

# Einführung in den Compilerbau

Prof. Dr.-Ing. Andreas Koch  
David Volz, Tim Noack, Jonas Renk



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Praktikum  
Wintersemester 23/24  
Übungsblatt 1

Abgabe bis Sonntag, 26.11.2023, 18:00 Uhr (MEZ)

---

## Einleitung

Im Rahmen dieses ersten Praktikums werden Sie einen rekursiven Abstiegsparser für MAVL implementieren. Grundlage dafür soll die LL (1) -Grammatik in Listing 1 bilden.

Wir stellen Ihnen eine Compilerumgebung zur Verfügung, die Sie als Archiv im Moodle-Kurs finden. Eine Anleitung, die Ihnen zeigt, wie Sie das Projekt auf Ihrem Rechner einrichten, bauen und ausführen können, ist im Archiv enthalten. Zusätzlich steht die Anleitung auch im Moodle-Kurs zur Ansicht bereit. Weitere Informationen zum Framework finden Sie in unseren Anleitungsvideos<sup>1</sup>.

In den Aufgaben 1.1 bis 1.7 werden Sie fehlende Methoden des rekursiven Abstiegsparsers ergänzen. Der Parser soll auf einem Tokenstrom (erzeugt von der Klasse `Scanner`) arbeiten und einen AST gemäß der Klassen aus dem Paket `mavlc.syntax.*` aufbauen. Die zugehörige Javadoc-Dokumentation finden Sie im Unterordner `doc` des von uns bereitgestellten Projekts.

Achten Sie bei Ihrer Implementierung auch auf einen gut verständlichen und lesbaren Code-Stil und dokumentieren Sie Ihre Abgabe mit ausreichend Kommentaren!

Bitte schreiben Sie Ihre Gruppennummer, sowie die Namen und Matrikelnummern aller Gruppenmitglieder, als Kommentar in *jede* abzugebende Datei.

**Unter keinen Umständen dürfen Sie die Signaturen der zu implementierenden Methoden verändern!**

---

## Testen und Bewertung

Um Ihre Implementierung zu testen, stellen wir Ihnen eine Reihe von öffentlichen Testfällen bereit, die die Ausgabe Ihres Compilers mit dem erwarteten Output vergleichen. Die erforderlichen Schritte zum Ausführen der Tests finden Sie in der Anleitung. Wenn Ihre Implementierung alle bereitgestellten öffentlichen Testfälle besteht<sup>2</sup>, erhalten Sie **mindestens 40** der erreichbaren 80 Punkte.

Im Rahmen der Bewertung durch Ihre Tutoren werden wir Ihren Compiler weiteren nicht-öffentlichen Tests unterziehen, deren Ergebnis Ihre Punktzahl ergibt. **Daher ist es unabdingbar, dass Sie Ihre Implementierung mit eigenen**

---

<sup>1</sup>[https://www.youtube.com/playlist?list=PLz70m4N3m7xvSCMk\\_FhW0Tl8EKklgLkrb](https://www.youtube.com/playlist?list=PLz70m4N3m7xvSCMk_FhW0Tl8EKklgLkrb)

<sup>2</sup>Falls Sie die Testfälle in Ihrer Entwicklungsumgebung ausführen, stellen Sie bitte sicher, dass Ihre Abgabe auch mit `gradle test` funktioniert.

---

**MAVL-Beispielprogrammen gründlich testen!** Um die Ausgabe Ihres Compilers zu überprüfen, können Sie, wie in der Anleitung beschrieben, den AST in das Graphviz DOT-Format exportieren.

Einige IDEs ergänzen die zu importierenden Bibliotheken automatisch. Beachten Sie daher bei der Abgabe, dass keine Ihrer Dateien ungenutzte imports von Bibliotheken beinhaltet, die möglicherweise nicht auf dem Korrekturrechner vorhanden sind.

