Einführung in den Compilerbau

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

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 18/19
Theorieblatt 2

Abgabe bis Sonntag, 02.12.2018, 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, 0, 1, 2}, P, S)`. Die Menge `P` enthält folgende Produktionen:

```
S ::= B a Y b
A ::= 0 | 1
B ::= A | ε
X ::= 0 | (a 1) | (2 2 b)
Y ::= a 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 Mehrdeutigkeit und Vorausschau *3 + 9 = 12 P*

Wie in §1 der MAVL-Sprachspezifikation erwähnt, orientiert sich die Syntax der Sprache in weiten Teilen an gängigen Programmiersprachen wie Java, C, C++, oder Scala. Records sind in dieser Hinsicht etwas unkonventionell, da z.B. Elementselektionen als `someRecord@someElement` geschrieben werden statt (wie in anderen Sprachen üblich) als `someRecord.someElement`. Um diese Diskrepanz aufzulösen, soll die Grammatik von MAVL abgeändert werden, um das `@`-Symbol loszuwerden. Die Änderungen werden am Beispiel des folgenden Programms demonstriert, welches hier zunächst in der aktuellen MAVL-Syntax angegeben ist:

```
1 record Point {
2   val float x;
3   val float y;
4 }
5
6 function void main() {
7   val Point p = @Point[1.0, 2.0];
8   val float x = p@x;
9 }
```

In dieser Aufgabe beschränken wir uns dabei auf das `@` in Record-Literalen (Zeile 7 im Beispiel). Wir versuchen also nicht, etwa die Elementselektion in Zeile 8 zu `val float x = p.x;` abzuändern.

a) Es kommt zunächst der Vorschlag, das Präfix `@` einfach wegzulassen. Zeile 7 des obigen Beispiels sähe dann also so aus: `val Point p = Point[1.0, 2.0];`
Leider führt das zu einer Mehrdeutigkeit in der Sprache. Identifizieren Sie die Quelle dieser Mehrdeutigkeit und geben Sie ein konkretes Beispiel in Form eines Ausdrucks, der sowohl als Record-Literal als auch anders geparst werden könnte.

b) Um die in a) festgestellte Mehrdeutigkeit zu vermeiden, werden die Namen der Elemente in Record-Literale mit aufgenommen und durch `=` von den dazugehörigen Ausdrücken getrennt. In obigem Beispiel würde Zeile 7 also `val Point p = Point[x=1.0, y=2.0];` lauten.

Der Einfachheit halber betrachten wir diese Änderung hier an einer Teilmenge von MAVL, beschrieben durch die Grammatik `G = ({expr, elementSelect, negation, recordLiteral}, {INT, ID, '[', ']', '-', '=', ','}, P, expr)` wobei die Produktionsmenge `P` diese Produktionen enthält:

```
expr          ::= INT | ID | elementSelect | negation | recordLit
elementSelect ::= ID '[' expr ']'
negation      ::= '-' expr
recordLiteral ::= ID '[' ID '=' expr (',' ID '=' expr)* ']'
```

Vervollständigen Sie den auf der folgenden Seite angegebenen rekursiven Abstiegsparser, indem Sie die Methode `parseExpr()` angeben. Verwenden Sie Java-ähnlichen Pseudocode analog zu `parseElementSelect()`, `parseNegation()` und `parseRecordLiteral()`. Sie haben unbegrenzte Vorausschau: `currentToken[0]` bezeichnet das aktuelle Token, `currentToken[1]` das nächste Token, `currentToken[2]` das übernächste, und so weiter. **Verwenden Sie so wenig Vorausschau wie möglich.** Der Parser soll die Sprache nur erkennen, der Aufbau eines ASTs o.Ä. ist nicht gefordert. Melden Sie bei Bedarf[1] mit `error()` einen Syntaxfehler.

[1] Es existiert eine Lösung, die kein explizites Melden von Syntaxfehlern benötigt.

```
parseExpr() {
    // TODO
}

parseElementSelect() {
  accept(ID);
  accept('[');
  parseExpr();
  accept(']');
}

parseNegation() {
  accept('-');
  parseExpression();
}


parseRecordLiteral() {
    accept(ID);
    accept('[');
    accept(ID);
    accept('=');
    parseExpr();
    while (currentToken[0] == ',') {
        accept(',');
        accept(ID);
        accept('=');
        parseExpr();
    }
    accept(']');
}
```

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 claire;
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!