Programmierparadigmen

0. Auflage, 20. Februar 2014 Martin Thoma

Vorwort

Dieses Skript wird/wurde im Wintersemester 2013/2014 von Martin Thoma geschrieben. Das Ziel dieses Skriptes ist vor allem in der Klausur als Nachschlagewerk zu dienen; es soll jedoch auch vorher schon für die Vorbereitung genutzt werden können und nach der Klausur als Nachschlagewerk dienen.

Ein Link auf das Skript ist unter
`martin-thoma.com/programmierparadigmen`
zu finden.

Anregungen, Verbesserungsvorschläge, Ergänzungen

Noch ist das Skript im Aufbau. Es gibt viele Baustellen und es ist fraglich, ob ich bis zur Klausur alles in guter Qualität bereitstellen kann. Daher freue ich mich über jeden Verbesserungsvorschlag.

Anregungen, Verbesserungsvorschläge und Ergänzungen können per Pull-Request gemacht werden oder mir per Email an info@martin-thoma.de geschickt werden.

Was ist Programmierparadigmen?

TODO

Erforderliche Vorkenntnisse

Grundlegende Kenntnisse vom Programmieren, insbesondere mit Java, wie sie am KIT in „Programmieren“ vermittelt werden, werden vorausgesetzt. Außerdem könnte ein grundlegendes Verständnis für das O-Kalkül aus „Grundbegriffe der Informatik“ hilfreich sein.

Die Unifikation wird wohl auch in „Formale Systeme“ erklärt; das könnte also hier von Vorteil sein.

Inhaltsverzeichnis

1 Programmiersprachen

Im folgenden werden einige Begriffe definiert anhand derer Programmiersprachen unterschieden werden können.

Definition 1

Eine **Programmiersprache** ist eine formale Sprache, die durch eine Spezifikation definiert wird und mit der Algorithmen beschrieben werden können. Elemente dieser Sprache heißen **Programme**.

Ein Beispiel für eine Sprachspezifikation ist die *Java Language Specification*.[1] Obwohl es kein guter Stil ist, ist auch eine Referenzimplementierung eine Form der Spezifikation.

Im Folgenden wird darauf eingegangen, anhand welcher Kriterien man Programmiersprachen unterscheiden kann.

1.1 Abstraktion

Wie nah an den physikalischen Prozessen im Computer ist die Sprache? Wie nah ist sie an einer mathematisch / algorithmischen Beschreibung?

Definition 2

Eine **Maschinensprache** beinhaltet ausschließlich Instruktionen, die direkt von einer CPU ausgeführt werden können. Die Menge dieser Instruktionen sowie deren Syntax wird **Befehlssatz** genannt.

[1] Zu finden unter `http://docs.oracle.com/javase/specs/`

Beispiel 1 (Maschinensprachen)
1) x86:

2) SPARC:

Definition 3
Assembler TODO

Beispiel 2 (Assembler)
TODO

1.2 Paradigmen

Die grundlegendste Art, wie man Programmiersprachen unterscheiden kann ist das sog. „Programmierparadigma“, also die Art wie man Probleme löst.

Definition 4 (Imperatives Paradigma)
In der imperativen Programmierung betrachtet man Programme als eine folge von Anweisungen, die vorgibt auf welche Art etwas Schritt für Schritt gemacht werden soll.

Definition 5 (Prozedurales Paradigma)
Die prozeduralen Programmierung ist eine Erweiterung des imperativen Programmierparadigmas, bei dem man versucht die Probleme in kleinere Teilprobleme zu zerlegen.

Definition 6 (Funktionales Paradigma)
In der funktionalen Programmierung baut man auf Funktionen und ggf. Funktionen höherer Ordnung, die eine Aufgabe ohne Nebeneffekte lösen.

Haskell ist eine funktionale Programmiersprache, C ist eine nichtfunktionale Programmiersprache.

Wichtige Vorteile von funktionalen Programmiersprachen sind:

- Sie sind weitgehend (jedoch nicht vollständig) frei von Seiteneffekten.

- Der Code ist häufig sehr kompakt und manche Probleme lassen sich sehr elegant formulieren.

Definition 7 (Logisches Paradigma)
In der logischen Programmierung baut man Unifikation.

genauer

1.3 Typisierung

Eine weitere Art, Programmiersprachen zu unterscheiden ist die stärke ihrer Typisierung.

Definition 8 (Dynamische Typisierung)
Bei dynamisch typisierten Sprachen kann eine Variable ihren Typ ändern.

