Grundlagen der Programmiersprachen, Prof. Dr. Uwe Kastens, Zusammenfassung von Florian Schoppmann

Spracheigenschaften

Ebenen der Spracheigenschaften

- **Grundsymbole:** Bezeichner, Literale, Wortsymbole, Spezialzeichen
- **Syntax:** Struktur von Grundkonstrukten, wie definierbar durch eine kontextfreie Grammatik
- **Statische Semantik:** Gültigkeit von Definitionen, Typregeln, … (zur Übersetzungszeit feststellbar)
- **Dynamische Semantik:** Ausführungsbedingungen (Fehler hier sind Laufzeitfehler)

Art der **Bindung** von Bezeichnern an Werte

- durch **Zuweisung** (imperative Programmiersprachen)
- durch **Unifikation**/Substitution (Prolog)
- durch **Definition** – in funktionalen Programmierung mit Mustern möglich, z.B.::

```
val (a, b) = (fn i => (i, 2*i)) 3;
```

Jeder freie Bezeichner im Muster `(a, b)` ist ein bindendes Auftreten. Ähnlichkeit mit Unifikation, da Ausdrücke auf beiden Seiten der Gleichung vorhanden.

Syntax

(Kontextfreie) **Grammatiken**

G = (T, N, P, S), wobei T Menge der Terminalsymbole, N Menge der Nichtterminalsymbole, V = T ∪ N das Vokabular, S ∈ N das Startsymbol und P ⊆ N × V* Menge der Produktionen

Beispiel/Konstruktionsschema:

```
p1:   Epxr   ::= Expr AddOpr Fact
p2:   Expr   ::= Fact
p3:   Fact   ::= Fact MulOpr Opd
p4:   Fact   ::= Opd
p5:   Opd    ::= '(' Expr ')'
p6:   Opd    ::= Ident
p7:   AddOpr ::= '+'
...
```

Ableitungsbäume: Gibt es mehrere zu einem Satz, so ist die Grammatik mehrdeutig

Präzedenzen: Im Beispiel hat `AddOpr` geringere Präzedenz als `MulOpr`, weil er höher in der Hierarchie der Kettenproduktionen steht.

Assoziativität: Im Beispiel sind `AddOpr` und `MulOpr` links-assoziativ, weil ihre Produktionen links-rekursiv sind.

Mehrdeutigkeit

- beweisen: Durch Gegenbeispiel
- vermeiden: Konstruktionsschema wie oben verwenden

EBNF (Extended Backus Naur Form)-Konstrukte

Erweiterungen:

`X ::= u (v) w`	Klammerung (zusätzl. Prod.-Stufe)
`X ::= u [v] w`	optional
`X ::= u s* w`	optionale Wiederholung
`X ::= u {s} w`	optionale Wiederholung
`X ::= u s… w`	Wiederholung
`X ::= u s+ w`	Wiederholung
`X ::= u (v || s) w`	Wdh. mit Trenner `s`
`X ::= u (v1 | v2) w`	Alternativen

Beispiele:

- Parameter Liste, Variablen-Deklaration

BNF sinnvoll in EBNF umwandeln

- Mehrfach vorkommende Produktionen eigenständig lassen

Variable, Laufzeitkeller, Funktionen

Verdeckungsregeln

Algol-Regel: Eine Definition gilt im kleinsten sie umfassenden Abschnitt überall, ausgenommen darin enthaltener Abschnitte mit einer Definition für denselben Bezeichner

C-Regel: Eine Definition eines Bezeichners gilt von der Definitionsstelle bis zu Ende des kleinsten sie umfassenden Abschnitts, ausgenommen die Gültigkeitsbereiche von Definitionen desselben Bezeichners in enthaltenen Abschnitten

Lebensdauer von Variablen

	Lebensdauer	**Unterbringung im Speicher**
globale Variable	gesamtes Programm	globaler Speicher
Klassenvariable		
Objektvariable	wie das Objekt	Halde (für gewöhnlich)
Parametervariable	während des Aufrufs	Laufzeitkeller
lokale Variable		

Laufzeitkeller

Enthält für jeden noch nicht beendeten Aufruf einen Speicherblock (**Schachtel**) mit Speicher für Parametervariable und lokale Variable.

Für rekursiven Funktionen können mehrere Schachtel gleichzeitig existieren

zeichnen

bei Aufruffolge: `h, q, q, q, r`

h	a=1, …
q	i=2
q	i=1
q	i=0
r	b=…

Geschachtelte Funktionen/Klassen

Statischer Vorgänger: Zeigt auf die Schachtel, die die Definition der aufgerufenen Funktion enthält. Zugriff auf Variable entlang der Kette statischer Vorgänger möglich, sofern nicht verdeckt.

