1 Einführung

1.1 Was ist eine Semantik?

Syntax und Semantik sind zwei unabdingbare Bausteine der Beschreibung von (Programmier-)Sprachen: Syntax kümmert sich darum, welche Zeichenfolgen gültige Sätze (Programme) der Sprache sind. Syntax umfasst also Vokabular (Schlüsselwörter) und Grammatik. Semantik beschreibt, was die Bedeutung eines gültigen Satzes (Programms) sein soll. Für Programmiersprachen heißt das: Wie verhält sich das Programm, wenn man es ausführt?

Syntax legt auch den Aufbau eines Satzes, i. d. R. ein Baum, fest und erklärt wie man von einer Zeichenfolge zum Syntaxbaum kommt. Eine Semantik beschreibt, wie man dieser syntaktischen Struktur eine Bedeutung gibt, d.h.: Was ist die Bedeutung eines einzelnen Konstrukts? Wie erhält man die Gesamtbedeutung aus den einzelnen Teilen?

Syntax und Semantik vieler Programmiersprachen sind standardisiert (C, C++, Pascal, Java, ...). Für die Definition der Syntax werden formale Techniken routinemäßig in der Praxis eingesetzt: kontext-freie Grammatiken (EBNF). Das Verhalten von Konstrukten der Programmiersprache und deren Zusammenwirken beschreiben die meisten dieser Standards allerdings nur in natürlicher Sprache, meistens auf englisch oder nur anhand konkreter Beispiele. Für umfangreiche Programmiersprachen ist es auf diese Weise fast unmöglich, alle möglichen Kombinationen eindeutig festzulegen und dabei trotzdem Widerspruchsfreiheit zu garantieren. Deswegen gibt es auch formale, d.h. mathematische, Beschreibungstechniken für Semantik, die das Thema dieser Vorlesung sind. Die funktionale Sprache ML wurde beispielsweise vollständig durch eine formale Semantik definiert; auch für Java gibt es formale Modellierungen, die zwar nicht Teil der Sprachdefinition sind, aber den Entwurf und die Weiterentwicklungen wesentlich beeinflussten (z.B. Featherweight Java, Java light, Jinja).

1.2 Warum formale Semantik?

Die einfachste Definition einer Sprache ist mittels eines Compilers: Alle Programme, die der Compiler akzeptiert, sind syntaktisch korrekt; die Semantik ist die des Zielprogramms. Eine solche Definition ist aber sehr problematisch:

1. *Keine Abstraktion.* Um das Verhalten eines Programmes zu verstehen, muss man den Compiler und das Kompilat in der Zielsprache verstehen. Für neue, abstrakte Konzepte in der Programmiersprache gibt es keine Garantie, dass die Abstraktionsschicht nicht durch andere Konstrukte durchbrochen werden kann – geht es doch, ist das eine der Hauptfehlerquellen bei der Entwicklung von Programmen.

   Beispiele:

   - Pointer-Arithmetik in C++ kann die Integrität von Objekten zerstören, weil damit beliebig (auch auf private) Felder zugegriffen werden kann.
   - `setjmp/longjmp` in C widerspricht dem Paradigma, dass Methoden stack-artig aufgerufen werden.
   - LaTeX ist nur ein Aufsatz auf TeX, der zwar in vielen Fällen eine gute Abstraktionsschicht schafft, aber diese nicht garantieren kann. Fast jeder ist schon über unverständliche TeX-Fehlermeldungen und Inkompatibilitäten zwischen LaTeX-Paketen gestoplert.

2. *Plattformabhängigkeit und Überspezifikation.* Ein Wechsel auf eine andere Zielsprache oder einen anderen Compiler ist fast unmöglich. Schließlich ist auch festgelegt, wie viele einzelne Ausführungsschritte (z. B. Anzahl der Prozessorzyklen) jedes einzelne Konstrukt genau benötigen muss.

Zwischen „Sprachdefinition“ und Implementierungsdetails des Compilers kann nicht unterschieden werden. Weiterentwicklungen aus der Compiler-Technik sind damit ausgeschlossen.

3. *Bootstrapping.* Auch ein Compiler ist nur durch seinen Programmtext definiert. Was ist aber die Semantik dieses Programmtextes?

Eine Semantikbeschreibung in Prosa wie bei den meisten Sprachstandards ist zwar besser, kann aber Mehrdeutigkeiten, Missverständnisse oder Widersprüche auch nicht verhindern. Demgegenüber stehen die Vorteile mathematischer Beschreibungen einer Semantik:

1. *Vollständige, rigorose Definition einer Sprache.* Jedes syntaktisch gültige Programm hat eine eindeutige, klar festgelegte Semantik. Mathematische Beschreibungen sind syntaktisch oft viel kürzer als englischer Fließtext. Programmierer können die Semantik als Nachschlagereferenz verwenden, um subtile Feinheiten der Sprache zu verstehen. Compiler-Bauer können die Semantik einer solchen Sprache als Korrektheitskriterium ihres Compilers verwenden. Damit verhalten sich Anwender-Programme gleich, unabhängig vom verwendeten (korrekten) Compiler.