Beispiele sind Python und PHP.

Definition 9 (Statische Typisierung)
Bei statisch typisierten Sprachen kann eine niemals ihren Typ ändern.

Beispiele sind C, Haskell und Java.

1.4 Kompilierte und interpretierte Sprachen

Sprachen werden überlicherweise entweder interpretiert oder kompiliert, obwohl es Programmiersprachen gibt, die beides unterstützen.

C und Java werden kompiliert, Python und TCL interpretiert.

1.5 Dies und das

Definition 10 (Seiteneffekt)
Seiteneffekte sind Veränderungen des Zustandes.

Das geht besser

Manchmal werden Seiteneffekte auch als Nebeneffekt oder Wirkung bezeichnet.

Definition 11 (Unifikation)

Was ist das?

2 Programmiertechniken

2.1 Rekursion

Definition 12 (rekursive Funktion)
Eine Funktion $f : X \to X$ heißt rekursiv definiert, wenn in der Definition der Funktion die Funktion selbst wieder steht.

Beispiel 3 (rekursive Funktionen)

1) Fibonacci-Funktion:

$$fib : \mathbb{N}_0 \to \mathbb{N}_0$$
$$fib(n) = \begin{cases} n & \text{falls } n \leq 1 \\ fib(n-1) + fib(n-2) & \text{sonst} \end{cases}$$

Erzeugt die Zahlen $0, 1, 1, 2, 3, 5, 8, 13, \ldots$

2) Fakultät:

$$! : \mathbb{N}_0 \to \mathbb{N}_0$$
$$n! = \begin{cases} 1 & \text{falls } n \leq 1 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

3) Binomialkoeffizient:

$$\binom{\cdot}{\cdot} : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$$
$$\binom{n}{k} = \begin{cases} 1 & \text{falls } k = 0 \vee k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{sonst} \end{cases}$$

Ein Problem von rekursiven Funktionen in Computerprogrammen ist der Speicherbedarf. Für jeden rekursiven Aufruf müssen alle Umgebungsvariablen der aufrufenden Funktion („stack frame“) gespeichert bleiben, bis der rekursive Aufruf beendet ist. Im Fall der Fibonacci-Funktion sieht ist der Call-Stack in Abbildung 2.1 abgebildet.

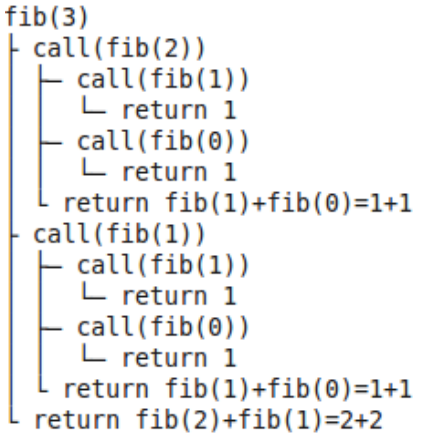

Abbildung 2.1: Call-Stack der Fibonacci-Funktion

Bemerkung 1

Die Anzahl der rekursiven Aufrufe der Fibonacci-Funktion f_C ist:

$$f_C(n) = \begin{cases} 1 & \text{falls } n = 0 \\ 2 \cdot fib(n) - 1 & \text{falls } n \geq 1 \end{cases}$$

Beweis:

- Offensichtlich gilt $f_C(0) = 1$
- Offensichtlich gilt $f_C(1) = 1 = 2 \cdot fib(1) - 1$
- Offensichtlich gilt $f_C(2) = 3 = 2 \cdot fib(2) - 1$
- Für $n \geq 3$:

$$f_C(n) = 1 + f_C(n-1) + f_C(n-2)$$

$$
\begin{aligned}
&= 1 + (2 \cdot fib(n-1) - 1) + (2 \cdot fib(n-2) - 1) \\
&= 2 \cdot (fib(n-1) + fib(n-2)) - 1 \\
&= 2 \cdot fib(n) - 1
\end{aligned}
$$

Mit Hilfe der Formel von Moivre-Binet folgt:

$$f_C \in \mathcal{O}\left(\frac{\varphi^n - \psi^n}{\varphi - \psi}\right) \text{ mit } \varphi := \frac{1+\sqrt{5}}{2} \text{ und } \psi := 1 - \varphi$$

Dabei ist der Speicherbedarf $\mathcal{O}(n)$. Dieser kann durch das Benutzen eines Akkumulators signifikant reduziert werden. TODO

Definition 13 (linear rekursive Funktion)
Eine Funktion heißt linear rekursiv, wenn in jedem Definitionszweig der Funktion höchstens ein rekursiver Aufruf vorkommt.