Übergabearten für Parameter

call-by-(strict)-value, call-by-reference, call-by-value-and-result

Typen

statische **Typbindung** (z.B. Java, C, C++, …) ⇒ dynamische Typbindung (z.B. Smalltalk, Perl)

Abstrakte Grundkonzepte

- **einfache Mengen:** Grundtypen, Aufzählungstypen
- **kartesisches Produkt:** Records (structs in C), Arrays
- **Vereinigung:** z.B. Unions, Ober-, Unterklassen
- **Funktionen:** Funktionen, Prozeduren, Methoden; auch Arrays
- **Potenzmenge:** z.B. Mengentypen in Pascal: `type Mischfarbe = set of farbe;`

Simulation durch Verwendung von Bit-Arrays/normalen Zahlen

Vereinigung

undiscriminated union: Durch den Zugriff wird unterstellt, welcher der vereinigten Typen vorliegt (unsicher!)

discriminated: „Tag Field" kennzeichnet Typzugehörigkeit als Teil des Wertes (In Pascal ebenfalls unsicher, da auf das Tag Field frei zugegriffen werden kann)

Union-Realisierung in Java

- allgemeine Oberklasse mit speziellen Unterklassen

Typdefinitionen

auch **rekursiv** in funktionalen Programmiersprachen möglich

```
datatype IntList = cons of (int * IntList) | IntNil;
```

In (den meisten) imperativen Sprachen werden solche Typen mit Zeigern implementiert.

rekursive für Bäume (abstrakte Typdefinition):

```
Baum = 'a * Liste
Liste = Liste * Baum | {nil}
LBaum = Baum | {nil};
```

Typisierung

streng (z.B. Java) oder schwach

Typumwandlung

ausweitend ⇒ einengend

- ausweitend ist meist sicher (z.B. von `float` zu `double`). Bei Java wird ausweitende Konversion deshalb für Grund- und Referenztypen implizit durchgeführt
- einengende müssen explizit angegeben werden

implizit (**Coercion**) ⇒ explizit (**Type Cast**)

Parametrisierte Typen (**Polytypen**): Haben andere Typen als Parameter

Generische Definitionen

z.B. templates in C++

Erst bei der Instantiierung werden aktuelle generische Parameter eingesetzt. Dies kommt prinzipiell einer Textersetzung gleich.

Typinferenz: Der Übersetzer berechnet Typen und prüft streng aus den benutzten Operationen

Funktionale Programmierung

Funktionen höherer Ordnung: Funktion mit einer Funktion als Parameter oder Ergebnis

Currying

Schrittweise Bindung der Parameter

```
fun CAdd x = fn y:int => x + y;
```

statt

```
fun Add (x, y:int) = x + y;
```

Kurzschreibweise in Curry-Form:

```
fun CAdd x y:int = x + y;
```

Parametermatching: Kurzschreibweise z.B. in SML z.B.

```
fun Sum (nil) = 0 | Sum (h::t) = h + Sum t;
```

Signatur: „polymorph", wenn polymorpher Typ enthalten

Transformation **zentral-rekursiver** Aufrufe einer Funktion in **end-rekursive** durch **akkumulierenden Parameter**

Beispiel-Funktionen auf Listenelementen:

map

```
fun Map (nil, f) = nil
|   Map (h::t,f) = (f h) :: Map (t, f);
```

LInsert

```
fun LInsert (nil, f, a) = a
|   LInsert (h::t, f, a)= LInsert (t, f, f (a, h));
```

Signaturen von Standardoperationen:

map: `('a list * ('a -> 'b)) -> 'b list`

LInsert: `('a list * (('b * 'a)->'b) * 'b) -> 'b`

Logische Programmierung

Prädikate

Eingeschränkt auf **Horn-Klauseln**, d.h. Klauseln der Form:

A_0 wenn A_1 und … und A_n.

Klauseln: Fakten, Regeln, Anfragen

Notation von Prolog-Programmen

Literale für Zahlen, Zeichen(reihen)	`127  "text"  'a'`
Symbole (erste Buchstabe klein)	`hans`
Variablen (erste Buchstabe groß)	`X  Person`
unbenannte Variable:	`_`
Listen-Notation:	`[a, b, c] []`
erstes Element `H`, Restliste `T` (wie `H::T` in SML)	`[H | T]`
Strukturen:	`kante(a, b)  a - b` `datum(T, M, J)`

Operatoren `kante`, `-` werden ohne Definition verwendet, nicht „ausgerechnet"

Listen

Beispiel:

```
last([X], X).
last([_|T], Y):- last(T, Y).
?-last([1,2,3], Z). bindet Z=3
```