Programmiersprachen

Vorlesung 1 vom 16.10.23

Einführung

Christoph Lüth

Universität Bremen

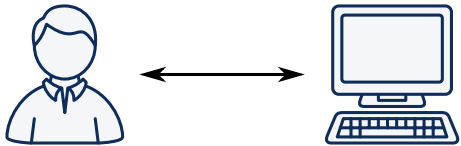
Wintersemester 2023/24

Einführung

Worum geht es?

- Es gibt über 700 Programmiersprachen (sagt Google)
- Wie kriegen wir da Ordnung rein?
- Zugrundeliegende Prinzipien
  - Was haben alle Programmiersprachen gemein?
  - Wo gibt es Unterschiede?
  - **Taxonomie** der Programmiersprachen
- Neue Programmiersprachen lernen (neue, alte, merkwürdige)

Warum Programmiersprachen?



- Wollen Programme in **verständlicher** Notation aufschreiben
- Maschine soll sich dem Menschen anpassen (nicht umgekehrt)
- Programme müssen **maschinenlesbar** und **auführbar** bleiben
- **Modellbildung** und **Abstraktion**

Konzept

Konzept der Veranstaltung

- Integrierter **Kurs** mit zwei Terminen pro Woche:
  - Montag 10- 12 MZH 1100
  - Donnerstag 08:30- 10 MZH 1100
- **Übungen**:
  - Ein Übungsblatt pro Woche.
  - Wird teilweise in den Terminen behandelt, Abgabe bis nächste Woche Montag.
  - **"Leichtgewichtige"** Korrektur.
  - Abgabe und Korrektur über FB3-gitlab.

  Bitte `clueth@uni-bremen.de` als Developer in euer Repo einladen!
- **Referate**:
  - Ab 11. Woche (*i.e.* ab 2024)
  - Studierende stellen je **eine** neue Sprache vor

Scheinbedingungen

1. Mind. 75% der **Übungsblätter** korrekt bearbeitet (50% oder mehr).
2. **Referat** über eine neue Sprache.
3. Mündliche **Prüfung** am Ende.

- Note: 25% Referat, 75% Prüfung

Welche Sprachen betrachten wir?

- Laufende Beispiele:
  - C
  - Rust
  - Java
  - Python
  - Haskell
- Weitere in den Referaten

Liste weiterer Sprachen

- Logische Programmierung: Prolog, Oz
- Dynamisch: JavaScript
- Nebenläufig/Reaktiv: Erlang, Golang
- Abhängige Typen: Idris, Agda, Liquid X (Dependent types)
- Prozedural: Julia, Kotlin, Swift
- Skriptsprachen: Lua, Tcl, sh/bash
- Funktional: SML, OCAML, Elm, Clojure, LISP, Scala
- Stack-basiert: Forth
- Historisch: COBOL, Algol-68, APL, Ada, Smalltalk
- Datenflusssprachen: Id, Lucid, Lustre
- DSLs: R, SQL, Postscript, TeX, Verilog/VHDL, SystemC, SpinalHDL

Struktur der Veranstaltung

- Einführung
- Einfache Ausdrücke und Werte
- Einfache Typen, Anweisungen und Seiteneffekte
- Variablen und Speichermodelle
- Aggregierende Typen
- Ausnahmen und Fehlerbehandlung
- Prozeduren und Funktionen
- Fortgeschrittene Typsysteme
- Datenabstraktion
- Programmierparadigmen
- Skriptsprachen
- Ab Woche 11 (2024): Studentische Vorträge.

Literatur und Basis

- David A. Watt: **Programming Language Design Concepts**, John Wiley & Sons, 2004.
- Maurizio Gabbrielli, Simone Martini: **Programming Languages: Principles and Paradigms**. Springer, 2010.
- Robert W. Sebesta: **Concepts of Programming Languages**. Pearson Education, 2016.
- Norman Ramsey: **Programming Languages. Build, Prove and Compare**. Cambridge University Press, 2023.

Vorkenntnisse

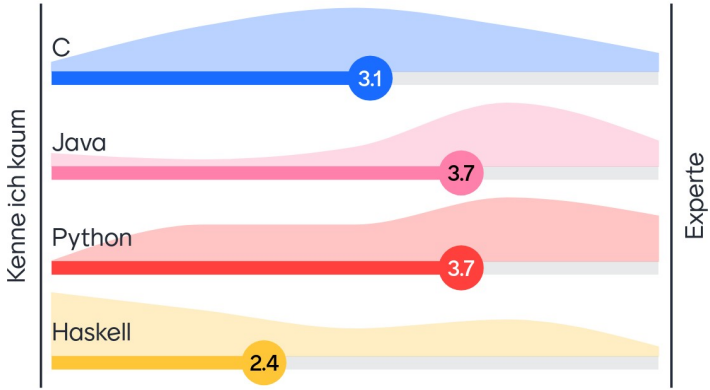
Online-Umfrage:
`https://www.menti.com/al8j27pcua32`

- Vorkenntnise in folgende Sprachen:
  - C
  - Java
  - Python
  - Haskell
- Welche weiteren Sprachen kennt ihr?

Ergebnisse I

Mentimeter

Vorkenntnisse



Ergebnisse II

Mentimeter

Welche anderen Sprachen kennt ihr?

66 responses

Grundlagen & Geschichte

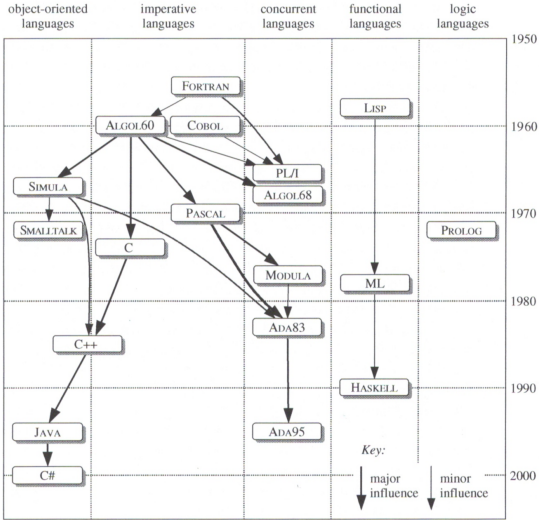
Was ist eine Programmiersprache?

1. Definierte, maschinenlesbare **Syntax**
2. Mathematisch, informell oder pragmatisch definierte **Semantik**
3. Die Sprache muss **ausführbar** und **Turing-mächtig** sein

Arten von Programmiersprachen

- Programmiersprachen sind immer **Abstraktionen** über einem Berechnungsmodell.
- **Imperativ**: Zustandsübergänge auf einem Speicher (Turing-Maschine)
  - Abstraktion durch Datentypen
  - Abstraktion durch Verkapselung
- **Funktional**: Rekursive Funktionen (Auswertung von Ausdrücken)
- Sonstige: logische, domänenspezifisch, Quantencomputer, ...

Historisches: Stammbaum einiger Programmiersprachen



Was ist ein Computer?

- Ein Computer:
  1. ist elektronisch und digital;
  2. beherrscht die vier arithmetischen Operationen ($+$, $-$, $\cdot$, $/$);
  3. kann programmiert werden;
  4. erlaubt die Speicherung von Programmen und Daten.
- (2)– (4) garantiert **Turing-Vollständigkeit**.

Die ersten Computer

- Zuse Z3 (Zuse, 1941): sogar mit Fließkommarithmetik, aber nicht elektronisch;
- ENIAC (Mauchly, Eckert, von Neumann, 1946): nicht reprogrammierbar
- EDSAC (Wilkes, 1949): erfüllt alle vier Kriterien
- Thomas Watson, IBM (1943): “Es gibt einen Weltmarkt von 5 Computern”

Generationen von Programmiersprachen:

- Programmiert in **Maschinensprache**
  - Programmiersprachen der ersten Generation — 1 GL
- Symbolische Repräsentation: **Assemblersprachen** (2 GL)
- Hochsprachen: 3GL
- 4GL: Nicht exakt definiert
  - “Programming without the Programmer”, CASE, . . .
  - Model-driven development, DSLs

Die erste Hochsprachen: FORTRAN

- Hardware war **teuer** als Arbeitskraft — deshalb **Programmeffizienz** wichtig
- FORTRAN (1957): FORmula TRANslator
- Erste Programmiersprache mit symbolischer Notation `a*2+ b`
- Entwickelt für numerische Berechnungen
- Lief auf der IBM 704

John Backus
(1924–2007)

- Turing Award (1977)
- FORTRAN
- Backus-Naur-Form

Die ersten Hochsprachen: COBOL

- COBOL (1960): COmmon Business Oriented Language
- Erste Sprache für Business-Anwendungen
- Standardisiert, Design by committee

Grace Hopper
(1906–1992)

- Erfand “Bugs”
- Allgemeinverständliche Programmierung
- Beeinflusste COBOL

Die ersten Hochsprachen: ALGOL

- ALGOL (Algorithmic Language): Ursprung der modernen Programmiersprachen
- Erste Version 1958, 1960 (ALGOL-60), spätere Versionen (ALGOL-68)
- Amerikanisch-Europäische Koproduktion
- Erste Sprachen mit formal definierter Grammatik (BNF), Blöcken, strukturierter Programmierung, call-by-name
- "ALGOL-Syntax":

```
for p := 1 step 1 until n do
  for q := 1 step 1 until m do
    if abs(a[p, q]) > y then
      begin
        y := abs(a[p, q]);
        i := p; k := q
      end
```

Funktionale Sprachen: LISP

- LISP (LISt Processor):
  die erste funktionale Sprache
- 1960 von John McCarthy am MIT entworfen
- Speziell für nicht-numerische Probleme (KI)

- Basiert auf dem Lambda-Kalkül
- Alles ist eine S-Expression

LISP-Maschine:
Symbolics 3640

John McCarthy
(1927–2011)

- Turing Award (1971)
- LISP
- Pionier der KI

Die 1970er Jahre

- C (Dennis Ritchie & Ken Thompson, 1972):
  - Portable Assembler-Sprache
  - Implementationssprache für UNIX
- Pascal (Niklaus Wirth, 1970):
  - Virtuelle Maschine (P-Code)
  - Strukturiert, block-orientiert, stark getypt
- Smalltalk (Alan Kay, 1970):
  - Objektorientiert, GUI integriert
- ML (Robin Milner, 1974):
  - Für den LCF-Beweiser, Hindley-Milner-Polymorphie
  - Standardisiert (1983), mathematisch formal definierte Semantik (1997)
- Prolog (Bob Kowalski, 1974):
  - Logische Programmierung, Ausführung durch **Resolution**

Die 1980er und 1990er Jahre

- C++ (Stroustrup, 1986): objektierientierte Erweiterung von C
- Ada (1983): vom DoD standardisiert, sehr komplexer Standard.
  - Erster Compiler 1986
- Erlang (Armstrong, 1986–92): für verteilte und nebenläufige Applikationen, Fa. Ericsson
- Java (Gosling *et al*, 1990): zuerst "Oak", für Set-Top-Boxen.
  - Objektorientiert, JVM, Applets — portabel und sicher
- Haskell (1987, 1998): nicht-strikt und funktional,
- Python (van Rossum, 1991): leichtgewichtig, einfach zu nutzen
- JavaScript (Eich, 1995): Skriptsprache für den Browser

Wie geht's weiter?

- Einführung
- Einfache Ausdrücke und Werte
- Einfache Typen, Anweisungen und Seiteneffekte
- Variablen und Speichermodelle
- Aggregierende Typen
- Ausnahmen und Fehlerbehandlung
- Prozeduren und Funktionen
- Fortgeschrittene Typsysteme
- Datenabstraktion
- Programmierparadigmen
- Skriptsprachen
- Ab Woche 11 (2024): Studentische Vorträge.

Programmiersprachen

Vorlesung 2 vom 19.10.23

Einfache Ausdrücke

Christoph Lüth

Universität Bremen

Wintersemester 2023/24

Wo sind wir?

- Einführung
- Einfache Ausdrücke und Werte
- Einfache Typen, Anweisungen und Seiteneffekte
- Variablen und Speichermodelle
- Aggregierende Typen
- Ausnahmen und Fehlerbehandlung
- Prozeduren und Funktionen
- Fortgeschrittene Typsysteme
- Datenabstraktion
- Programmierparadigmen
- Skriptsprachen
- Ab Woche 11 (2024): Studentische Vorträge.

Ausdrücke

Ausdrücke und Anweisungen

- Viele Sprachen unterscheiden **Ausdrücke** und **Anweisungen**
  - Ausdrücke bezeichnen die zu manipulierenden **Werte**
  - Anweisungen beschreiben **Kontrollfluß**
  - **Imperatives** Konstrukt: Programm ist abstrakte Sicht auf Speichermanipulation
- Funktionale Sprachen, logische Sprachen (**deklarative**) uvm. haben diese Unterscheidung **nicht**.

Werte

- Was sind Werte?
  "Any entity that can be manipulated by a program."
  - **Primitive** Werte:
    Booleans, ganze Zahlen, Fließkommazahlen, Adressen (Pointer, Referenzen)
  - **Zusammengesetzte** Werte (composite values):
    Strukturen, Felder, Listen, Objekte, ...
- Semantisch gesehen:
  - Abstraktionen über dem Speicherinhalt
  - Nichtreduzierbare Ausdrücke

Typen

- Was sind Typen?
  - "A set of values." (Mengen von Werten)
  - Durch die Operationen darauf charakterisiert. z.B. $\{13, \textit{Monday}, \textit{true}\}$ ist kein Typ.
    — kann man drüber streiten
- Arten von Typen:
  - **Primitive** Typen (primitive types)
  - **Zusammengesetzte** Typen (composite types)
- Typüberprüfung und Typsysteme

Vorgegebene Primitive Typen

- Anwendungsabhängig: COBOL vs. FORTRAN; BCPL vs. C vs. TEX
- Ganze Zahlen (feste Wortbreite):
  - Java: `int`
  - C: `char`, `short`, `int`, `long`, `long long`, etc.
  - Rust: `i8`,...,`i128`, `u8`,...,`u128`, `isize`, `usize`
  - Haskell: `Int`
- C und Rust haben `signed` und `unsigned` ($\mathbb{Z}$ und $\mathbb{N}$)
- Vorzeichen meist mit **Zweierkomplement.**

Vorgegebene Primitive Typen

- Beliebig große ganze Zahlen:
  - `Integer`, `Natural` in Haskell
  - `BigInteger` in Java
  - Basiert auf Gnu MP Bücherei
- Booleans:
  - In C **kein** expliziter Typ (`bool\_t` ist `int`)
- Fließkommazahlen:
  - `float` und `double` (in Rust: `f32` und `f64`).
  - Meist nach IEEE 754

Übung 1.1: Welche Wortbreite haben ganze Zahlen in C, Java, Python, Haskell?

Selbstdefinierte Primitive Typen

- C, Java, Haskell erlauben **Aufzählungen:**

```
enum colour {red, green, blue}
```

- In C nur syntaktischer Zucker für `int`, kein separater Typ.
- In Java: Klasse mit konstanter Anzahl von Objekten (s. hier)
- In Haskell und Rust: algebraischer Typ mit ausschließlich konstanten Konstruktoren.

```
data Colour = Red | Green | Blue
```

```
enum Colour  { Red, Green, Blue }
```

Zeichenketten

- Zeichenketten (Strings) spielen **meist** eine Sonderrolle
- Konzeptionell: Array oder Liste von Zeichenketten, *e.g.* C und Haskell:

```c
char txt1[] = "Foo";
char txt2[4] = {'F', 'o', 'o', 0};
```

```haskell
type String = [Char]
```

- Aus Effizienzgründen: **Unveränderliche** Strings
  - Java und Python
  - `bytestring` in Haskell

Auswertung

Ausdrücke

- Wir beschreiben die Semantik mit Hilfe einer **vereinfachten Sprache**:

$$
\begin{aligned}
e ::= \mathbb{Z} \mid \textbf{Idt} \mid \mathit{true} \mid \mathit{false} \\
\mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \\
\mid e_1 == e_2 \mid e_1 < e_2 \\
\mid \,!\, e \mid e_1 \,\&\&\, e_2 \mid e_1 \,||\, e_2
\end{aligned}
$$

- Dieses ist **abstrakte Syntax**. Terme können als **Bäume** repräsentiert werden.

Strukturelle Operationale Semantik

- Induktiv definiert durch **Regeln** der Form

$$\frac{\langle a', t' \rangle \to b' \qquad \phi(a')}{\langle a, t \rangle \to b}$$

- t ist ein Baum der Tiefe 1 (ein oberstes Symbol mit variablen Kindern)
- a sind Eingabedaten (*e.g.* der Zustand), b Rückgabe (*e.g.* ein Wert)
- Die Anwendung einer Regel entspricht einem **Zustandsübergang**

Zustände

Zustände sind **partielle, endliche Abbildungen** (finite partial maps) repräsentiert durch **rechtseindeutige** Relationen

$$f : X \rightharpoonup A \subseteq X \times A \text{ so dass } \forall x, a, b.\, (x, a) \in f \wedge (x, b) \in f \implies a = b$$

Notation:

- $\langle x \mapsto a, y \mapsto b, z \mapsto c \rangle$ für $\{(x, a), (y, b), (z, c)\}$
- $f(x)$ für den Wert von x in f (*lookup*)
- $f(x) = \bot$ wenn x nicht in f (*undefined*) und $\mathrm{def}(f(x))$ für $(x, y) \in f$ (*defined*)
- $f \setminus x$ um x aus f zu entfernen (*remove*)
- $f[x \mapsto n]$ für den **Update** an der Stelle x mit dem Wert n.