---

## Arbeit im Team

---

Es ist Ihnen freigestellt, wie Sie die gemeinsame Arbeit am Code organisieren (gemeinsam an einem Rechner, SVN, Git, ...). Falls Sie ein Versionsverwaltungssystem benutzen, stellen Sie jedoch bitte sicher, dass Ihr **Repository nicht öffentlich zugänglich** ist. Eine Missachtung dieser Regel wird als Täuschungsversuch gewertet und führt zu einer Bewertung der Abgabe mit **0 P.**

---

## Abgabe

---

Nutzen Sie zur Vorbereitung des Abgabearchivs bitte unbedingt die in der Anleitung beschriebenen Kommandos. Beachten Sie, dass Ihre Abgabe **nur die Klasse Parser** umfasst. **Nehmen Sie daher keine Änderungen an anderen von uns bereitgestellten Quelldateien vor, da wir Ihren Parser gegen die Originalversionen der übrigen Klassen testen werden!**

---

## Fehlermeldungen

---

Für einen Compiler sind korrekte und informative Fehlermeldungen elementar. Nutzen Sie in den folgenden Aufgaben Instanzen der Klasse `SyntaxError`, um dem Nutzer Syntaxfehler im Eingabeprogramm anzuzeigen. Detaillierte Informationen zum Interface der Klasse `SyntaxError` finden Sie in der Javadoc-Dokumentation. **Korrekte und vollständige Fehlermeldungen sind Teil der Tests!**

---

## Zugehörige Vorlesungen und Folien

---

Zur Bearbeitung dieser Aufgabe sollten Sie im Kapitel 2 (Syntaktische Analyse) die Folien bis einschließlich **Folie 74** bearbeitet haben.

---

## Aufgabe 1.1: Variablen- und Wertdefinitionen (14 Punkte)

---

In dieser Aufgabe sollen Sie den Parser für MAVL so erweitern, dass Variablendeklarationen (`var`), Wertdefinitionen (`val`) und Variablenzuweisungen verarbeitet werden können. Implementieren Sie dafür die folgenden 3 Methoden in der Klasse `Parser`.

- `parseValueDef()`
- `parseVarDecl()`
- `parseAssign(String name, SourceLocation location)`

---

### Aufgabe 1.2: Arithmetische Ausdrücke (16 Punkte)

---

Im Rahmen dieser Teilaufgabe soll das Parsing von arithmetischen Ausdrücken mit den Operationen Addition, Subtraktion, Multiplikation, Division, Exponentiation und Vergleich von zwei Werten implementiert werden. Füllen Sie dazu die folgenden 4 Methoden in Parser.

- `parseAddSub()`
- `parseMulDiv()`
- `parseExponentiation()`
- `parseCompare()`

Beachten Sie bei Ihrer Implementierung unbedingt auch die korrekte Operatorpräzedenz, wie in der Grammatik (Listing 1) und in Tabelle 3 in der MAVL Sprachspezifikation gegeben. Berücksichtigen Sie auch die Rechtsassoziativität des Potenz-Operators in MAVL.

Ergänzen Sie außerdem, unter Beachtung der Operatorpräzedenzen, die fehlenden Aufrufe in den Methoden `parseNot()`, `parseUnaryMinus()` und `parseTranspose()`.

---

### Aufgabe 1.3: Vektor- und Matrixindizierung (6 Punkte)

---

Erweitern Sie den MAVL-Parser so, dass Indezierungsausdrücke `v[1][2]` verarbeitet werden können. Implementieren Sie dafür die Methode `parseElementSelect()`.

---

### Aufgabe 1.4: For-Schleife (4 Punkte)

---

Für diese Aufgabe sollen Sie den MAVL-Parser so erweitern, dass For-Schleifen verarbeitet werden können. Fügen Sie dafür den passenden Code in die Methode `parseFor()` ein. Auch hier darf die Methodensignatur nicht verändert werden!

