Programmiersprachen und Übersetzer

Hans H. Brüggemann

Institut für Praktische Informatik
Fakultät für Elektrotechnik und Informatik
Leibniz Universität Hannover

SoSe 2016

Teil I

Sprache als Kommunikationsmittel für Menschen und Rechner

Das Wort „Computer“ in natürlichen Sprachen

Programmiersprachen zur Beschreibung von Algorithmen

- Natürliche Sprachen dienen zur **mündlichen** und **schriftlichen Kommunikation** zwischen Menschen.
- Programmiersprachen dienen zur Kommunikation mit dem Rechner, aber auch zur Kommunikation mit anderen Programmierern.
- Programmiersprachen ermöglichen die Beschreibung von Algorithmen. Die Beschreibung soll für Rechner und Menschen lesbar sein.
- Programmiersprachen haben großen Einfluss auf die Art der Problemlösung und den Algorithmus-Entwurf.

Syntax, Semantik und Pragmatik von Sprachen

Jede Sprache –auch eine Programmiersprache– hat die Aspekte Syntax, Semantik und Pragmatik.

Die Syntax einer Sprache beschreibt
die Beziehung der Zeichen untereinander:

- Nach welchen Regeln des Satzbaus muss ich einen Text schreiben?
- Aus welchen Bestandteilen besteht ein gegebener Text und ist er grammatikalisch korrekt?

Beispiel

Werden Elemente einer Liste durch Komma oder Zwischenraum getrennt?

Syntax, Semantik und Pragmatik von Sprachen

Jede Sprache –auch eine Programmiersprache– hat die Aspekte Syntax, Semantik und Pragmatik.

Die Semantik einer Sprache beschreibt
die Beziehung der Zeichen zur realen Welt:

- Welche Bedeutung will ich mit einem Text vermitteln?
- Was sagt ein gegebener Text aus?

Beispiel

Wird die Abbruchbedingung einer Schleife am Anfang oder Ende geprüft?

Syntax, Semantik und Pragmatik von Sprachen

Jede Sprache –auch eine Programmiersprache– hat die Aspekte Syntax, Semantik und Pragmatik.

Die Pragmatik einer Sprache beschreibt
die Beziehung der Zeichen zu ihren Benutzern:

- Wie formuliere ich, damit mich die Zielgruppe gut versteht?
- Kann ich aus der Formulierung eines Textes die Absicht des Schreibers ableiten?

Beispiel

Ist ein Sprachkonstrukt (z.B. Sprung, Zeiger) fehleranfällig?
Kann ich es oft ersetzen
(z.B. durch Fallunterscheidung bzw. Schleife, durch rekursiven Datentyp)?

Auch die Pragmatik ist wichtig,
denn ein Programm soll nicht nur vom Computer ausgeführt werden,
sondern Menschen (u.a. der Programmierer selbst!) sollen es verstehen.

Von Maschinensprachen zu Assemblersprachen

Ein Programm einer **Maschinensprache** besteht aus Maschinenbefehlen. Jeder Maschinenbefehl ist eine Bitfolge.

- Der **Operationsteil** bestimmt die Klasse des Maschinenbefehls,
- der **Operandenteil** beschreibt den/die Operand(en).

Vom Operationsteil hängt ab, welche und wie viele Operanden folgen und wie lang (in bit) der Maschinenbefehl ist.

Von Maschinensprachen zu Assemblersprachen

Assemblersprachen verbessern vor allem die Lesbarkeit.

- Operationsteil als **mnemotechnischer Code** leicht zu merken, z.B. `jmp`, `add`
- Operanden durch Komma getrennt statt mehrere Bitfolgen fester Länge.
- Konstanten dezimal (oder hexadezimal) statt nur dual.

Beispiel

add 3,2	statt	01010000 00000011 00000010

Absolute Speicher- und Sprungadressen stören die Wiederverwendbarkeit von Programmteilen:

- Speicherplätze werden daher durch (Variablen-)**Namen**,
- Sprungzieladressen durch **symbolische Marken** (*label*) beschrieben.

Von Assemblersprachen zu höheren Programmiersprachen

Höhere Programmiersprachen abstrahieren vom Maschinenmodell:

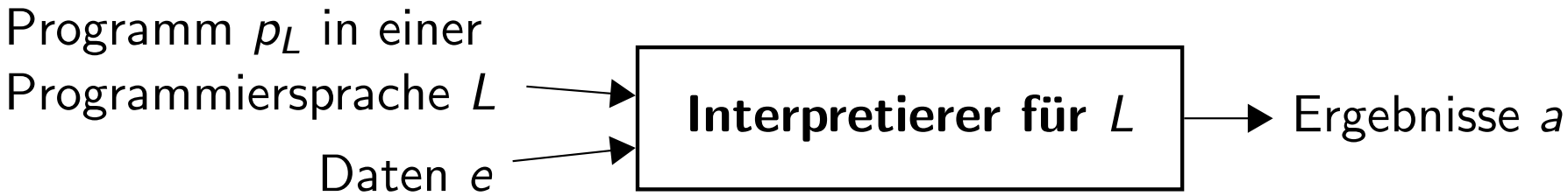
- Ausdrücke können beliebig geschachtelt werden (z.B. $2 + \sin(pi/3) * 3$).
- Zut Steuerung des Programmablaufs werden häufig genutzte Programmiermuster verwendet, z.B. Alternativen (`if...then...else...fi`) und Schleifen (`while...do...od`).
- Durch Deklaration von Namen können Schreibfehler erkannt werden (ermöglicht Fehlermeldung „*undeclared identifier*“ statt Behandlung als zwei verschiedene Variable).
- Neue Einheiten können gebildet werden:
  - Mehrere Ausdrücke bzw. Anweisungen werden zu einer **Funktion** bzw. **Prozedur**.
  - Zusammengehörige Variable werden zum **Datenverbund** oder **Datenfeld**.

Sprachprozessoren: Compiler und Interpretierer

Problem: Rechner verstehen nur ihre eigene Maschinensprache.

Lösung 1: Übersetzung in die Maschinensprache

Lösung 2: Simulation eines Rechners, der L als Maschinensprache hat



Sprachklassen für höhere Programmiersprachen

Höhere **Programmiersprachen** abstrahieren –unterschiedlich stark– von der Struktur und den Details der Rechner, auf denen die Programme ausgeführt werden sollen.

Die wichtigsten Klassen von **universell einsetzbaren Programmiersprachen** enthalten:

- imperative (prozedurale) Programmiersprachen,
- objektorientierte Programmiersprachen,
- funktionale (applikative) Programmiersprachen und
- logische Programmiersprachen.

Sprachklassen für höhere Programmiersprachen

Imperative Programmiersprachen,
z.B. Algol 60, Algol 68, Fortran, Cobol, Pascal, Ada, Modula-2, C.

Sie orientieren sich eng an der Architektur des **von-Neumann-Rechners**

- aus (aktiver) Zentraleinheit (*central processing unit*, CPU),
- (passivem) Speicher und einem
- Bus für den Datenaustausch zwischen Zentraleinheit und Speicher.

Grundeinheit dieser Sprachen ist die **Anweisung** (*statement*).

Anweisungen verändern den **Zustand** (*state*) des Programms und steuern den Programmablauf.

Imperative Sprachen kennen neben Anweisungen auch Ausdrücke. Ausdrücke liefern Werte. Evtl. wird der Zustand sogar bei der Auswertung von Ausdrücken verändert (**Nebenwirkung**). Imperative Sprachen bieten **funktionale Abstraktion** an (aber ggf. mit Nebenwirkungen!).

Sprachklassen für höhere Programmiersprachen

Objektorientierte Programmiersprachen,
z.B. Simula, Smalltalk, Eiffel, C++, Modula-3, Java.

Objektorientierte Sprachen enthalten **Klassen** zur **Datenabstraktion** und das Konzept der **Vererbung** ermöglicht eine „evolutionäre“ Art der Software-Entwicklung.

Sie sind im Kern meist imperativ.

Sprachklassen für höhere Programmiersprachen

Funktionale Programmiersprachen,
z.B. (pure) Lisp, Scheme, Standard ML, OCaml, Haskell.

Wie bei einer mathematischen Funktion liefert ein Funktionsaufruf stets das gleiche Ergebnis – unabhängig vom Programmzustand („frei von Nebenwirkungen“). Dies vereinfacht die Verifikation eines Programms.