Definition 14 (endrekursive Funktion)
Eine Funktion heißt endrekursiv, wenn in jedem Definitionszweig der Rekursive aufruf am Ende des Ausdrucks steht. Der rekursive Aufruf darf also insbesondere nicht in einen anderen Ausdruck eingebettet sein.

Auf Englisch heißen endrekursive Funktionen *tail recursive*.

Beispiel 4 (Linear- und endrekursive Funktionen)

1) `fak n = if (n==0) then 1 else (n * fak (n-1))`
ist eine linear rekursive Funkion, aber nicht endrekursiv, da nach der Rückgabe von `fak (n-1)` noch die Multiplikation ausgewertet werden muss.

2) `fakAcc n acc = if (n==0) then acc else fakAcc (n-1) (n*acc)`
ist eine endrekursive Funktion.

3) `fib n = n <= 1 ? n : fib(n-1) + fib (n-2)`
ist weder linear- noch endrekursiv.

2.2 Backtracking

2.3 Funktionen höherer Ordnung

Funktionen höherer Ordnung sind Funktionen, die auf Funktionen arbeiten. Bekannte Beispiele sind:

- `map(function, list)`
  `map` wendet `function` auf jedes einzelne Element aus `list` an.
- `filter(function, list)`
  `filter` gibt eine Liste aus Elementen zurück, für die `function` mit `true` evaluiert.
- `reduce(function, list)`
  `function` ist für zwei Elemente aus `list` definiert und gibt ein Element des gleichen Typs zurück. Nun steckt `reduce` zuerst zwei Elemente aus `list` in `function`, merkt sich dann das Ergebnis und nimmt so lange weitere Elemente aus `list`, bis jedes Element genommen wurde.

3 Haskell

Haskell ist eine funktionale Programmiersprache, die von Haskell Brooks Curry entwickelt und 1990 in Version 1.0 veröffentlicht wurde.

Wichtige Konzepte sind:

1. Funktionen höherer Ordnung
2. anonyme Funktionen (sog. Lambda-Funktionen)
3. Pattern Matching
4. Unterversorgung
5. Typinferenz

Haskell kann mit „Glasgow Haskell Compiler“ mittels `ghci` interpretiert und mittels

3.1 Erste Schritte

Haskell kann unter `www.haskell.org/platform/` für alle Plattformen heruntergeladen werden. Unter Debian-Systemen ist das Paket `ghc` bzw. `haskell-platform` relevant.

3.1.1 Hello World

Speichere folgenden Quelltext als `hello-world.hs`:

hello-world.hs
```
1 main = putStrLn "Hello, World!"
```

Kompiliere ihn mit `ghc -o hello hello-world.hs`. Es wird eine ausführbare Datei erzeugt.

3.2 Syntax

3.2.1 Klammern und Funktionsdeklaration

Haskell verzichtet an vielen Stellen auf Klammern. So werden im Folgenden die Funktionen $f(x) := \frac{\sin x}{x}$ und $g(x) := x \cdot f(x^2)$ definiert:

```
f :: Floating a => a -> a
f x = sin x / x

g :: Floating a => a -> a
g x = x * (f (x*x))
```

Die Funktionsdeklarationen mit den Typen sind nicht notwendig, da die Typen aus den benutzten Funktionen abgeleitet werden.

Zu lesen ist die Deklaration wie folgt:

```
[Funktionsname] :: [Typendefinitionen] =>
                   Signatur
```

T. Def. Die Funktion `f` benutzt als Parameter bzw. Rückgabewert einen Typen. Diesen Typen nennen wir `a` und er ist vom Typ `Floating`. Auch `b`, `wasweisich` oder etwas ähnliches wäre ok.

