Compilerbau
Vorlesung
Wintersemester 2008, 09, 10, 11

Johannes Waldmann, HTWK Leipzig

16. Oktober 2011

Einleitung

Compilerbau

Vorlesung

Wintersemester 2008, 09, 10, 11

Johannes Waldmann, HTWK Leipzig

17. Oktober 2011

Einleitung

Beispiel

Eingabe (Java):

```
{ int i;
  float Beispiel
    float ♭20♯  a,
  float ♭20♯ / b,i
  prod  /  0,
  i  /  1,
  do  ı
     prod  /  prod
          →  a♭i♯    ⇀b♭i♯,
     i  /  i→1,
  ℘  while  ↼i  </  20↽,
℘
```

Ausgabe (Drei-Adress-Code):

```
L1: prod = 0
L3: i =
L4.  t1 /          ⇀  8
     t2  /  a  ♭  t1  ♯
     t3  /  i      ⇀  8
     t4  /  b  ♭  t3  ♯
     t5  /  t2       ⇀  t4
     prod  /  prod  →  t5
L6.  i  /  i  →  1
L5.  if  i  </  20  goto  L4
L2.
```

Inhalt

- Motivation, Hintergründe
- lexikalische und syntaktische Inhalt (Kombinator-Parser)
- te Übersetzung (Attributgrammatiken) • Code-
- statische Typsysteme
- Laufzeitumgebungen

Sprachverarbeitung

- mit Compiler:
  - Quellprogramm → Compiler → Zielprogramm
  - Eingaben → Zielprogramm → Ausgaben
- mit Interpreter:
  - Quellprogramm, Eingaben → Interpreter → Ausgaben
- Mischform:
  - Quellprogramm → Compiler → Zwischenprogramm
  - Zwischenprogramm, Eingaben → virtuelle Maschine → Ausgaben

Compiler und andere Werkzeuge

- Quellprogramm
- Präprozessor → modifiziertes Quellprogramm
- Compiler → ... erprogramm *Compiler und andere Werkzeuge*
- Assembler → verschieblicher Maschinencode
- Linker, Bibliotheken → ausführbares Maschinenprogramm

Phasen eines Compilers

- Zeichenstrom
- lexikalische Analyse → Tokenstrom
- syntaktische Analyse → Syntaxbaum
- semantische Analyse → annotierter Syntaxbaum
- Zwischencode-Erzeugung → Zwischencode
- maschinenunabhängige Optimierungen → Zwischencode
- Zielcode-Erzeugung → Zielcode
- maschinenabhängige Optimierungen → Zielcode

Methoden und Modelle

- le Analyse: reguläre Ausdrücke, Automaten
  Automaten
- syntaktische Analyse: kontextfreie Grammatiken, Kellerautomaten
- semantische Analyse: Attr ... semantische An

Code-Erz bei Registerzuordnung: Graphenfärbung

Anwendungen von Tec

Anwendungen von Tecdes Compilerbaus
- Implementier Implementierh
  öhererProgrammiersprachen
- architekturspezifischeOptimierungen (Parallelisierung, Speicherhierarchien)
- Entwurf neuer Architekturen (RISC, spezielleHardware)
- Progr
  Übersetzungen (Binär-Übersetzer, Hardwaresynthese, Datenbankanfragesprachen)
- Softw
  Softwerkzeuge

Literatur

- Franklyn Turbak, David Gifford, Mark Sheldon: *Design Concepts in Programming Languages*, MIT Press, 2008.
  Design fturbak
- Guy Steele, Gerald Sussman: *Lambda: The Ultimate Imperative*, MIT AI Lab Memo AIM-353, 1976
  (the original 'lambda `http://library.readscheme.org/` *http.library.readscheme.org/page1.html*)
- Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman: *Compilers: Principles, Techniques, and Tools* (2nd edition), Addison-Wesley, 2007,
  `http://dragonbook.stanford.edu/`

Organisation

- pro Woche eine Vorlesung, eine Übung.
- Prüfungszulassung: regelmäßiges und erfolgreiches Bearbeiten von Übungsaufgaben
- Prüfung: Klausur (120 min, keine Hilfsmittel)

Beispiel: Interpreter (I)

arithmetische Ausdrücke:

```
data Exp = Const Int | Plus Exp Exp | Times Exp
    deriving ( Show )
ex1 :: Exp
ex1 = Times ( Plus ( Const 1 ) ( Const 2 ) ) (
value :: Exp -> Int
value x = case x of
    Const i -> i
    Plus x y -> value x + value y
    Times x y -> value x * value y
```

Beispiel: Interpreter (II)

Variablen und Umgebungen:

```
Exp=...
         |LetStringExpExp|RefString
ex2::Exp
ex2=Let"x"(Const3)
     (Times(Ref"x")(Ref"x"))
typeEnv=(String->Integer)
value::Env->Exp->Integer
valueenvx=casexof
    Refn->envn
    Letnxb->value(\m->
      ifn==mthenvalueenvxelseenvm)  b
    Const i -> i
    Plus x y -> value env x + value env y
    Times x y -> value env x* value env y
```

