
Theorie der Programmierung I

Kurt Sieber

Fachbereich Mathematik/Theoretische Informatik
Universität Siegen

Semantik von Programmiersprachen und Programmverifikation

Zentrale Fragestellungen

- Wie definiert man (mit mathematischer Präzision) die Syntax und **Semantik** einer Programmiersprache?
- Wie beweist man auf der Basis dieser Definition
 - Eigenschaften der Programmiersprache?
 - Eigenschaften einzelner Programme, z.B. die **Korrektheit** von Programmen?

Einleitung

Einzelne 'Phasen' der Definition einer Programmiersprache

- **Lexikalische Syntax:**
Festlegung der kleinsten syntaktischen Einheiten wie Zahldarstellungen, Namen, Schlüsselwörter
- **Kontextfreie Syntax:**
Festlegung der Struktur von Programmen durch eine kontextfreie Grammatik
- **Kontextbedingungen (oder "statische Semantik"):**
Einschränkung der kontextfreien Sprache durch zusätzliche Bedingungen, z.B.
 - ein Name muss deklariert sein, bevor er verwendet wird
 - Anzahl und Typen der Parameter einer Funktion (oder Prozedur oder Methode) müssen passen
- **Semantik (oder "dynamische Semantik"):**
Festlegung des Laufzeitverhaltens von Programmen

Zum Vergleich: Phasen eines Compilers

- **Lexikalische Analyse:**
Erkennung der kleinsten syntaktischen Einheiten (**tokens**) durch den **Scanner**
- **Syntaktische Analyse:**
Erkennung der Baumstruktur eines Programms durch den **Parser**
- **Semantische Analyse:**
Überprüfung der Kontextbedingungen
- **Codegenerierung:**
Übersetzung des Programms in Maschinencode

Einleitung

Methoden für die einzelnen Phasen

- **Lexikalische Syntax:**
Reguläre Ausdrücke (bekannt aus GTI)
- **Kontextfreie Syntax:**
Kontextfreie Grammatiken (bekannt aus GTI)
- **Kontextbedingungen:**
Hier: **Typregeln** (Im Compilerbau: Attributierte Grammatiken)
- **Semantik:**
Unterschiedliche Methoden:
 - **Operationelle Semantik**
 - * **Small step:** Der Programmablauf wird **Schritt für Schritt** angegeben.
 - * **Big step:** Nur das **Endergebnis** einer Berechnung wird angegeben.
 - **Denotationelle Semantik:**
Jedem Programm(stück) wird eine **Bedeutung** in einem mathematischen Modell zugeordnet (meist eine Funktion).

Einleitung

Schwerpunkte der Vorlesung

- Typsysteme
- operationelle Semantik
- Zusammenhänge zwischen beiden

Vorgehensweise

- Einübung der Methoden an einer (kleinen) funktionalen Programmiersprache.
- Schrittweise Erweiterung der Programmiersprache

Unsere Programmiersprache orientiert sich an **OCaml** (= **Objective Caml**), einer Erweiterung der funktionalen Programmiersprache ML um objektorientierte Konzepte.

Eine einfache funktionale Sprache

1. Eine einfache funktionale Sprache

1.1. Lexikalische Syntax

Details unwichtig!

Vorgegeben:

- die Menge *Int* aller (Darstellungen von) **ganzen Zahlen** n , z.B.
 $Int = \mathbb{Z}$ oder $Int = \{n \in \mathbb{Z} \mid \min_{Int} \leq n \leq \max_{Int}\}$
- die Menge *Bool* der **booleschen Werte** b
 $Bool = \{true, false\}$
- die Menge *Unit*, die nur das **unit-Element** $()$ enthält
- eine unendliche Menge *Id* von **Namen** id (engl. **identifier**)

Kontextfreie Syntax

1.2. Kontextfreie Syntax

Mit einer kontextfreien Grammatik werden definiert:

- die Menge Op der Operatoren op

$op ::= not$	Negation
$+$ $-$ $*$ $/$ mod	arithmetische Operatoren
$<$ $>$ \leq \geq	Vergleichsoperatoren
$=$	Gleichheit
$\#n$ ($n \geq 1$)	Projektionen

Kontextfreie Syntax

- die Menge *Const* der Konstanten c

$$\begin{aligned} c ::= & () \text{ unit-Element} \\ & | b \text{ boolescher Wert} \\ & | n \text{ ganze Zahl} \\ & | op \text{ Operator} \end{aligned}$$

- die Menge *BType* der Basistypen β

$$\beta ::= \text{unit} \mid \text{bool} \mid \text{int}$$

- die Menge *Type* der Typen τ

$$\begin{aligned} \tau ::= & \beta && \text{Basistyp} \\ & | \tau_1 * \dots * \tau_k \quad (k \geq 2) && \text{Produkttyp} \\ & | \tau_1 \rightarrow \tau_2 && \text{Funktionstyp} \end{aligned}$$

Kontextfreie Syntax

- die Menge Exp der **Ausdrücke** e

$e ::= c$		Konstante
id		Name
(e_1, \dots, e_k)	$(k \geq 2)$	Tupel
$e_1 e_2$		Applikation
if e_0 then e_1 else e_2		bedingter Ausdruck
let $id = e_1$ in e_2		let-Ausdruck
$\lambda id : \tau. e_1$		λ -Abstraktion