Signatur Die Signatur liest man am einfachsten von hinten:

- `f` bildet auf einen Wert vom Typ `a` ab und
- `f` hat genau einen Parameter `a`

Gibt es Funktionsdeklarationen, die äquivalent? (bis auf wechsel des namens und der Reihenfolge)

3.2.2 if / else

Das folgende Beispiel definiert den Binomialkoeffizienten (vgl. Beispiel 3.3)

```
binom :: (Eq a, Num a, Num a1) => a -> a -> a1
binom n k =
    if (k==0) || (k==n)
    then 1
    else binom (n-1) (k-1) + binom (n-1) k
```

```
$ ghci binomialkoeffizient.hs
GHCi, version 7.4.2:
    http://www.haskell.org/ghc/
    :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
[1 of 1] Compiling Main
        ( binomialkoeffizient.hs, interpreted )
Ok, modules loaded: Main.
*Main> binom 5 2
10
```

Guards

3.2.3 Rekursion

Die Fakultätsfunktion wurde wie folgt implementiert:

$$fak(n) := \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot fak(n) & \text{sonst} \end{cases}$$

```
fak :: (Eq a, Num a) => a -> a
fak n = if (n==0) then 1 else n * fak (n-1)
```

Diese Implementierung benötigt $\mathcal{O}(n)$ rekursive Aufrufe und hat einen Speicherverbrauch von $\mathcal{O}(n)$. Durch einen **Akkumulator** kann dies verhindert werden:

```
fakAcc :: (Eq a, Num a) => a -> a -> a
fakAcc n acc = if (n==0)
               then acc
               else fakAcc (n-1) (n*acc)

fak :: (Eq a, Num a) => a -> a
fak n = fakAcc n 1
```

3.2.4 Listen

- `[]` erzeugt die leere Liste,
- `[1,2,3]` erzeugt eine Liste mit den Elementen $1, 2, 3$
- `:` wird **cons** genannt und ist der Listenkonstruktor.
- `head list` gibt den Kopf von `list` zurück, `tail list` den Rest:

```
Prelude> head []
*** Exception: Prelude.head: empty list
Prelude> tail []
*** Exception: Prelude.tail: empty list
Prelude> tail [1]
[]
Prelude> head [1]
1
Prelude> null []
True
```

```
Prelude> null [[]]
False
```

- `null list` prüft, ob `list` leer ist.
- `length list` gibt die Anzahl der Elemente in `list` zurück.
- `maximum [1,9,1,3]` gibt 9 zurück (analog: `minimum`).
- `last [1,9,1,3]` gibt 3 zurück.
- `reverse [1,9,1,3]` gibt `[3,1,9,1]` zurück.
- `elem item list` gibt zurück, ob sich `item` in `list` befindet.

Beispiel in der interaktiven Konsole

```
Prelude> let mylist = [1,2,3,4,5,6]
Prelude> head mylist
1
Prelude> tail mylist
[2,3,4,5,6]
Prelude> take 3 mylist
[1,2,3]
Prelude> drop 2 mylist
[3,4,5,6]
Prelude> mylist
[1,2,3,4,5,6]
Prelude> mylist ++ sndList
[1,2,3,4,5,6,9,8,7]
```

List-Comprehensions

List-Comprehensions sind kurzschreibweisen für Listen, die sich an der Mengenschreibweise in der Mathematik orientieren. So

entspricht die Menge

$$myList = \{\, 1, 2, 3, 4, 5, 6 \,\}$$
$$test = \{\, x \in myList \mid x > 2 \,\}$$

in etwa folgendem Haskell-Code:

```
Prelude> let mylist = [1,2,3,4,5,6]
Prelude> let test = [x | x <- mylist, x>2]
Prelude> test
[3,4,5,6]
```

3.2.5 Strings

- Strings sind Listen von Zeichen:
  `tail ÄBCDEF"` gibt `"BCDEF"` zurück.

3.3 Typen

In Haskell werden Typen aus den Operationen geschlossfolgert. Dieses Schlussfolgern der Typen, die nicht explizit angegeben werden müssen, nennt man **Typinferent**.

Haskell kennt die Typen aus Abbildung 3.1.

Ein paar Beispiele zur Typinferenz:

```
Prelude> let x = \x -> x*x
Prelude> :t x
x :: Integer -> Integer
Prelude> x(2)
4
Prelude> x(2.2)
<interactive>:6:3:
    No instance for (Fractional Integer)
      arising from the literal `2.2'
```

```
    Possible fix: add an instance declaration for
                        (Fractional Integer)
    In the first argument of `x', namely `(2.2)'
    In the expression: x (2.2)
    In an equation for `it': it = x (2.2)


Prelude> let mult = \x y->x*y
Prelude> mult(2,5)
<interactive>:9:5:
    Couldn't match expected type `Integer' with
                actual type `(t0, t1)'
    In the first argument of `mult', namely `(2, 5)'
    In the expression: mult (2, 5)
    In an equation for `it': it = mult (2, 5)
Prelude> mult 2 5
10
Prelude> :t mult
mult :: Integer -> Integer -> Integer

Prelude> let concat = \x y -> x ++ y
Prelude> concat [1,2,3] [3,2,1]
[1,2,3,3,2,1]
Prelude> :t concat
concat :: [a] -> [a] -> [a]
```