---

### Aufgabe 1.5: ForEach-Schleife (8 Punkte)

---

Implementieren Sie analog die Methoden `parseForEach()` und `parseIteratorDeclaration()`, sodass ForEach-Schleifen verarbeitet werden können.

---

### Aufgabe 1.6: Funktionsaufrufe (14 Punkte)

---

Ermöglichen Sie das Parsen von Funktionsaufrufen mit den folgenden drei Methoden:

- `parseCall(String name, SourceLocation location)`
- `parseReturn()`
- `parseAssignOrCall()`

---

### Aufgabe 1.7: Switch-Statement (18 Punkte)

---

Ermöglichen Sie das Parsen des MAVL `switch`-Statements in den folgenden drei Methoden:

- `parseSwitch()`
- `parseCase()`
- `parseDefault()`

Listing 1: LL (1)-Grammatik für MAVL

```

module      ::= (function | recordTypeDecl)*
function    ::= 'function' type ID '(' ( formalParameter (',' formalParameter)* ) ? ')' '{
    ' statement* '}'
recordTypeDecl ::= 'record' ID '{' recordElemDecl+ '}'
recordElemDecl ::= ( 'var' | 'val' ) type ID ';';
formalParameter ::= type ID
type        ::= 'int' | 'float' | 'bool' | 'void' | 'string'
              | 'vector' '<' ( 'int' | 'float' ) '>' '[' expr ']'
              | 'matrix' '<' ( 'int' | 'float' ) '>' '[' expr ']' '[' expr ']'
              | ID


statement   ::= valueDef
              | varDecl
              | return
              | assignOrCall
              | for
              | foreach
              | if
              | compound
              | switch

valueDef    ::= 'val' type ID '=' expr ';';
varDecl     ::= 'var' type ID ';';
return      ::= 'return' expr ';';
assignOrCall ::= ID ( assign | call ) ';';
assign      ::= ( ( '[' expr ']' ( '[' expr ']' )? ) | '@' ID )? '=' expr
call        ::= '(' ( expr ( ',' expr )* )? ')'
for         ::= 'for' '(' ID '=' expr ';' expr ';' ID '=' expr ')' statement
foreach     ::= 'foreach' '(' iteratorDecl ':' expr ')' statement
iteratorDecl ::= ( 'var' | 'val' ) type ID
if          ::= 'if' '(' expr ')' statement ( 'else' statement )?
switch      ::= 'switch' '(' expr ')' '{' switchSection* '}'
switchSection ::= case
                | default

case        ::= 'case' expr ':' statement
default     ::= 'default' ':' statement
compound    ::= '{' statement* '}'

expr        ::= select
select      ::= or ( '?' or ':' or )?
or          ::= and ( '|' and ) *
and         ::= not ( '&' not ) *
not         ::= '!' ? compare
compare     ::= addSub ( ( '>' | '<' | '<=' | '>=' | '==' | '!=' ) addSub ) *
addSub      ::= mulDiv ( ( '+' | '-' ) mulDiv ) *
mulDiv      ::= unaryMinus ( ( '*' | '/' ) unaryMinus ) *
unaryMinus  ::= '-' ? exponentiation
exponentiation ::= dotProd ( '^' dotProd ) *
dotProd     ::= matrixMul ( '.' matrixMul ) *
matrixMul   ::= transpose ( '#' transpose ) *
transpose   ::= '~' ? dim
dim         ::= subrange ( '.rows' | '.cols' | '.dimension' )?
subrange    ::= elementSelect ( '{' expr ':' expr ':' expr '}' ( '{' expr ':' expr ':'
    expr '}' )? )?
elementSelect ::= recordElemSel ( '[' expr ']' ) *
recordElemSel ::= atom ( '@' ID )?
atom          ::= INT | FLOAT | BOOL | STRING
              | ID ( call )?
              | '(' expr ')'
              | ('@' ID)? initializerList

```



---

```
initializerList ::= '[' expr ( ',' expr )* ']'
```