2. *Nachweis von Programmeigenschaften.* Ohne formale Semantik als Grundlage lassen sich Eigenschaften eines Programms nicht beweisen, ja nicht einmal mathematisch aufschreiben. Dabei unterscheidet man zwischen Eigenschaften, die alle Programme erfüllen und damit Meta-Eigenschaften der Sprache bzw. Semantik sind (z. B. Typsicherheit), und solchen eines einzelnen Programms (z. B. Korrektheit eines Algorithmus).

3. *Unterstützung beim Programmiersprachenentwurf.* Eine Programmiersprache mit vielen verschiedenen Konzepten, die klar, verständlich und ohne unnötige Sonderfälle zusammenwirken, zu entwerfen, ist sehr schwierig. Bereits der Versuch, eine formale Semantik für eine Programmiersprache zu entwerfen, deckt viele Inkonsistenzen und unerwartete Interaktionen auf.

   Hat man eine formale Beschreibung der Semantik, lässt sich automatisch ein prototypischer Interpreter für die Sprache erzeugen, z. B. als Prolog-Programm. Dadurch können verschiedene Designentscheidungen beim Entwurf der Semantik an konkreten Beispielen praktisch ausprobiert werden.

   Programmverifikation ist einfacher, wenn die Semantik der Programmiersprache mathematisch einfach und klar ist. Eine zufällig, nach Gutdünken entworfene Programmiersprache hat in der Regel ein sehr kompliziertes mathematisches Modell. Dementsprechend ist es auch schwierig, darüber Beweise zu führen, da stets viele Sonderfälle berücksichtigt werden müssen.

4. *Klare und konsistente Begrifflichkeit.* Formale Semantik arbeitet mit der Sprache und den Begriffen der Mathematik, über deren Bedeutung man sich im Klaren ist. Damit werden Mehrdeutigkeiten und Missverständnisse von vornherein ausgeschlossen.

Beispiele:

- Was ist der Wert von `x` am Ende?

  ```
  b = true; c = false;
  if (b) if (c) then x = 1; else x = 2;
  ```

- Was ist der Wert von `x` bzw. `i` nach Ausführung des folgenden C-Fragments? Was wäre er in Java?

  ```
  int i = 1;
  i += i++ + ++i;
  ```

- Sind die beiden Initialisierungen von `b` äquivalent?

```
boolean f(int a, int b) {
  return (a == 0) && (b == 0);
}

boolean b = (1 == 0) && (2 / 0 == 0);
boolean b = f(1, 2 / 0);
```

1.3 Operational, denotational und axiomatisch

In dieser Vorlesung werden die drei bekanntesten Beschreibungsarten für Semantiken vorgestellt – mit einem Schwerpunkt auf operationaler Semantik:

Operational Die Bedeutung eines Konstrukts ergibt sich aus seinen (abstrakten) Ausführungsschritten, die durch symbolische Regeln beschrieben werden. Neben dem Ergebnis der Berechnung wird auch modelliert, wie die Berechnung zum Ergebnis kommt. Die Regeln beschreiben dabei nur eine mögliche (idealisierte) Implementierung der Sprache, reale Implementierungen können andere, äquivalente Abläufe verwenden.

Denotational Jedes Konstrukt wird als mathematisches Objekt realisiert, welches *nur* den Effekt seiner Ausführung modelliert. Im Gegensatz zur operationalen Semantik ist irrelevant, wie der Effekt zustande kommt.

Axiomatisch Die Bedeutung eines Programms wird indirekt festgelegt, indem beschrieben wird, welche Eigenschaften des Programms (in einem zu entwerfenden Logik-Kalkül) beweisbar sind.

Die drei verschiedenen Ansätze ergänzen sich: Für Compiler und Implementierungen sind operationale Semantiken gut geeignet. Denotationale Konzepte finden sich oft in der Programmanalyse. Programmverifikation stützt sich auf die axiomatische Semantik. Zu jedem Ansatz werden wir eine Semantik für eine einfache imperative Sprache – ggf. mit verschiedenen Spracherweiterungen – entwickeln und die Beziehung der verschiedenen Semantiken zueinander untersuchen.

Beispiel 1. Semantik des Programms `z := x; x := y; y := z`

1. Die *operationale Semantik* beschreibt die Semantik, indem sie definiert, wie das Programm auszuführen ist: Eine Sequenz von zwei durch `;` getrennten Anweisungen führt die einzelnen Anweisungen nacheinander aus. Eine Zuweisung der Form $< Variable >$ `:=` $< Ausdruck >$ wertet zuerst den Ausdruck zu einem Wert aus und weist diesen Wert dann der Variablen zu.