Regeln der Operationalen Semantik

Ein Ausdruck e wertet unter Zustand σ zu einer ganzen Zahl n oder einen boolschen Wert b aus.

$$e ::= \mathbb{Z} \mid \mathbf{Idt} \mid \mathit{true} \mid \mathit{false} \mid \ldots \qquad \langle e, \sigma \rangle \to_{Exp} n \mid b$$

Regeln:

$$\frac{i \in \mathbb{Z}}{\langle i, \sigma \rangle \to_{Exp} i} \qquad \frac{b \in \mathbb{B}}{\langle b, \sigma \rangle \to_{Exp} b} \qquad \frac{x \in \mathbf{Idt}, x \in \mathrm{dom}(\sigma), \sigma(x) = v}{\langle x, \sigma \rangle \to_{Exp} v}$$

Operationale Semantik: Arithmetische Ausdrücke

$$e ::= \ldots \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1/e_2 \mid \ldots \qquad \langle e, \sigma \rangle \to_{Exp} n$$

Regeln:

$$\frac{\langle e_1, \sigma \rangle \to_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \to_{Exp} n_2 \qquad n_i \in \mathbb{Z}}{\langle e_1 + e_2, \sigma \rangle \to_{Exp} n_1 + n_2}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \to_{Exp} n_2 \qquad n_i \in \mathbb{Z}}{\langle e_1 - e_2, \sigma \rangle \to_{Exp} n_1 - n_2}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \to_{Exp} n_2 \qquad n_i \in \mathbb{Z}}{\langle e_1 * e_2, \sigma \rangle \to_{Exp} n_1 \cdot n_2}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \to_{Exp} n_2 \qquad n_i \in \mathbb{Z}, n_2 \neq 0}{\langle e_1/e_2, \sigma \rangle \to_{Exp} n_1 \div n_2}$$

Operationale Semantik: Prädikate

$$e ::= \ldots \mid e_1 == e_2 \mid e_1 < e_2 \mid \ldots \quad \langle e, \sigma \rangle \rightarrow_{Exp} b$$

Regeln:

$$\frac{\langle e_1, \sigma \rangle \rightarrow_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \rightarrow_{Exp} n_2 \qquad n_i \in \mathbb{Z}, n_1 = n_2}{\langle e_1 == e_2, \sigma \rangle \rightarrow_{Lexp} \mathit{true}}$$

$$\frac{\langle e_1, \sigma \rangle \rightarrow_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \rightarrow_{Exp} n_2 \qquad n_i \in \mathbb{Z}, n_1 \neq n_2}{\langle e_1 == e_2, \sigma \rangle \rightarrow_{Lexp} \mathit{false}}$$

$$\frac{\langle e_1, \sigma \rangle \rightarrow_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \rightarrow_{Exp} n_2 \qquad n_i \in \mathbb{Z}, n_1 < n_2}{\langle e_1 < e_2, \sigma \rangle \rightarrow_{Lexp} \mathit{true}}$$

$$\frac{\langle e_1, \sigma \rangle \rightarrow_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \rightarrow_{Exp} n_2 \qquad n_i \in \mathbb{Z}, n_1 \geq n_2}{\langle e_1 < e_2, \sigma \rangle \rightarrow_{Lexp} \mathit{false}}$$

Operationale Semantik: Konnektive

$$e ::= \ldots \mid !\, e \mid e_1 \,\&\&\, e_2 \mid e_1 \,||\, e_2 \quad \langle e, \sigma \rangle \to_{Exp} b$$

Regeln:

$$\frac{\langle e, \sigma \rangle \to_{Lexp} true}{\langle !\, e, \sigma \rangle \to_{Lexp} false} \qquad \frac{\langle e, \sigma \rangle \to_{Lexp} false}{\langle !\, e, \sigma \rangle \to_{Lexp} true}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Lexp} false}{\langle e_1 \,\&\&\, e_2, \sigma \rangle \to_{Lexp} false} \qquad \frac{\langle e_1, \sigma \rangle \to_{Lexp} true \qquad \langle e_2, \sigma \rangle \to_{Lexp} t}{\langle e_1 \,\&\&\, e_2, \sigma \rangle \to_{Lexp} t}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Lexp} true}{\langle e_1 \,||\, e_2, \sigma \rangle \to_{Lexp} true} \qquad \frac{\langle e_1, \sigma \rangle \to_{Lexp} false \qquad \langle e_2, \sigma \rangle \to_{Lexp} t}{\langle e_1 \,||\, e_2, \sigma \rangle \to_{Lexp} t}$$

Zusammenfassung

- Ausdrücke werden im Kontext eines **Zustands** zu **Werten** ausgewertet.
- Zustände sind partielle endliche Abbildungen.
- Strukturelle Operationale Semantik beschreibt diese Auswertung **mathematisch**.
- Nicht alle Ausdrücke lassen sich auswerten.
- Typen versuchen diese Undefiniertheit im Vorfeld auszuschließen.

Programmiersprachen

Vorlesung 3 vom 23.10.23

Einfache Typen, Anweisungen und Seiteneffekte

Christoph Lüth

Universität Bremen

Wintersemester 2023/24

Organisatorisches

- Übungen: 8 Repos, Dateiaustauschtest
- Bitte `uebung-XX.md` nicht umbenennen.
- Vorlesung am 30.10.2023 **findet statt**.

Wo sind wir?

- Einführung
- Einfache Ausdrücke und Werte
- Einfache Typen, Anweisungen und Seiteneffekte
- Variablen und Speichermodelle
- Aggregierende Typen
- Ausnahmen und Fehlerbehandlung
- Prozeduren und Funktionen
- Fortgeschrittene Typsysteme
- Datenabstraktion
- Programmierparadigmen
- Skriptsprachen
- Ab Woche 11 (2024): Studentische Vorträge.

Typen

Warum Typen?

- Evaluation kann “stecken bleiben” (Bsp. $x/0$).

Übung 2.1: Aus welchen Gründen reduziert ein Ausdruck nicht zu einem Wert?

Warum Typen?

- Evaluation kann “stecken bleiben” (Bsp. $x/0$).

Übung 2.1: Aus welchen Gründen reduziert ein Ausdruck nicht zu einem Wert?

- Wie können wir das verhindern?

- Durch **Typisierung**

Typüberprüfung

- **Statische** Typsysteme:
  - Typen werden zur Compile-Zeit geprüft.
  - Zur Laufzeit werden keine Typinformationen benötigt (“type erasure”)
  - Sicher und effizient, aber unflexibel
  - Bsp: C, Rust, Haskell, Java
- **Dynamische** Typsysteme
  - Typen werden zur Laufzeit geprüft.
  - Flexibel, aber fehleranfällig.
  - Bsp: Python, Java (dynamische Bindung von Methoden)

Implizit vs. explizit

- **Explizit** oder manifeste Typsysteme: alle Typen werden **explizit** angegeben
  - Compiler muss nur prüfen
  - Beispiel: Java
- **Implizite** Typsysteme: Typen werden abgeleitet
  - Compiler muss Typ inferieren
  - Beispiel: Rust, Haskell, Hindley-Milner-Typsystem
  - Problem: Entscheidbarkeit, bspw. mit Subtyping unentscheidbar

Frage: Ist Python implizit oder explizit?

Typinferenz

- *type judgements* sind von der Form $\Gamma \vdash e : t$
- $\Gamma \in \mathbf{Idt} \rightharpoonup \mathcal{T}ypes$ ist eine **Typumgebung**
- e ist ein Ausdruck
- t ist ein Typ $t \in \mathcal{T}ypes = \{\mathbf{int}, \mathbf{bool}, \mathbf{unit}\}$
- *type judgements* werden durch **Typinferenz** hergeleitet

Regeln für die Typinferenz

$$\frac{}{\Gamma \vdash n : \textbf{int}} \qquad \frac{x \in \mathrm{dom}(\Gamma), \Gamma(x) = T}{\Gamma \vdash x : T}$$

$$\frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 + e_2 : \textbf{int}} \qquad \frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 - e_2 : \textbf{int}}$$

$$\frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 * e_2 : \textbf{int}} \qquad \frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 / e_2 : \textbf{int}}$$

$$\frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 == e_2 : \textbf{bool}} \qquad \frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 < e_2 : \textbf{bool}}$$

$$\frac{\Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : \textbf{bool}}{\Gamma \vdash e_1 \,\&\&\, e_2 : \textbf{bool}} \qquad \frac{\Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : \textbf{bool}}{\Gamma \vdash e_1 \,||\, e_2 : \textbf{bool}} \qquad \frac{\Gamma \vdash e : \textbf{bool}}{\Gamma \vdash !\, e : \textbf{bool}}$$

Eigenschaften der Typinferenz

- Typinferenz ist **entscheidbar**.

- Was bedeuten folgende Eigenschaften (und gelten sie)?

$$\forall \Gamma, \sigma, e, t. \Gamma \vdash e : t \Longrightarrow \exists v. \langle e, \sigma \rangle \to v \tag{1}$$

$$\Gamma \vdash e : \textbf{int} \wedge \langle e, \sigma \rangle \to v \Longrightarrow v \in \mathbb{Z} \tag{2}$$

Anweisungen

Einfache Anweisungen:

- Kernsprache:
  - Zuweisung
  - Sequenzierung und leere Anweisung — **Sequenzen** von Anweisungen
  - Fallunterscheidung
  - Iteration
    - `while`, `repeat`, Rekursion
  - $\Longrightarrow$ Turing-mächtig

Mehr Anweisungen

- `for`-Schleifen:
  - C, Java: syntaktischer Zucker für `while`
  - Rust, Python: **Iterator**

```
m = {'a' : 1, 'b' : 5, 'c': 7 }
for k in m:
  print(k)
```

```
let array = [1u32, 3, 3, 7];
for i in array.iter() {
    println!("> {}", i);
}
```

Mehr Anweisungen

- Sprünge: `goto` etc. — "considered harmful"[1]
  - C kennt `goto` und `longjmp()`
  - Java und Rust haben Labels
- Manche Sprachen unterscheiden Ausdrücke und Anweisungen nicht
  - In C sind Zuweisungen Ausdrücke, und Ausdrücke Anweisungen
  - In Haskell ist alles ein Ausdruck

[1] Edsger W. Dijkstra. 1968. Letters to the editor: go to statement considered harmful. Commun. ACM 11, 3 (March 1968), 147-148. https://doi.org/10.1145/362929.362947

Operationale Semantik

Abstrakte Syntax

- Anweisungen:

$$
\begin{aligned}
c ::= \ & \mathbf{Idt} := \mathbf{Exp} \\
\mid \ & c_1; c_2 \\
\mid \ & \mathbf{nil} \\
\mid \ & \mathbf{if}\ (e)\ \mathbf{then}\ c_1\ \ \mathbf{else}\ \ c_2 \\
\mid \ & \mathbf{while}\ (e)\ c
\end{aligned}
$$

- Operationale Semantik: $\langle c, \sigma \rangle \rightarrow_{Stmt} \sigma'$

Regeln der operationalen Semantik I

- Regeln:

$$\frac{\langle e, \sigma\rangle \to_{Exp} n \qquad n \in \mathbb{Z}}{\langle x := e, \sigma\rangle \to_{Stmt} \sigma[x \mapsto n]} \qquad \frac{}{\langle \textbf{nil}, \sigma\rangle \to_{Stmt} \sigma}$$

$$\frac{\langle c_1, \sigma\rangle \to_{Stmt} \sigma' \qquad \langle c_2, \sigma'\rangle \to_{Stmt} \sigma''}{\langle c_1; c_2, \sigma\rangle \to_{Stmt} \sigma''}$$

- Beispiel: $\sigma \stackrel{def}{=} \langle\rangle$

```
x := 6;
y := 4+ x;
```

Regeln der operationalen Semantik II

- Regeln:

$$\frac{\langle b, \sigma\rangle \to_{Lexp} \mathit{true} \qquad \langle c_1, \sigma\rangle \to_{Stmt} \sigma'}{\langle \textbf{if } (b) \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \to_{Stmt} \sigma'} \qquad \frac{\langle b, \sigma\rangle \to_{Lexp} \mathit{false} \qquad \langle c_2, \sigma\rangle \to_{Stmt} \sigma'}{\langle \textbf{if } (b) \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \to_{Stmt} \sigma'}$$

- Beispiel: $\sigma \stackrel{def}{=} \langle x \mapsto 6, y \mapsto 10\rangle$

```
if (x != 0) {
  y := y/x;
  } else {
  y := 0;
  }
```

Regeln der operationalen Semantik III

- Regeln:

$$\frac{\langle b, \sigma\rangle \to_{Lexp} \mathit{false}}{\langle \textbf{while } (b)\ c, \sigma\rangle \to_{Stmt} \sigma}$$

$$\frac{\langle b, \sigma\rangle \to_{Lexp} \mathit{true} \qquad \langle c, \sigma\rangle \to_{Stmt} \sigma' \qquad \langle \textbf{while } (b)\ c, \sigma'\rangle \to_{Stmt} \sigma''}{\langle \textbf{while } (b)\ c, \sigma\rangle \to_{Stmt} \sigma''}$$

- Beispiel: $\sigma \stackrel{def}{=} \langle x \mapsto 3\rangle$

```
f := 1;
while (x > 0) {
  f := f* x;
  x := x-1;
}
```

Auswertung von Ausdrücken mit Seiteneffekten

- Imperative Sprachen haben Ausdrücke mit **Seiteneffekt**
- Auswertung verändert Zustand.
- Dazu **Änderung** der Sprache:

$$
\begin{aligned}
e ::= {} & \mathbb{Z} \mid \mathit{I} \mid \mathit{true} \mid \mathit{false} \\
& \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \\
& \mid e_1 == e_2 \mid e_1 < e_2 \\
& \mid \,!\, e \mid e_1 \,\&\&\, e_2 \mid e_1 \,||\, e_2 \\
& \mid \mathbf{Idt} := \mathbf{Exp}
\end{aligned}
$$

$$
c ::= e \mid \mathbf{if}\ (e)\ \mathbf{then}\ c_1\ \ \mathbf{else}\ \ c_2 \mid \mathbf{while}\ (e)\ c \mid c_1; c_2 \mid \mathbf{nil}
$$

Neue Regeln

Auswertung: $\langle e, \sigma \rangle \to_{Exp} \langle v, \sigma' \rangle$

$$\frac{i \in \mathbb{Z}}{\langle i, \sigma \rangle \to_{Exp} \langle i, \sigma \rangle} \qquad \frac{b \in \mathbb{B}}{\langle b, \sigma \rangle \to_{Exp} \langle b, \sigma \rangle} \qquad \frac{x \in \textbf{Idt}, x \in \mathrm{dom}(\sigma), \sigma(x) = v}{\langle x, \sigma \rangle \to_{Exp} \langle v, \sigma \rangle}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Exp} \langle n_1, \sigma_1 \rangle \qquad \langle e_2, \sigma_1 \rangle \to_{Exp} \langle n_2, \sigma_2 \rangle \qquad n_i \in \mathbb{Z}}{\langle e_1 + e_2, \sigma \rangle \to_{Exp} \langle n_1 + n_2, \sigma_2 \rangle}$$

$$\frac{\langle e, \sigma \rangle \to_{Exp} \langle n, \sigma' \rangle \qquad n \in \mathbb{Z}}{\langle x := e, \sigma \rangle \to_{Exp} \langle n, \sigma'[x \mapsto n] \rangle}$$

$$\frac{\langle e, \sigma \rangle \to_{Exp} \langle v, \sigma' \rangle}{\langle e, \sigma \rangle \to_{Stmt} \sigma'}$$

Auswertung

- Was ist der **Wert** von $x := e$?
- In welcher Reihenfolge werden die Argumente von $+$ ausgewertet?
  - Gilt auch für $-$, $*$, $/$, aber **nicht** für &&, ||

Übung 2.6: Gegeben $\sigma = \langle x \mapsto 0, y \mapsto 0 \rangle$.
Berechne die Auswertung von `(x := y+1)+(y := x+5)`

Alternative Regeln

- Auswertung von rechts nach links:

$$\frac{\langle e_1, \sigma_1\rangle \to_{Exp} \langle n_1, \sigma_2\rangle \qquad \langle e_2, \sigma\rangle \to_{Exp} \langle n_2, \sigma_1\rangle \qquad n_i \in \mathbb{Z}}{\langle e_1 + e_2, \sigma\rangle \to_{Exp} \langle n_1 + n_2, \sigma_2\rangle}$$

- Unbestimmt:

$$\frac{\langle e_1, \sigma\rangle \to_{Exp} \langle n_1, \sigma_1\rangle \qquad \langle e_2, \sigma_1\rangle \to_{Exp} \langle n_2, \sigma_2\rangle \qquad n_i \in \mathbb{Z}}{\langle e_1 + e_2, \sigma\rangle \to_{Exp} \langle n_1 + n_2, \sigma_2\rangle}$$

$$\frac{\langle e_1, \sigma_1\rangle \to_{Exp} \langle n_1, \sigma_2\rangle \qquad \langle e_2, \sigma\rangle \to_{Exp} \langle n_2, \sigma_1\rangle \qquad n_i \in \mathbb{Z}}{\langle e_1 + e_2, \sigma\rangle \to_{Exp} \langle n_1 + n_2, \sigma_2\rangle}$$

Übung 2.7: In welcher Reihenfolge werden die Argumente mehrstelliger Operationen in C, Rust, Java, Python, Haskell ausgewertet?

Zusammenfassung

- Typsystem können implizit oder explizit, statisch oder dynamisch sein.
- Die Typableitung $\Gamma \vdash e : t$ wird durch **Regeln** definiert.
- Die Auswertung von **Anweisungen** erzeugt einen Zustandsübergang: $\langle c, \sigma \rangle \rightarrow_{Stmt} \sigma'$
- Die Auswertung von Ausdrücken mit Seiteneffekten erzeugt einen Wert **und** einen neuen Zustand: $\langle e, \sigma \rangle \rightarrow_{Exp} \langle v, \sigma' \rangle$

Programmiersprachen
Vorlesung 4 vom 30.10.23
Variablen und Speichermodelle

Christoph Lüth

Universität Bremen

Wintersemester 2023/24

Wo sind wir?

- Einführung
- Einfache Ausdrücke und Werte
- Einfache Typen, Anweisungen und Seiteneffekte
- Variablen und Speichermodelle
- Aggregierende Typen
- Ausnahmen und Fehlerbehandlung
- Prozeduren und Funktionen
- Fortgeschrittene Typsysteme
- Datenabstraktion
- Programmierparadigmen
- Skriptsprachen
- Ab Woche 11 (2024): Studentische Vorträge.

Variablen, Namen, Deklarationen

- Was genau macht eine **Variablendeklaration?**

```
int x;

...
```

Variablen, Namen, Deklarationen

- Was genau macht eine **Variablendeklaration?**

```
int x;

...
```

1. Sie führt den **Namen** (Bezeichner) `x` ein.
2. Sie reserviert Platz im **Speicher**.

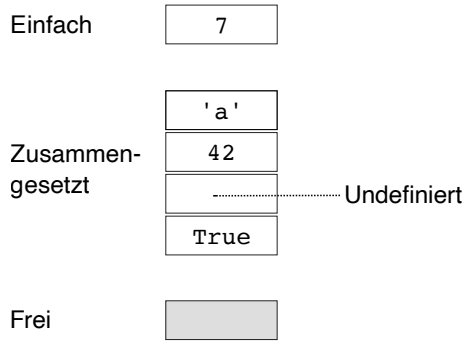
- Vergleiche:

```
let x= 37 in...
```

- Hier wird **nur** der **Name** eingeführt.