3.4 Beispiele

3.4.1 Quicksort

qsort.hs

```
1 qsort []     = []
2 qsort (p:ps) = (qsort (filter (\x -> x<=p) ps))
3           ++ p:(qsort (filter (\x -> x> p) ps))
```

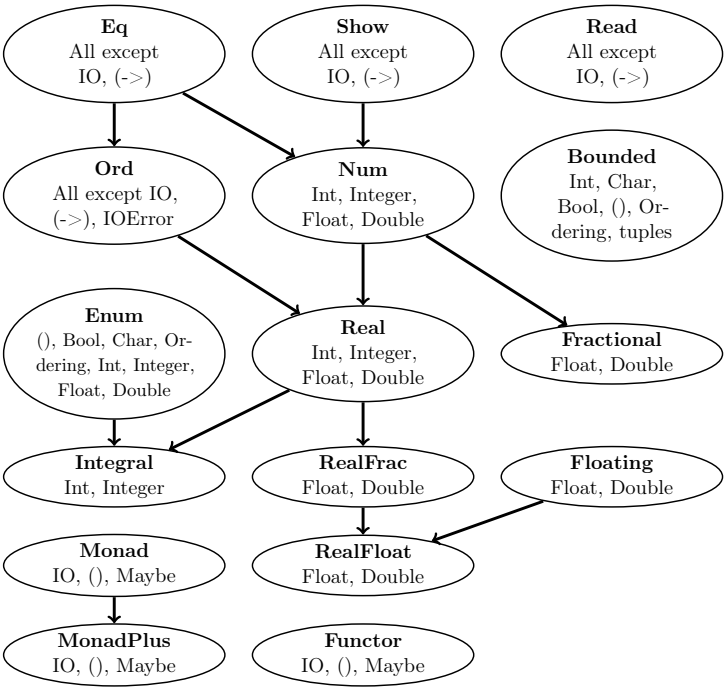

Abbildung 3.1: Hierarchie der Haskell Standardklassen

- Die leere Liste ergibt sortiert die leere Liste.
- Wähle das erste Element `p` als Pivotelement und teile die restliche Liste `ps` in kleinere und gleiche sowie in größere Elemente mit `filter` auf. Konkateniere diese beiden Listen mit `++`.

Durch das Ausnutzen von Unterversorgung lässt sich das ganze sogar noch kürzer schreiben:

qsort.hs

```
1 qsort []     = []
2 qsort (p:ps) = (qsort (filter (<=p) ps))
3           ++ p:(qsort (filter (> p) ps))
```

3.4.2 Fibonacci

fibonacci.hs

```
1 fib n
2     | (n == 0) = 0
3     | (n == 1) = 1
4     | otherwise = fib (n - 1) + fib (n - 2)
```

fibonacci-akk.hs

```
1 fibAkk n n1 n2
2     | (n == 0) = n1
3     | (n == 1) = n2
4     | otherwise = fibAkk (n - 1) n2 (n1 + n2)
5 fib n = fibAkk n 0 1
```

fibonacci-zip.hs

```
1 fib = 0 : 1 : zipWith (+) fibs (tail fibs)
```

fibonacci-pattern-matching.hs

```
1 fib 0 = 0
2 fib 1 = 1
```

```
3 fib n = fib (n - 1) + fib (n - 2)
```

3.4.3 Quicksort

3.4.4 Funktionen höherer Ordnung

3.5 Weitere Informationen

- `hackage.haskell.org/package/base-4.6.0.1`: Referenz
- `haskell.org/hoogle`: Suchmaschine für das Haskell-Manual
- `wiki.ubuntuusers.de/Haskell`: Hinweise zur Installation von Haskell unter Ubuntu

4 Prolog

Prolog ist eine Programmiersprache, die das logische Programmierparadigma befolgt.

Eine interaktive Prolog-Sitzung startet man mit `swipl`.

In Prolog definiert man Terme.