Funktionale Sprachen unterscheiden nicht zwischen Anweisungen und Ausdrücken.
Namen bezeichnen Ausdrücke und Funktionen, aber keine Speicherzellen.
Argumente und Ergebnisse von Funktionen können auch Funktionen sein.

Grundeinheit dieser Sprachen ist der **Ausdruck** (*expression*).
Es gibt keinen explizit angegebenen Programmablauf.

Ausführungsprinzip ist die **Reduktion** (*reduction*): Ein Ausdruck wird schrittweise durch einen äquivalenten, einfacheren ersetzt, bis eine **Normalform** (*normal form*) erreicht wird.

Sprachklassen für höhere Programmiersprachen

Logische Programmiersprachen, z.B. Prolog (und seine Dialekte).

Diese Sprachen nutzen eine operationale Sicht der **Prädikatenlogik** (*predicate calculus*) und erlauben **freie Variablen** im Aufruf.

Grundeinheit dieser Sprachen ist die **Hornklausel** (*horn clause*).

Ausführungsprinzip ist die **Resolution** (*resolution*), ein Verfahren um Implikationen in der Prädikatenlogik erster Stufe zu beweisen.

Sprachen für spezielle Anwendungen

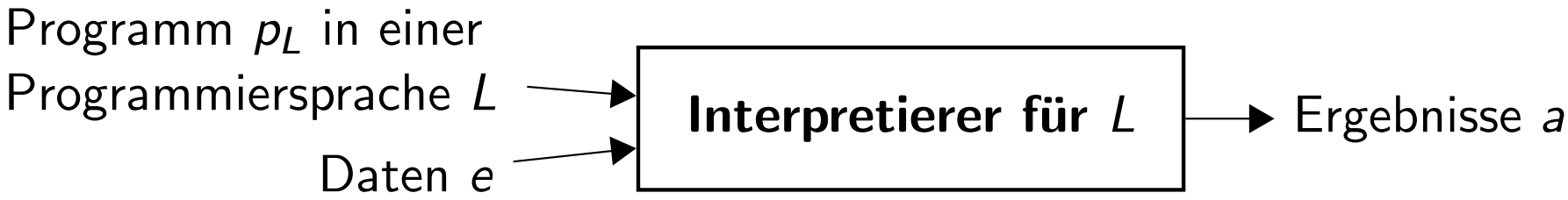
Außerdem gibt es **Sprachen für spezielle Anwendungen**, z.B.

- **Hardware-Beschreibungssprachen** (z.B. VHDL) spezifizieren (Komponenten von) Rechner(n), z.B. ihr funktionales Verhalten, ihren hierarchischen Aufbau oder die räumliche Anordnung ihrer Komponenten.
- **Kommandosprachen von Betriebssystemen** (z.B. Unix shell) ermöglichen, **Prozesse** zu erzeugen, mehrere Prozesse koordiniert zusammenwirken zu lassen oder zu beenden.
- **Datenbanksprachen** (z.B. SQL) für Anfragen an große Datenbestände, oft ohne Ausführungsdetails vorzugeben.
- **Spezifikationssprachen für Druckseiten, Grafikobjekte oder Animationen** (z.B. Postscript) müssen neben den geometrischen Eigenschaften der darzustellenden Objekte auch zeitliche Abläufe und Reaktionen auf Ereignisse beschreiben können.
- **Sprachen für** semi-**strukturierte Dokumente** (z.B. SGML, XML) zur Beschreibung und zum Austausch von Daten(-familien).

Teil II

Sprachprozessoren für Programmiersprachen

Compiler versus Interpretierer



Compiler versus Interpretierer

Im allgemeinen gilt:

- Compiler liefern nach der Übersetzung Programme, deren Ausführung schneller abläuft als die Abarbeitung des Ausgangsprogramms durch einen Interpretierer.
- Eine Änderung des Ausgangsprogramms führt bei der Verwendung eines Compilers zu größerem Aufwand (neues Übersetzen, Binden usw.) als bei Verwendung eines Interpretierers.
- Eine Übersetzung ist zwingend notwendig, wenn die Maschine, auf der das Zielprogramm laufen soll, für einen Sprachprozessor zu klein ist oder dort Ein-/Ausgabemöglichkeiten fehlen.
  Dies kommt oft vor bei **eingebetteten Systemen**, bei dem ein Mikroprozessor Teil einer komplexen Anlage (z.B. Auto) ist.
  Ein **Cross-Compiler** erzeugt dann auf einem größeren Prozessor das Zielprogramm für das eingebettete System.

Übersetzungszeit versus Laufzeit

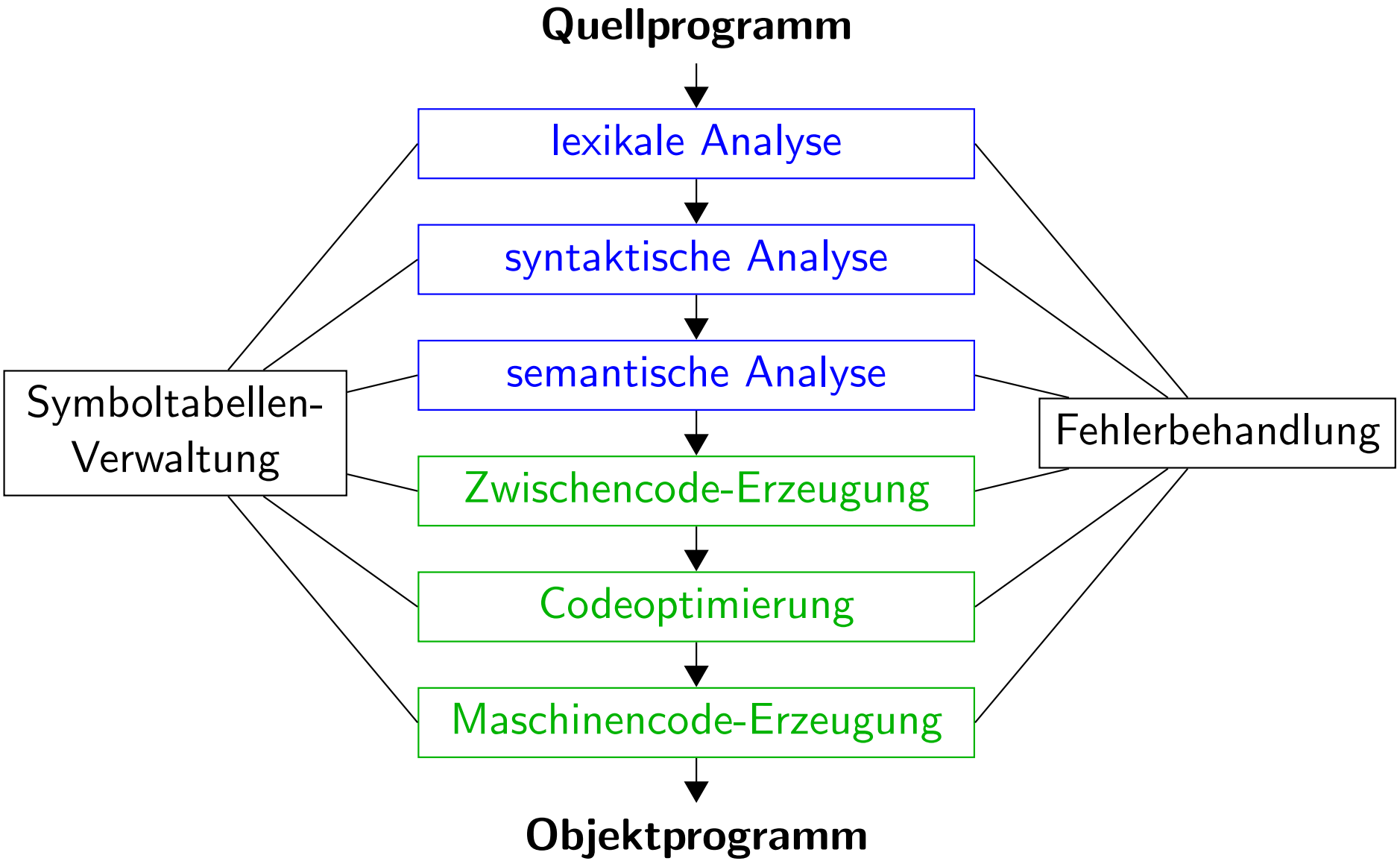
Das Programm und die Eingabe wird zu verschiedenen Zeiten verarbeitet:

- Zur **Übersetzungszeit** wird das Programm vorverarbeitet, d.h. unabhängig von Eingabedaten analysiert und in eine Form überführt, die die effiziente Ausführung des Programms mit beliebigen Eingaben erlaubt.
- Zur **Ausführungszeit** oder **Laufzeit** wird das erzeugte Zielprogramm mit der Eingabe ausgeführt.

