Einführung in den Compilerbau

Lösungsblatt Nr. 1

von Patrick Elsen, Viola 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.

Abweichend von den Vorlesungsfolien soll die Grammatik jedoch in erweiterter Backus-Naur-Form (EBNF) beschrieben werden: Sie können und sollen in den Produktionen den ?-Operator (0 oder 1 Wiederholung) und den $*$-Operator (0 oder mehr Wiederholungen) verwenden. Beispielsweise können Sie eine Sequenz eines Nichtterminals A statt durch

```
A ::= A single-A
    | single-A
```

auch einfacher durch

```
A ::= single-A (single-A)*
```

ausdrücken.

Verwenden Sie bitte folgende Terminalsymbole (Tokens):
Whitespace-Symbole und Kommentare brauchen Sie nicht zu beachten. Lassen Sie in Ihrer Lösung nicht die einfachen Anführungsstriche weg – sie dienen zur Unterscheidung der Tokens von den Operatoren der EBNF.

> **Hinweis**
> Beachten Sie, dass Sie die in der Spezifikation angegebenen kontextuellen Einschränkungen, z.B. Typkompatibilität, hier nicht prüfen müssen. Verwenden Sie eine möglichst kompakte Beschreibung der Syntaxelemente. Die Grammatik darf durch Ihre Produktionen mehrdeutig werden. Ignorieren Sie für diese Aufgabe außerdem § 7.9 der MAVL-Spezifikation: Ihre Produktionen müssen nicht die Mehrfachanwendung von unären Operatoren ausschließen.

Terminalsymbole	**Bedeutung**
`ID`	Berzeichner, z. B. `someVar_42`.
`INT`, `FLOAT`, `BOOL`, `STRING`	Vorzeichenlose Literale des entsprechenden Typs, z. B. `17`, `3.14`, `false`, `"foo"`.
`'function'`, `'val'`, `'var'`, `'for'`, `'if'`, `'else'`, `'return'`, `'foreach'`, `'switch'`, `'case'`, `'default'`, `'record'`	Schlüsselwörter.
`'int'`, `'float'`, `'bool'`, `'void'`, `'string'`, `'matrix'`, `'vector'`	Schlüsselwörter für eingebaute Typen.
`'('`, `')'`, `'{'`, `'}'`, `'['`, `']'`, `'<'`, `'>'`	Klammern.

a) Das Nichtterminal `expr` dient zur Erkennung eines beliebig komplexen Ausdrucks wie z. B. `a * b + (c - m[3][1])`. In einer kontextfreien Grammatik lassen sich Ausdrücke als rekursive Produktionen beschreiben: Ein Ausdruck ist entweder ein atomarer Ausdruck oder entsteht durch die Anwendung eines Operators auf weitere (atomare oder zusammengesetzte) Ausdrücke.

`expr` wird durch folgende Produktion beschrieben:

```
expr ::= ID | INT | FLOAT | STRING
       | '(' expr ')'
       | ...
       | mulExpr
       | subvectorExpr
       | recordElementSelectExpr
       | ...
```

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`-Werte als Parameter und ist Linksassoziativ. Also kann man diesen grammatikalisch folgendermaßen definieren.

```
mulExpr ::= (INT | FLOAT) '*' expr
```

Die `subvectorExpr` ist in dem Sprachstandard unter *§ 7.7.4 Submatrix und Subvektor* definiert. Hier wird definiert, dass eine solche Beispielsweise als `v{-1:i:1}` geschrieben werden kann, wobei `v` ein Vektor und `i` eine Zahl sein muss. Dieser Ausdruck extrahiert einen Subvektor mit den Elementen $[i-1, i+1]$.

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

Unter *§ 7.8 Selektion von Record-Elementen* ist definiert, wie der Syntax funktioniert.

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

b) Das Nichtterminal `type` dient zur Erkennung von Typbezeichnern wie z. B. `vector<float>[5]`. `type` wird durch folgende Produktion beschrieben:

```
type ::= primitiveType
       | vectorType
       | ...
```

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.

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

c) Das Nichtterminal `statement` dient zur Erkennung von Befehlen und wird durch folgende Produktion beschrieben:

```
statement ::= returnStmt
            | varDecl
            | callStmt
            | forStmt
            | ...
```

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

Aufgabe 1.2: AST zu MAVL

Abstrakte Syntaxbäume (engl. *Abstract Syntax Trees, AST*) sind eine weitverbreitete Zwischendarstellung, die nur essentielle Informationen enthält und Details der konkreten Syntax einer Programmiersprache abstrahiert.

In dieser Aufgabe zeigen wie 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.

a) Geben Sie den zum folgenden AST zugehörigen MAVL-Code an.

Den Syntaxbaum kann man, von oben nach unten und links nach rechts, einfach wie Code lesen.

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