



Compilerbau - INF3339c WS 2023/24

Jan Keller
Timon Martins
Eric Sabel
Timo Schmidberger



Allgemeiner Aufbau



Scanner

Parser

Typecheck


Constant pool

Bytecode



Scanner

- Erkennen/ Ignorieren von Kommentaren
 - inline (`// ...`)
 - multiline (`/* ... */`)
- Position jedes Tokens (`-> [PositionedToken]`)
- Validierung von Tokens
 - unclosed Char (`char c = 'a;`)
 - integer too large (`int i = 2.147.483.648;`)



```
public class Main {  
    int a = 0;  
    int b = 0;  
}
```

wird eingelesen als:

```
"public class Main { \n    int a = 0; \n    int b = 0; \n}"
```

Scanner wandelt zunächst um in:

```
[PUBLIC, SPACE, CLASS, SPACE, NAME("main"), {, NEWLINE ...]
```

und nach der Positionsbestimmung in:

```
[(PUBLIC,pos), (CLASS,pos), (NAME("main"),pos), ({,pos), ...]
```



Parser

Aufbau durch Parser-Kombinatoren

```
type Parser toks a = [toks] -> [(a, [toks])]
```

```
[Tok-1, Tok-2, Tok-2, ... Tok-n] -> [(AST, [])]
```



```
[PUBLIC, CLASS, NAME(main), {, BODY, }]
```

```
lexem(PUBLIC) [...] -> [(PUBLIC, [CLASS, NAME(main), {, BODY, }])]
```

```
lexem(PUBLIC) ++ lexem(CLASS) [...]  
-> [(PUBLIC, (CLASS, [Name(main), {, BODY, }]))]
```

```
lexem(PUBLIC) ++ lexem(CLASS) ++ lexem(name) ++ lexem({) .. [...]  
-> [(PUBLIC, (CLASS, ((name(Main), ({, (Body, ({, [])))))])]
```

Bauen der AST-Struktur

```
parseClass <<< \(_, (_, (cname, (_, (cbody, _))))  
-> Class{ name = cname, body = body}
```

```
-> [(Class {name = main, body = BODY}, [] ) ]
```

Parser - Operatorpräzedenz

~~$Expr = \dots \mid BinExpr \mid \dots$~~

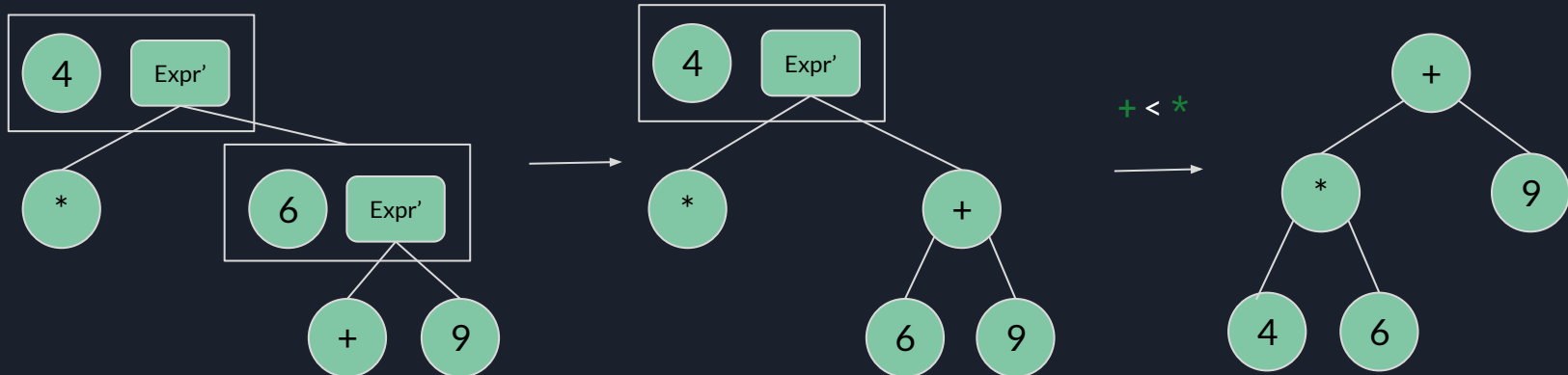
~~$BinExpr = Expr + Expr$~~

$Expr = TExpr : Expr'$
 $\mid TExpr$

$TExpr = \dots \mid Intliteral \mid \dots$

$Expr' = \dots \mid + Expr \mid \dots$

Beispiel: 4 * 6 + 9





Typecheck

Überladung/Vererbung

Schöne Fehler

Standardbibliothek

Umsetzung



Feature: Überladung/Vererbung

Code-Beispiel:

```
class Super() {}
```

```
class Sub extends Super() {}
```




Feature: Überladung/Vererbung

Code-Beispiel:

B.foo();

```
class A {  
    static foo() {};  
}  
  
class B extends A {  
    static foo(int i);  
}
```




Feature: Überladung/Vererbung

Code-Beispiel:

```
B.foo(new Sub());
```

```
class A {  
    static foo(Sub s) {};  
}  
  
class B extends A {  
    static foo(Super s) {};  
}
```



Feature: Überladung/Vererbung

Code-Beispiel:

```
B.foo(new Sub());
```

```
class A {  
    private static foo(Sub s) {};  
}  
  
class B extends A {  
    static foo(Super s) {};  
}
```



Feature: Überladung/Vererbung

Erkennung von Fehlern

- `@Override` muss Methode überschreiben
- Return-Typ darf nicht spezieller sein
- Sichtbarkeit darf nicht verringert werden
- Etc.

Feature: Schöne Fehler

Error: Expected right hand side to be a subtype of int, but has type java/io/PrintStream.

```
2 |  
3 | class Error {  
4 |     // some init stuff  
5 |     int i = System.out;  
6 |  
7 |     public static void main(String[] args) {  
8 |         // some code
```



Feature: Standardbibliothek

`System.out.println(...);`

=> resolve System to java/lang/System

```
printStream :: Class
printStream = Class
{ cposition      = AutoGenerated
, caccess       = withAuthGen Public
, cname         = withAuthGen "java/io/PrintStream"
, cextends      = Just $ withAuthGen "java/lang/Object"
, cfields       = []
, cmethods      =
  [ printFunc "print"      Type.Char,
    printFunc "print"      Type.Int,
    printFunc "print"      $ Type.Instance "java/lang/Object",
    printFunc "println"    Type.Bool,
    printFunc "println"    Type.Char,
    printFunc "println"    Type.Int,
    printFunc "println"    $ Type.Instance "java/lang/Object"
  ]
, cconstructors = [ defConstructor "java/io/PrintStream" ]
}
```



Umsetzung

1

Invarianten prüfen

⋮

Zyklische Vererbung
Valide Typ Klassen-Referenzen

2

AST modifizieren

⋮

Default-Konstruktor einfügen
Instanz-Felder initialisieren

3

AST übersetzen

⋮

Stmts/Exprs übersetzen
Typen berechnen



Aufbau des Constant Pools's



Allg. Klassen
Informationen

Feld
Informationen

konstruktoren

Methoden

Aufbau des Constant Pool's

```
class A {  
}  
  
class B extends A {  
    B() {  
        super();  
    }  
    int i;  
  
    bool f(int) {  
        C a = new C();  
        int b = a.h();  
        return true;  
    }  
}
```

1. Utf-8 "Code"
2. Utf-8 "B"
3. ClassRef #2 // Class B
4. Utf-8 "A"
5. ClassRef #4 // Class A
6. Utf-8 "I"
7. Utf-8 "i"
8. NameAndType #7:#6
9. FieldRef #3:#8 // Field int i
10. Utf-8 "<init>"
11. Utf-8 "()V"
12. NameAndType #10:#11
13. MethodRef #3:#12 // Constructor B
14. MethodRef #5:#12 // Constructor A
15. Utf-8 "f"
16. Utf-8 "(I)Z"
17. NameAndType #15:#16
18. MethodRef #3:#17
19. Utf-8 "C"
20. ClassRef #19
21. MethodRef #20:#12
22. Utf-8 "h"
23. Utf-8 "()I"
24. NameAndType #22:#23
25. MethodRef #21:#24
26. Integer "1";



Suchen im Constant Pool

Suchen nach der Klasse BeispielKlasse

- Suche Utf-8 "BeispielKlasse"
→ Index : 1
- Suche Klasse mit Utf-8 referenz auf #1

⇒ #2 Klasse BeispielKlasse

1: Utf-8 "BeispielKlasse"
2: Klasse #1
3: Utf-8 "BeispielFeld"
4: Utf-8 "BeispielMethode"
5: Utf-8 "(IZLB;)V"
6: Utf-8 "Ljava/lang/String;"
7: Klasse #6
8: Utf-8 "BeispielFeld"
9: Utf-8 "NeuerString"
10: NameAndType #8:#6
11: NameAndType #4:#5
12: MethodRef #2:#11
13: Fieldref #3:#10
14: Utf-8 "neueMethode"
15: Methodref #7:#11