Interpreter (Übung)

(bis Woche 41:)

Verzweigungen mit C-ähnlicher Semantik:

Bedingung ist arithmetischer Ausdruck, verwende 0 als Falsch und alles andere als Wahr.

```
data Exp = ...
         | If Exp Exp Exp
```

(evtl. später:)

neuer AST-Knoten für Mehrfachbindung

```
data Exp = ...
         | Lets [ (String, Exp) ] Exp
```

- Semantik definieren/direkt implementieren
- Semantik durch Transformation in geschachtelte `Let`

Inferenz-Systeme

ein *Inferenz-System* I besteht aus

- Regeln (besteht aus Prämissen, Konklusion)
  Schreibweise $\frac{P_1,\dots,P_n}{K}$
- Axiomen (= Regeln ohne Prämissen)

eine *Ableitung* für F bzgl. I ist ein Baum:

- jeder Knoten ist mit einer Formel beschriftet
- jeder Knoten (mit Vorgängern) entspricht Regel von I
- Wurzel ist mit F beschriftet

Schreibweise: $I \vdash F$

Inferenz-Systeme (Beispiel 2)

- Grundbereich: Zeichenketten aus $\{0,1\}^*$
- Axiom:

$$\frac{}{01}$$

- Regel-Schemata (für jedes u, v):

$$\frac{0u, v0}{u1v}, \quad \frac{1u, v1}{u0v}, \quad \frac{u}{\text{reverse}(u)}$$

Leite 11001 ab. Wieviele Wörter der Länge k sind ableitbar?

Inferenz von Werten

- Grundbereich: Aussagen der Form $\mathsf{wert}(f, b)$ mit $f \in$Formeln, $b \in \mathbb{B} = \{0, 1\}$
- Axiome: $\mathsf{wert}(\mathsf{T}, 1), \mathsf{wert}(\mathsf{F}, 0)$
- Regeln:

$$\frac{\mathsf{wert}(X,1), \mathsf{wert}(Y,1)}{\mathsf{wert}(X \wedge Y, 1)}, \frac{\mathsf{wert}(X,0)}{\mathsf{wert}(X \wedge Y, 0)}, \frac{\mathsf{wert}(Y,0)}{\mathsf{wert}(X \wedge Y, 0)}, \ldots$$

Beispiel: leite $\mathsf{wert}((T \wedge F) \wedge (T \wedge T), 0)$ ab.

Durch ein Inferenzsystem kann man die Semantik von Ausdrücken (Programmen) spezifizieren.
(hier: dynamische Semantik)

Inferenz mit Umgebungen

Motivation: Inferenzsystem für (aussagenlogische) Formeln mit Variablenbindung

- Grundbereich: Aussagen der Form $\mathsf{wert}(E, F, B)$

  (in Umgebung E hat Formel F den Wert B)
- Regeln für Operatoren $\dfrac{\mathsf{wert}(E, X, 1), \mathsf{wert}(E, Y, 1)}{\mathsf{wert}(E, X \wedge Y, 1)}, \ldots$
- Regeln für Umgebungen

$$\frac{}{\mathsf{wert}(E[v := b], v, b)}, \quad \frac{\mathsf{wert}(E, v', b')}{\mathsf{wert}(E[v := b], v', b')} \text{ für } v \neq v'$$

- Regeln für Bindung: $\dfrac{\mathsf{wert}(E, X, b), \mathsf{wert}(E[v := b], Y, c)}{\mathsf{wert}(E, \mathtt{let}\ v = X\ \mathtt{in}\ Y, c)}$

Umgebungen

Umgebung ist (partielle) Funktion von Name nach Wert

Realisierungen: `type Env = String -> Int`

`import Data.Map ; type Env = Map String Int`

Operationen:

- `empty :: Env` leere Umgebung
- `lookup :: Env -> String -> Int`

  Notation: $e(x)$
- `extend :: Env -> String -> Int -> Env`

  Notation: $e[x/v]$

Spezifikation:

- $e[x/v](x) = v, \quad x \neq y \Rightarrow e[x/v](y) = e(y)$

Semantische Bereiche

bisher: Wert eines Ausdrucks ist Zahl.

jetzt erweitern (Motivation: if-then-else mit richtigem Typ):

```
data Val = ValInt Int
         | ValBool Bool
```

Dann brauchen wir auch

- `data Val = ... | ValErr String`
- vernünftige Notation (Kombinatoren) zur Einsparung von Fallunterscheidungen bei Verkettung von Rechnungen

```
with_int  :: Val -> (Int  -> Val) -> Val
```

Continuations

Programmablauf-Abstraktion durch Continuations:

Definition:

```
with_int  :: Val -> (Int  -> Val) -> Val
with_int v k = case v of
    ValInt i -> k i
    _ -> ValErr "expected ValInt"
```

Benutzung:

```
value env x = case x of
    Plus l r ->
        with_int ( value env l ) $ \ i ->
        with_int ( value env r ) $ \ j ->
        ValInt ( i + j )
```

Aufgabe: if/then/else mit `with_bool`