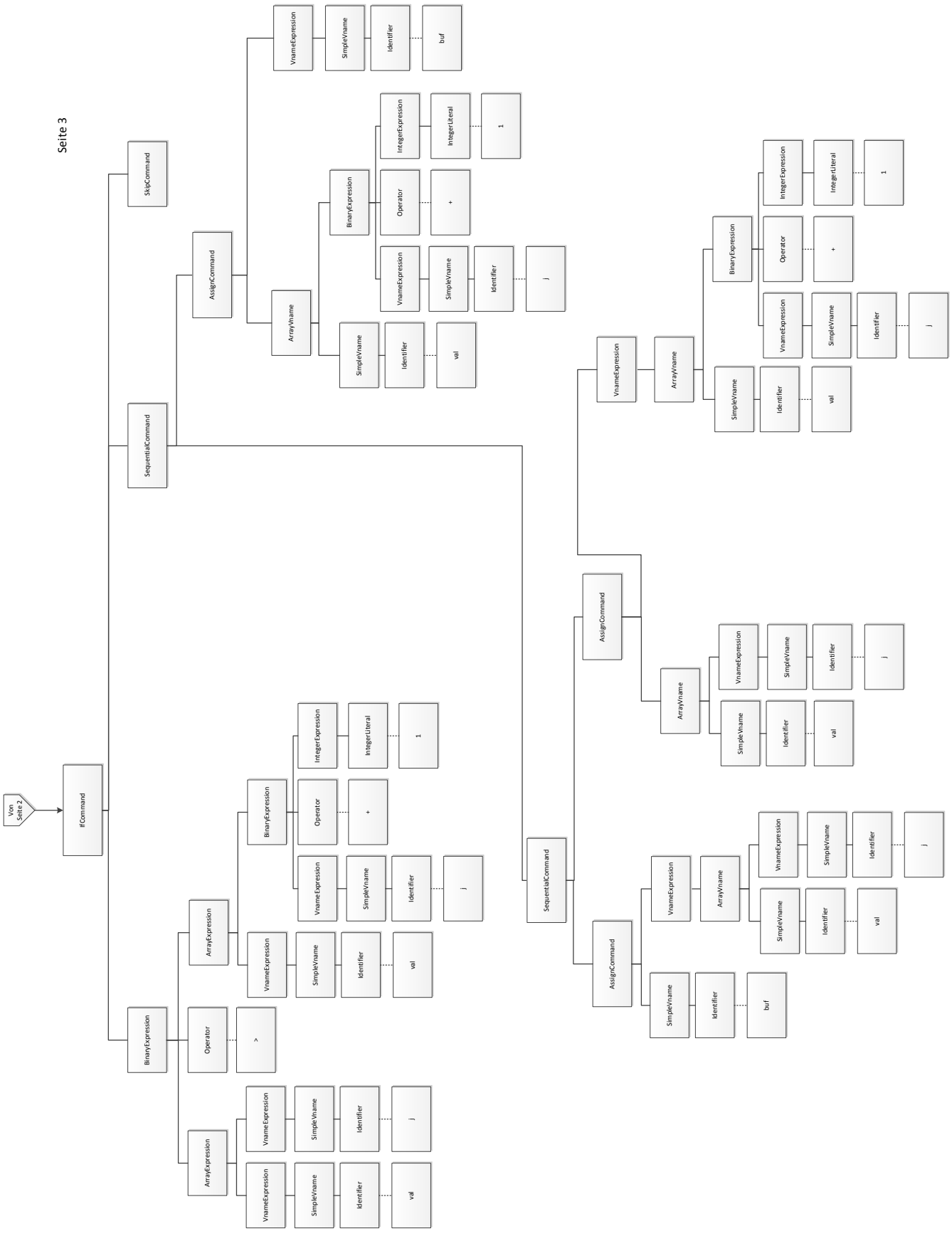
Token ::= Identifier | Operator | IntegerLiteral | XLiteral | ...
Identifier ::= Letter (Letter | Digit) $*$
IntegerLiteral ::= (Digit) +
XLiteral ::= 0x(Digit)+

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

Hinweis: Die Grammatik soll nicht hexadezimale Literale erkennen.


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


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
Operator
>
ArrayExpression
VnameExpression
SimpleVname
Identifier
val
[
VnameExpression
SimpleVname
Identifier
j
]
ArrayExpression
VnameExpression
SimpleVname
Identifier
val
[
BinaryExpression
Operator
+
IntegerExpression
IntegerLiteral
1
VnameExpression
SimpleVname
Identifier
j
]
SkipCommand
SequentialCommand
AssignCommand
VnameExpression
SimpleVname
Identifier
buf
:=
ArrayVname
SimpleVname
Identifier
val
[
BinaryExpression
Operator
+
IntegerExpression
IntegerLiteral
1
VnameExpression
SimpleVname
Identifier
j
]
SequentialCommand
AssignCommand
VnameExpression
SimpleVname
Identifier
buf
:=
ArrayVname
SimpleVname
Identifier
val
[
VnameExpression
SimpleVname
Identifier
j
]
AssignCommand
VnameExpression
SimpleVname
Identifier
val
:=
ArrayVname
SimpleVname
Identifier
val
[
VnameExpression
SimpleVname
Identifier
j
]