Eigenschaften eines Programms,
die nicht von den Eingabedaten abhängen, heißen **statisch**;
solche, die von den Eingabedaten abhängen, heißen **dynamisch**.

Statische Eigenschaften können zur Übersetzungszeit ermittelt werden,
dynamische Eigenschaften erst zur Laufzeit.

Prinzipieller Aufbau eines Compilers



Aufgabe der lexikalen Analyse

- Lesen der Eingabe Zeichen für Zeichen

Beispiel (31 Zeichen)

Eingabe:

```
position := initial + rate * 60
```

- Zusammenfassen von Zeichen zu einer größeren Einheit (Lexem).
  Im Compiler wird ein Lexem durch ein Token dargestellt.

Beispiel (7 Token)

Eingabe:

```
position := initial + rate * 60
```

- Überlesen von Leerzeichen und Kommentaren

Aufgabe der lexikalen Analyse

Ein Token repräsentiert eine Folge von Eingabezeichen, die als Einheit betrachtet wird.
Ein Token wird einer Tokenklasse zugeordnet und hat oft einen Tokenwert.

Diese Komponente des Übersetzers heißt (Lexical) Scanner oder Tokenizer.

Lexikale Analyse

Beispiel

Die Eingabe `position := initial + rate * 60` könnte folgende Tokenfolge liefern:

Tokenklasse:	Tokenwert:
`Identifier`	`position`
`Wertzuweisungszeichen`	
`Identifier`	`initial`
`Pluszeichen`	
`Identifier`	`rate`
`Multiplikationszeichen`	
`Zahl`	`60`

Aufgabe der syntaktischen Analyse

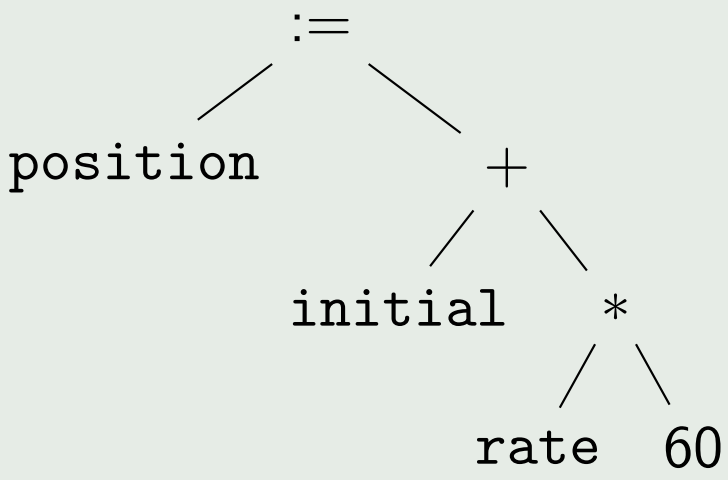
- Prüfung, ob die von der lexikalen Analyse gelieferten Token in einer Reihenfolge auftreten, die der (syntaktischen) Beschreibung der Programmiersprache entspricht.
- Gruppieren der Token zu größeren syntaktischen Strukturen, etwa in Form eines Ableitungsbaumes oder eines Syntaxbaumes.

Diese Komponente des Übersetzers heißt Syntaxanalysator oder Parser.

Syntaktische Analyse

Beispiel

Die Token können zu folgendem Syntaxbaum zusammengefasst werden:



Aufgabe der semantischen Analyse

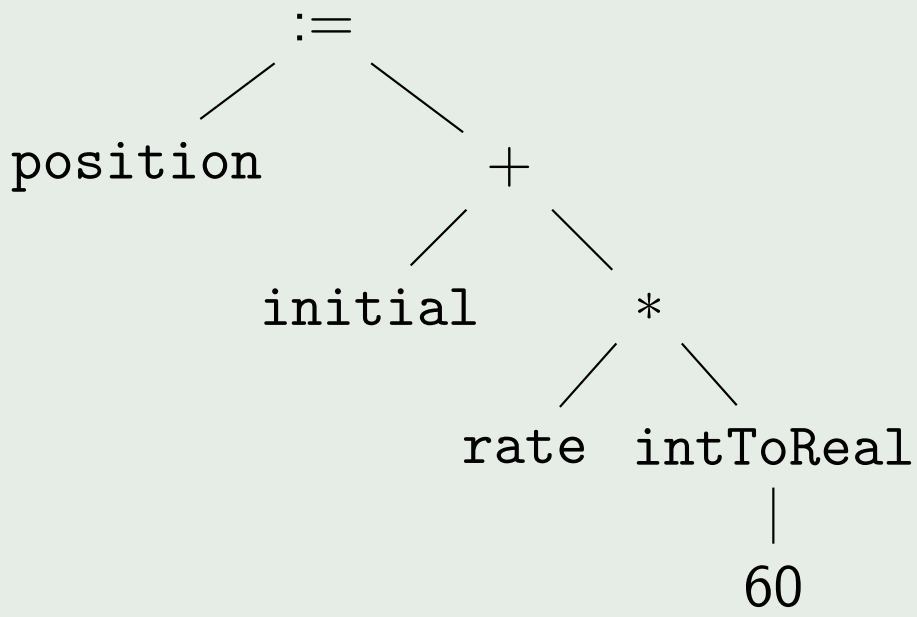
Die semantische Analyse überprüft z.B. folgendes:

- Wurden die verwendeten Identifier auch im Programm deklariert?
- Passen die Typen von Konstanten und Variablen zu den auf sie angewendeten Operationen?
  (Ggf. Fehlermeldung oder automatische Typanpassung)
- Stimmt die Anzahl der Parameter?

Semantische Analyse

Beispiel

Sind alle Identifier vom Typ `real`, muss die `int`-Konstante 60 in die `real`-Konstante 60.0 konvertiert werden:

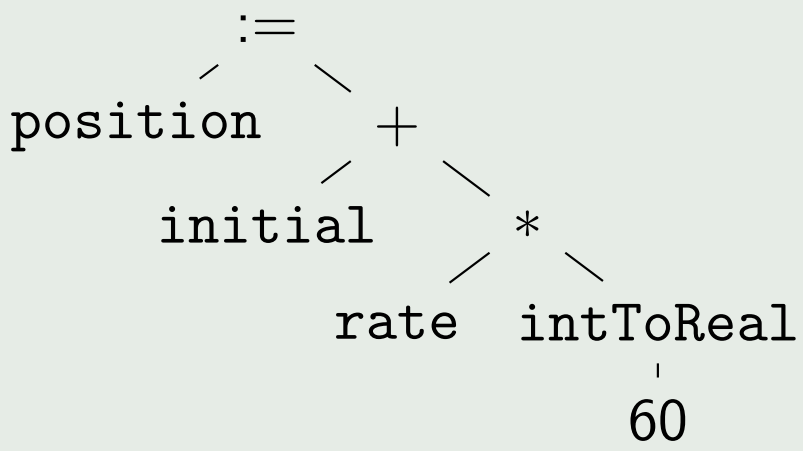


Aufgabe der Zwischencode-Erzeugung

- Übersetzung in einen Zwischencode,
  d.h. in die Maschinensprache einer **virtuellen Maschine**, die eine möglichst einfache Struktur hat.
  - Eine Möglichkeit sind erweiterte Syntaxbäume oder Graphen.
  - Eine andere Möglichkeit eines Zwischencodes sind 3-Adress-Befehle:
    `A := B op C` ,
    wobei `A` das Ergebnis, `B` und `C` die Operanden und op den Operator bezeichnet.

Zwischencode-Erzeugung

Beispiel



Das Beispiel könnte folgende Übersetzung in 3-Adress-Befehle liefern:

```
t1 := intToReal 60
t2 := rate * t1
t3 := initial + t2
position := t3
```

Dabei sind `t1`, `t2` und `t3` neu erzeugte Namen.

Aufgabe der Codeoptimierung

Verbesserung des Zwischencodes bzgl.

