Praktikum zu Einführung in den Compilerbau

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

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 17/18
Praktikum 1

Abgabe bis Sonntag, 19.11.2017, 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.

In Aufgabe 1.1 -Aufgabe 1.6 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.ast.*` aufbauen. Die zugehörige Javadoc-Dokumentation finden Sie im Unterordner `doc` des von uns bereitgestellten Projekts sowie online unter `https://www.esa.informatik.tu-darmstadt.de/campus/mavl/`.

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

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, 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 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.

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. Detailierte Informationen zum Interface der Klasse `SyntaxError` finden Sie in der Javadoc-Dokumentation. **Korrekte und vollständige Fehlermeldungen sind Teil der Tests!**

Aufgabe 1.1 Variablen- und Wertdefinitionen *15 P*

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 Funktionen in der Klasse `Parser`.

- `parseValueDef()`
- `parseVarDecl()`
- `parseAssign(String name, int line, int column)`

Aufgabe 1.2 Arithmetische Ausdrücke *15 P*

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

- `parseAddSub()`
- `parseMulDiv()`
- `parseExponentation()`
- `parseCompare()`

Beachten Sie bei Ihrer Implementierung unbedingt auch die korrekte Operatorpräzedenz, wie in der Grammatik (Listing 1) und in Tabelle 2 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()` und `parseUnaryMinus()`.

Aufgabe 1.3 Ternärer Operator *5 P*

Implementieren Sie die Unterstützung für den ternären-Operator (`? :`) im Parser, indem Sie Methode `parseSelect()` implementieren.

Aufgabe 1.4 Records *20 P*

In dieser Aufgabe sollen Sie den Parser für MAVL so erweitern, dass Record-Typdeklarationen und Zugriffe auf einzelne Elemente von Records verarbeitet werden können. Implementieren Sie dafür die folgenden 3 Funktionen in der Klasse `Parser`.

- `parseRecordTypeDeclaration()`
- `parseRecordElementDeclaration()`
- `parseRecordElementSelect()`

Zusätzlich sind Modifikationen an der Methode `parseAssign(String name, int line, int column)` erforderlich, die Sie im Rahmen von Teilaufgabe 1.1 implementiert haben.

Aufgabe 1.5 Switch-Statement *15 P*

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

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

Aufgabe 1.6 Schleifen *10 P*

MAVL unterstützt `foreach`-Schleifen. Implementieren Sie die Methoden `parseForEach()` und `parseIteratorDeclaration()`, so dass Schleifen vom Parser verarbeitet werden können, ohne dabei die Methodensignatur zu verändern.

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' ) '>' '[' INT ']'
                 |  'matrix' '<' ('int' | 'float' ) '>' '[' INT ']' '[' INT ']'
                 |  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')' '{' singleCase* '}'
singleCase      ::=  case
                 |   default
case            ::= 'case' '-'? INT ':' 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      ::= '-' ? pow
pow             ::= (dim '^')* dim
dim             ::= dotProd ( '.yDimension' | '.xDimension' | '.dimension' )?
dotProd         ::= matrixMul ( '.*' matrixMul )*
matrixMul       ::= subrange ( '#' subrange )*
subrange        ::= elementSelect ( '{' expr ':' expr ':' expr '}' ( '{' expr ':' expr ':' expr '}' )? )?
elementSelect   ::= recordElemSel ( '[' expr ']' )*
recordElemSel   ::= atom ( '@' ID )?
atom            ::= INT | FLOAT | BOOL | STRING
                 |  ID ( call )?
                 |  '(' expr ')'
                 |  ('@' ID)? '[' expr ( ',' expr )* ']'
```