Ein Einfaches Speichermodell

- Der Speicher hat eine Menge von **Speicherzellen** mit einer eindeutigen **Adresse**
- Speicherzellen haben einen **Status**:
  - **Belegt** (allocated) oder **frei** (unallocated)
  - Belegte Speicherzellen haben einen **Inhalt**, entweder ein **Wert** oder **undefiniert**.
- Zusammengesetzte Werte belegen mehrere Speicherzellen (“composite variables”)
- Abstraktion über Wortbreite etc.



Einfache Variablen

- Unterschied: Name `n` als **Adresse** der Variable vs. `n` als **Wert** der Speicherzelle mit dieser Adresse
- Unterschied nach Kontext:
  - Links der Zuweisung ("L-Wert") vs. rechts der Zuweisung ("R-Wert")

```
x= x+1
```

- In funktionalen Sprachen sind Variablen **unveränderlich**
  - Es gibt nur lokale **Namen**

```
let x = 37 in x+ 1
```

  - Keine Zuweisung, keine L-Werte
  - **Verwirrend**: Rust nutzt `let x = 37u32;` zur Deklaration von **Variablen**.

Lebenszyklus einer Variablen

- Generell haben Variablen einen **Lebenszyklus**: Allokation, Nutzung, Deallokation
  - Bei der **Allokation** wird Platz im Speicher reserviert
  - Bei der **Deallokation** wird der Speicher wieder freigegeben
- Klassifikation von Variablen nach der Lebensdauer:
  - **Global** oder statisch — ganze Laufzeit des Programmes
  - **Lokal** oder automatisch — innerhalb eines **Blocks**
  - **Heap** — beliebig, aber höchstens bis Programmende
  - **Persistent** — länger als das Programm (e.g. Dateien, Datenbanken)

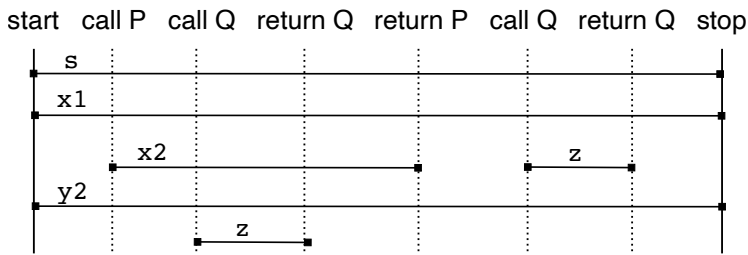
Block

- Ein **Block** ist ein Programmabschnitt zusammen mit **lokalen Deklarationen**
- Blöcke dienen zur
  - **Gruppierung** von Anweisung
  - **Verkapselung** (durch lokale Deklarationen)
- Fast alle Programmiersprachen haben **verschachtelte Blöcke**
- Blöcke bestimmen die Lebensdauer und Sichtbarkeit der lokalen Variablen
- NB: Lebensdauer $\neq$ Sichtbarkeit

Lebensdauer — Beispiel

```
char s[] = "Foo";
void main()
{ int x1;
  ... P(); ... Q(); ...
}
```

```
void P()
{ int *x2; static float y2;
  ... Q(); ...
}
void Q()
{ float z;
  ...
}
```



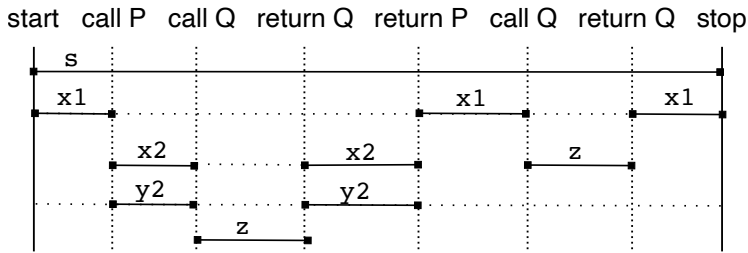
Bindung und Scope

- Eine **Bindung** assoziert lexikalische Bezeichner mit einem semantischen Wert
  - Abstrakt: symbolische Bezeichner der ausführenden abstrakten Maschine
  - Konkret: Speicheradresse
- Eine **Umgebung** ist eine Menge von Bindungen
- Der **Scope** eines Bezeichners ist sein Gültigkeitsbereich oder Sichtbarkeitsbereich
- Unterscheidung:
  - Statischer (oder lexikalischer) Scope — Gültigkeitsbereich wird zur Übersetzungszeit festgelegt
  - Dynamischer Scope — Gültigkeitsbereich wird während der Laufzeit festgelegt

Sichtbarkeit ist nicht Lebensdauer — Beispiel

```
char s[] = "Foo";
void main()
{ int x1;
  ... P(); ... Q(); ...
}
```

```
void P()
{ int *x2; static float y2;
  ... Q(); ...
}
void Q()
{ float z;
  ...
}
```



Static vs. Dynamic Scope

Ein Beispielprogramm (fiktive Syntax):

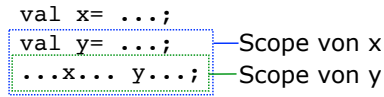
```
s= 2

def foo(x):
  print(s*x)

def baz(y):
  foo(y)

def bah(z):
  s= 4 # Implicitly declared as local,
  foo(z)

bah(5)
baz(5)
```

- **Statisch**
  - C, Java, Python, Haskell:
  - Ausgabe `10, 10`
  - Python hat “late binding”
  - Alternative Ausgabe: 20, 20
    (dann ist `s` in `bah` global)
- **Dynamisch**
  - Perl, shell:
  - Ausgabe `20, 10`

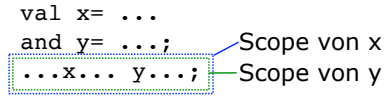
Deklarationen

- Deklarationen führen eine Bindung ein.
- **Komposition** von Deklarationen:
  - Sequential
  - Rekursiv
  - Kollateral
- Beispiel Standard ML: kann alles

Übung 2.1: Wie werden Deklarationen in C, Java, Python, Haskell gehandhabt?

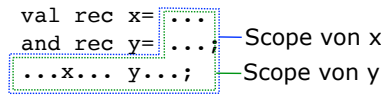


Sequentielle Deklarationen



Kollaterale Deklarationen

Rekursive Deklaration



Rekursive, kollaterale Deklarationen

Speicherverwaltung

- Der Speicher wird meist unterteilt in einen **Stack** und einen **Heap**
- Der Stack verwaltet lokale Variablen:
  - Für jeden Aufruf einer Funktion ein **Stack Frame**
  - Wird am Ende der Funktion wieder entfernt
- Der Heap verwaltet Heap-Variablen
  - Allokation manuell (C, `malloc`) oder durch Konstruktor (`new`)
  - Deallokation manuell (C, `free`) oder durch **Garbage collector**
- Garbage-Collection Algorithmen:
  - reference counting, mark&sweep, copy
- Problemquellen:
  - Dangling pointers, memory leaks

Sonderfall: Ownership in Rust

- Rust hat eine neuartiges Konzept — **Ownership**:
  - Variable hat **einen** Besitzer, der die Lebenszeit bestimmt:

```
let s1 = String::from("hello");
let s2 = s1;    // Does not work!
```

  - Variablen können **verliehen** werden:

```
let s1 = String::from("hello");
let len = calculate_length(&s1);
```

- Verliehene Variablen dürfen nicht **verändert** werden.
- Es gibt immer **einen** Besitzer, oder **beliebig viele** unveränderliche Referenzen.
- Referenzen sind immer **gültig**:

```
fn dangle() -> &String {
    let s = String::from("hello");
    &s
}
```

Namen und Substitution

Lokale Namen

$$
\begin{aligned}
e ::= \; & \mathbb{Z} \mid i \mid \mathit{true} \mid \mathit{false} \\
& \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \\
& \mid e_1 == e_2 \mid e_1 < e_2 \\
& \mid \; ! \, e \mid e_1 \;\&\&\; e_2 \mid e_1 \;||\; e_2 \\
& \mid i := e \\
& \mid \textbf{let} \;\; x = e_1 \;\; \textbf{in} \;\; e_2
\end{aligned}
$$

- **let** $x = e$ **in** f führt den lokalen Namen x in f ein, und setzt ihn auf den Wert e.
- x ist **nicht veränderlich** (keine **Variable**)!

Semantik lokaler Namen

- **let** $x = e_1$ **in** e_2 ist wie e_2 in dem x durch e_1 ersetzt ist.

- Problem: Schachtelung lokaler Namen:

```
let x= 5 in
  let y= 2*x in
    (let x= 3 in x+y)+ x
```

Substitution

- Definition $e_1[e_2/x]$:

$$
\begin{aligned}
v[e/x] &= v \\
y[e/x] &= \begin{cases} e & x = y \\ y & \text{otherwise} \end{cases} \\
(e_1 + e_2)[e/x] &= (e_1[e/x]) + (e_2[e/x]) \\
(x := e_1)[e_2/y] &= (x := (e_1[e_2/y])) \\
(\textbf{let}\ \ x = e_1\ \ \textbf{in}\ \ e_2)[e_3/y] &= \textbf{???}
\end{aligned}
$$

Freie Variablen

$$
\begin{aligned}
\mathrm{FV}(v) &= \emptyset \\
\mathrm{FV}(x) &= \{x\} \\
\mathrm{FV}(e_1 + e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
&\dots \\
\mathrm{FV}(i := e) &= \mathrm{FV}(e) \\
\mathrm{FV}(\textbf{let } x = e_1 \textbf{ in } e_2) &= \mathrm{FV}(e_1) \cup (\mathrm{FV}(e_2) \setminus \{x\})
\end{aligned}
$$

Übung 3.2: Berechne die freien Variablen in
$(\textbf{let } z = x + 3 * y \textbf{ in } z + x) + (\textbf{let } y = 5 * x \textbf{ in } y + x)$.

Substitution

- Damit Definition der Substitution:

$$(\textbf{let } x = e_1 \textbf{ in } e_2)[e_3/y] = \begin{cases} \textbf{let } x = e_1[e_3/y] \textbf{ in } e_2 & x = y \\ \textbf{let } x = e_1[e_3/y] \textbf{ in } e_2[e_3/y] & x \neq y, x \notin \mathrm{FV}(e_3) \\ \textbf{let } z = e_1[e_3/y] \textbf{ in } (e_2[z/x])[e_3/y] & x \neq y, x \in \mathrm{FV}(e_3), \\ & z \notin \mathrm{FV}(e_3) \cup \mathrm{FV}(e_2) \end{cases}$$

- z ist eine **frische** Variable.

Frage: Berechne $(\textbf{let } x = x + y \textbf{ in } x + 5 == 7 - y)[\textbf{let } x = x + 3 \textbf{ in } 2 * x/y]$.

Übung 3.3: Sei
$e \stackrel{def}{=} \textbf{let } x = x + y \textbf{ in } 4 * ((\textbf{let } y = x + y \textbf{ in } y + 5) + (\textbf{let } x = 3 * x * y \textbf{ in } y + x))$.
Berechne $e[3 * x + y/y]$.

Auswertung

- Zwei Möglichkeiten:

$$\frac{\langle e_2[e_1/x], \sigma\rangle \to_{Exp} \langle v, \sigma'\rangle}{\langle \textbf{let } x = e_1 \textbf{ in } e_2, \sigma\rangle \to_{Exp} \langle v, \sigma'\rangle} \quad (1)$$

$$\frac{\langle e_1, \sigma\rangle \to_{Exp} \langle v_1, \sigma'\rangle \qquad \langle e_2[v_1/x], \sigma'\rangle \to_{Exp} \langle v_2, \sigma''\rangle}{\langle \textbf{let } x = e_1 \textbf{ in } e_2, \sigma\rangle \to_{Exp} \langle v_2, \sigma''\rangle} \quad (2)$$

- (1) ist nicht-strikt ("call-by-name")
- (2) ist strikt ("call-by-value")

Variablen und Speichermodelle

Variablen und Speichermodelle

- Speichermodell $\Sigma = \mathbf{Loc} \rightharpoonup \mathbf{V}$ mit $\mathbf{Loc} = \mathbb{N}$
- Trennung von Bezeichner **Idt** und Adressen **Loc**
- Speicherzellen können frei oder belegt (uninitialisiert oder mit Wert) sein
- Daher $\mathbf{V} = \mathbb{Z}_\perp = \mathbb{Z} \uplus \{\perp\}$

Erweiterung der Sprache

- Neues Kommando:

$$c ::= e \mid \textbf{if } (e) \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{while } (e)\ c \mid c_1; c_2 \mid \textbf{nil} \mid \textbf{new } x : t \textbf{ in } c$$

- Sequential, nicht kollateral
- Typisierung:

$$\frac{\Gamma[x \mapsto t] \vdash c : \textbf{unit}}{\Gamma \vdash \textbf{new } x : t \textbf{ in } c : \textbf{unit}}$$

- Erweiterung der Semantik:

$$\Gamma = \textbf{Idt} \rightharpoonup \textbf{Loc} \qquad \Gamma \vdash \langle e, \sigma \rangle \rightarrow_{Exp} \langle v, \sigma' \rangle$$
$$\sigma = \textbf{Loc} \rightharpoonup V \qquad \Gamma \vdash \langle c, \sigma \rangle \rightarrow_{Stmt} \sigma'$$

Γ ist eine **Umgebung** (statisch, nur zur Übersetzungszeit)

Semantik: alte Regeln

$$\frac{\Gamma \vdash \langle c_1, \sigma\rangle \to_{Stmt} \sigma' \qquad \Gamma \vdash \langle c_2, \sigma'\rangle \to_{Stmt} \sigma''}{\Gamma \vdash \langle c_1; c_2, \sigma\rangle \to_{Stmt} \sigma''}$$

$$\frac{\Gamma \vdash \langle b, \sigma\rangle \to_{Exp} \langle true, \sigma'\rangle \qquad \Gamma \vdash \langle c_1, \sigma'\rangle \to_{Stmt} \sigma''}{\Gamma \vdash \langle \textbf{if } (b) \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \to_{Stmt} \sigma''}$$

$$\frac{\Gamma \vdash \langle b, \sigma\rangle \to_{Exp} \langle false, \sigma'\rangle \qquad \Gamma \vdash \langle c_2, \sigma'\rangle \to_{Stmt} \sigma''}{\Gamma \vdash \langle \textbf{if } (b) \textbf{ then } c_1 \textbf{ else } c_2, \sigma''\rangle \to_{Stmt} s}$$

$$\frac{\Gamma \vdash \langle b, \sigma\rangle \to_{Exp} \langle false, \sigma'\rangle}{\Gamma \vdash \langle \textbf{while } (b)\ c, \sigma'\rangle \to_{Stmt} \sigma'}$$

$$\frac{\Gamma \vdash \langle b, \sigma\rangle \to_{Exp} \langle true, \sigma'\rangle \qquad \Gamma \vdash \langle c, \sigma'\rangle \to_{Stmt} \sigma'' \qquad \Gamma \vdash \langle \textbf{while } (b)\ c, \sigma''\rangle \to_{Stmt} \sigma'''}{\Gamma \vdash \langle \textbf{while } (b)\ c, \sigma\rangle \to_{Stmt} \sigma'''}$$

Semantik: neue Regeln

$$\frac{x \in \mathbf{Idt}, x \in \mathrm{dom}(\Gamma), \sigma(\Gamma(x)) = v}{\Gamma \vdash \langle x, \sigma\rangle \to_{Exp} \langle v, \sigma\rangle}$$

$$\frac{\Gamma \vdash \langle e, \sigma\rangle \to_{Exp} \langle n, \sigma'\rangle \qquad n \in \mathbb{Z}}{\Gamma \vdash \langle x := e, \sigma\rangle \to_{Stmt} \sigma'[\Gamma(x) \mapsto n]}$$

$$\frac{\Gamma[x \mapsto l] \vdash \langle c, \sigma[l \mapsto \bot]\rangle \to_{Stmt} \sigma' \qquad l \notin \mathrm{dom}(\sigma)}{\Gamma \vdash \langle \mathbf{new}\ \ x : t\ \ \mathbf{in}\ \ c, \sigma\rangle \to_{Stmt} \sigma' \setminus l}$$

Beispiel

Auswertung mit $\Gamma = \langle z \mapsto 0 \rangle, \sigma = \langle 0 \mapsto \bot \rangle$

```
new x: int in
  x := 0;
  new y: int in
    y := 5;
    x := y+ 3;
    new x: int= 0 in
      z := x+ y;
    y:= x;
```

Beispiel

Auswertung mit $\Gamma = \langle z \mapsto 0 \rangle, \sigma = \langle 0 \mapsto \bot \rangle$

```
new x: int in
  x := 0;
  new y: int in
    y := 5;
    x := y+ 3;
    new x: int= 0 in
      z := x+ y;
    y:= x;
```

```
new x: int = 0 in
  new y: int = 5 in
    x := y+ 3;
    new x: int = 0; in
      z := x+ y;
    y:= x;
```

Zusammenfassung

- Namen vs. (veränderliche) Variablen, L-Werte vs. R-Werte
- Variablen haben einen **Lebenszyklus**
  - Global/statisch, lokal/automatisch, Heap
- Lebenszeit $\neq$ Sichtbarkeit
- Scope: Statisch vs. Dynamisch
- Deklarationen: sequentiell, kollateral, rekursiv
- Speicherverwaltung: Stack und Heap, Garbage Collection vs. manuell

Programmiersprachen
Vorlesung 5 vom 05.11.23
Aggregierende Typen

Christoph Lüth

Universität Bremen

Wintersemester 2023/24

Wo sind wir?

- Einführung
- Einfache Ausdrücke und Werte
- Einfache Typen, Anweisungen und Seiteneffekte
- Variablen und Speichermodelle
- Aggregierende Typen
- Ausnahmen und Fehlerbehandlung
- Prozeduren und Funktionen
- Fortgeschrittene Typsysteme
- Datenabstraktion
- Programmierparadigmen
- Skriptsprachen
- Ab Woche 11 (2024): Studentische Vorträge.

Aggregierende Typen

- Auch genannt: Compound Types, Aggregate Types, Composite Types
- Kartesische Produkte (Tupel und Strukturen)
- Endliche Abbildungen (Arrays, Dictionaries)
- Vereinigungstypen (algebraische Typen, discriminated records, Objekte)
- Rekursive Typen

Kartesische Produkte

- Mathematisch:
  - Paare: $A \times B = \{(a, b) \mid a \in A, b \in B\}$
  - n-Tupel: $\prod_i A_i = \{(a_1, \ldots, a_n) \mid a_i \in A_i\}$
- Fast alle Programmiersprachen haben kartesische Produkte
- Direkt als Tupel (Haskell, Scala),
- `struct` sind kartesische Produkte mit **benannten** Projektionen

```
struct pair {
    int fst;
    int snd;
}
```

- Tupel sind "Listen fester Länge"

Tupel und Funktionen

- Ungenaue Notation bei Funktionsaufruf:

Haskell:

```
f1 :: Int → Int → Int
f2 :: (Int, Int) → Int
```

C:

```
int f(int x, int y)
```

`f1` hat zwei Argumente, `f2` eines (ein Tupel)

`f` hat **zwei** Argumente, nicht eins.