- Speicherplatzbedarf und
- Laufzeit

Codeoptimierung

Dazu benötigt man Algorithmen, die z.B.

- überflüssige Berechnungen entfernen,
- nicht ereichbaren Code entfernen,
- Berechnungen zur Laufzeit durch Berechnungen zur Übersetzungszeit ersetzen,
- schleifeninvariante Berechnungen erkennen und diese Teile vor den Schleifenbeginn verschieben.

Beispiel

Vor der Codeoptimierung:

```
t1 := intToReal 60
t2 := rate * t1
t3 := initial + t2
position := t3
```

Nach der Codeoptimierung:

```
t2 := rate * 60.0

position := initial + t2
```

Aufgabe der Code-Erzeugung

Erzeugung von verschiebbarem Maschinencode (d.h. eines Objektprogramms) bzw. von Assemblercode.

Die Hardware-Register sollen so zugeordnet werden, dass möglichst wenig Speicherzugriffe nötig sind.

Beispiel

Die Floating-Point-Register R1 und R2 müssen frei sein.

```
MOVF rate, R2
MULF #60.0, R2
MOVF initial, R1
ADDF R2, R1
MOVF R1, position
```

Symboltabellen-Verwaltung

Die Symboltabelle speichert die im Programm vorkommenden **Namen**.

Zusätzlich werden weitere Informationen abgespeichert, z.B.

bei Variablennamen

- der Typ der Variable,
- die zugeordnete Relativadresse im Datenspeicher und
- ggf. die Schachtelungstiefe der Deklaration
  (um verschiedene Variablen gleichen Namens zu unterscheiden);

bei Typnamen z.B.

- die Größe des Typs.

Fehlerbehandlung

Programmierer wären unzufrieden, wenn der Übersetzer die Übersetzung nur mit der Meldung

„Ihr Programm ist kein Wort der Programmiersprache“

oder

„Ihr Programm enthält noch Fehler“

beenden würde.

Gewünscht sind hilfreiche Angaben über **Position** und **Art der Fehler**.

Dazu muss der Übersetzer nicht nur merken, dass das Programm kein Wort der Programmiersprache ist, sondern aus der Art der Abweichung auf mögliche Fehlerquellen schließen.

Reale und virtuelle Maschinen

Programmierer hat **reale Maschine**: Rechner mit Prozessor und Speicher.

Die Zielsprache der Übersetzung ist durch den Prozessortyp bestimmt, für den man **nativen Code** erzeugt.

Probleme:

- Für die Übersetzung würde man oft gern Befehle verwenden, die es für die reale Maschine nicht gibt.
- Befehlssatz moderner Rechner ändert sich schnell.

Abhilfe:
Code für idealisierte **virtuelle Maschine** erzeugen, deren Befehle dann auf den verschiedenen realen Zielrechnern leicht zu implementieren sind.

Vorteile:

- **Portierbarkeit** des Übersetzers wird leichter.
- Übersetzung wird einfacher, da Befehlssatz passend zur Programmiersprache gewählt werden kann.

Reale und virtuelle Maschinen

Internet-Anwendungen machen virtuelle Maschinen zusätzlich attraktiv.

Durch die Portierbarkeit können Systeme unter verschiedenen Betriebssystemen –**plattformunabhängig**– lauffähig werden.

Dann kann der Code **mobil** (über das Internet verbreitet) werden.
Das ist eine der Ideen hinter der Programmiersprache Java.

Ausführung fremden Codes auf dem eigenen Rechner erzeugt Gefahren!

Lösung durch virtuelle Maschine:
Da Code nicht unmittelbar auf der eigenen Hardware läuft, kann man das Verhalten des auszuführenden Codes beobachten und Zugriffsrechte auf Resourcen des Rechners einschränken: **Sandkasten-Prinzip** (*sand boxing*).

Sprachprozessoren können kombiniert werden

Kombinationen von Übersetzung und Interpretation sind möglich:

Meist werden die in die Sprache einer virtuellen Maschine übersetzten Quellprogramme durch einen Interpretierer interpretiert.

Allerdings kann man die Sprache der virtuellen Maschine auch weiter übersetzen, z.B. in die Sprache eines realen Rechners.

Sprachprozessoren können kombiniert werden

Wann und auf welcher Maschine erfolgt dieser zweite Übersetzungsschritt?

Die zweite Übersetzung kann auch zur Ausführungszeit erfolgen (**just-in-time Übersetzung**, **JIT**).
Ziel der JIT-Übersetzung: Kombination von Effizienz und Portierbarkeit.

Beide Übersetzungsschritte auf verschiedenen Rechnern:

- Der erste Schritt passiert auf dem Rechner des Programmierers.
  Erzeugt wird portabler Zwischencode für virtuelle Maschine.
  Dieser sehr aufwändige Übersetzungsschritt wird nur einmal gemacht.
- Der Zwischencode wird auf anderen Rechner heruntergeladen.
  Hier findet die zweite, einfache Übersetzung statt und erzeugt einen effizienten nativen Code für diesen Rechner.

  Der native Code wird in einem lokalen Speicher (*cache*) gesammelt, so dass jede Funktion nur einmal übersetzt wird.

Die Architektur der virtuellen Maschinen

Jede virtuelle Maschine stellt **Befehle** (*instruction*) zur Verfügung, die auf einer virtuellen Hardware ausgeführt werden.

Diese virtuelle Hardware wird meist in Software **emuliert**.

Der Ausführungszustand wird dabei in Datenstrukturen gespeichert, auf die die Befehle zugreifen und die vom **Laufzeitsystem** verwaltet werden.

Der Befehlssatz der virtuellen Maschinen wird schrittweise eingeführt, so wie wir sie gerade für die aktuellen Konzepte der Quellsprache brauchen.

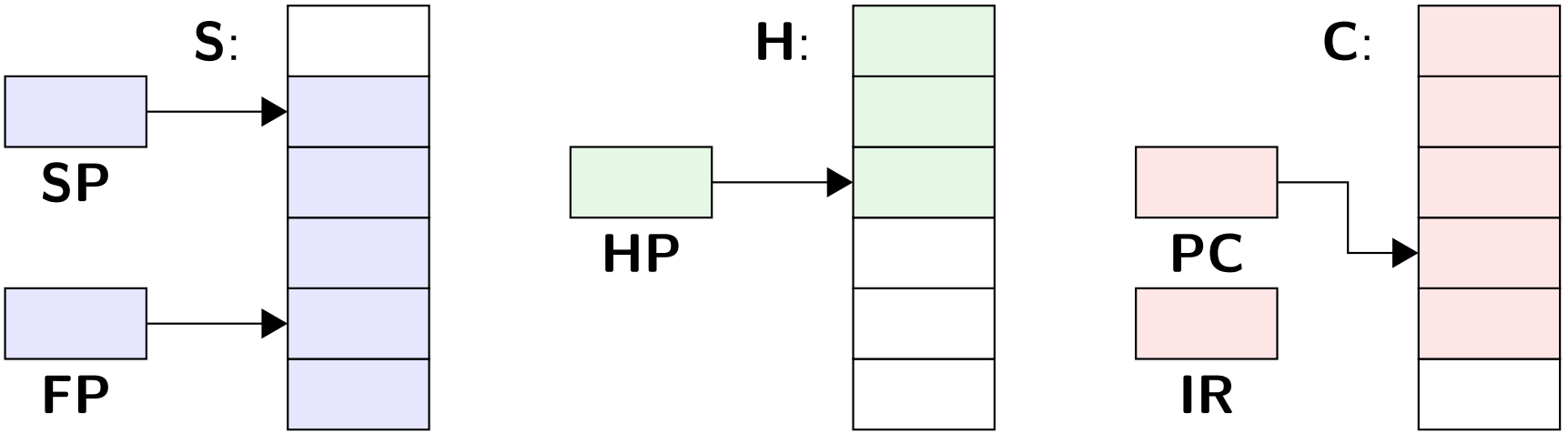
Die Architektur der virtuellen Maschinen

Die Architektur der virtuellen Maschinen ist recht ähnlich.

