Einführung in den Compilerbau

Lösungsblatt Nr. 1

Patrick Elsen, Viola Hofmeister und Michael Matthé

Andreas Koch

Wintersemester 2018-2019
Technische Universität Darmstadt

Einleitung

Auf diesem Aufgabenblatt sollen Sie sich mit der *Matrix and Vector Language*, kurz MAVL, vertraut machen. Studieren Sie bitte zunächst die MAVL-Sprachspezifikation, die Sie im Moodle-Kurs der Veranstaltung finden.

Aufgabe 1.1: MAVL-Syntax

Die MAVL-Sprachspezifikation enthält nur eine informelle Beschreibung der Syntaxelemente der Sprache. In den folgenden Teilaufgaben sollen Sie einige der Syntaxelemente in Produktionen einer kontextfreien Grammatik überführen.

Aufgabe 1.1a: Produktionen

Geben Sie Produktionen für die Nichtterminale `mulExpr` *(Multiplikations-Operator),* `subvectorExpr` *(Subvektor-Operator), sowie* `recordElementSelectExpr` *(Selektion von Record-Elementen) an.*

Ein Multiplikationsausdruck ist in der Sprachspezifikation unter *§ 7.5 Ternärer Operator* definiert. Ein solcher Ausdruck nimmt `int` oder `float`-Ausdrücke als Parameter und ist Linksassoziativ. Also kann man diesen grammatikalisch folgendermaßen definieren.

```
mulExpr ::= expr '*' expr
```

Die `subvectorExpr` ist in dem Sprachstandard unter *§ 7.7.4 Submatrix und Subvektor* definiert. Hier wird definiert, dass eine solche Beispielsweise als $v\{l : e : u\}$ geschrieben werden kann, wobei v ein Vektor, e eine `int`-Ausdruck, und l und u konstante `int`-Ausdrücke sein müssen, so dass $l < u$. Dieser Ausdruck extrahiert einen Subvektor mit den Elementen $[i + l, i + u]$. Zuerst brauchen wir also ein Hilfs-Nichtterminalsymbol `constExpr` für l und u.

```
constExpr ::= '-'? INT (('*' | '+' | '-' | '/' | '^') constExpr)?
            | '(' constExpr ')'
```

Mit diesem Hilfsausdruck können wir nun die `subvectorExpr` definieren. Da *e* ein beliebiger Ausdruck sein kann, können wir hier eine `expr` verwenden.

```
subvectorExpr ::= expr '{' constExpr ':' expr ':' constExpr '}'
```

Unter *§ 7.8 Selektion von Record-Elementen* ist definiert, wie der Syntax der Recordelementselektion funktioniert. Mit diesem kann man auf ein bestimmtes Element eines Records zugreifen. Mit dem Ausdruck `person@name` greift man zum Beispiel auf das Namenselement des Records `person` zu.

```
recordElementSelectExpr ::= expr '@' ID
```

Aufgabe 1.1b

Geben Sie Produktionen für die Nichtterminale `primitiveType` (primitive Typen) und `vectorType` (Vektortypen) an.

Die primitiven Typen sind unter *§ 4.2 Primitive Datentypen* definiert. Hier sind nur `int`, `float` und `bool` als eingebaute, primitive Typen angegeben. Also könnte eine Grammatik folgendermaßen aussehen:

```
primitiveType ::= 'int' | 'float' | 'bool'
```

Der `vectorType` ist bei *§ 4.5 Vektoren* definiert. Ein Vektor muss, mit einem Elementtyp (entweder `int` oder `float`) und einer Länge (positive, ganze Zahl) definiert werden. Die Länge muss ein konstanter Ausdruck sein, also können wir hier wieder `constExpr` benutzen.

```
vectorType ::= 'vector' '<' ('int' | 'float') '>' '[' constExpr ']'
```

Aufgabe 1.1c

Geben Sie Produktionen für die Nichtterminale `returnStmt` (Rückgabebefehl), `varDecl` Variablendeklaration), `callStmt` (Aufruf-Befehl, ohne Rückgabewert) sowie `forStmt` (For-Schleife) an.

Der Rückgabebefehl ist unter *§ 6.8 Rückgabe* im Sprachstandard definiert. Ein solcher Befehl besteht aus dem Keyword `'return'`, einem Ausdruck und einem abschließenden Semikolon. Also kann man einen solchen folgendermaßen definieren.

```
returnStmt ::= 'return' expr ';'
```

Die Variablendeklaration ist unter *§ 6.2 Variablendeklaration* im Sprachstandard definiert. Eine solche Deklaration besteht aus dem Keyword `var`, dem Typen der Variablen (primitiver Typ wie `int`, `float` oder `bool`, Vektor, Matrix oder Record), sowie ein Variablenname.

```
varDecl ::= 'var' (primitiveType | vectorType | matrixType | ID) ID ';'
```

