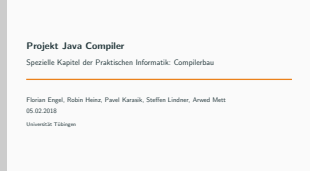
Projekt Java Compiler

Spezielle Kapitel der Praktischen Informatik: Compilerbau

Florian Engel, Robin Heinz, Pavel Karasik, Steffen Lindner, Arwed Mett

05.02.2018

Universität Tübingen



Agenda

Test-Framework

Parser

Code Generator

Agenda
Test-Framework
Parser
Code Generator

Allgemein

Aufgabenstellung:

Entwickeln eines Mini-Java Compilers mit den zugehörigen Schritten: Lexer, Parser, TypChecker und Codegenerierung.

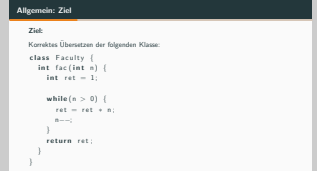
Allgemein
Aufgabenstellung:
Entwickeln eines Mini-Java Compilers mit den zugehörigen Schritten: Lexer, Parser, TypChecker und Codegenerierung.

Allgemein: Ziel

Ziel:

Korrektes Übersetzen der folgenden Klasse:

```
class Faculty {
  int fac(int n) {
    int ret = 1;

    while(n > 0) {
      ret = ret * n;
      n--;
    }
    return ret;
  }
}
```

Test-Framework

Test-Framework

Test-Framework

Das Test-Framework wurde selbst implementiert. Es enthält diverse Funktionen zum automatisierten überprüfen der Testfälle.

Tests werden in korrekte und falsche Testfälle unterschieden.

Test-Framework
Das Test-Framework wurde selbst implementiert. Es enthält diverse Funktionen zum automatisierten überprüfen der Testfälle.
Tests werden in korrekte und falsche Testfälle unterschieden.

Test-Suite: Token-Coverage & Testfälle

Die Test-Suite umfasst eine Token-Coverage von 100%.

Zusätzlich umfasst die Test-Suite insgesamt 21 gültige und 12 ungültige Testfälle.

Ungültige Testfälle werden in Syntaxfehler (Parser) und Typfehler (Typchecker) unterschieden.

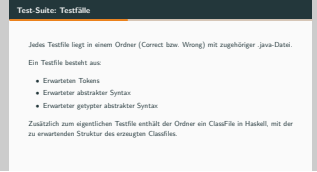
Test-Suite: Testfälle

Jedes Testfile liegt in einem Ordner (Correct bzw. Wrong) mit zugehöriger .java-Datei.

Ein Testfile besteht aus:

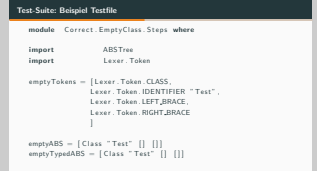
- Erwarteten Tokens
- Erwarteter abstrakter Syntax
- Erwarteter getypter abstrakter Syntax

Zusätzlich zum eigentlichen Testfile enthält der Ordner ein ClassFile in Haskell, mit der zu erwartenden Struktur des erzeugten Classfiles.



Test-Suite: Beispiel Testfile

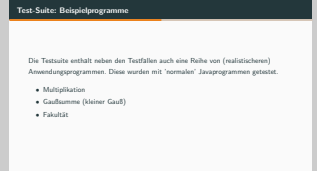
```
module Correct.EmptyClass.Steps where

import          ABSTree
import          Lexer.Token

emptyTokens = [Lexer.Token.CLASS,
               Lexer.Token.IDENTIFIER "Test",
               Lexer.Token.LEFT_BRACE,
               Lexer.Token.RIGHT_BRACE
               ]

emptyABS = [Class "Test" [] []]
emptyTypedABS = [Class "Test" [] []]
```

Test-Suite: Beispielprogramme

Die Testsuite enthält neben den Testfällen auch eine Reihe von (realistischeren) Anwendungsprogrammen. Diese wurden mit 'normalen' Javaprogrammen getestet.