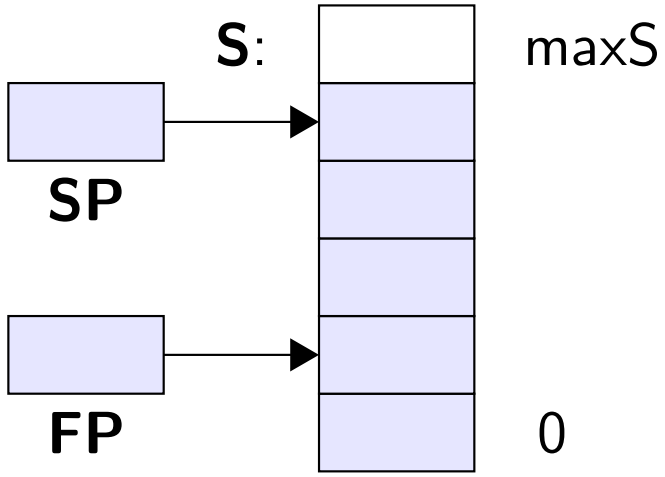
Wir beschreiben daher im folgenden die Gemeinsamkeiten:
die Speichersegmente, die Register und den Hauptausführungszyklus.

Weitere Speichersegmente und Register werden bei Bedarf eingeführt.

Virtuelle Maschinen haben zwei Daten- und einen Programmspeicher:

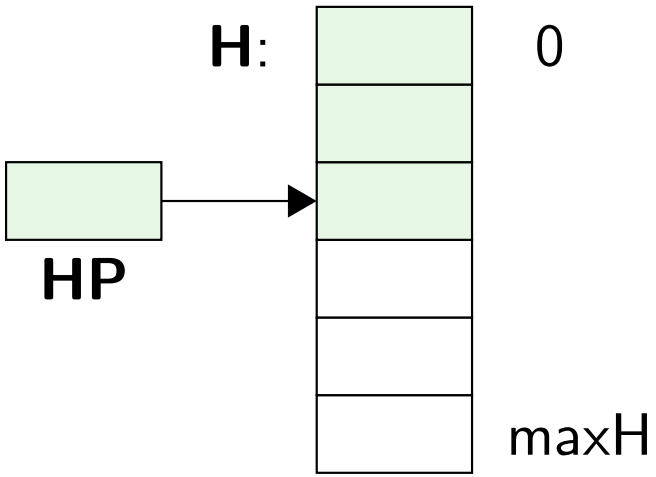


Architektur der virtuellen Maschinen: Keller



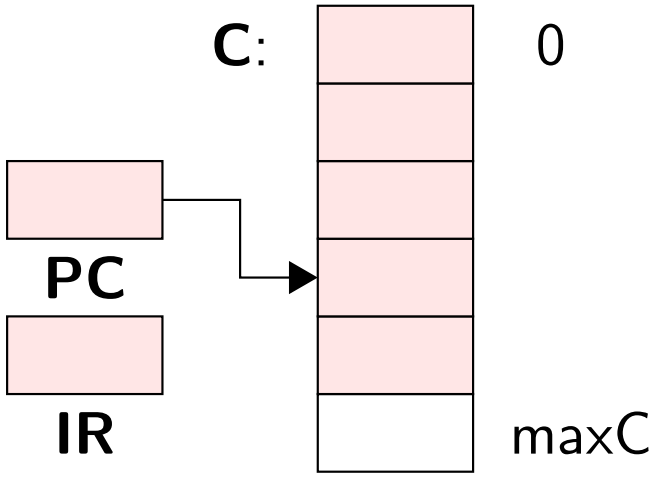
- einen **Keller** (*stack*) S für deklarierte Daten und Zwischenergebnisse.
  Jede Zelle sollte ein Datenelement (Zahl, Adresse) aufnehmen können.
  Der **Kellerzeiger**, ein Register **SP** (*stack pointer*), zeigt stets auf die letzte (oberste) belegte Zelle des Kellers.
  Der **Rahmenzeiger** (*frame pointer*), ein Register **FP**, zeigt auf den Kellerrahmen, in dem die lokalen Variablen des aktuellen Funktionsaufrufs liegen.

Architektur der virtuellen Maschinen: Halde



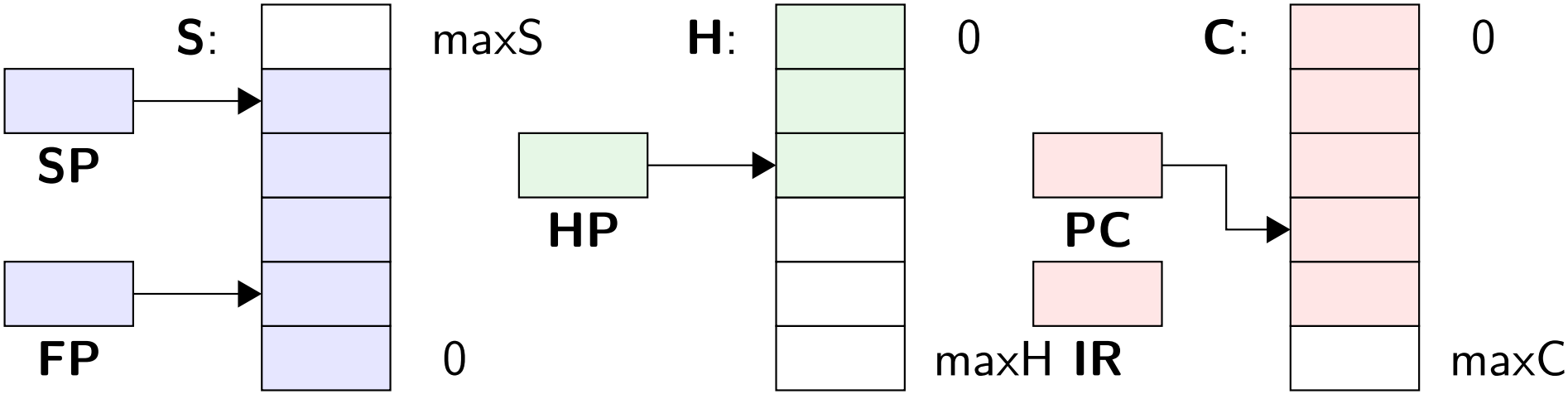
- eine **Halde** (*heap*) H für dynamische Daten.
  Der **Haldenzeiger** (*heap pointer*), ein Register **HP**, zeigt stets auf die letzte (unterste) belegte Zelle der Halde.

Architektur der virtuellen Maschinen: Programmspeicher



- einen **Programmspeicher** C für das auszuführende Programm.
  Der **Befehlszähler**, ein Register **PC** (*program counter*), zeigt auf den nächsten auszuführenden Befehl im Programmspeicher.
  Dieser wird ins **Befehlsregister IR** (*instruction register*) geladen und anschließend ausgeführt.
  Alle Befehle der virtuellen Maschine belegen genau eine Zelle im Programmspeicher, so dass der Inhalt des Registers PC nach jedem Befehl um 1 erhöht wird. Nur bei einem Sprung wird der Inhalt des Befehlszählers PC mit der Zieladresse überschrieben.

Die Architektur der virtuellen Maschinen



Aus praktischen Gründen zählen wir Adressen

- beim Keller von unten nach oben,
  so dass der Keller –wie gewohnt– nach oben wächst und
- sonst von oben nach unten.

Architektur der virtuellen Maschinen: Hauptzyklus

Der **Hauptausführungszyklus** (*main cycle*) der virtuellen Maschinen lautet daher:

```
while (true) {
    IR ← C[PC];
    PC++;
    execute(IR);
}
```

Anfangs hat das Register PC den Wert 0, d.h. die Programmausführung beginnt mit dem Befehl C[0].

Die virtuelle Maschine hält an, indem sie den Befehl **halt** ausführt.

Teil III

Zwischencode-Erzeugung: Funktionale Abstraktion und imperative Programmiersprachen

Code-Erzeugung für eine imperative Programmiersprache

Dieses und das nächste Kapitel sollen eine Vorstellung vermitteln, **was** ein Übersetzer tut, ohne schon genau zu erklären, **wie** er das tut.

Wir definieren dazu intuitiv die Korrespondenz zwischen Programmen einer imperativen Quellsprache und den übersetzten Zielprogrammen.

Als Quellsprache wählen wir einen Ausschnitt der Programmiersprache C. Allerdings haben wir die konkrete Syntax geringfügig angepasst:

- Eine Wertzuweisung $x = e$ schreiben wir $x \leftarrow e$ und
- einen Vergleich $e_1 == e_2$ schreiben wir $e_1 = e_2$.
- Eine Bedingung klammern wir mit Schlüsselwörtern statt mit runden Klammern, z.B. schreiben wir **if** e **then** s für **if** (e) s.

