Einführung in den Compilerbau

Prof. Dr.-Ing. Andreas Koch
Julian Oppermann, Lukas Sommer

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 17/18
Theorieblatt 2

Abgabe bis Sonntag, 03.12.2017, 18:00 Uhr (MEZ)

Aufgabe 2.1 Nachweis der LL(1)-Eigenschaft

6 + 2 + 2 + 6 + 2 = 18 P

Gegeben sei die Grammatik `G = ({S, A, B, X, Y}, {a, b, c, 0, 1}, P, S)`. Die Menge `P` enthält folgende Produktionen:

```
S ::= B 0 Y 1
A ::= a | b
B ::= A | ε
X ::= a | (0 b) | (c c 1)
Y ::= 0 X (X*)
```

a) Geben Sie den Inhalt der starters- und follow-Mengen jeweils für die Nichterminale `A`, `X` und `Y` an.

Weisen Sie in den folgenden Teilaufgaben **formal** nach, dass `G` die LL(1)-Eigenschaft erfüllt. Geben Sie für alle Schritte des Nachweises zunächst den Ansatz (mit starters-/follow-Mengen) an, und setzen Sie anschließend alle konkreten Mengen ein (auch wenn Sie diese an anderer Stelle bereits angegeben haben).

b) Weisen Sie formal nach, dass die `A`-Produktion die LL(1)-Eigenschaft erfüllt.

c) Weisen Sie formal nach, dass die `B`-Produktion die LL(1)-Eigenschaft erfüllt.

d) Weisen Sie formal nach, dass die `X`-Produktion die LL(1)-Eigenschaft erfüllt.

e) Weisen Sie formal nach, dass die `Y`-Produktion die LL(1)-Eigenschaft erfüllt.

Aufgabe 2.2 Mehr als ein Token Vorausschau *6 + 6 = 12 P*

Gegeben sei die Grammatik `H = ({S, A}, {a, b}, P, S)`. Die Menge `P` enthält folgende Produktionen:

```
S ::= ( a A a a b ) | ( b A b a b )
A ::= b | ε
```

a) Vervollständigen Sie den folgenden rekursiven Abstiegsparser, indem Sie eine Implementierung der Methode `parseA()` angeben. Falls das Nichtterminal `A` zu ε abgeleitet wird, soll `epsilonCounter` um 1 inkrementiert werden. Andernfalls, d.h. wenn `A` zu `b` abgeleitet wird, soll das `b`-Token mit `accept('b')` konsumiert werden. Verwenden Sie Java-ähnlichem Pseudocode analog zu `parseS()`. Sie haben unbegrenzte Vorausschau: `currentToken[0]` bezeichnet das aktuelle Token, `currentToken[1]` das nächste Token, `currentToken[2]` das übernächste Token, usw.. **Verwenden Sie so wenig Vorausschau wie möglich.**

```
void parseS() {
  if (currentToken[0] == 'a') {
    accept('a'); parseA(); accept('a'); accept('a'); accept('b');
  } else if (currentToken[0] == 'b') {
    accept('b'); parseA(); accept('b'); accept('a'); accept('b');
  }
}

int epsilonCounter = 0;

void parseA() {
  // TODO
}
```

b) Vervollständigen Sie den folgenden (modifizierten) rekursiven Abstiegsparser, indem Sie eine Implementierung der Methode `parseA(boolean parsedLeftSideOfS)` angeben. Über diesen Methodenparameter können Sie unterscheiden, ob die linke (`parsedLeftSideOfS == true`) oder rechte (`parsedLeftSideOfS == false`)) Seite der Alternative in `S` genommen wurde. Wie in Teilaufgabe a) gilt: Falls das Nichtterminal `A` zu ε abgeleitet wird, soll `epsilonCounter` um 1 inkrementiert werden. Andernfalls, d.h. wenn `A` zu `b` abgeleitet wird, soll das `b`-Token mit `accept('b')` konsumiert werden. Melden Sie bei Bedarf[1] mit `error()` einen Syntaxfehler. Verwenden Sie Java-ähnlichem Pseudocode analog zu `parseS()`.

Sie haben unbegrenzte Vorausschau: `currentToken[0]` bezeichnet das aktuelle Token, `currentToken[1]` das nächste Token, `currentToken[2]` das übernächste Token, usw.. **Verwenden Sie so wenig Vorausschau wie möglich.**

```
void parseS() {
  if (currentToken[0] == 'a') {
    accept('a'); parseA(true); accept('a'); accept('a'); accept('b');
  } else if (currentToken[0] == 'b') {
    accept('b'); parseA(false); accept('b'); accept('a'); accept('b');
  }
}

int epsilonCounter = 0;

void parseA(boolean parsedLeftSideOfS) {
  // TODO
}
```

[1] Es existiert eine Lösung, die kein explizites Melden von Syntaxfehler erfordert.

Aufgabe 2.3 Identifikationstabelle *5 P*

Gegeben sei folgender Ausschnitt aus einem MAVL-Programm:

```
1  ...
2  {
3    var int alice;
4    var int claire;
5    {
6      var int alice;
7      {
8        var int bob;
9        var int claire;
10     }
11     var int bob;
12     {
13       var int alice;
14       var int claire;
15       // <-- HIER
16     }
17   }
18 }
19 ...
```

Skizieren Sie den Inhalt der Datenstrukturen `idents` und `scopes` zum gekennzeichneten Zeitpunkt gemäß der in der Vorlesung vorgestellten Implementierung einer Identifikationstabelle (3. Foliensatz, Folie 20f). Verwenden Sie als "Attribut" die Zeilennummer der Deklaration, und **kennzeichnen Sie deutlich das oberste Element** der beiden Stacks.

Aufgabe 2.4 Kontextuelle Einschränkungen *5 P*

Geben Sie die Regeln zur Überprüfung der kontextuellen Einschränkungen für den **Kopf** einer for-Schleife in MAVL gemäß der Sprachspezifikation an.

Verwenden Sie eine kurze und präzise textuelle Beschreibung der Regeln in Ihren eigenen Worten[2]; es ist kein bestimmter Formalismus gefordert. Referenzieren Sie Kindknoten/Bezeichner mit den entsprechenden Namen aus der Implementierung des MAVL-Compilers (siehe `https://www.esa.informatik.tu-darmstadt.de/campus/mavl/mavlc/ast/nodes/statement/ForLoop.html`). Die Felder `initVarDecl` und `incrVarDecl` seien noch uninitialisiert und sollen in Ihren Regeln nicht verwendet werden.

[2] Kopieren Sie keine Textpassagen aus der MAVL-Spezifikation!