- Es gilt $A \times B \to C \cong A \to B \to C$ ("Currying")
- Wird nicht von allen Programmiersprachen unterstützt
- $A \to B \to C$ ist eine Funktion höherer Ordnung

Endliche Abbildungen

- Eine endliche Abbildung ist eine Funktion $A \to B$ mit A endlich.
- Wenn $A = 0, \ldots, n$ für $n \in \mathbb{N}$, dann ist $A \to B$ ein **Feld** (Array)
  - Felder können effizient als zusammenhängende Speicherbereiche implementiert werden
  - Indizierung ($a[i]$) sehr billig $O(1)$
  - Manchmal Indizierung auch ab 1 oder von $n \ldots m$
- Wenn A eine Menge von **Bezeichnern**, dann ist $A \to B$ ein **Dictionary** (Python)

```
a = { "foo" : 12, "baz": "hello", "bah" : { "a" : 1, "b" : False }}
a.get("foo")
a["foo"]
a["bah"]["a"]
```

Disjunkte Vereinigung

- Mathematisch:
  - Binär: $A + B = (\{1\} \times A \cup \{2\} \times B)$
  - Verallgemeinert: $\sum_i A_i = \cup_i \{i\} \times A_i$
- Disjunkte Vereinigungen werden sehr **heterogen** gehandhabt:
  - C kennt `union`
  - Meist kann man nicht einfach zwei Typen vereinigen.
  - In Java über Subtyping.

Disjunkte Vereinigung in C

- Der Union-Typ vereinigt alle Komponenten an der gleichen Adresse:

```
union { int x; double y; } u;
```

- Extrem fehleranfällig
- Keine **disjunkte** Vereinigung
- Zur Unterscheidung (discriminated records in Ada):

```
enum u_tag { u_a, u_b };
struct { enum u_tag tag; union { int a; double b; } cont; } u;

switch (u.tag) {
  case u_b: printf("Double: %f\n", u.cont.b); break
}
```

Disjunkte Vereinigung in Java

- Unterklassen einer Oberklasse sind eine Vereinigung (aber nicht disjunkt)
- Sehr indirekt durch Subtyping:

```
class A {
  C f() { ... };
}
class B {
  C g() { ... };
}
```

```
abstract class AB {
   private class InlA extends AB {
     private A a;
     C fg() { a.f(); }
   }
   private class InrB extends AB {
     private B b;
     C fg() { b.g(); }
   }
   C fg();
}
```

Disjunkte Vereinigung in Haskell

- Algebraische Datentypen (mit Fallunterscheidung):

```
data Either a b = Left a | Right b

foldEither :: (a → c) → (b → c) → Either a b → c
foldEither l r (Left a) = l a
foldEither l r (Right b) = r b
```

- Unterschiedliche Konstruktoren für einen Typen sind eine disjunkte Vereinigung (wie `Left, Right` oben).

Sonderfälle

- Leere Vereinigung — der **leere** Typ: $T = \emptyset$
- `void` in C und Java, `Nothing` in Scala; in Haskell nicht vordefiniert und nutzlos
- Leeres Tupel – der **einelementige** Typ: $T = ()$
- `()` in Haskell und Rust, `Null` in Scala

Rekursive Typen

- Rekursive Typen sind durch **Gleichungen** definiert:

$$T = F(T)$$

- Beispiele:
  - Listen: $L(A) = 1 + A \times L(A)$
  - Binäre Bäume: $T(A) = 1 + T(A) \times A \times T(A)$
  - Variadische Bäume: $R(A) = A \times L(R(A))$

Frage: Wieso definieren diese Gleichungen den Typ?

Rekursive Typen in C

- C erlaubt **keine** direkt rekursiven Typen
  - Umweg über Zeiger und “incomplete types”:

```
typedef struct list_el {
    void *head;
    struct list_el *tail;
    } *list;

typedef struct tree_el {
  struct tree_el *le; void *node; struct tree_el *ri;
  } *tree;
```

- Der `NULL`-Pointer übernimmt die Rolle des Unit-Typen.
- Polymorphie durch `void *`.

Rekursive Typen in Java

Rekursive Typen durch rekursive Klassen:

```
class List {
  public Object head;
  public List tail;
  }
```

```
class Tree {
  public Tree left;
  public Object node;
  public Tree right;
}
```

- In Java ist alles[1] **implizit** eine Referenz (Pointer), daher eigentlich ähnlich C einschließlich `null` für den Unit-Typ.
- Polymorphie durch `Object`, mehr dazu später.

[1] Bis auf primitive Typen.

Rekursive Typen in Haskell

- Hier können wir die Domänengleichungen direkt abschreiben:

```
data List a = Null | Cons a (List a)

data Tree a = Node { le :: Tree a, node :: a, ri :: Tree a}

data NTree a = Node a (List (Ntree a))
```

- Listen sind mit syntaktischem Zucker vordefiniert.

Rekursive Typen in Python

- Listen sind vordefiniert (Typ `list`)
- Definition von rekursiven Typen als Klasse.
- Binäre Bäume:

```
class Tree:
  def __init__(self, node):
    self.left = None
    self.right = None
    self.node = node
```

```
class NTree:
  def __init__(self, node):
    self.node= node
    self.children= []
```

- `__init__` ist der Konstruktor, der Typ selbst wird dynamisch definiert.

Variablen von aggregierendem Typ

- Variablen von aggregierendem Typ belegen einen Block von mehreren Speicherzellen
- Bei Feldern zusammenhängend
- Bei Tupeln nicht notwendigerweise
- Speicherlayout und alignment
  - Nur für systemnahe Programmiersprachen (e.g. C)
- Totales und selektives Update

```
struct date { int y, m, d; } d1, d2;
d1.m= 11; // selektiv
d2= d1;   // total
```

- C erlaubt **Speicherarithmetik**: `a[i] == *(a+i)`

Felder

- Statische Felder haben **feste**, **unveränderliche** Länge (C, Rust, Java)
- Bei **dynamischen** Felder kann die Länge verändert werden (Haskell, `Vec` in Rust, `Vector` in Java)
- Bei **flexiblen** Feldern ist die Länge variabel (aber fest)

```
double a1[] = {2.0, 3.0, 5.0};

static void prtVec(double [] v) {
  for (int i= 0; i< v.length; i++)
    System.out.println(v[i]+" ")
  }
```

Copy Semantics vs Reference Semantics

- Was passiert bei einer Zuweisung `x= e`, wenn `x` einen zusammengesetzten Typ hat?
- **Copy semantics**: `x` enthält danach eine **Kopie** von `e`, alle Komponenten von `e` werden in die Komponenten von `x` kopiert
- **Reference semantics**: `x` ist eine **Referenz** auf `e`

Copy Semantics vs Reference Semantics

- Was passiert bei einer Zuweisung `x= e`, wenn `x` einen zusammengesetzten Typ hat?
- **Copy semantics**: `x` enthält danach eine **Kopie** von `e`, alle Komponenten von `e` werden in die Komponenten von `x` kopiert
- **Reference semantics**: `x` ist eine **Referenz** auf `e`
- C kopiert (Referenzen sind in der Sprache **explizit**)
- Java und Python referenzieren (alles ist eine Referenz, Kopie explizit über `clone`, `copy`, `deepcopy`)
- Haskell referenziert, aber Werte sind **unveränderlich**

Verwandt damit: Gleichheit

- Identität vs. strukturelle Gleichheit
- Identität: Referenz auf das gleiche Objekt im **Speicher**
- Strukturelle Gleichheit: gleicher "Inhalt"
  - Java: == für Identität (der Referenzen), `equals` für strukturelle Gleichheit
  - Python: `is` für Identität (der Referenzen), == für strukturelle Gleichheit
  - C: == auf Referenzen für Identität, == auf zusammengesetzten Typen für strukturelle Gleichheit
  - Haskell: **nur** strukturelle Gleichheit (==, Typklasse `Eq`)

Referenzen

- In C sind Referenzen **explizit**: `struct c *x` deklariert `x` als Referenz auf `struct c`.
- Zwei Operatoren:
  - `&e` erzeugt Referenz auf `e` (muss ein l-value sein)
  - `*e` dereferenziert `e` (gibt ein l-value zurück)
- Erlaubt **dynamische Datenstrukturen**
- Der `NULL`-Pointer: Tony Hoares "One-Billion Dollar Mistake"[2]
- Alle anderen Sprachen: **implizite Referenzen**

[2] `https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/`

Semantik von Aggregierten Typen

Was brauchen wir?

- Erweiterung der **Sprache**: mehr Typen, mehr Ausdrücke
- Erweiterung des **Speichermodells**
- Semantik für die neuen **Ausdrücke**

Erweiterung der Sprache

$$
\begin{aligned}
t &::= \textbf{int} \mid \textbf{bool} \mid \textbf{array}\ n\ \textbf{of}\ t \mid \textbf{struct}(i : t)^* \\
l &::= i \mid l.i \mid l[e] \\
e &::= \mathbb{Z} \mid \mathit{true} \mid \mathit{false} \mid l \\
&\quad \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \\
&\quad \mid e_1 == e_2 \mid e_1 < e_2 \\
&\quad \mid !\, e \mid e_1 \,\&\&\, e_2 \mid e_1 \,||\, e_2 \\
&\quad \mid l := e \\
&\quad \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2
\end{aligned}
$$

Größe eines Typs

$$\begin{aligned}
\text{size}(\textbf{int}) &= 1 \\
\text{size}(\textbf{bool}) &= 1 \\
\text{size}(\textbf{array}\ \ n\ \ \textbf{of}\ \ t) &= n \cdot \text{size}(t) \\
\text{size}(\textbf{struct}(f_i : t_i)_{i=1,\ldots,n}) &= \sum_{i=1}^{n} \text{size}(t_i)
\end{aligned}$$

Offset eines Feldes g in einer Struktur:

$$\text{offset}_{\textbf{struct}(f_i:t_i)_{i=1,\ldots,n}}(g) \stackrel{def}{=} \sum_{k=1}^{l-1} \text{size}(t_k) \quad f_l = g$$

Erweiterung des Speichermodells

- Kann eigentlich bleiben:

$$\Sigma = \mathbf{Loc} \rightharpoonup \mathbf{V}, \mathbf{Loc} = \mathbb{N}, \mathbf{V} = (\mathbb{B} + \mathbb{Z} + \mathbb{N})_\perp$$

- Braucht zwei Hilfsfunktionen:

$$\text{mem\_cp}(\sigma, n, k, m) = \begin{cases} \sigma & k \leq 0 \\ \text{mem\_cp}(\sigma[n \mapsto \sigma(m)], n+1, k-1, m+1) & k > 0 \end{cases}$$

$$\text{mem\_alloc}(\sigma, n, k) = \begin{cases} \sigma & k \leq 0 \\ \text{mem\_alloc}(\sigma[n \mapsto \perp], n+1, k-1) & k > 0 \end{cases}$$

Semantik: Auswertung von L-Werten

$$\Gamma \vdash \langle l, \sigma \rangle \to_{Lexp} \langle n, \sigma' \rangle$$

$$\frac{}{\Gamma \vdash \langle l, \sigma \rangle \to_{Lexp} \langle \Gamma(l), \sigma \rangle}$$

$$\frac{\Gamma \vdash \langle l, \sigma \rangle \to_{Lexp} \langle n, \sigma' \rangle \qquad \Gamma \vdash l : \textbf{array}\ \ n\ \ \textbf{of}\ \ t \qquad \Gamma \vdash \langle e, \sigma' \rangle \to_{Exp} \langle v, \sigma'' \rangle, v \in \mathbb{Z}}{\Gamma \vdash \langle l[e], \sigma \rangle \to_{Lexp} \langle l + \mathrm{size}(t) \cdot v, \sigma'' \rangle}$$

$$\frac{\Gamma \vdash \langle l, \sigma \rangle \to_{Lexp} \langle n, \sigma' \rangle \qquad \Gamma \vdash l : \textbf{struct}(f_1 : t_1 \ldots f_n : t_n) \qquad \exists k.\, f_k = i}{\Gamma \vdash \langle l.i, \sigma \rangle \to_{Lexp} \langle n + \mathrm{offset}_{\textbf{struct}(f_1:t_1 \ldots f_n:t_n)}(i), \sigma' \rangle}$$

Semantik: Ausdrücke

- Zu was wertet ein Ausdruck von aggregiertem Typ aus?
  - Ein **strukturierter Wert**? (Python, Haskell)
  - Eine **Referenz**? (Java)
- Ausdruck von aggregiertem Typ benötigen Sonderfälle für
  - L-Werte als Ausdrücke (*i.e.* auf der **rechten** Seite von Zuweisungen)
  - Zuweisungsausdrücke
- Wie allozieren wir **Variablen** aggregierten Typs?
  - Speicherblock von der Größe des Typs auf dem Stack?
  - Referenz in den Heap?

Semantics: Die Regeln

$$\frac{\Gamma \vdash l : \textbf{int} \text{ or } \Gamma \vdash l : \textbf{bool} \qquad \Gamma \vdash \langle l, \sigma \rangle \to_{Lexp} \langle n, \sigma' \rangle}{\Gamma \vdash \langle l, \sigma \rangle \to_{Exp} \langle \sigma'(n), \sigma' \rangle}$$

$$\frac{\Gamma \vdash l : \textbf{array } n \textbf{ of } t \text{ or } \Gamma \vdash l : \textbf{struct} \qquad \Gamma \vdash \langle l, \sigma \rangle \to_{Lexp} \langle n, \sigma' \rangle}{\Gamma \vdash \langle l, \sigma \rangle \to_{Exp} \langle n, \sigma' \rangle}$$

$$\frac{\Gamma \vdash e : \textbf{int} \text{ or } \Gamma \vdash e : \textbf{bool} \qquad \Gamma \vdash \langle l, \sigma \rangle \to_{Lexp} \langle n, \sigma' \rangle \qquad \Gamma \vdash \langle e, \sigma \rangle \to_{Exp} \langle v, \sigma' \rangle}{\Gamma \vdash \langle l := e, \sigma \rangle \to_{Exp} \langle v, \sigma'[n \mapsto v] \rangle}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : t \text{ with } t = \textbf{array } n \textbf{ of } t_1 \text{ or } t = \textbf{struct} \\ \Gamma \vdash \langle l, \sigma \rangle \to_{Lexp} \langle n_1, \sigma' \rangle \qquad \Gamma \vdash \langle e, \sigma \rangle \to_{Exp} \langle n_2, \sigma' \rangle\end{array}}{\Gamma \vdash \langle l := e, \sigma \rangle \to_{Exp} \langle n_2, \mathsf{mem\_cp}(\sigma', n_1, \mathsf{size}(t) n_2,) \rangle}$$

Allokation von Variablen

$$\frac{\Gamma[x \mapsto l] \vdash \langle c, \text{mem\_alloc}(\sigma, l, \text{size}(t))\rangle \to_{Stmt} \sigma' \qquad \{l, \ldots, l + \text{size}(t) - 1\} \cap \text{dom}(\sigma) = \emptyset}{\Gamma \vdash \langle \textbf{new}\ \ x : t\ \ \textbf{in}\ \ c, \sigma\rangle \to_{Stmt} \sigma' \setminus \{l, \ldots, l + \text{size}(t) - 1\}}$$

- Java: Aggregierte Werte (**Objekte**) nur auf dem Heap
- C: Referenzen als **expliziter Typ** (“pointer-to”)

Zusammenfassung

- Aggregierte Typen sind zusammengesetzt:
  - Kartesische Tupel
  - Endliche Abbildungen (dictionaries)
  - Disjunkte Vereinigungen
  - Rekursive Typen
- Bei der Behandlung von aggregierten Typen gibt es verschieden Möglichkeiten:
  - Zu was werten Ausdrücke aggregierten Typs aus?
  - Was passiert bei Zuweisungen und Gleichheit?
  - Wie werden Variablen behandelt?

Programmiersprachen
Vorlesung 6 vom 13.11.23
Ausnahmen und Fehlerbehandlung

Christoph Lüth

Universität Bremen

Wintersemester 2023/24

Organisatorisches

- Nächste Woche (20.11.): Stromabschaltung

Wo sind wir?

- Einführung
- Einfache Ausdrücke und Werte
- Einfache Typen, Anweisungen und Seiteneffekte
- Variablen und Speichermodelle
- Aggregierende Typen
- Ausnahmen und Fehlerbehandlung
- Prozeduren und Funktionen
- Fortgeschrittene Typsysteme
- Datenabstraktion
- Programmierparadigmen
- Skriptsprachen
- Ab Woche 11 (2024): Studentische Vorträge.

Ausnahmen (Exceptions)

- Motivation:
  - Fehlerbehandlung in geschachtelten Funktionen
  - Ohne Sprünge nur umständliche Fallunterscheidungen

Ausnahmen (Exceptions)

- Motivation:
  - Fehlerbehandlung in geschachtelten Funktionen
  - Ohne Sprünge nur umständliche Fallunterscheidungen
  - Nicht alle Fehlermöglichkeiten **können** im Vorfeld ausgeschlossen werden
  - Klassisches Beispiel: Dateizugriff

```
if os.path.exists(filenm):
  # someone deletes file
  fd= open(filenm) # FEHLER!
```

```
int main() {
  fd= open("data");
  ... P(fd); ...;
  close(fd);
  }

void P(int fd) { ... Q(fd); ... }

void Q(int fd) { ... R(fd); ... }

void R(int fd) {
  if ((x= read(fd, 1024))== -1)
    // Fehler!
  ...
  }
```

Ausnahmen

- Konzeptionell: Ausnahmen sind **Erweiterung des Definitionsbereichs**:

$$\texttt{f} : A \to B \texttt{ throws } C \;=\; \texttt{f} : A \to B + C$$

$$B[f(x)] \texttt{ catch } e \Rightarrow E \;=\; \begin{cases} B[b] & f(x) = b \\ E & f(x) = e \end{cases}$$

- Semantisch: **Programmfehler** sind "undeklarierte Ausnahmen"
  - Aber: nicht alle können **gefangen** werden
- Unterstützung von Ausnahmen durch eine Programmiersprache benötigt:
  - Ausnahmen **deklarieren** — meist eigener Typ (SML) oder Klasse (Java, Haskell)
  - Ausnahmen **auslösen** (`throw`, `raise`)
  - Ausnahmen **fangen** (`catch`)

Ausnahmen in Java

- Ausnahmen sind Objekte der Klassen `Throwable`, `Exception`
- Geworfene Ausnahmen müssen **deklariert** werden (`throws ...`)
  - Warum? Static Scoping (siehe Beispiel)
- Schema:

```
try
  block
catch (exception_type e)
  block
catch (exception_type e)
  block
finally
  block
```

Ausnahmen in Haskell

- Ausnahmen sind Instanzen der Typklasse `Exception` (Modul `Control.Exception`)
  - Situation in Haskell98 anders
- Werfen und fangen:

```
throw :: Exception e ⇒ e → a
catch :: Exception e ⇒ IO a → (e → IO a) → IO a
```

- Exceptions können überall geworfen werden, aber nur als Aktion (`IO`) gefangen werden.
  - Warum? Bricht referentielle Transparenz
- Durch Nicht-Striktheit (verzögerte Auswertung) werden Ausnahmen später geworfen als man denkt (siehe Beispiel)

Ausnahmen in C

- C hat **keine** Exceptions
- Alternativen:
  - `goto`
  - `setjmp` and `longjmp`
  - `switch`, siehe Duff's Device

Duff's Device:

```
void send(short *to, short *from,
          int count)
{
  int n = (count+ 7)/8;
  switch (count % 8) {
   case 0: do {
              *to = *from++;
   case 7:    *to = *from++;
   case 6:    *to = *from++;
   case 5:    *to = *from++;
   case 4:    *to = *from++;
   case 3:    *to = *from++;
   case 2:    *to = *from++;
   case 1:    *to = *from++;
            } while (--n > 0);
} }
```

Quelle: Wikipedia

Programmieren mit Ausnahmen

Ausnahmen sollten **unerwartete** und **seltene** Situationen modellieren.