Code-Erzeugung für eine imperative Programmiersprache

Als **Zielsprache** der Übersetzung wählen wir die Maschinensprache einer virtuellen Maschine, die wir dafür während der Übersetzung entwerfen. Diese virtuelle Maschine nennen wir **C-Maschine**.

Die bei der Übersetzung erzeugten Befehlsfolgen werden schrittweise für die behandelten C-Konstrukte mit Hilfe einer Übersetzungsfunktion code und einer Speicherbelegungsfunktion ρ angegeben.

Wir kümmern uns noch nicht um das Problem, wie ein C-Programm syntaktisch analysiert wird.

Ebenso sei die Typkorrektheit des Eingabeprogramms schon geprüft.

Konzepte imperativer Programmiersprachen

- Der **Zustand** eines Programms kann durch eine **Anweisung** verändert werden (z.B. Wertzuweisung).

  Ein **Wert** kann in einer (Programm-)**Variablen** gespeichert werden, die als Behälter für Datenobjekte dient.
  (Der mathematische Begriff einer Variable ist ein anderer!)

  Der Wert kann sich im Laufe der Programmausführung ändern und damit der Zustand des Programms.

  Variable (Konstante, Funktionen) werden durch **Namen** bezeichnet.

  Einer Variablenbezeichnung wird die **Adresse** einer Speicherzelle der Maschine zugeordnet, diese Speicherzelle (und ggf. die nachfolgenden) enthalten den aktuellen Wert der Variablen.

Konzepte imperativer Programmiersprachen

- Ein **Ausdruck** ist ein Term aus Konstanten, Namen und Operatoren, der bei der Programmausführung **ausgewertet** wird.
  Der Wert eines Ausdrucks hängt normalerweise vom Zustand ab, da bei jeder Auswertung die aktuellen Werte der im Ausdruck enthaltenen Variablen (und Funktionsaufrufe) benutzt werden.

- **Anweisungen** steuern den **Programmablauf** explitzit:
  Der **Sprung** (goto) –falls noch zugelassen– kann direkt in den unbedingten Sprungbefehl der Zielmaschine übersetzt werden.

  Eine **Alternative** (if) oder eine **Schleife** (while, do-while, repeat, for) wird mit Hilfe bedingter Sprünge übersetzt:
  Auf die Befehlsfolge für die Bedingung folgt ein bedingter Sprung.

  Eine allgemeine **Fallunterscheidung** (case, switch) lässt sich durch indizierte Sprünge realisieren. Dazu wird die im Befehl angegebene Sprungadresse mit einem vorher berechneten Wert modifiziert.

Konzepte imperativer Programmiersprachen

- Die Definition einer **Funktion** macht aus einer Folge von Anweisungen eine neue Anweisung (**funktionale Abstraktion**).

  Ein **Aufruf** dieser Funktion erzeugt für die in der Definition angegebene Folge von Anweisungen eine **Inkarnation** dieser Funktion.

  Nach ihrer Abarbeitung wird mit dem Funktions-**Ergebnis** an die Aufrufstelle zurückgekehrt.

  Ein Unterprogramm-Sprung muss sich daher seine Herkunft merken.

  Durch **Parameter** kann man unterschiedliche Inkarnationen der Funktion aufrufen. Der Rumpf der Funktion muss bei jedem Aufruf mit **aktuellen Parametern** versorgt werden.

Konzepte imperativer Programmiersprachen

- Der Rumpf einer **rekursiven Funktion** enthält das aktuell zu definierende Funktionssymbol. Streng genommen ist diese Gleichung keine Definition, sondern eine einschränkende Bedingung.

  Diese Gleichung hat evtl. keine Lösung (wenn z.B. die Ausführung nicht terminiert) oder mehrere Lösungen (und eine wird ausgewählt: „kleinster Fixpunkt“).

- Bei jedem Aufruf einer Funktion benötigt die Inkarnation Speicherplatz für ihre lokalen Variablen.

  Nach Verlassen dieser Funktion wird dieser Speicherplatz nicht mehr benötigt.

  Dieser Speicher wird daher wie ein **Keller** organisiert.

Speicherbelegung für Variablen

Jede Variable des Programms erhält zur Übersetzungszeit eine **Adresse** im Datenspeicher S, wo ihr Wert zur Laufzeit gespeichert wird.

Der Übersetzer nutzt diese Adressen, um Code zu erzeugen, der die Werte der Variablen von dort lädt oder dort speichert.

Diese Information wird den Übersetzungsfunktionen zur Verfügung gestellt als Funktion ρ, die zu jeder Variablen x deren Adresse liefert.
ρ heißt **Speicherbelegungsfunktion** oder **Adressumgebung**.

Speicherbelegungsschema

Für jeden Typ t müssen wir seine **Größe** $|t|$ festlegen, nämlich die Anzahl der benötigten Speicherzellen für einen Wert dieses Typs.

Die einfachen Datentypen **int**, **float**, **char** sollen mit *einer* Speicherzelle auskommen (Zeiger ebenso).

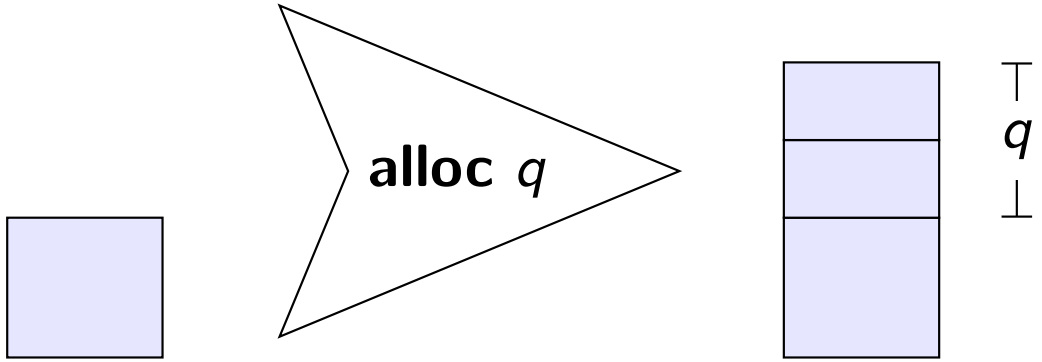
Schema zur Speicherbelegung: Den Variablen im Deklarationsteil des Programms ordnen wir in der Reihenfolge ihres Vorkommens lückenlos Adressen am Anfang des Kellerspeichers zu.

Seien im C-Programm die Variablen $t_1\ x_1; \ldots; t_k\ x_k;$ deklariert. Dann gilt:

$$\rho(x_1) = 1 \qquad \text{und} \qquad \rho(x_i) = \rho(x_{i-1}) + |t_{i-1}| \text{ , für } i > 1.$$

Befehl **alloc**

Um q Speicherplätze im Keller *zur Laufzeit* zu reservieren, führen wir die Befehle **alloc** q ein. Für die Freigabe von Speicherplatz im Keller nutzen wir ebenfalls die **alloc**-Befehle – mit einem negativen Argument.



$SP \leftarrow SP + q;$

Der Befehl **alloc** q.

Bevor die Auswertung von Ausdrücken beginnt, wird mit *einem* **alloc**-Befehl der Speicherplatz für *alle* deklarierten Variablen (des Hauptprogramms oder einer Funktionsinkarnation) reserviert.

Relativadressen

Die Adresse einer Variablen ist eine **Relativadresse**,
d.h. eine konstante Differenz zwischen zwei absoluten Adressen in S:
der tatsächlichen Adresse der Zelle für diese Variable und der Anfangsadresse des Speicherbereichs ihrer Funktion (für das Hauptprogramm 0).