4.1 Syntax

4.2 Beispiele

4.2.1 Humans

Erstelle folgende Datei:

human.pro

```
1 human(bob).
2 human(socrates).
3 human(antonio).
```

Kompiliere diese mit

```
$ swipl -c human.pro
% library(swi_hooks) compiled into pce_swi_hooks
%                 0.00 sec, 2,224 bytes
% human.pro compiled 0.00 sec, 644 bytes
% /usr/lib/swi-prolog/library/listing compiled into
%                 prolog_listing 0.00 sec, 21,648 bytes
```

Dabei wird eine `a.out` Datei erzeugt, die man wie folgt nutzen kann:

```
$ ./a.out
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version
Copyright (c) 1990-2011 University of Amsterdam, VU Amste
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is fre
software, and you are welcome to redistribute it under ce
conditions. Please visit http://www.swi-prolog.org for de

For help, use ?- help(Topic). or ?- apropos(Word).

?- human(socrates).
true.
```

4.2.2 Zebrarätsel

Folgendes Rätsel wurde von `https://de.wikipedia.org/w/index.php?title=Zebrar%C3%A4tsel&oldid=126585006` entnommen:

1. Es gibt fünf Häuser.
2. Der Engländer wohnt im roten Haus.
3. Der Spanier hat einen Hund.
4. Kaffee wird im grünen Haus getrunken.
5. Der Ukrainer trinkt Tee.
6. Das grüne Haus ist direkt rechts vom weißen Haus.
7. Der Raucher von Altem-Gold-Zigaretten hält Schnecken als Haustiere.
8. Die Zigaretten der Marke Kools werden im gelben Haus geraucht.
9. Milch wird im mittleren Haus getrunken.

10. Der Norweger wohnt im ersten Haus.
11. Der Mann, der Chesterfields raucht, wohnt neben dem Mann mit dem Fuchs.
12. Die Marke Kools wird geraucht im Haus neben dem Haus mit dem Pferd.
13. Der Lucky-Strike-Raucher trinkt am liebsten Orangensaft.
14. Der Japaner raucht Zigaretten der Marke Parliaments.
15. Der Norweger wohnt neben dem blauen Haus.

Wer trinkt Wasser? Wem gehört das Zebra?

zebraraetsel.pro

```
1 Street=[Haus1,Haus2,Haus3],
2 mitglied(haus(rot,_,_),Street),
3 mitglied(haus(blau,_,_),Street),
4 mitglied(haus,(grün,_,_),Street),
5 mitglied(haus(rot,australier,_),Street),
6 mitglied(haus(_,italiener,tiger),Street),
7 sublist(haus(_,_,eidechse),haus(_,chinese,_),Street),
8 sublist(haus(blau,_,_),haus(_,_,eidechse),Street),
9 mitglied(haus(_,N,nilpferd),Street).
```

4.3 Weitere Informationen

- `wiki.ubuntuusers.de/Prolog`: Hinweise zur Installation von Prolog unter Ubuntu

5 Scala

Scala ist eine funktionale Programmiersprache, die auf der JVM aufbaut und in Java Bytecode kompiliert wird.

5.1 Syntax

5.2 Beispiele

6 X10

6.1 Syntax

6.2 Beispiele

7 C

C ist eine imperative Programmiersprache. Sie wurde in vielen Standards definiert. Die wichtigsten davon sind:

Wo sind unterschiede?

- C89
- C99
- ANSI C
- C11

7.1 Datentypen

Die grundlegenden C-Datentypen sind

Typ	**Größe**
char	1 Byte
int	4 Bytes
float	4 Bytes
double	8 Bytes
void	0 Bytes

zusätzlich kann man `char` und `int` noch in `signed` und `unsigned` unterscheiden.

Dez.	Z.	Dez.	Z.	Dez.	Z.	Dez.	Z.
0		31		64	@	96	’
1				65	A	97	a
2				66	B	98	b
3					C	99	c
4					D	100	d
5					E		
6					F		
7					G		
8					H		
9					I		
10							
11							
12							
13							
14							
15							
16							
17							
18							
19							
20							
21							
22							
23							
24							
25							
26							
27							
28							
29							
31						127	