1. "Ask forgiveness, not permission"
   - Bessere Robustheit
2. "Let it fail"
3. Nur Ausnahmen fangen, die auch behandelt werden
   - Unbehandelte Fehler werden nur schlimmer
4. Ausnahmen können der Effizienz und der Übersichtlichkeit dienen.

Semantik

Ausnahmen

- Einfacher Ansatz: es gibt eine feste Menge von Ausnahmen $X = \{x_1, \ldots, x_n\}$.
- Dazu Laufzeitfehler $E = \{E_0, E_1\}$ (Division durch Null, undefinierter Speicherzugriff)
- Sprachkonstrukte:
  - **throw**(x) (mit $x \in X$) wirft Ausname x
  - **try** c_1 **catch** $f \to c_2$ fängt Ausnahme oder Laufzeitfehler $f \in X + E$

Erweiterung der Sprache

$$
\begin{aligned}
l &::= i \mid l.i \mid l[e] \\
e &::= \mathbb{Z} \mid \mathit{true} \mid \mathit{false} \mid l \\
&\quad \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \\
&\quad \mid e_1 == e_2 \mid e_1 < e_2 \\
&\quad \mid \,!\, e \mid e_1 \,\&\&\, e_2 \mid e_1 \,||\, e_2 \\
&\quad \mid l := e \\
&\quad \mid \textbf{throw}(x)
\end{aligned}
$$

$$
\begin{aligned}
c &::= e \mid \textbf{if}\ (e)\ \textbf{then}\ c_1\ \ \textbf{else}\ \ c_2 \mid \textbf{while}\ (e)\ c \mid c_1; c_2 \mid \textbf{nil} \\
&\quad \mid \textbf{try}\ \ c_1\ \ \textbf{catch}\ \ x \to c_2
\end{aligned}
$$

Semantik

- Für Ausdrücke und Anweisungen:

$$F = X + E$$
$$\Sigma \rightharpoonup (\mathbf{V} + F) \times \Sigma$$
$$\Sigma \rightharpoonup (() + F) \times \Sigma$$

- Reguläre Auswertung (ohne Fehler oder Ausnahmen) gibt Wert in $\mathbf{V}$ bzw. () (dient als "Marker") zurück
- Laufzeitfehler oder Ausnahmen geben $f \in X + E$ zurück.

Regeln I: L-Werte

Es werden nur die **zusätzlichen** Regeln aufgeführt:

$$\frac{\Gamma \vdash \langle l, \sigma\rangle \to_{Lexp} \langle f, \sigma'\rangle \qquad f \in X + E}{\Gamma \vdash \langle l.i, \sigma\rangle \to_{Lexp} \langle f, \sigma'\rangle}$$

$$\frac{\Gamma \vdash \langle l, \sigma\rangle \to_{Lexp} \langle f, \sigma'\rangle \qquad f \in X + E}{\Gamma \vdash \langle l[e], \sigma\rangle \to_{Lexp} \langle f, \sigma'\rangle}$$

$$\frac{\Gamma \vdash \langle e, \sigma\rangle \to_{Exp} \langle f, \sigma'\rangle \qquad f \in X + E}{\Gamma \vdash \langle l[e], \sigma\rangle \to_{Lexp} \langle f, \sigma'\rangle}$$

$$\frac{\Gamma \vdash l : \textbf{int} \text{ or } \Gamma \vdash l : \textbf{bool} \qquad \Gamma \vdash \langle l, \sigma\rangle \to_{Lexp} \langle n, \sigma'\rangle \qquad n \notin \mathrm{dom}(\sigma')}{\Gamma \vdash \langle l, \sigma\rangle \to_{Exp} \langle E_1, \sigma'\rangle}$$

Regeln II: Ausdrücke

Es werden nur **zusätzliche** Regeln aufgeführt:

$$\frac{\Gamma \vdash \langle e_1, \sigma\rangle \to_{Exp} \langle f, \sigma'\rangle \qquad f \in X + E}{\Gamma \vdash \langle e_1 / e_2, \sigma\rangle \to_{Exp} \langle f, \sigma'\rangle}$$

$$\frac{\Gamma \vdash \langle e_1, \sigma\rangle \to_{Exp} \langle v, \sigma'\rangle, v \in \mathbb{Z} \qquad \Gamma \vdash \langle e_2, \sigma'\rangle \to_{Exp} \langle f, \sigma''\rangle \qquad f \in X + E}{\Gamma \vdash \langle e_1 / e_2, \sigma\rangle \to_{Exp} \langle f, \sigma''\rangle}$$

$$\frac{\Gamma \vdash \langle e_1, \sigma\rangle \to_{Exp} \langle v, \sigma'\rangle, v \in \mathbb{Z} \qquad \Gamma \vdash \langle e_2, \sigma'\rangle \to_{Exp} \langle 0, \sigma''\rangle}{\Gamma \vdash \langle e_1 / e_2, \sigma\rangle \to_{Exp} \langle E_0, \sigma''\rangle}$$

$$\frac{\Gamma \vdash \langle e_1, \sigma\rangle \to_{Exp} \langle v_1, \sigma'\rangle, v_1 \in \mathbb{Z}, \qquad \Gamma \vdash \langle e_2, \sigma'\rangle \to_{Exp} \langle v_2, \sigma''\rangle, v_2 \in \mathbb{Z}, v_2 \neq 0}{\Gamma \vdash \langle e_1 / e_2, \sigma\rangle \to_{Exp} \langle v_1 \div v_2, \sigma''\rangle}$$

Regeln III: Anweisungen

Es werden nur **zusätzliche** Regeln aufgeführt:

$$\frac{\Gamma \vdash \langle c_1, \sigma\rangle \to_{Stmt} \langle (), \sigma'\rangle}{\Gamma \vdash \langle \textbf{try}\ \ c_1\ \ \textbf{catch}\ \ x \to c_2, \sigma\rangle \to_{Stmt} \langle (), \sigma'\rangle}$$

$$\frac{\Gamma \vdash \langle c_1, \sigma\rangle \to_{Stmt} \langle e_1, \sigma'\rangle \qquad e_1 \neq e_2}{\Gamma \vdash \langle \textbf{try}\ \ c_1\ \ \textbf{catch}\ \ e_2 \to c_2, \sigma\rangle \to_{Stmt} \langle e_1, \sigma'\rangle}$$

$$\frac{\Gamma \vdash \langle c_1, \sigma\rangle \to_{Stmt} \langle f, \sigma'\rangle \qquad \Gamma \vdash \langle c_2, \sigma'\rangle \to_{Stmt} \langle r, \sigma''\rangle}{\Gamma \vdash \langle \textbf{try}\ \ c_1\ \ \textbf{catch}\ \ f \to c_2, \sigma\rangle \to_{Stmt} \langle r, \sigma''\rangle}$$

Zusammenfassung

- Ausnahmen: reglementierte Sprünge
  - In Java, Python, Haskell recht ähnlich
  - In C nicht vorhanden
  - Sollten **unerwartete** und **seltene** Situationen behandeln

Programmiersprachen

Vorlesung 7 vom 20.11.23

Prozeduren und Funktionen

Christoph Lüth

Universität Bremen

Wintersemester 2023/24

Organisatorisches

- Vorträge

Liste möglicher Sprachen

- Logische Programmierung: Prolog, Oz
- Nebenläufig/Reaktiv: Erlang, Golang
- Abhängige Typen: Idris, Agda, Liquid X (Dependent types)
- Prozedural: Julia, Kotlin, Swift
- Skriptsprachen: Lua, Tcl
- Funktional: SML/OCAML, Elm, Clojure/Lisp, Scala, Elixir
- Stack-basiert: Forth
- Historisch: COBOL, Algol-68, APL, Ada, ABAP, Smalltalk
- Datenflusssprachen: Id, Lucid, Lustre
- Shiny: Mojo, Dart
- DSLs: R, SQL, Postscript, TeX, Verilog/VHDL, SystemC, SpinalHDL

Wo sind wir?

- Einführung
- Einfache Ausdrücke und Werte
- Einfache Typen, Anweisungen und Seiteneffekte
- Variablen und Speichermodelle
- Aggregierende Typen
- Ausnahmen und Fehlerbehandlung
- Prozeduren und Funktionen
- Fortgeschrittene Typsysteme
- Datenabstraktion
- Programmierparadigmen
- Skriptsprachen
- Ab Woche 11 (2024): Studentische Vorträge.

Prozeduren und Funktionen

- **Prozeduren** sind benannte, parameterisierte **Blöcke**
  - Meist ohne Rückgabewert
- **Funktionen** sind Prozeduren mit Rückgabewert
  - **Reine** Funktionen (pure functions): referentiell transparent, ohne Seiteneffekt
  - Meist **mit** Seiteneffekten
- Viele Programmiersprachen unterscheiden das nicht
- Funktionsdefinition hat **(formale) Parameter**, beim Aufruf **Parameterwerte** (Argumente)

```
int f(x) { return x*10; }    // 'x' ist formaler Parameter

... f(29+2) ...  // '29+2' ist Parameterwert
```

Parameterübergabe (Parameter Passing)

- Aus **konzeptioneller** Sicht gibt es drei Arten von Parametern:
  - **Eingabeparameter** — erlaubt Kommunikation vom Aufrufer an die Funktion
  - **Ausgabeparameter** — erlaubt Kommunikation von der Funktion an den Aufrufer
  - **Ein/Ausgabeparameter** — erlaubt bidirektionale Kommunikation

Parameterübergabe (Parameter Passing)

- Aus **konzeptioneller** Sicht gibt es drei Arten von Parametern:
  - **Eingabeparameter** — erlaubt Kommunikation vom Aufrufer an die Funktion
  - **Ausgabeparameter** — erlaubt Kommunikation von der Funktion an den Aufrufer
  - **Ein/Ausgabeparameter** — erlaubt bidirektionale Kommunikation
- Beispiel (Ada; Eingabeparater `v`, `w`, Ausgabeparameter `sum`)

```
type Vector is array (1 .. n) of Float;

procedure add (v, w: in Vector; sum: out Vector) is
  begin for in 1 .. n loop
    sum(i) := v(i) + w(i);
  end loop
```

Parameterübergabe (Parameter Passing)

- Aus **konzeptioneller** Sicht gibt es drei Arten von Parametern:
  - **Eingabeparameter** — erlaubt Kommunikation vom Aufrufer an die Funktion
  - **Ausgabeparameter** — erlaubt Kommunikation von der Funktion an den Aufrufer
  - **Ein/Ausgabeparameter** — erlaubt bidirektionale Kommunikation
- Beispiel (Ada; Eingabeparater `v`, `w`, Ausgabeparameter `sum`)

```
type Vector is array (1 .. n) of Float;

procedure add (v, w: in Vector; sum: out Vector) is
  begin for in 1 .. n loop
    sum(i) := v(i) + w(i);
  end loop
```

- Aus **operationaler** Sicht gibt verschiedene Arten der **Parameterübergabe**

Call by Value

- **Parameterwert** ist **beliebiger** Ausdruck (R-Wert)
- **Funktionsaufruf**:
  - Parameter wird zu v ausgewertet
  - Formaler Parameter wird lokale Variable im Funktionsrumpf, mit v initialisert
  - Funktionsrumpf wird ausgeführt
- Für **Eingabeparameter**
- Klare Semantik (kein Effekt auf Aufrufer)
- Effizient für “kleine”, ineffizient für “große” Datenstrukturen (Felder etc.)
- Wertet eventuell zu viel aus

Call by Reference (Call by Variable)

- Parameterwert muss ein L-Wert sein
- Funktionsaufruf:
  - Umgebung des Funktionsrumpfes wird erweitert
  - Formaler Parameter wird zu L-Wert aufgelöst (aliasing)
  - Funktionsrumpf wird ausgeführt
- Für **Ausgabeparameter** und **Ein/Ausgabeparameter**
- Funktion kann Parameterwert verändern
- Effizient aber fehleranfällig (wegen Aliasing)

```
void foo(reference int x)
{ x= x+1; }

char V[10];
i= 2;
V[2]= 5;
foo(V[i]);
// V[2] == 6
```

Call By Name

- Parameterwert ist beliebiger Ausdruck `e` (R-Wert)
- Aufruf von Funktion `f(x)` ist semantisch äquivalent zur Ausführung des Rumpfes, in dem alle `x` durch `e` ersetzt werden.
- Semantisch sauber, aber subtil
- Stammt von ALGOL-60, wird heute nur noch wenig benutzt

```
int x= 0;
int foo(name int y)
{
  int x= 2;
  return x+y;
}
...
int a= foo(x+1);
  // = int x= 2; x+ x+ 1 ???
```

Jensen's Device

- Call-by-Name erlaubt **Metaprogrammierung** (Macros)
- Beispiel: Jensen's Device

```
int sum(name int exp; name int i; int fr; int to)
{
  int acc= 0;
  for (i= fr; i<= to; i++) acc= acc+ exp;
  return acc;
}
int x= ...;
int y= sum(2*x*x- 1, x, 1, 10)
```

- Berechnet

$$y = \sum_{x=1}^{10} 2x^2 - 1$$

Variationen

- Call by **constant**:
  - Wenn Funktionsrumpf den formalen Parameter nicht modifiziert kann call-by-value durch call-by-reference implementiert werden.
- Call by **need** (Haskell):
  - Ähnlich call-by-name, Parameterwert wird **nur** ausgewertet, wenn er benutzt wird
- Call by **value** mit Zeigern (C, Java, Python, Rust)
  - Wenn Werte Zeiger (Referenzen) sind kann der Aufruf Seiteneffekte haben
  - Parameter vom Typ Pointer (C) oder `Object` (Java, Python) sind Ein/Ausgabe-Parameter

Parameterübergabe in C

C-Standard (C99, §6.5.2.2)

1. The expression that denotes the called function[77] shall have type pointer to function returning void or returning an object type other than an array type.
4. An argument may be an expression of any object type. In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument.[a]

[a] A function may change the values of its parameters, but these changes cannot affect the values of the arguments. On the other hand, it is possible to pass a pointer to an object, and the function may change the value of the object pointed to. A parameter declared to have array or function type is adjusted to have a pointer type as described in 6.9.1.

Paramterübergabe in Rust

- Ähnlich C/Java (call by value), aber **restriktiver**: **Ownership**
- Datenstrukturen haben einen **Owner**.
- Parameterübergabe:
  - Entweder als **unveränderliche Referenz**,
  - oder als **veränderliche Referenz**.
  - Muss in der Funktionsdefinition deklariert werden.

Falsch:

```
fn main() {
  let s = String::from("hello");

  let len= calc_len(&s); // Works
  change(&s); // Does not work
}

fn change(some_string: &String) {
  some_string.push_str(", world")
}

fn calc_len(str: &String)-> usize {
  s.len()
}
```

Paramterübergabe in Rust

- Ähnlich C/Java (call by value), aber **restriktiver**: **Ownership**
- Datenstrukturen haben einen **Owner**.
- Parameterübergabe:
  - Entweder als **unveränderliche Referenz**,
  - oder als **veränderliche Referenz**.
  - Muss in der Funktionsdefinition deklariert werden.
- Vergleiche `const` in C.

Richtig:

```
fn main() {
  let mut s = String::from("hello");

  let len= calc_len(&s); // Works
  change(&mut s); // Works
}

fn change(some_string: &mut String) {
  some_string.push_str(", world")
}

fn calc_len(str: &String)-> usize {
  s.len()
}
```

Parameterübergabe und Auswertungsstrategie

- Auswertungsstrategien in funktionalen Sprachen:
  - Innermost-first
  - Outermost-first
- Innermost-first $\sim$ call-by-value, eager evaluation
- Outermost-first $\sim$ call-by-need, lazy evaluation
- Outermost-first: nicht-strikt

```
f 7 undefined  ⇝  14
```

Beispiel:

```
f x y = x+ x
```

Auswertung innermost:

```
    f (f 7 3) (f 5 9)
⇝   f (7+7) (5+ 5)
⇝   f 14 20
⇝   14+14  ⇝  28
```

Auswertung outermost:

```
    f (f 7 3) (f 5 9)
⇝   f 7 3+ f 7 3
⇝   (7+ 7)+ (7+ 7)
⇝   14+14  ⇝  28
```

Funktionen höherer Ordnung

Funktionen Höherer Ordnung

- Funktionen höherer Ordnung sind Funktionen $A \to B$ mit A oder B eine Funktion.
- Funktion als Argument, Beispiel (Python):

```
map(str, [1, 18, true, "foo"])
```

- Funktion als Resultat, Beispiel `3 <=` (vom Typ `Int-> Bool`) in (Haskell):

```
filter (3 ≤) [0,7,1,8,2,9,-2]
```

- Dabei hilfreich: “anonyme” Funktionen (Lambda-Ausdrücke), Beispiel (Python):

```
filter (lambda x: 3<= x, [0,7,1,8,2,9,-2])
```

- Komplikationen: Scoping
- Python und besonders Haskell unterstützen Funktionen höher Ordnung

Funktionen höherer Ordnung in C

- Auch C unterstützt Funktionen höherer Ordnung — durch Zeiger
- Beispiel:

```
typedef struct list_t {
     void          *elem;
     struct list_t *next;
     } list_t;
extern list_t *filter(int f(void *x), list_t *l);
extern list_t *map(void *f(void *x), list_t *l);
```

- Problem: Speicherverwaltung, Typsystem nicht expressiv genug
- Wird genutzt für Sprungtabellen, Signalhandler, Callbacks.