Aufrufe sind unter *§ 6.4 Aufrufe* im Sprachstandard definiert. Diese bestehen aus dem Namen einer Funktion, einer Klammer `'('`, den Parametern (durch Kommata separiert) und einer Klammer `')'`.

```
callStmt ::= ID '(' (expr (',' expr)*)? ')' ';'
```

Die For-Schleife ist unter *§ 6.6.1 For-Schleife* definiert. Es besteht aus dem Keyword `for`, einer offenen Klammer, einer initialen Zuweisung, einer Kondition (boolscher Ausdruck, der bei jedem Durchlauf ausgewertet wird), eine Zuweisung, die nach jeden Durchlauf ausgeführt wird (typischerweise, um eine Zählervariable zu inkrementieren), eine schließende Klammer, sowie entweder ein einzelner Ausdruck oder mehrere Ausdrücke in geschweiften Klammern. Daraus ergibt sich folgende Grammatik.

```
forStmt ::= 'for' '(' ID '=' expr ';' expr ';' ID '=' expr ')'
            (statement | '{' statement* '}')
```

Aufgabe 1.2: AST zu MAVL

Abstrakte Syntaxbäume (engl. *Abstract Syntax Tree*) sind eine weitverbreitete Zwischendarstellung, die nur essentielle Informationen enthält und Details der konkreten Syntax einer Programmiersprache abstrahiert. In dieser Aufgabe zeigen wir Ihnen eine mögliche Repräsentation von MAVL-Code als AST. Die darin verwendeten AST-Knoten korrespondieren auf natürliche Weise mit den in der Spezifikation beschriebenen Syntaxelementen.

Artikel 1.2a

Geben Sie den zum AST zugehörigen MAVL-Code an.

```
if(flag & r > q) {
  r = -1;
  e(q, r);
} else {
  r = a - (q * d);
}
```

Artikel 1.2b

Geben Sie den zum AST zugehörigen MAVL-Code an.

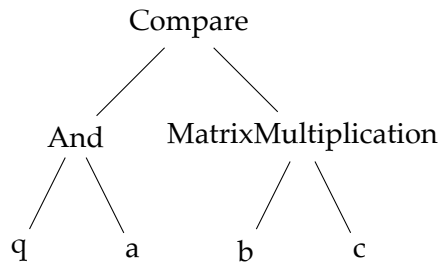
```
var vector<int>[3 + 1] p;
foreach(var int i : p) {
  i = k;
}
```

Artikel 1.3: Ausdrücke

Ausdrücke in typischen Programmiersprachen lassen sich einfach durch mehrdeutige Grammatiken beschreiben, die aber als Grundlage für die syntaktische Analyse ungeeignet sind.

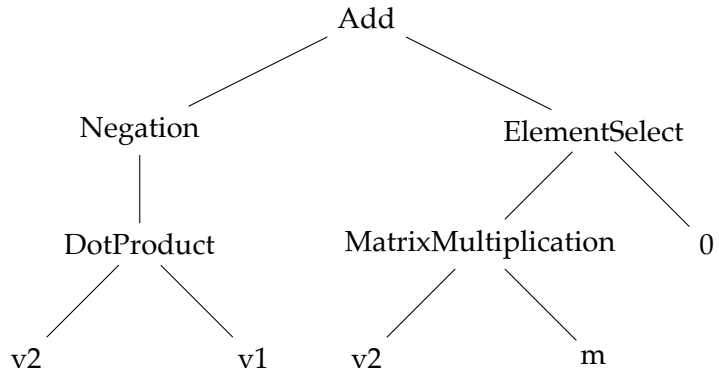
Artikel 1.3a

Zeichnen Sie den AST für den MAVL-Ausdruck `q & a == b # c`.



Artikel 1.3b

Zeichnen Sie den AST für den MAVL-Ausdruck `- v2 .* v1 + (v2 # m)[0]`.



Aufgabe 1.3c

Gegeben seien folgende Wertedefinitionen.

```
val matrix <int>[2][2] m = [[1, 2],
                            [3, 4]];
val vector<int>[2]    v1 = [4, 2];
val vector<int>[2]    v2 = [2, 3];
```

Welchen Wert liefert der Ausdruck aus Teilaufgabe 1.3b?

Dieses Programm kann man als ein kleines Ruby-Skript ausdrücken, was (hoffentlich) das richtige Ergebnis liefert. Wichtig ist hier, dass Ruby andere Operatopräzedenzregeln hat, weswegen anders geklammert werden muss.

```
require "matrix"

m = Matrix[[1, 2], [3, 4]]
v1 = Vector[4, 2]
v2 = Vector[2, 3]

puts (-v1.inner_product(v2) + (v2.covector * m)[0,0])
```

Dieses Programm liefert als Resultat der Berechnung das Ergebis `-3`.