Suchen im Constant Pool

Suchen nach der Methode

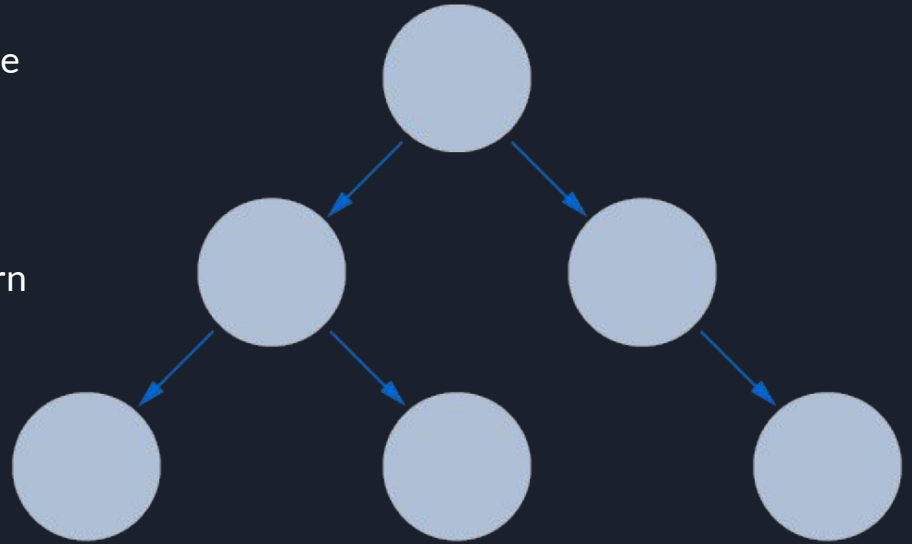
BeispielMethode(int, bool, B) -> void

- Suche Utf-8 "BeispielMethode"
→ Index : 4
- Suche Utf-8 "(IZLB;)V"
→ Index : 5
- Suche NameAndType "#4:#5"
→ Index : 11
- Suche MethodRef #2:#11(siehe Bsp1.)
⇒ Eindeutig #12

1: Utf-8 "BeispielKlasse"
2: Klasse #1
3: Utf-8 "BeispielFeld"
4: Utf-8 "BeispielMethode"
5: Utf-8 "(IZLB;)V"
6: Utf-8 "Ljava/lang/String;"
7: Klasse #6
8: Utf-8 "BeispielFeld"
9: Utf-8 "NeuerString"
10: NameAndType #8:#6
11: NameAndType #4:#5
12: MethodRef #2:#11
13: Fieldref #3:#10
14: Utf-8 "neueMethode"
15: Methodref #7:#11

Bytecode-Generierung

- Verwendung des JVM-Moduls
 - ClassFile -> Method_Info -> AttributeCode
- Erfassung aller lokalen Variablen aus dem TAST
- Rekursive Bytecode-Generierung mittels Pattern Matching aus dem TAST
- Berechnung der Max-Stacksize aus erstelltem Bytecode





Lokalen Variablen

- Anzahl der lokalen Variablen für jede Methode muss angegeben werden
- Informationen werden für die Bytecode-Generierung benötigt
 - Erstellung eines Arrays mit lokalen Variablen
- Bei nicht-statischen Methoden:
 - "This" befindet sich an der Stelle null im Array
 - Gefolgt von den Parametern und den innerhalb der Methode definierten Variablen
- Unterschied bei statischen Methoden
 - "This" befindet sich nicht an der Stelle null im Array
 - Array beginnt direkt mit den Parametern der Methode
- Erkennung der innerhalb der Methode definierten Variablen
 - TAST wird rekursiv durchsucht
 - Gefundene Variablen werden in dem Array gespeichert

Bytecode-Generierung

```
if(1 == 2) {  
    return;  
}
```

```
Block [  
    If (Binary Bool EQ (Literal Int (IntLit 1))  
      (Literal Int (IntLit 2)))  
      (Block [Return Nothing])  
]
```

19,0,14, (load 1 from cp)

19,0,15, (load 2 from cp)

100, (sub 1 - 2)

153,0,7, (ifeq)

3, (iconst_0)

167,0,4, (goto)

4, (iconst_1)

153,0,4, (ifeq)

177 (return)

...



Max-Stacksize

19,0,14, (load 1)	+1
19,0,15, (load 2)	+1
100, (sub 1 - 2)	-1
153,0,7, (ifeq)	-1
3, (iconst_0)	+1
167,0,4, (goto)	+0
4, (iconst_1)	+1
153,0,4, (ifeq)	-1
177 (return)	+0

Max-Stacksize = 2



Funktionsumfang

- Vererbung
- Überladung
- Instanzvariablen mit Initialisierung
- Dead-Code Erkennung
- Schöne Fehlermeldungen durch Position-Tracking
- Einfügen von Default-Konstruktor, `super ()` ; und `return ;`
- Operatorpräzedenz
- Debug-Modus