Dynamische Bindung

- Methode `f` einer Klasse kann auf allen Untertypen angewandt werden.
- Konkrete Klasse der Instanz bestimmt konkrete Methode (**dynamische Bindung**)

```
class C:
  def f(self):
    print("Foo.")
  def g(self):
    print("Baz.")

class D(C):
  def f(self):
    print("Wibble.")
```

Semantik

Funktionsparameter

```
int foo(int x)
{
  x= 2*x +1;
  return x;
}
```

- Was ist der Parameter `x`?
- Lokale Variable mit unbestimmten (aber definierten) initialen Wert.
- Initialer Wert wird bei Funktionsaufruf **übergeben**.

Funktionsparameter

```
int foo(int x)
{
  x= 2*x +1;
  return x;
}
```

- Was ist der Parameter `x`?
- Lokale Variable mit unbestimmten (aber definierten) initialen Wert.
- Initialer Wert wird bei Funktionsaufruf **übergeben**.
- Benötigen semantischen Mechanismus, der diese **Übergabe** modelliert.

Ein Programm

$$
\begin{aligned}
\phi \equiv \quad & \textbf{fun} \;\; f_1(x_{1,1}, \ldots, x_{1,n_1}) = b_1 \\
& \textbf{fun} \;\; f_2(x_{2,1}, \ldots, x_{2,n_2}) = b_2 \\
& \ldots \\
& \textbf{fun} \;\; f_m(x_{m,1}, \ldots, x_{m,n_m}) = b_m
\end{aligned}
$$

- Programme haben an sich keine Semantik.
- Stattdessen: jede Funktion hat eine Semantik.

Parameterized Blocks

- Block mit **Parameter**

$$pb ::= \lambda i.pb \mid c$$

- Damit:

$$\begin{aligned} \phi \equiv \quad & \textbf{fun} \;\; f_1 = \lambda x_{1,1}, \ldots, x_{1,n_1}.\, b_1 \\ & \textbf{fun} \;\; f_2 = \lambda x_{2,1}, \ldots, x_{2,n_2}.\, b_2 \\ & \ldots \\ & \textbf{fun} \;\; f_m = \lambda x_{m,1}, \ldots, x_{m,n_m}.\, b_m \end{aligned}$$

- Müssen Umgebung Γ mit **Funktionen** erweitern: $\Gamma_{\text{fun}}(\phi) = \{(f_i, pb_i) \mid (f_i, pb_i) \in \phi\}$

Funktionsaufruf

Modelliert durch **Substitution**:

$$
\begin{aligned}
n[e/x] &= n \\
y[e/x] &= \begin{cases} e & x = y \\ y & \text{otherwise} \end{cases} \\
(e_1 + e_2)[e/x] &= (e_1[e/x]) + (e_2[e/x]) \\
&\dots \\
(c_1; c_2)[e/x] &= (c_1[e/x]); (c_2[e/x]) \\
(\textbf{if } (b) \textbf{ then } c_1 \textbf{ else } c_2)[e/x] &= \textbf{if } (b[e/x]) \textbf{ then } c_1[e/x] \textbf{ else } c_2[e/x] \\
(\textbf{while } (b)\ c)[e/x] &= \textbf{while } (b[e/x])\ (c[e/x]) \\
(\lambda y.\, c)[e/x] &= \begin{cases} \lambda y.\, c & x = y \\ \lambda y.\, (c[e/x]) & x \neq y, y \notin \mathrm{FV}(e) \\ \lambda z.\, ((c[z/y])[e/x]) & x \neq y, y \in \mathrm{FV}(e), z \notin \mathrm{FV}(e) \cup \mathrm{FV}(c) \end{cases}
\end{aligned}
$$

Extending the Language

$$
\begin{aligned}
l &::= i \mid l.i \mid l[e] \\
e &::= \mathbb{Z} \mid \mathit{true} \mid \mathit{false} \mid l \\
&\quad \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1/e_2 \\
&\quad \mid e_1 == e_2 \mid e_1 < e_2 \\
&\quad \mid \,!\, e \mid e_1 \,\&\&\, e_2 \mid e_1 \,||\, e_2 \\
&\quad \mid l := e \mid \textbf{throw}(x) \\
&\quad \mid f(e_1, \ldots, e_n) \\
c &::= e \mid \textbf{if}\ (e)\ \textbf{then}\, c_1\ \ \textbf{else}\ \ c_2 \mid \textbf{while}\ (e)\ c \mid c_1; c_2 \mid \textbf{nil} \mid \textbf{try}\ \ c_1\ \ \textbf{catch}\ \ x \to c_2 \\
&\quad \mid \textbf{return}\ \ e \\
pb &::= \lambda i.\, pb \mid c
\end{aligned}
$$

Funktionsaufruf

- Werte zurückgeben: $E = \{E_R\} \times \mathbf{V} \cup \{E_0, E_1\}$
- Damit:

$$\frac{\Gamma(f) = \lambda x.\, b \qquad \langle b[e/x], \sigma\rangle \rightarrow_{Stmt} \langle (E_R, v), \sigma'\rangle \qquad v \in \mathbf{V}}{\Gamma \vdash \langle f(e), \sigma\rangle \rightarrow_{Exp} \langle v, \sigma'\rangle} \tag{1}$$

Funktionsaufruf

- Werte zurückgeben: $E = \{E_R\} \times \mathbf{V} \cup \{E_0, E_1\}$
- Damit:

$$\frac{\Gamma(f) = \lambda x.\, b \qquad \langle b[e/x], \sigma\rangle \to_{Stmt} \langle (E_R, v), \sigma'\rangle \qquad v \in \mathbf{V}}{\Gamma \vdash \langle f(e), \sigma\rangle \to_{Exp} \langle v, \sigma'\rangle} \tag{1}$$

$$\frac{\begin{array}{c}\Gamma(f) = \lambda x.\, b \\ \Gamma \vdash \langle e, \sigma\rangle \to_{Exp} \langle v_1, \sigma'\rangle \qquad \langle b[v_1/x], \sigma'\rangle \to_{Stmt} \langle (E_R, v_2), \sigma'\rangle \qquad v_1, v_2 \in \mathbf{V}\end{array}}{\Gamma \vdash \langle f(e), \sigma\rangle \to_{Exp} \langle v_2, \sigma''\rangle} \tag{2}$$

Was fehlt hier?

- Mehrere Argumente
- Exceptions im Funktionsrumpf
- “Fall-through”: kein `return` am Ende

Programmiersprachen
Vorlesung 8 vom 27.11.23
Fortgeschrittene Typsysteme

Christoph Lüth

Universität Bremen

Wintersemester 2023/24

Organisatorisches: Vorträge

08.01.2024	Fabian: Abap	Kenneth: COBOL	Carolin: SQL
11.01.2024	Corinna: VHDL	Matthias: Verilog	Mika: Lua
15.01.2024	Moritz: Postscript	Niklas: Julian	Aljoscha: Kotlin
18.01.2024	Niklas: Swift	Finn B: Dart	Daniel: Golang
22.01.2024	Simeon: Elixir	Raphael: Uiua	Yannic: Liquid X
29.01.2024	Kelke: Mojo	Johannes & Finn: cQASM	

Wo sind wir?

- Einführung
- Einfache Ausdrücke und Werte
- Einfache Typen, Anweisungen und Seiteneffekte
- Variablen und Speichermodelle
- Aggregierende Typen
- Ausnahmen und Fehlerbehandlung
- Prozeduren und Funktionen
- Fortgeschrittene Typsysteme
- Datenabstraktion
- Programmierparadigmen
- Skriptsprachen
- Ab Woche 11 (2024): Studentische Vorträge.

Typäquivalenz

- Grundsätzliche Frage: wann ist $T_1 \cong T_2$? (Wichtig für die Typüberprüfung)
- **Nominal**: wenn sie den gleichen Namen haben.
- **Strukturell**: wenn sie die gleiche Struktur haben.

```
typedef struct a {int a;
                  double d;
                 } t1;

void f(t1 x) {
  printf("Double is %f\n", x.d);
}
```

```
typedef struct b {int a;
                  double d;
                 } t2;

void g(t2 y) {
  f(y);
}
```

Typäquivalenz konkret

- C, Rust[1], Haskell, Java nutzen nominale Typäquivalenz
  - `typedef` in C, `type` in Haskell sind **Typsynonyme**, definiert keinen neuen Typ
- Python nutzt (schwächere Form der) strukturellen Typäquivalenz
  - "Duck typing"

Polymorphie

- Von griechisch $\pi o \lambda \nu \zeta$ (viel), $\mu o \rho \phi \eta$ (Gestalt)
- Ganz allgemein: Funktionen (Methoden), die auf **mehr als einen** Typ anwendbar sind.
- Im speziellen:
  - In **objektorientierten Sprachen**: jede Methode ist auch auf allen Untertypen anwendbar.
  - **Parametrische Polymorphie**: nach Hindley-Milner, uniform auf **allen** Typen definiert.
  - **Ad-Hoc Polymorphie**: Überladene Funktionen, auf **einigen** Typen spezifisch definiert.

Polymorphie in Java und Python

- Methode `f` eine Klasse kann auf allen Untertypen angewandt werden.
- Klasse des Objektes bestimmt konkrete Methode (**dynamische Bindung**)

```
class C:
  def f(self):
    print("Foo.")
  def g(self):
    print("Baz.")

class D(C):
  def f(self):
    print("Wibble.")
```

Parametrische Polymorphie:

- Parametrische Polymorphie: Abstraktion über **Typen**
- Sowohl für Funktionen (Methoden) als auch für Datentypen (Klassen)
- Beispiel: Listen
  - Haben für alle Typen die gleiche Struktur
  - Viele Funktionen auf Listen sind vom Inhalt der Listen unabhängig.
- Typisierung nach dem Hindley-Milner-Damas-Algorithmus

Parametrische Polymorphie in Java: Generics

- Beispiel: Listen

```
class List<T> {
  public T elem;
  public List<T> next;

  public List(T el, List<T> tl) {
    this.elem= el;
    this.next= tl;
  }
```

- Benutzung umständlich weil Java keine Typen inferiert:

```
List<Integer> l1= new List<>(1, new List<>(2, null));
```

Parametrische Polymorphie in Haskell

- Typ-Parameter, an Funktionen oder Typen:

```
data List a = Cons a (List a) | Null

map :: (a → b) → List a → List b
map f Null = Null
map f (Cons a l) = Cons (f a) (map f l)
```

- Haskell leitet Typen ab, vergleicht dann (ggf.) mit deklarierten Typen
- Elegante Benutzung

Parametrische Polymorphie in Rust

- Mehr Java als Haskell
- Beispiel: Option

```
enum Option<T> {
    Some(T),
    None,
}
```

- Muss an Funktionen **explizit** annotiert werden:

```
  fn just<T>(opt: &Option<T>)-> &T {
   match opt { Some(t) => t }
}
```

Parametrische Polymorphie in C

- Der Typ `void *` ist mit `t *` für alle Typen `t` kompatibel.
  - C-Standard (C-90), 6.3.2.3 Pointers:
    A pointer to void may be converted to or from a pointer to any incomplete or object type.
- Manuelle Typannotation nötig — Typinformation geht verloren (bspw. für `map` und `filter`)
- Funktionen höherer Ordnung durch Zeiger auf Funktionen.
- Vergleiche Beispiel.

Theoretische Aspekte:

- Parametrische Polymorphie ist **entscheidbar** (Hindley-Milner-Damas).
  - Mit exponentiellem Aufwand.
  - In der Praxis unerheblich.
- Wird schnell unentscheidbar:
  - Konstruktorklassen
  - Rank-2 Polymorphie
  - Subtyping

Ad-Hoc-Polymorphie und Überladen

- **Ad-Hoc-Polymorphie** ist ein Aspekt von **Überladung:** ein Bezeichner, mehrere Funktionen.
  - Beispiel für Ad-hoc-Polymorphie: Addition + auf verschiedenen numerischen Typen.
  - Beispiel für Überladen: – unär für Negation, binäre für Subtraktion
- **Java** erlaubt Überladen, Ad-Hoc-Polymorphie über Interfaces
- **Rust** erlaubt kein Überladen, Ad-Hoc-Polymorphie über Traits
  - Gleicher Bezeichner mit mehreren, aber **unterschiedlichen** Signaturen
- **C** hat überladene numerische Operatoren (+, −, ∗) und erlaubt sonst **kein** Überladen
- **Python** erlaubt kein Überladen (hat aber default-Parameter, Traits (Mix-Ins), u.ä.)

Ad-Hoc-Polymorphie in Haskell

- Haskell hat Typklassen:

```
class Eq a where
  (==) :: a → a → Bool
class Eq a ⇒ Num a where
  (+) :: a → a → a
instance Num Int where
  a+ b=...
```

- Ansonsten kein Überladen
- Typklassen für Datentypen (Konstrukturklassen)

```
class Monad m where
  (>>=)  :: m a → (a → m b) → m b
  return :: a → m a
```

Untertypen (Subtyping)

- Semantisch ist S ein **Untertyp** von T gdw.

$$S \subseteq T$$

  - Wo immer T gefordert ist, kann auch S benutzt werden.
- Beispiel numerische Typen

$$\mathbb{N} \subseteq \mathbb{Z} \subset \mathbb{R} \tag{1}$$

  - Siehe Typkonversionen in C: `unsigned int`, `int`, `double` sowie verschieden Wortbreiten
- Verallgemeinert: S ist Untertyp von T wenn es eine **Einbettung** gibt:

$$\iota : S \hookrightarrow T \quad \iota \text{ injektiv}$$

- Einbettung: **explizite** Konversion
- Beispiel numerische Typen in Haskell

Record Subtyping

- Vererbung erzeugt semantisch gesehen keine Subtypen

```
class Point {
    double x;
    double y;
}
```

```
class ColouredPoint
      extends Point {
    Colour col;
}
```

- `Point` ist $\mathbb{R} \times \mathbb{R}$, `ColouredPoint` ist $\mathbb{R} \times \mathbb{R} \times \mathit{Colour}$
- Wir können aus jedem `ColouredPoint` einen `Point` machen
  - Allerdings nicht injektiv
  - Solange wir **nur** auf die **Felder** zugreifen
- Deshalb können wir `ColouredPoint` als Untertyp von `Point` **definieren** (“record subtyping” oder “structural subtyping”)

Inheritance is not Subtyping

- Record Subtyping funktioniert nur bei Argumenten:

```
void move(Point p) {
  this.x += p.x;
  this.y += p.y;
}
```

- Kann auch auf `ColouredPoints` angewandt werden.

```
static Point
  fromPolar(double phi, double r) {
  return new Point(r*Math.cos(phi),
                   r* Math.sin(phi));
}
Point move1(Point p) {
  return new Point(this.x+ p.x,
                   this.y+ p.y);
}
```

- Welche Farbe soll der Ergebnistyp haben?
- Wenn $A \subseteq A'$, dann ist $A \to B \subseteq A' \to B$, aber $B \to A \not\subseteq B \to A'$

Subtyping und Parametrische Polymorphie (Generics)

- Frage: sind Typkonstruktoren F **monoton**

$$A \subseteq B \Longrightarrow F(A) \subseteq F(B)?$$

- Ganz allgemein gilt:

$$A \subseteq A' \Longrightarrow \begin{array}{l} A \times B \subseteq A' \times B \\ B \times A \subseteq B \times A' \\ A + B \subseteq A' + B \\ B + A \subseteq B + A' \\ A' \to B \subseteq A \to B \\ B \to A \subseteq B \to A' \end{array}$$

- **Polynomiale** Typkonstruktoren sind Datentypen aus Produkt und Koprodukt (vgl. `data` in Haskell ohne Funktionsräume)
- Polynomiale Typkonstruktoren sind monoton.
- Funktionsräume $A \to B$ machen den Unterschied.

Kovarianz und Kontravarianz

- Ein Typkonstruktor $F(X)$ ist **kovariant**, wenn X nur in **Argumentposition** des Funktionsraums $\rightarrow$ auftaucht.
- Ein Typkonstruktor ist $F(X)$ ist kontravariant, wenn X nur in **Resultatposition** des Funktionsraums $\rightarrow$ auftaucht.
- Wenn F **kovariant**, dann ist F monoton: $A \subseteq B \Longrightarrow F(A) \subseteq F(B)$
- Wenn F **kontravariant**, dann ist F anti-monoton: $A \subseteq B \Longrightarrow F(B) \subseteq F(A)$

Subtyping und Generics

- Praktische Auswirkungen — ein klassisches Beispiel:

```
class Ref<T> {
  private T curr;
  Ref(T init) { this.curr= init; }

  T get() { return curr; }
  void set(T x) { curr= x; }
  }
```

- Consider this:

```
Ref<String> c1 = new Ref<>("foo");
Ref<Object> c2 = c1;   // String ⊆ Object, also Ref<String> ⊆ Ref<Object>
c2.set(1);             // Ändert auch c1
String s = c1.get();   // Liest c1, wo aber jetzt eine Zahl steht... ↯
```

- Problem ist zweite Zeile, `Ref<String>` $\not\subseteq$ `Ref<Object>`

Arrays und Generics

- Lösung: Generics sind **invariant** (Typkonstruktoren nicht monoton)
  - Typ `<? extends T>`, `<? super T>`
- Arrays sind **nicht** invariant:

```
String[] c1 = { "foo" };
Object[] c2 = c1;
c2[0]= 99;
String s = c1[0];
System.out.println(s); // What will happen?
```

Arrays und Generics

- Lösung: Generics sind **invariant** (Typkonstruktoren nicht monoton)
  - Typ `<? extends T>`, `<? super T>`
- Arrays sind **nicht** invariant:

```
String[] c1 = { "foo" };
Object[] c2 = c1;
c2[0]= 99;
String s = c1[0];
System.out.println(s); // What will happen?
```

- Grund: Arrays sind **reifizierbar** — Typ wird zur Laufzeit **nicht** gelöscht.
- Bei Generics wird der Typ zur Laufzeit gelöscht (type erasure) — effizientere Ausführung.

Subtyping, Overloading and Dynamic Binding

- Java kombiniert Subtyping, Overloading und dynamisches Binden.
- Im ersten Argument (Methodenauswahl) wird die Methode **dynamisch** ausgewählt.
- Ansonsten wird der Typ **statisch** bestimmt.
- **Overloading** wird **statisch** aufgelöst.
- Siehe Beispiel.