- Multiplikation
- Gaußsumme (kleiner Gauß)
- Fakultät



Parser

Parser

Lexer

Lexer

Parser - Operatoren Priorität

```
%right in
%right ASSIGN ADD ...
%right QUESTIONMARK COLON
%left OR
 ...
%nonassoc LESSER GREATER LESSER_EQUAL...
 ...
%nonassoc INCREMENT DECREMENT
```

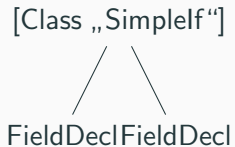

Parser - Operatoren Priorität
%right in
%right ASSIGN ADD ...
%right QUESTIONMARK COLON
%left OR
...
%nonassoc LESSER GREATER LESSER_EQUAL...
...
%nonassoc INCREMENT DECREMENT

Struktur Happy File

```
Program
    : Class                 { [$1] }
    | Program Class         { $1 ++ [$2] }
    | Program SEMICOLON     { $1 }

Statement
    : SingleStatement SEMICOLON             { $1 }
      ...
    | IF LEFT_PARANTHESES Expression RIGHT_PARANTHESES
        Statement ELSE Statement
                                            { If $3 $5 (Just $7) }
    | IF LEFT_PARANTHESES Expression
        RIGHT_PARANTHESES Statement
        %prec THEN                          { If $3 $5 Nothing }
    | Switch                                { $1 }
```


Beispiel

```
class SimpleIf {
        int i;
        void dolf() {
                int a;
                a = 5;
                i = 0;
                if(a < 5) {
                        i = a;
                }
                else {
                        i = 2;
                }
        }
}
```



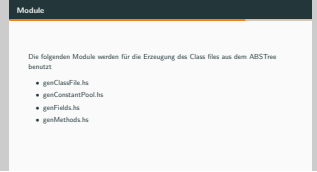
Code Generator

Code Generator

Module

Die folgenden Module werden für die Erzeugung des Class files aus dem ABSTree benutzt

- genClassFile.hs
- genConstantPool.hs
- genFields.hs
- genMethods.hs



Module

In den nachfolgenden Modulen sind die Datentypen enthalten die für den abstrakten Bytecode benutzt werden

- Data/Assembler.hs
- Data/ClassFile.hs

Aus dem abstrakten Bytecode wird im Module Module BinaryClass.hs der Bytecode erzeugt

Module
In den nachfolgenden Modulen sind die Datentypen enthalten die für den abstrakten Bytecode benutzt werden
- Data/Assembler.hs
- Data/ClassFile.hs
Aus dem abstrakten Bytecode wird im Module Module BinaryClass.hs der Bytecode erzeugt

Constanten Pool

Der Constantenpool ist in einer hashMap die ein Eintrag auf dessen Position abbildet.

Im Module genConstantPool.hs sind Funktionen enthalten die ein Eintrag erzeugen und dessen Position zurückgeben bzw nur die Position zurückgeben.

Constanten Pool
Der Constantenpool ist in einer hashMap die ein Eintrag auf dessen Position abbildet.
Im Module genConstantPool.hs sind Funktionen enthalten die ein Eintrag erzeugen und dessen Position zurückgeben bzw nur die Position zurückgeben.

Beispiel genConstantPool

```
genMethodRefSuper :: String
                  -> Type
                  -> State ClassFile IndexConstantPool
genMethodRefSuper name typ =
  do indexClassName <- view (super . indexSp) <$> get
     indexNameType <- genNameAndType name typ
     genInfo MethodRefInfo
               { _tagCp             = TagMethodRef
               , _indexNameCp       = indexClassName
               , _indexNameandtypeCp = indexNameType
               , _desc              = ""
               }
```


Beispiel genConstantPool
genMethodRefSuper :: String
-> Type
-> State ClassFile IndexConstantPool
genMethodRefSuper name typ =
do indexClassName <- view (super . indexSp) <$> get
indexNameType <- genNameAndType name typ
genInfo MethodRefInfo
{ _tagCp = TagMethodRef
, _indexNameCp = indexClassName
, _indexNameandtypeCp = indexNameType
, _desc = ""
}

Generieren der Methoden

Bei der Generierung der Methoden werden auch gleichzeitig die Einträge im constanten pool erstellt. Im State wird folgender Datentyp verwendet.

```
data Vars
  = Vars { _localVar :: [HM.HashMap LocVarName LocVarIndex]
         , _allLocalVar :: S.Set LocVarName
         , _classFile :: ClassFile
         , _curStack :: Int
         , _maxStack :: Int
         , _line :: LineNumber
         , _continueLine :: [LineNumber]
         }
makeLenses ''Vars
```


Als Major Version wird 48 anstatt 53 verwendet da die StackMapTable nicht implemntiert wurde

Als Major Version wird 48 anstatt 53 verwendet da die StackMapTable nicht implemntiert wurde