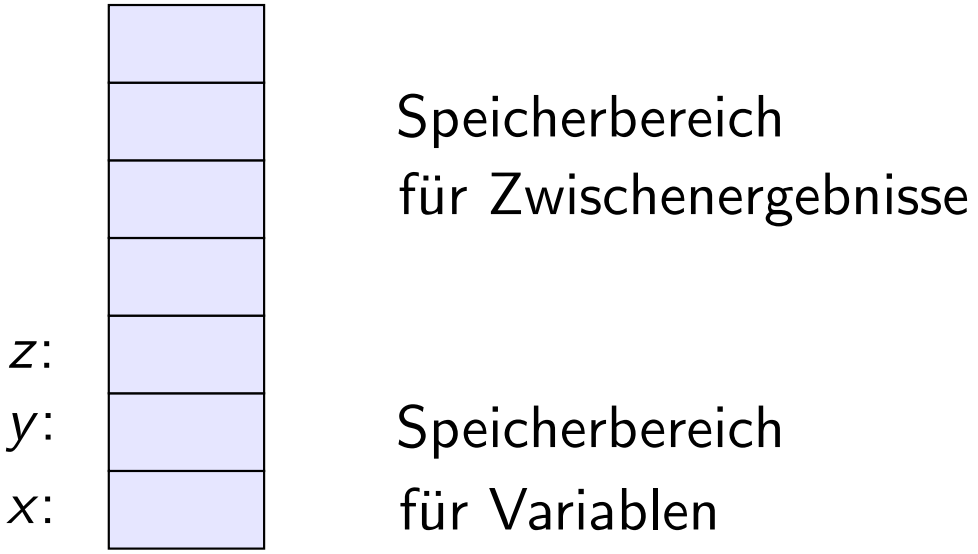
Diese Relativadressen sind statische Größen;
denn sie ergeben sich aus dem Quellprogramm,
genauer aus der Position der Variablen im Deklarationsteil.

Speicherbelegung für Variablen

Diese Adressen liegen im Keller der C-Maschine.

Später wird klar, dass es zwei ineinander geschachtelte Keller gibt:

- einen „großen“ Keller, der aus den Datenbereichen aller zu einem Zeitpunkt aktiven Funktionen besteht und daher wächst bzw. schrumpft, wenn eine Funktion betreten bzw. verlassen wird, und
- einen „kleinen“ Keller zu jeder aktiven Funktion für ihre Variablen und Zwischenergebnisse.



Der „kleine“ Keller.

Übersetzung von arithmetischen und logischen Ausdrücken

Problem: Erzeuge eine Befehlsfolge für die virtuelle Maschine, die den Wert eines Ausdrucks berechnet und oben auf dem Keller hinterlässt,

$$\text{z.B. } (1+7)\cdot(5-2)$$

Lösungsidee:

- Berechne zunächst die Werte der Teilausdrücke,
- lege diese Argumente oben auf den Keller,
- wende den Operator an und
- ersetze die Argumente durch das Ergebnis.

Wir berechnen also zuerst die Werte der Teilausdrücke $1+7$ und $5-2$, hinterlassen ihre Werte in den oberen Kellerzellen und wenden dann den äußeren Operator „$\cdot$“ auf diese Werte an.

Die Anwendung des Operators löscht die Zwischenergebnisse 8 und 3 auf dem Keller und hinterlässt das Ergebnis 24 oben auf dem Keller.

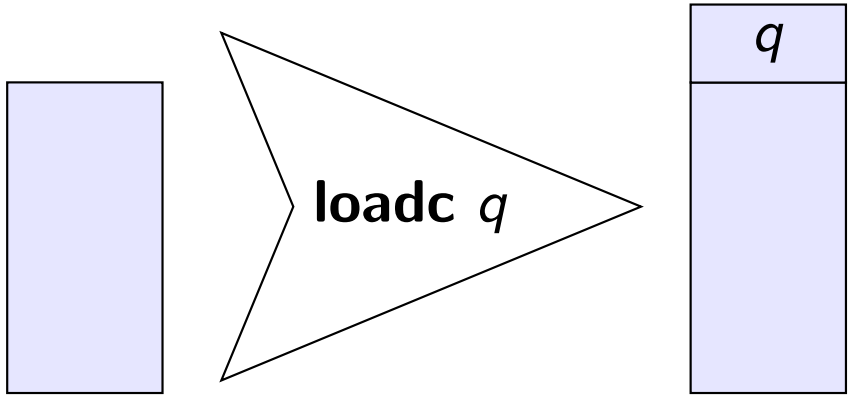
Übersetzung von arithmetischen und logischen Ausdrücken

Entwurfsentscheidung zur Berechnung von Ausdrücken:
Ein Befehl erwartet seine Operanden oben auf dem Keller und ersetzt sie diese durch das Ergebnis.

Das rekursive Vorgehen (zur Berechnung der Ausdrücke) passt gut zu unserer Entwurfsentscheidung.

Besteht der Ausdruck nur aus einer Konstanten (z.B. 7), benötigen wir einen Befehl, der diesen Wert oben auf den Keller schreibt.

Der Befehl **loadc** q erwartet keine Operanden auf dem Keller. Er legt als Ergebnis die Konstante q oben auf dem Keller ab.

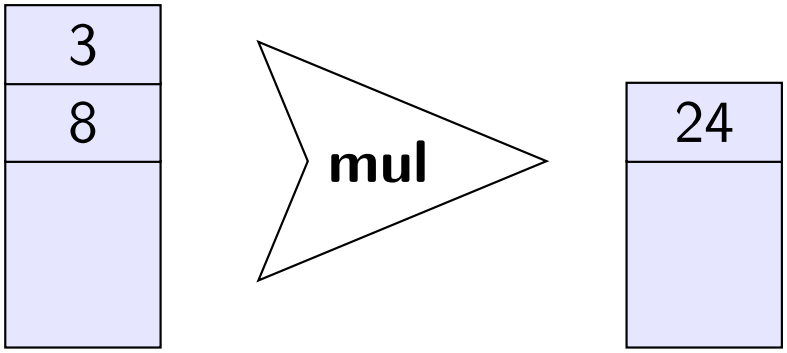


$SP{+}{+}; \quad S[SP] \leftarrow q;$

(Der Inhalt von SP wird durch die Höhe des Kellers beschrieben.)

Übersetzung von arithmetischen und logischen Ausdrücken

Der binäre Multiplikations-Befehl **mul** erwartet zwei Argumente oben auf dem Keller, multipliziert sie, löscht die Operanden und schreibt das Ergebnis oben auf den Keller: $8 \cdot 3$.



$$S[SP-1] \leftarrow S[SP-1] \cdot S[SP]; \quad SP--;$$

Analog arbeiten die **binären Befehle**:

- die arithmetischen Befehle **add**, **sub**, **div**, **mod**,
- die Vergleiche **eq**, **neq**, **le**, **leq**, **gr** und **geq** sowie
- die logischen Befehle **and** und **or**.

Übersetzung von arithmetischen und logischen Ausdrücken

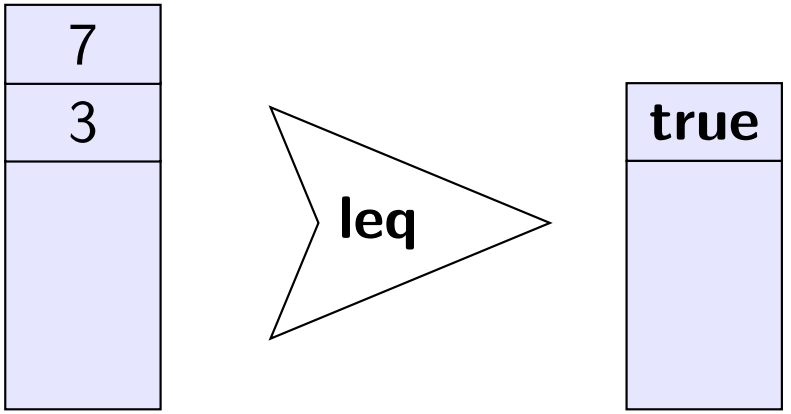
Ergebnis einer Vergleichsoperation ist ein logischer Wert, also **true** oder **false**.

In C werden Wahrheitswerte durch ganze Zahlen dargestellt:

0 steht für **false**,

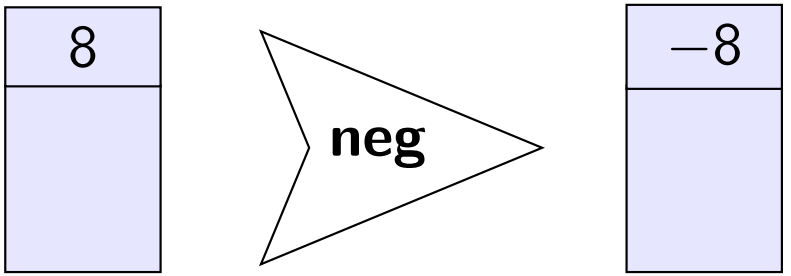
alle anderen Werte für **true**.