Abhängige Typen (Dependent Types)

- Abhängige Typen vermischen Typen und Terme
- Typen können mit Termen gebildet werden
- Beispiel (Idris): `Vec l` sind Listen der Länge `l :: Int`

```
  data Vect : Nat -> Type -> Type where
  Nil  : Vect Z a
  (::) : a -> Vect k a -> Vect (S k) a

(++) : Vect n a -> Vect m a -> Vect (n + m) a
Nil ++ ys   ys = ys
(x :: xs) ++ ys = x :: xs ++ ys
```

- Nicht mehr entscheidbar.
- Erlaubt Kodierung von **Spezifikation** im Typ.

Abhängige Paare

- Einfach wenn die Länge statisch berechenbar ist:

```
map : (a-> b)-> Vect n a-> Vect n b
map f []        = []
map f (x::xs) = f x :: map f xs
```

- Was ist, wenn wir die Länge nicht kennen:

```
filter : (a -> Bool) -> Vect n a -> (m ** Vect m a)
filter p Nil = (_ ** [])
filter p (x :: xs) with (filter p xs)
  | ( _ ** xs' ) = if (p x) then ( _ ** x :: xs' ) else ( _ ** xs' )
```

- Typ gegeben als abhängiges Paar:

```
data DPair : (a : Type) -> (P : a -> Type) -> Type where
   MkDPair : {P : a -> Type} -> (x : a) -> P x -> DPair a P
```

Zusammenfassung

- Fortgeschrittene Aspekte der Typsysteme
- Arten der Polymorphie
  - Polymorphie über Subtypen
    - Java und Python
  - Parametrische Polymorphie
    - In Haskell, Java (Generics); rudimentär in C (`void *`)
  - Ad-Hoc-Polymorphie und Overloadimg
    - Overloading in Java, Ad-Hoc-Polymorphie in Haskell
- Subtyping
  - Record Subtyping in Java, Python.
  - Kombination mit parametrischer Polymorphie delikat.

Programmiersprachen
Vorlesung 9 vom 04.12.23
Datenabstraktion

Christoph Lüth

Universität Bremen

Wintersemester 2023/24

Organisatorisches

- Vorträge — kleine Verschiebungen

- Aktuelle Version immer auf der Webseite:
  `https://user.informatik.uni-bremen.de/clueth/lehre/ps.ws23/`

Wo sind wir?

- Einführung
- Einfache Ausdrücke und Werte
- Einfache Typen, Anweisungen und Seiteneffekte
- Variablen und Speichermodelle
- Aggregierende Typen
- Ausnahmen und Fehlerbehandlung
- Prozeduren und Funktionen
- Fortgeschrittene Typsysteme
- Datenabstraktion
- Programmierparadigmen
- Skriptsprachen
- Ab Woche 11 (2024): Studentische Vorträge.

Abstraktion

- Definition Wikipedia:

> Das Wort Abstraktion[a] bezeichnet meist den [...] Denkprozess des erforderlichen Weglassens von Einzelheiten und des Überführens auf etwas Allgemeineres oder Einfacheres. Daneben gibt es spezifische sowie unspezifische Verwendungen des Begriffes in bestimmten Einzelwissenschaften und einzelnen Theorien, Thesen sowie Behauptungen.
>
> [a] lat. *abstractus* "abgezogen", Partizip Perfekt Passiv von abs-trahere "abziehen", "trennen"

- Weiter: "In der Mathematik [...] werden Abstrakta meist mit Äquivalenzklassen identifiziert."
- Im Lambda-Kalkül ist Abstraktion $\lambda x.\, t$ — die Einführung des Funktionsparameters

Arten der Abstraktion

- Kontrollabstraktion (done):
  - Strukturierte Programmierung (statt GOTO)
  - Ausnahmen (strutkurierte Fehlerbehandlung)
  - Prozeduren und Parameter
- Datenabstraktion (heute):
  - Abstrakte Datentypen
  - Verkapselung und Objekte
  - Module
  - Packages

Wozu Datenabstraktion?

- Kontrollabstraktion hilft uns, Programme **verständlich** zu machen.
- Datenabstraktion hilft uns, **große** Programme verständlich zu machen.
- Indem wir existierende Daten und Funktionen (Methoden) zu neuen Datentypen zusammenfassen erlauben wir Abstraktion in der Sprache.
  - Abstraktion = "geordnetes Weglassen", hier: von Implementationsdetails.
  - Triviales Beispiel: `Int` als `Bool`, ist `0` jetzt `True` oder `False`?

Abstrakte Datentypen

Abstrakter Datentyp (ADT)

Ein ADT besteht aus besteht aus einem (oder mehreren) **Typen** und **Operationen** darauf, mit folgenden Eigenschaften:

1. Werte des Typen können nur über die Operationen **erzeugt** werden
2. Eigenschaften von Werten des Typen werden nur über die Operationen **beobachtet**.
3. Die Einhaltung von **Invarianten** über dem Typ kann garantiert werden

- Damit eine Programmiersprache ADTs unterstützt, müssen wir die **Sichtbarkeit** einschränken können (*information hiding*).
- **Repräsentationsunabhängigkeit**: Eigenschaften sollten von konkreter Implementation unabhängig sein.

Beispiel: Stack

Ein Stack (von ganzen Zahlen) hat mehrere Operationen:

- den leeren Stack (**empty**),
- eine Zahl auf den Stack schieben (**push**),
- die oberste Zahl vom Stack nehmen (**top** und **pop**),
- und einen Test, ob der Stack leer ist (**isEmpty**).

mit folgenden Eigenschaften:

1. der leere Stack ist leer, und nur der;
2. das oberste Element das Stacks ist dasjenige, was zuletzt darauf geschoben worden ist;
3. wenn ich von einem Stack, auf den ich ein Element geschoben habe, das oberste Element wieder herunternehme, ändert sich nichts.

Spezifikation

- Wollen Stacks **mathematisch** beschreiben
  - Möglichst unzweideutig
  - Können daraus Tests generieren
- Hier:

$$\text{top}(\text{push}(s, x)) = x \qquad \text{isEmpty}(\text{empty})$$

$$\text{pop}(\text{push}(s, x)) = s \qquad \neg(\text{isEmpty}(\text{push}(s, x)))$$

- Gleichungen gelten nur für **unveränderliche** (zustandsfreie) Stacks
- Was bedeutet Gleichheit?
  - Gleichheit für `int` — bekannt
  - Gleichheit für `stack` — nicht gleich, nur **beobachtbar** gleich

Stack: Implementation

- Beispiel: Stack in Python
  - Stack als Liste
  - Stack als Feld
- Unveränderliche Stacks (immutable): `push` und `pop` liefern **neuen Stack**
- Veränderliche Stacks (mutable): `push` und `pop` haben Seiteneffekte
- Problem: Python schränkt Sichtbarkeit bedingt ein
  - Keine Trennung zwischen Interface und Implementation
  - Private Felder **syntaktisch** gekennzeichnet (`__name`), Zugriff trotzdem möglich (als `_Class__name`)
  - Python Design-Prinzip: "Wir sind alle erwachsen."

Stacks in anderen Sprachen

- Beispiel: Stack in C
  - Interface `stack.h` syntaktisch getrennt
- In Haskell:
  - Wir können Sichtbarkeit einschränken, aber syntaktisch nicht trennen
  - **Nur** funktionale Lösung
- In Java:
  - **Interface** als separates Konstrukt

Sprachmittel zur Unterstützung von Datenabstraktion

- Sichtbarkeitseinschränkungen aka. Module
- Syntax zur Beschreibung von Schnittstellen
- Trennung der Schnittstelle von der Implementation
- Modularisierung der Schnittstellen (`interface` in Java, `trait` in Rust, Mix-Ins in Python)

Module

- Ein **Modul** ist die Zusammenfassung mehrerer Definition zu einer Einheit
  - Oft mit Verkapselung — Modul hat definierte **Schnittstelle**
  - Module dienen oft auch zur **getrennten Übersetzung** (aber nicht notwendigerweise)
- Module werden deklariert, definiert und benutzt (importiert).
  - Trennt die Sprache das?
  - Wie erfolgt die Benutzung?
  - Wie verhalten sich Module zu Quelldateien?
- Flacher Namensraum für Module vs. Pfade für Module

Import

- Wird **qualifiziert** oder **unqualifiziert** importiert?
  - Qualifizierter Import: Bezeichner `f` aus Modul `M` wird zu `M.f`.
- Wird immer **alles** importiert, oder kann der Importeur selektieren (wie)?
  - Positive Selektion: importiere `f`, negative Selektion: alles bis auf `f`
- Kann beim Import **umbenannt** werden?
  - Importiere `f` aus `M` und nenne es `g`; importiere `f` aus `M` und nenne das Module `N`

Module in Python

- Module sind Quelldateien
- Keine Interface-Definition
- Kaum Sichtbarkeitseinschränkungen
- Keine Datenabstraktion
- Import: mit Namen, qualifiziert/unqualifiziert, alles oder selektiv, Umbenennung möglich

Module in Haskell

- Jede Quelldatei ist ein Modul
- Interface: Sichtbarkeit von Bezeichnern kann eingeschränkt werden
- Aber:
  - algebraische Datentypen und Konstruktoren bleiben erkennbar
  - Typsynonyme sind nicht abstrakt
  - Klasseninstanzen werden immer exportiert
- Datenabstraktion möglich (manchmal umständlich)
- Interface keine separate Datei
- Import: mit Namen, qualifiziert/unqualifiziert, alles oder selektiv, Umbenennung möglich

Module in Java

- Module sind **Klassen** (nicht an Quelldatei gebunden)
- Klassen verkapseln interne Repräsentation, Einschränkung der Sichtbarkeit (`public`, `private`, `protected`)
  - Aber Konstruktoren bleiben erkennbar
- Datenabstraktion möglich (durch Reflektion zu durchbrechen?)
- Interfaces sind separates Konstrukt
- Import: nur ganze Klassen, keine Umbenennung, impliziter Import möglich

Module in C

- Module sind Quelldateien (*translation units*)
- Sichtbarkeitseinschränkungen für Bezeichner (`static`, `extern`)
  - Local and global linkage
- Interfaces sind **per Konvention** separate Dateien (`.h`)
  - Konvention wird durch den Präprozessor ermöglicht
- Datenabstraktion möglich (durch Zeigeroperationen zu durchbrechen)
- Import: immer alles, nur unqualifziert, keine Umbenennung

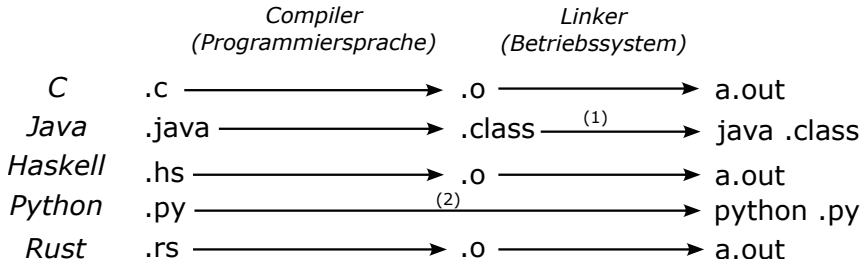
Packages

- Packages schränken die Sichtbarkeit von Modulen ein.
- Existiert in Java, Python.
- Haskell kennt nur hierarchische Module.
- Wenn Module Quelldateien entsprechen, sind Packages Verzeichnisse.

Sprachneutral Interfaces: IDL

- IDL ist die **Interface Definition Language** der OMG
- Definition der Schnittstelle von Komponenten in CORBA
- Syntax an C angelehnt
- Compiler erzeugt aus IDL “Rumpf” in entsprechender Programmiersprache
- Alle Funktionsaufrufe gehen über einen **Broker**
- Nicht mehr ganz aktuell.

Übersetzung: Von der Quelle zum executable



- Java hat plattformübergreifendes Objektdateiformat (1)
- Python interpretiert Quellcode direkt (2)

Der Build-Prozess: Am Anfang war `make`

- Programmiersprachenunabhängig
- Programmierer spezifiziert **Abhängigkeiten manuell**:

```
a.o:  a.c b.h c.h
    gcc -c a.c

b.o:  b.c c.h
    gcc -c b.c

c.o:  c.c
    gcc -c c.c

abc: a.o b.o c.o
    ld -o abc a.o b.o c.o -lfoo
```

Vereinfacht durch vordefinierte **Regeln**:

```
a.o: b.h c.h

b.o:  c.h

abc: a.o b.o c.o
    ld -o abc a.o b.o c.o -lfo
```

Unterstützung des Build Prozesses

- Analyse der Abhängigkeiten in viele Übersetzer **integriert** (Haskell, Rust)
- Verwaltung **externer Büchereien**:
  - Erweitertes Abhängigkeitsprinzip
  - Globale Büchereien (zentrales Repository)
- Werkzeuge:

Sprache	Werkzeugk	Zentr. Repository
C	(CMake)	—
Java	Maven	Maven
Haskell	stack/cabal	Hackage
Python	pip	PyPI
Rust	cargo	`crates.io`

Zusammenfassung

- Datenabstraktion durch **abstrakte Datentypen**:
  - Typ mit Operationen darüber
  - Zugriff nur über definierte Schnittstelle
  - Repräsentationsunabhängigkeit
- Module: **Verkapselung** durch Einschränkung der Sichtbarkeit
  - Was wird verkapselt: Sichtbarkeit der Bezeichner, Typrepräsentation, Konstruktoren?
  - Sind Interfaces explizit oder implizit?
  - Modul = Quelldatei?
  - Wie wird importiert?
  - Qualifizierte Bezeichner `M.f`
- Packages: Sammlungen von Modulen
- Fallbeispiel: Module in SML

Programmiersprachen
Vorlesung 10 vom 11.12.23
Programmierparadigmen

Christoph Lüth

Universität Bremen

Wintersemester 2023/24

Organisatorisches

Die Vorlesung am **Do, 14.12.2023** fällt aus.

Wo sind wir?

- Einführung
- Einfache Ausdrücke und Werte
- Einfache Typen, Anweisungen und Seiteneffekte
- Variablen und Speichermodelle
- Aggregierende Typen
- Ausnahmen und Fehlerbehandlung
- Prozeduren und Funktionen
- Fortgeschrittene Typsysteme
- Datenabstraktion
- Programmierparadigmen
- Skriptsprachen
- Ab Woche 11 (2024): Studentische Vorträge.

Was ist ein Paradigma?

Definition nach dem Duden:

Bedeutungen (2)

1. Beispiel, Muster; Erzählung mit beispielhaftem Charakter

   Gebrauch **bildungssprachlich**

2. Gesamtheit der Formen der Flexion eines Wortes, besonders als Muster für Wörter, die in gleicher Weise flektiert werden

   Gebrauch **Sprachwissenschaft**

Definition nach Merriam-Webster:

Full Definition of *paradigm*

1 : EXAMPLE, PATTERN

especially **:** an outstandingly clear or typical example or archetype

// ... regard science as the *paradigm* of true knowledge.

— G. C. J. Midgley

2 : an example of a conjugation or declension showing a word in all its inflectional forms

3 : a philosophical and theoretical framework of a scientific school or discipline within which theories, laws, and generalizations and the experiments performed in support of them are formulated

// the Freudian *paradigm* of psychoanalysis

broadly **:** a philosophical or theoretical framework of any kind

Was ist ein Paradigma?

Definition nach dem Duden:

Bedeutungen (2)

1. Beispiel, Muster; Erzählung mit beispielhaftem Charakter

   Gebrauch **bildungssprachlich**

2. Gesamtheit der Formen der Flexion eines Wortes, besonders als Muster für Wörter, die in gleicher Weise flektiert werden

   Gebrauch **Sprachwissenschaft**

Definition nach Merriam-Webster:

Full Definition of *paradigm*

1 : EXAMPLE, PATTERN

especially : an outstandingly clear or typical example or archetype

// ... regard science as the *paradigm* of true knowledge.
— G. C. J. Midgley

2 : an example of a conjugation or declension showing a word in all its inflectional forms

3 : a philosophical and theoretical framework of a scientific school or discipline within which theories, laws, and generalizations and the experiments performed in support of them are formulated

// the Freudian *paradigm* of psychoanalysis

broadly : a philosophical or theoretical framework of any kind

Was ist ein Programmierparadigma?

Programmierparadigma

Ein Programmierparadigma ist eine **grundlegende Herangehensweise** an die Programmierung, und umfasst:

- ein **Berechnungsmodell** — wie rechne ich?
- ein **Weltmodell** — was sind die Objekte meiner Berechnungen?
- sowie darauf aufbauende Konzepte.

- Programmierparadigmen sind mehr als nur eine bestimmte Programmiersprache.
- Ein Programmierparadigma bedingt auch eine Modellierung und Systemarchitektur.

Bekannte Programmierparadigmen

- Prozedural-Imperativ
- Objektorientiert
- Funktional
- Logisch

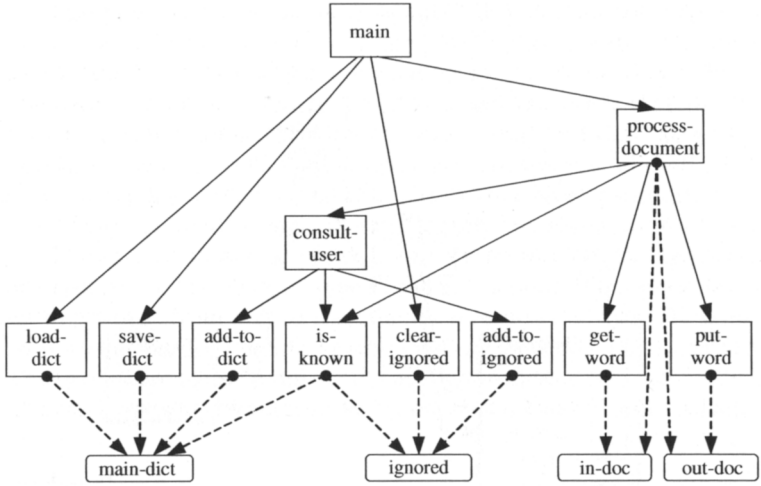
Fallstudie

- Ein einfacher Spellchecker:
- Ablauf:
  - Eingabe: Text aus einer Datei
  - Zerlegt Text in Wörter
  - Prüft Wort gegen Wörterbuch
  - Wenn Wort nicht in Wörterbuch: ignorieren oder hinzufügen
  - Ausgabe: korrigierter Text in neuer Datei
- Quelle: Watt, Programming Language Design Concepts.