7.2 ASCII-Tabelle

7.3 Syntax

7.4 Beispiele

7.4.1 Hello World

Speichere den folgenden Text als `hello-world.c`:

hello-world.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hello, World\n");
6     return 0;
7 }
```

Compiliere ihn mit `gcc hello-world.c`. Es wird eine ausführbare Datei namens `a.out` erzeugt.

8 MPI

Message Passing Interface (kurz: MPI) ist ein Standard, der den Nachrichtenaustausch bei parallelen Berechnungen auf verteilten Computersystemen beschreibt.

8.1 Syntax

8.2 Beispiele

9 Compilerbau

Wenn man über Compiler redet, meint man üblicherweise „vollständige Übersetzer“:

Definition 15

Ein **Compiler** ist ein Programm C, das den Quelltext eines Programms A in eine ausführbare Form übersetzen kann.

Jedoch gibt es verschiedene Ebenen der Interpretation bzw. Übersetzung:

1. **Reiner Interpretierer**: TCL, Unix-Shell
2. **Vorübersetzung**: Java-Bytecode, Pascal P-Code, Python[1], Smalltalk-Bytecode
3. **Laufzeitübersetzung**: JavaScript[2]
4. **Vollständige Übersetzung**: C, C++, Fortran

Zu sagen, dass Python eine interpretierte Sprache ist, ist in etwa so korrekt wie zu sagen, dass die Bibel ein Hardcover-Buch ist.[3]

Reine Interpretierer lesen den Quelltext Anweisung für Anweisung und führen diese direkt aus.

Bild

[1] Python hat auch `.pyc`-Dateien, die Python-Bytecode enthalten.

[2] JavaScript wird nicht immer zur Laufzeit übersetzt. Früher war es üblich, dass JavaScript nur interpretiert wurde.

[3] Quelle: stackoverflow.com/a/2998544, danke Alex Martelli für diesen Vergleich.

Bei der *Interpretation nach Vorübersetzung* wird der Quelltext analysiert und in eine für den Interpretierer günstigere Form übersetzt. Das kann z. B. durch

?

- Zuordnung Bezeichnergebrauch - Vereinbarung
- Transformation in Postfixbaum
- Typcheck, wo statisch möglich

geschehen. Diese Vorübersetzung ist nicht unbedingt maschinennah.

Bild

Die *Just-in-time-Compiler* (kurz: JIT-Compiler) betreiben Laufzeitübersetzung. Folgendes sind Vor- bzw. Nachteile von Just-in-time Compilern:

- schneller als reine Interpretierer
- Speichergewinn: Quelle kompakter als Zielprogramm
- Schnellerer Start des Programms
- Langsamer (pro Funktion) als vollständige Übersetzung
- kann dynamisch ermittelte Laufzeiteigenschaften berücksichtigen (dynamische Optimierung)

Was ist hier gemeint?

Moderne virtuelle Maschinen für Java und für .NET nutzen JIT-Compiler.

Bei der *vollständigen Übersetzung* wird der Quelltext vor der ersten Ausführung des Programms A in Maschinencode (z. B. x86, SPARC) übersetzt.

Bild

9.1 Funktionsweise

Üblicherweise führt ein Compiler folgende Schritte aus:

1. Lexikalische Analyse
2. Syntaktische Analyse
3. Semantische Analyse
4. Zwischencodeoptimierung
5. Codegenerierung
6. Assemblieren und Binden

9.2 Lexikalische Analyse

In der lexikalischen Analyse wird der Quelltext als Sequenz von Zeichen betrachtet. Sie soll bedeutungstragende Zeichengruppen, sog. *Tokens*, erkennen und unwichtige Zeichen, wie z. B. Kommentare überspringen. Außerdem sollen Bezeichner identifiziert und in einer *Stringtabelle* zusammengefasst werden.

Beispiel 5

Beispiel erstellen

9.2.1 Reguläre Ausdrücke

Beispiel 6 (Regulärere Ausdrücke)

Folgender regulärer Ausdruck erkennt Float-Konstanten in C nach ISO/IEC 9899:1999 §6.4.4.2:

$((0|\ldots|9)^*.(0|\ldots|9)^+)|((0|\ldots|9)^+.)$

Satz 9.1
Jede reguläre Sprache wird von einem (deterministischen) endlichen Automaten akzeptiert.

TODO: Bild einfügen

Zu jedem regulären Ausdruck im Sinne der theoretischen Informatik kann ein nichtdeterministischer Automat generiert werden. Dieser kann mittels Potenzmengenkonstruktion[4] in einen deterministischen Automaten überführen. Dieser kann dann mittels Äquivalenzklassen minimiert werden.

Alle Schritte beschreiben

9.2.2 Lex

Lex ist ein Programm, das beim Übersetzerbau benutzt wird um Tokenizer für die lexikalische Analyse zu erstellen. Flex ist eine Open-Source Variante davon.

Eine Flex-Datei besteht aus 3 Teilen, die durch `%%` getrennt werden:

```
Definitionen: Definiere Namen
%%
Regeln: Definiere reguläre Ausdrücke und
        zugehörige Aktionen (= Code)
%%
Code: zusätzlicher Code
```

x	Zeichen 'x' erkennen
"xy"	Zeichenkette 'xy' erkennen
\	Zeichen 'x' erkennen (TODO)
$[xyz]$	Zeichen x, y oder z erkennen
$[a-z]$	Alle Kleinbuchstaben erkennen
$[-z]$	Alle Zeichen außer Kleinbuchstaben erkennen
$x\|y$	x oder y erkennen
(x)	x erkennen
x*	0, 1 oder mehrere Vorkommen von x erkennen
x+	1 oder mehrere Vorkommen von x erkennen
x?	0 oder 1 Vorkommen von x erkennen
{Name}	Expansion der Definition Name
\t, \n, \rq	Tabulator, Zeilenumbruch, Wagenrücklauf erkennen

Reguläre Ausdrücke in Flex

9.3 Syntaktische Analyse

In der syntaktischen Analyse wird überprüft, ob die Tokenfolge zur kontextfreien Sprachegehört. Außerdem soll die hierarchische Struktur der Eingabe erkannt werden.

Warum kontextfrei?

Ausgegeben wird ein **abstrakter Syntaxbaum**.

Beispiel 7 (Abstrakter Syntaxbaum)
TODO

Was ist gemeint?

9.4 Semantische Analyse

Die semantische Analyse arbeitet auf einem abstrakten Syntaxbaum und generiert einen attributierten Syntaxbaum.

Sie führt eine kontextsensitive Analyse durch. Dazu gehören:

[4] http://martin-thoma.com/potenzmengenkonstruktion/

- **Namensanalyse**: Beziehung zwischen Deklaration und Verwendung

?

- **Typanalyse**: Bestimme und prüfe Typen von Variablen, Funktionen, ...

?

- **Konsistenzprüfung**: Wurden alle Einschränkungen der Programmiersprache eingehalten?

?

Beispiel 8 (Attributeriter Syntaxbaum)
TODO

9.5 Zwischencodeoptimierung

Hier wird der Code in eine sprach- und zielunabhängige Zwischensprache transformiert. Dabei sind viele Optimierungen vorstellbar. Ein paar davon sind:

- **Konstantenfaltung**: Ersetze z. B. $3 + 5$ durch 8.
- **Kopienfortschaffung**: Setze Werte von Variablen direkt ein
- **Code verschieben**: Führe Befehle vor der Schleife aus, statt in der Schleife

?

- **Gemeinsame Teilausdrücke entfernen**: Es sollen doppelte Berechnungen vermieden werden

Beispiel?

- **Inlining**: Statt Methode aufzurufen, kann der Code der Methode an der Aufrufstelle eingebaut werden.

9.6 Codegenerierung

Der letzte Schritt besteht darin, aus dem generiertem Zwischencode den Maschinencode oder Assembler zu erstellen. Dabei muss folgendes beachtet werden:

- **Konventionen**: Wie werden z. B. im Laufzeitsystem Methoden aufgerufen?
- **Codeauswahl**: Welche Befehle kennt das Zielsystem?
- **Scheduling**: In welcher Reihenfolge sollen die Befehle angeordnet werden?
- **Registerallokation**: Welche Zwischenergebnisse sollen in welchen Prozessorregistern gehalten werden?
- **Nachoptimierung**

Bildquellen

Abb. **??** S^2: Tom Bombadil, tex.stackexchange.com/a/42865

Abkürzungsverzeichnis

Beh. Behauptung

Bew. Beweis

bzgl. bezüglich

bzw. beziehungsweise

ca. circa

d. h. das heißt

DEA Deterministischer Endlicher Automat

etc. et cetera

ggf. gegebenenfalls

sog. sogenannte

Vor. Voraussetzung

vgl. vergleiche

z. B. zum Beispiel

z. z. zu zeigen

Symbolverzeichnis

Reguläre Ausdrücke

$\emptyset$ Leere Menge
ϵ Das leere Wort
α, β Reguläre Ausdrücke
$L(\alpha)$ Die durch α beschriebene Sprache
$L(\alpha|\beta) = L(\alpha) \cup L(\beta)$
$L(\alpha \cdot \beta) = L(\alpha) \cdot L(\beta)$
$\alpha^+ = L(\alpha)^+$ TODO: Was ist $L(\alpha)^+$
$\alpha^* = L(\alpha)^*$ TODO: Was ist $L(\alpha)^*$

Stichwortverzeichnis