   Für einen Zustand $[\mathtt{x} \mapsto 5, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 0]$, der den Variablen `x`, `y` und `z` die Werte 5, 7 und 0 zuweist, ergibt sich folgende Auswertungssequenz:

$$
\begin{aligned}
& \langle \mathtt{z := x;\ x := y;\ y := z},\ [\mathtt{x} \mapsto 5, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 0] \rangle \\
\to_1\ & \langle \mathtt{x := y;\ y := z},\ [\mathtt{x} \mapsto 5, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 5] \rangle \\
\to_1\ & \langle \mathtt{y := z},\ [\mathtt{x} \mapsto 7, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 5] \rangle \\
\to_1\ & [\mathtt{x} \mapsto 7, \mathtt{y} \mapsto 5, \mathtt{z} \mapsto 5]
\end{aligned}
$$

2. Die *denotationale Semantik* kümmert sich nur um den Effekt der Ausführung, nicht um die einzelnen Berechnungsschritte. Entsprechend ist die Semantik eines solchen Programms eine

Funktion, die einen Ausgangszustand in einen Endzustand überführt. Für eine Sequenz erhält man die Funktion durch Komposition (Hintereinanderausführung) der Funktionen der beiden Anweisungen. Die Bedeutung einer Zuweisung ist die Funktion, die den übergebenen Zustand so ändert, dass der Variablen dann der Wert des Ausdrucks zugewiesen ist.

Für das Programm ergibt sich dann:

$$\begin{aligned}
\mathcal{D}[\![\texttt{z := x; x := y; y := z}]\!](\sigma) &= (\mathcal{D}[\![\texttt{y := z}]\!] \circ \mathcal{D}[\![\texttt{x := y}]\!] \circ \mathcal{D}[\![\texttt{z := x}]\!])(\sigma) \\
&= \mathcal{D}[\![\texttt{y := z}]\!](\mathcal{D}[\![\texttt{x := y}]\!](\mathcal{D}[\![\texttt{z := x}]\!](\sigma))) \\
&= \mathcal{D}[\![\texttt{y := z}]\!](\mathcal{D}[\![\texttt{x := y}]\!](\sigma[\texttt{z} \mapsto \sigma(\texttt{x})])) \\
&= \mathcal{D}[\![\texttt{y := z}]\!](\sigma[\texttt{z} \mapsto \sigma(\texttt{x}),\ \texttt{x} \mapsto \sigma[\texttt{z} \mapsto \sigma(\texttt{x})](\texttt{y})]) \\
&= \mathcal{D}[\![\texttt{y := z}]\!](\sigma[\texttt{z} \mapsto \sigma(\texttt{x}),\ \texttt{x} \mapsto \sigma(\texttt{y})]) \\
&= \sigma[\texttt{z} \mapsto \sigma(\texttt{x}),\ \texttt{x} \mapsto \sigma(\texttt{y}),\ \texttt{y} \mapsto \sigma[\texttt{z} \mapsto \sigma(\texttt{x}),\ \texttt{y} \mapsto \sigma(\texttt{y})](\texttt{z})] \\
&= \sigma[\texttt{x} \mapsto \sigma(\texttt{y}),\ \texttt{y} \mapsto \sigma(\texttt{x}),\ \texttt{z} \mapsto \sigma(\texttt{x})]
\end{aligned}$$

Für $\sigma = [\texttt{x} \mapsto 5, \texttt{y} \mapsto 7, \texttt{z} \mapsto 0]$ ergibt dies wieder:

$$\mathcal{D}[\![\texttt{z := x; x := y; y := z}]\!](\sigma) = [\texttt{x} \mapsto 7, \texttt{y} \mapsto 5, \texttt{z} \mapsto 5]$$

3. *Axiomatische Semantik* konzentriert sich auf die Korrektheit eines Programms bezüglich Vor- und Nachbedingungen. Immer, wenn ein Startzustand die Vorbedingung erfüllt, dann sollte – sofern das Programm terminiert – nach der Ausführung des Programms mit diesem Startzustand der Endzustand die Nachbedingung erfüllen.

   $$\{\,\texttt{x} = n \wedge \texttt{y} = m\,\}\,\texttt{z := x; x := y; y := z}\,\{\,\texttt{x} = m \wedge \texttt{y} = n\,\}$$

   Dabei ist $\texttt{x} = n \wedge \texttt{y} = m$ die Vorbedingung und $\texttt{x} = m \wedge \texttt{y} = n$ die Nachbedingung. Die Hilfsvariablen (logical variables) n und m, die nicht im Programm vorkommen, merken sich die Anfangswerte von `x` und `y`.

   Eine axiomatische Semantik definiert ein Regelsystem, mit dem Aussagen wie obige hergeleitet werden können. Beispielsweise ist $\{\,P\,\}\,c_1\texttt{;}\ c_2\,\{\,Q\,\}$ für eine Sequenz $c_1\texttt{;}\ c_2$ herleitbar, wenn $\{\,P\,\}\,c_1\,\{\,R\,\}$ und $\{\,R\,\}\,c_2\,\{\,Q\,\}$ für eine Bedingung R herleitbar sind. Dadurch ergeben sich viele Bedingungen an das Programmverhalten, die dieses dadurch (möglichst vollständig) beschreiben. Wichtig dabei ist, dass die Regeln korrekt sind, d.h, keine widersprüchlichen Bedingungen herleitbar sind. Umgekehrt sollen sich möglichst alle Eigenschaften eines Programms durch das Kalkül herleiten lassen (*Vollständigkeit*).