Hier vergleicht der **leq**-Befehl zwei ganze Zahlen: $3 \leq 7$.



$S[SP-1] \leftarrow S[SP-1] \leq S[SP];$
$SP--;$

Unäre Befehle wie **neg** und **not** haben nur einen Operanden.
Hier dreht der **neg**-Befehl das Vorzeichen einer Zahl um.



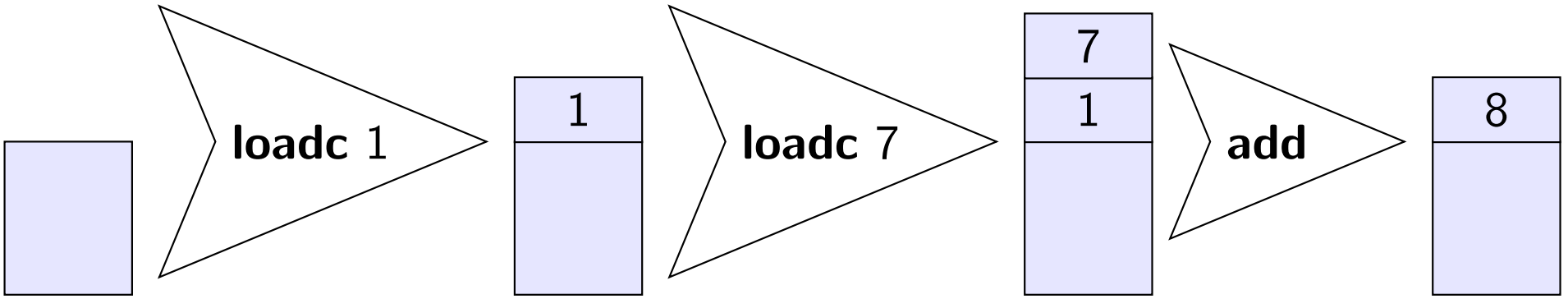
$S[SP] \leftarrow -S[SP];$

Übersetzung von arithmetischen und logischen Ausdrücken

Der Ausdruck $1 + 7$ wird also in folgende Befehle der virtuellen Maschine übersetzt:
loadc 1, **loadc** 7, **add**

loadc 1
loadc 7
add

Der Programmspeicher nach Übersetzung des Ausdrucks $1 + 7$.



Abarbeitung der Befehlsfolge für $1 + 7$ im Keller zur Laufzeit.

Übersetzung von arithmetischen und logischen Ausdrücken

Die Code-Erzeugung wird durch **Übersetzungsfunktionen** beschrieben.

Diese Funktionen bekommen als Argumente ein Programmfragment und eine Adressumgebung ρ.

Sie zerlegen das Programmfragment rekursiv und setzen die für die Komponenten erzeugten Befehlsfolgen zu einer Befehlsfolge zusammen:

$$\text{code}^{\rho}_{W}\ (e_1 + e_2) = \text{code}^{\rho}_{W}\ e_1,\ \text{code}^{\rho}_{W}\ e_2,\ \textbf{add}\ \ //\ \text{andere binäre Op. analog}$$
$$\text{code}^{\rho}_{W}\ (-e) = \text{code}^{\rho}_{W}\ e,\ \textbf{neg}\qquad //\ \text{andere unäre Operatoren analog}$$
$$\text{code}^{\rho}_{W}\ q = \textbf{loadc}\ q\qquad //\ \text{Konstante}$$

Der Index W kennzeichnet die Übersetzungsfunktion zur Berechnung von Werten.

Wertzuweisung

In imperativen Sprachen werden Variablen auf zwei Arten verwendet.

Für eine Wertzuweisung $x \leftarrow y + 1$ benötigen wir offenbar

- den **Wert** der Variablen y, aber
- die **Adresse** der Variablen x.

Eine Variable auf der *linken* Seite einer Wertzuweisung muss daher anders übersetzt werden als eine auf der *rechten* Seite.

Wir unterscheiden daher verschiedene Übersetzungsfunktionen:

- code_W erzeugt Befehle zur **Berechnung eines Wertes**;
- code_A erzeugt Befehle zur **Berechnung einer Adresse**;
- code (ohne Index) übersetzt **Anweisungen**.

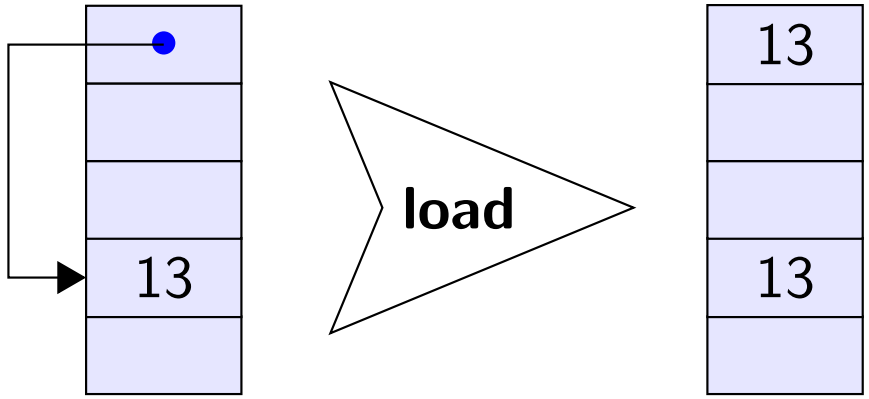
Wertzuweisung

$\text{code}^{\rho}_{A}\ x = \textbf{loadc}\ \rho(x)$ // Variable auf der linken Seite ...

$\text{code}^{\rho}_{W}\ x = \text{code}^{\rho}_{A}\ x, \textbf{load}$ // ... rechten Seite einer Wertzuweisung

Die Adresse der Variablen x wird der Adressumgebung ρ entnommen.

Um den Wert einer Variablen x zu ermitteln, benötigen wir einen Befehl **load**, der den Inhalt einer Speicherzelle, deren Adresse oben auf dem Keller liegt, oben auf den Keller lädt.



$S[SP] \leftarrow S[S[SP]];$

Jeder Ausdruck hat einen Wert, aber nicht immer eine Adresse.
Z.B. ist der Wert des Ausdrucks $y + 1$ nur temporär vorhanden und deshalb nicht adressierbar.

Wertzuweisung

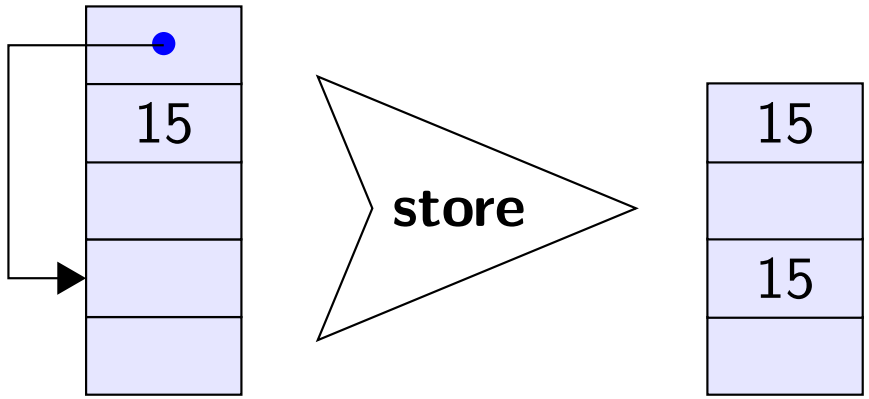
In C ist eine **Wertzuweisung** ein **Ausdruck**!
Der Wert dieses Ausdrucks ist der Wert der rechten Seite der Zuweisung.

Der Wert der Variablen auf der linken Seite der Zuweisung ändert sich als **Nebenwirkung** der Auswertung des Ausdrucks.

Für die Wertzuweisung benötigen wir den Befehl **store**.

Der Befehl **store** erwartet zwei Argumente auf dem Keller: einen Wert w und darüber eine Adresse a.

Den Wert w schreibt er an der Adresse a in den Speicher und lässt ihn als Ergebnis oben auf dem Keller zurück.



$S[S[SP]] \leftarrow S[SP-1]; \quad SP--;$

Eine Wertzuweisung $x \leftarrow e$ übersetzen wir durch:
$\text{code}_W^\rho(x \leftarrow e) = \text{code}_W^\rho\ e, \text{code}_A^\rho\ x,$ **store**