Prozedural-Imperativ

- Berechnungsmodell: Zustandsübergänge
- Weltmodell: Abstraktion des Speichers
- Schlüsselkonzepte:
  - Variablen und Befehle
  - Prozeduren und Funktionen
  - Einfache Datentypen
- Prozedural: Abstraktion durch Funktionen und Prozeduren
- Mutter aller Programmierparadigmen
- Programmiersprachen: C (uvm)

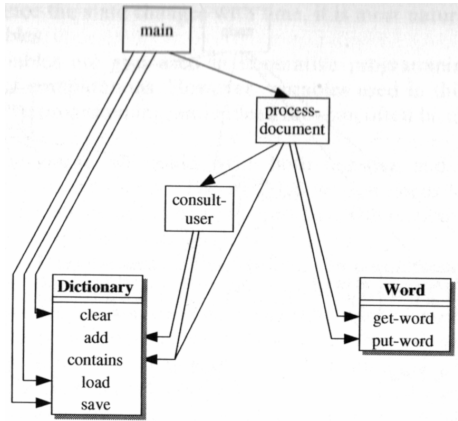
Imperative Systemarchitektur



Fallstudie imperativ

- Ablauf ist imperativ
- Keine Datenabstraktion:
  - Wörterbuch und Ignore-Liste globale Variablen
  - Ein- und Ausgabedatei global
  - Wörter sind `char *`, müssen anders deklariert werden

Prozedurale Systemarchitekure



Fallstudie prozedural

- Datenabstraktionen:
  - Funktionen zu `Dictionary` zusammengefasst
  - Dadurch Vereinfachung: `main_dict` und `ignored` sind beides Dictionaries.
  - Datenabstraktion zu `Word` immer noch unvollständig
- Erste Anflug von Objekt-Orientierung

Objekt-Orientiertes Paradigma

- Objekte der Berechnung sind **Objekte**
- Berechnung sind (abstrakte) Nachrichten (**Methoden**), welche die Objekte austauschen.
- Im einzelnen:
  - Verkapselung der Objekte als **abstrakte Datentypen**
  - Typisierung der Objekte durch Klassen
  - Strukturierung der Klassen durch **Vererbung**
  - Methodenaufruf durch **dynamische Bindung**
  - **Polymorphie** durch Subtyping

Geschichtliches

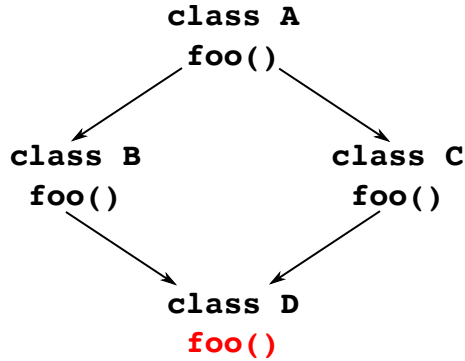
- Erste Sprache: Simula (1960s), Ole-Johann Dahl und Kristen Nygaard
- Simula kannte Objekte, Klassen, Vererbung, Koroutinen und hatte Speicherverwaltung.
- Spätere Sprachen: Smalltalk-80, C++, Eiffel, Java
- Noch spätere Sprachen: Ruby, Python, C#, Scala, OCaml

Vererbung

- Vererbung löst ein Problem mit abstrakten Datentypen:
  - ADTs sind **verkapselt** und könnten gut erweitert und wiederverwendet werden.
  - Aber: Verkapselung verhindert Wiederverwendung
  - Ferner: keine **hierarchischen** Definitionen
- Arten der Vererbung:
  - Einfache Vererbung (jede Klasse hat **eine** Oberklasse)
  - Mehrfachvererbung (jede Klasse hat **mehrere** Oberklassen)

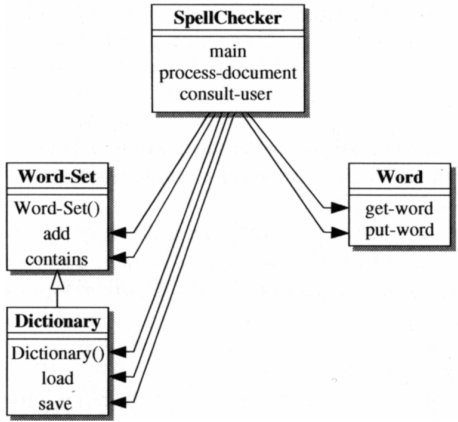
Probleme mit Mehrfachvererbung

1. Das **Diamanten-Problem** (diamond problem):
   - `B` erbt von `A` und überschreibt `foo`
   - `C` erbt von `A` und überschreibt `foo`
   - `D` erbt von `B` und `C` — was ist `foo` in `D`?
2. Implementation



- Sprachen mit Mehrfachvererbung: C++, Eiffel, Python
- Sprachen mit Einfachvererbung: Java, Scala, ...
  - Dafür Interfaces und/oder Traits

Die Fallstudie objekt-orientiert



Funktionales Programmierparadigma

- Berechnungsmodell: rekursive Funktionen
- Weltmodell: algebraische Datentypen
  - frei definierbar — SML, Haskell
  - fest — LISP
- Schlüsselkonzepte:
  - Sprachen können pur (Haskell) oder gemischt sein (SML, LISP)
  - Getypt (Haskell, SML) oder ungetypt (LISP)
  - Strikt (SML, LISP) oder nicht-strikt (Haskell)

Die Fallstudie funktional

- Erste Annäherung:
  - Modularchitektur ähnlich vorher
  - Implementation ähnlich Java
- Zweite Annäherung: funktionale Modellierung
  - `spellcheck` als `stream processor`

```
consultUser :: String → Dictionaries → IO (String, Dictionarie
dropWord :: String → (String, String)
getWord :: String → Maybe (String, String)
```

Zusammenfassung

- Programmierparadigmen bestehen aus einem Weltmodell und einem Berechnungsmodell
  - Was sind die Objekte meiner Berechnung?
  - Wie berechne ich?
- Beispiele:
  - imperativ und prozedural
  - objekt-orientiert
  - funktional
  - logisch
- Die meisten Programmiersprachen sind **gemischt** und unterstützen mehrere Paradigmen.
  - Aber meistens ein bestimmtes besonders gut.

Programmiersprachen
Vorlesung 11 vom 18.12.23
Skriptsprachen

Christoph Lüth

Universität Bremen

Wintersemester 2023/24

Wo sind wir?

- Einführung
- Einfache Ausdrücke und Werte
- Einfache Typen, Anweisungen und Seiteneffekte
- Variablen und Speichermodelle
- Aggregierende Typen
- Ausnahmen und Fehlerbehandlung
- Prozeduren und Funktionen
- Fortgeschrittene Typsysteme
- Datenabstraktion
- Programmierparadigmen
- Skriptsprachen
- Ab Woche 11 (2024): Studentische Vorträge.

Was ist das?

- Kennzeichen von Skriptsprachen:
  1. Komposition komplexer Subsysteme zu einem Gesamtsystem;
  2. Hohe Abstraktionsstufe gegenüber der Hardware;
  3. Kleiner Sprachkern;
  4. Schneller Entwicklungszyklus;
  5. Nicht vorranging effizient, meist interpretiert;
  6. Meist auf ein Anwendungsgebiet zugeschnitten (**domänenspezifisch**).
- Kein **Programmierparadigma**, eher Benutzungspragmatik.

Zitat:

A scripting language is one where the main effect of a program is to drive another system, while in a programming language the program itself is the main action.

Quelle: `http://www.cs.man.ac.uk/~pjj/cs211/langdes/script.html`

Warum?

- Die meisten Skriptsprachen entstehen **evolutionär**:
  - Große Anwendungen mit komplexer Funktionalität
  - Anwender muss Ablauf steuern
  - Repetitiv und schematisch — Bedarf der Automatisierung
- Skriptsprachen **automatisieren**, was der Benutzer händisch macht.

Geschichtliches

- Stapelverabeitung (**batch control**) für Mainframe-Rechner: JCL (IBM) (1960?)
  "JCL ... is, I am convinced, the worst computer programming language ever devised by anybody, anywhere. It was developed under my supervision; there is blame enough to go around among all the supervisory levels." — Fred Brooks.
- Mainframes wurden zu Minicomputern zu PCs ...
  - Benutzerinteraktion durch eine **Shell**
  - Dazu Shell-Sprachen: sh, REXX (ab 1970)
- Später kamen als Anwendungsgebiete dazu: GUIs, Datenbanken, Office-Anwendungen, Webanwendungen (ab 1980)

Moderne Skriptsprachen

- **Allzweckskriptsprachen**:
  - Python, Ruby
- "**Glue languages**":
  - Tcl, bash
- Spezielle **Anwendungsgebiete**:
  - PHP, JavaScript (Webanwendungen)
  - Office-Anwendungen (Visual Basic)
  - Textverarbeitung (awk, sed, Perl)
- **Eingebettete** Sprachen:
  - Lua, Tcl, JavaScript

Schlüsselkonzepte

- Gute Unterstützung von **Zeichenketten**
  - Leichtgewichtige Parsierung durch **reguläre Ausdrücke**
  - Repräsentation strukturierter Daten durch XML oder JSON
- Integration von **Hostanwendungen**
  - GUI, Webschnittstelle, ...
  - Machen die "eigentliche" Arbeit
- Dynamisch getypt oder ungetypt
  - Flexibler, schnellerer Entwicklungszyklus

Beispielsprache: Shell

- Die **Bourne-Shell** und die **Bourne-Again-Shell**
- Erste Version 1976 für Unix V7
- Reimplementierung als Bourne-Again-Shell (`bash`) für GNU/Linux ab 1999.
- Was macht eine **Shell**?
  - Behandlung der **Benutzereingabe** (am Terminal oder auf der Kommandozeile)
  - Starten von Kommandos, Parameterexpansion, Umgebungsvariablen
    - Standardeingabe/-ausgabe
    - Als besondere "Filedeskriptoren"
  - Signalbehandlung (Unterbrechen, Suspendieren)
  - Umleitung der Ein/Ausgabe, Verkettung durch Pipes, Umgebungsvariablen
  - ... und sie kann **programmiert** werden!

Die Shell-Sprache

- Datentypen: Zeichenketten.
  - Komplett ungetypt (Was braucht man mehr?)
  - Ganze Zahlen sind auch nur Zeichenketten
  - Auswertung von Ausdrücken durch *arithmetic expansion*
- Elaborate Unterstützung von Literalen:

```
echo "Hello $World."
echo 'Hello $World.'
```

- Macros (Antiquotation)

```
x=`ls -la`
```

- Syntax: von ALGOL inspiriert

Programmierstrukturen

- Variablen (global, lokal)
- Volle Turingmächtigkeit
- `while`, `if`, `case`
- Ausführungsmodell:
  - Sequentielle Ausführung von Kommandos
  - Parameterexpansion
- Funktionen:
  - Positionale Parameterübergabe
  - Dynamisches Scoping

Beispiele:

- Aus dem echten Leben ...
- Fakultät iterativ (`fac1.sh`)
- Fakultät rekursiv (`fac2.sh`)

Beispielsprache: Tcl

Was ist Tcl?

- Tcl ist eine **Skriptsprache**
  - Dynamisch und komplett ungetypt
  - Als **eingebette** und **erweiterbare** “glue language” konzipiert
- Erfolgreich durch
  - Expect
  - Tk, das GUI Toolkit (Tcl/Tk)
- Inzwischen etwas überholt (insbesondere Tk)

Geschichte

- Entstanden 1987, Autor John Ousterhout (damals University of California at Berkeley)
- Erste Version 1990
- 1993 erste Tcl/Tk-Konferenz
- 1994 Ousterhout wechselt zu Sun, gründet Tcl-Abteilung
- 1997 ACM Software Award für Tcl/Tk
- 2012 Objekt-Orientierte Erweiterung

Syntax und Semantik

- Tcl-Programme (Skripte) bestehen aus einer Sequenz von **Kommandos** der Form

```
cmd arg1 arg2 arg3 ...
```

- Trennung der Kommandos durch Zeilenumbruch oder Semikolon
- Auswertung in drei Schritten:
  1. Gruppierung der Argumente (durch `{ ... }` oder `" ... "`)
  2. Substitution der Argumente
  3. Aufruf des Kommandos

Datentypen

- Strings
- **Keine** numerische Datentypen (aber es gibt ein Kommando dafür)
- Assoziativlisten (`array`, wie in Python) und Listen

Kommandos

- Variablenzuweisung: `set`
- Auswertung arithmetischer Ausdrücke: `expr`
- Auswertung eines Tcl-Kommandos: `eval`
- Fallunterscheidung und Schleifen:
  - `if`,
  - `while`, `foreach`, `for`, `break`, `continue`
  - `if`, `while` erwarten numerische Argumente (`0` ist `false`, wie in C)
- Ausnahmen: `try` und `catch`
- Arraymanipulation: `array`
- Ein/Ausgabe: `gets`, `puts`
- Fallunterscheidung: `switch`
  - Auf Werten oder **regulären Ausdrücken**

Substitution

- Variablen (brauchen nicht deklariert zu werden)

```
set x 5
puts {x is $x}
puts "x is $x"
```

- Kommandosubstitution: `[ ... ]` (kann geschachtelt werden)

```
set x [string length "foo"]
```

- Backslashes: `\$`, `\n`, `\u001b`, ...
- Gruppierung **vor** Substitution

Metaprogrammierung durch `eval`

- `eval` ruft den Tcl-Interpreter auf das Argument auf:

```
set cmd {puts {Hello, World!}}
...
eval $cmd
```

- Auswertung der Variablen zur **Ausführungszeit**:

```
set string "Hello, world!"
set cmd {puts $string}
unset string
eval $cmd
```

- Nutzung: **callbacks**, *e.g.* für GUI-Elemente

Beispiel

- Die Fakultätsfunktion, iterativ:

```
proc Factorial {x} {
  set i 1; set product 1
  while {$i <= $x} {
    set product [expr $product * $i]
    incr i
  }
  return $product
}
```

Beispiel

- Die Fakultätsfunktion, rekursiv:

```
proc Factorial {x} {
  if {$x <= 1} {
    return 1
  } else {
    return [expr $x * [Factorial [expr $x - 1]]]
  }
}
```

Die GUI-Bibliothek Tk

- Integraler Bestandteil der Sprache, und Grund für die Popularität
- Inzwischen etwas out-of-date, aber sehr stabil
- Grundprinzip:
  - GUI läuft asynchron
  - Erzeugt **Events**, an die **Callbacks** gebunden werden
- Eigene Bücherei, hat daher kein (kaum) natives Look&Feel
  - Sieht überall gleich (schlecht) aus

Tcl/Tk im Beispiel: Hello, World!

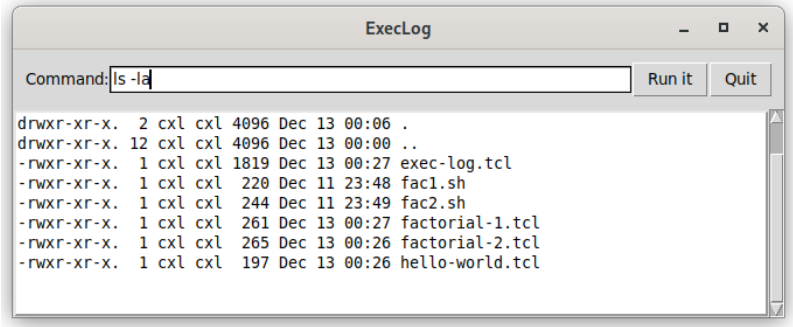
- Ein erstes Beispiel:

```
button .hello -text Hello \
              -command {puts stdout {Hello, World!}}
pack .hello -padx 20 -pady 10
```

- `button` deklariert Knopf
- `pack` plaziert Knopf im GUI

Ein längeres Beispiel

- `exec-log.tcl`



Steckbrief Tcl

Name	Tcl
Entstehung	Ab 1990 von John Ousterhout entwickelt
Programmierparadigma	Imperativ
Typen	Strings, Arrays, Listen
Typsystem	Ungetypt (schwach dynamisch getypt)
Parameterübergabe	call-by-name
Datenabstraktion	Wenig (lokale/global Variablen)
Anwendungsgebiet	"Glue language"
Besondere Kennzeichen	GUI-Bücherei Tk

Zusammenfassung

Zusammenfassung

- Skriptsprachen sind kein Programmierparadigma, eher eine Frage der Nutzpragmatik
- Verschiedene Kennzeichen von Skriptsprachen:
  - Kleiner Sprachkern, interpretiert
  - Schneller Entwicklungszyklus
  - Dient meist dazu, andere Programme zusammenzufügen
- Beispielsprachen: sh, Tcl

Programmiersprachen
Vorlesung 12 vom 21.12.23
Weihnachtsvorlesung

Christoph Lüth

Universität Bremen

Wintersemester 2023/24

Was machen wir heute?

- Eine kleine Sprache
- Eine besondere Sprache

Eine kleine Sprache

Kleine Sprachen

- Wieviele Anweisungen braucht man **mindestens**, um turingmächtig zu sein?

Kleine Sprachen

- Wieviele Anweisungen braucht man **mindestens**, um turingmächtig zu sein?
- **Eine** reicht.
- Beweis: Subleq (Subtract and Branch on result Less than or Equal to zero)
- Beispiel für OISC (One Instruction Set Computer)

Subleq Basics

- Syntax:

```
subleq a b c
```

- Semantik:

$$
\begin{aligned}
r &\leftarrow mem[B] - mem[A] \\
mem[B] &\leftarrow r \\
pc &\leftarrow \begin{cases} pc + C & r \leq 0 \\ pc + 1 & \text{otherwise} \end{cases}
\end{aligned}
$$

Das ist genug?

- Invariante: $mem[0] = 0$ (genannt Z)
- Unbedingte Sprünge: branch c

```
subleq 0, 0, c    ;; 0 := 0- 0 = 0
                  ;; if 0 <= 0 then PC := PC+ C
```

- Addition: **add** a b

```
subleq a, 0, 0    ;; Z := 0- a = -a
subleq 0, b, 0    ;; b := b- Z = b+ a
subleq 0, 0, 0    ;; Z := Z- Z= 0
```

- Kopieren (Move): mov a b

```
subleq b, b, 0    ;; a := b- b = 0
subleq a, 0, 0    ;; Z := Z- a = -a
subleq 0, b, 0    ;; b := b- Z = 0+ a
subleq 0, 0, 0    ;; Z := 0
```

Source: https://en.wikipedia.org/wiki/One-instruction_set_computer#Subtract_and_branch_if_less_than_or_equal_to_zero

Ist das Turing-mächtig?

- Nicht ganz, aber es demonstriert das Prinzip.
- Weitere Instruktionen: **add** a b c, **beq** a c
- Formaler Beweis: Übersetzung einer Registermaschine nach Subleq.
- Nützlichkeit von Subleq:
  1. Einfach in Hardware zu implementieren
  2. Basis für kompakte Mikroarchitektur

Eine besondere Sprache

Brainfuck

- Brainfuck wurde 1993 von Urban Müller erfunden, um den "kleinstmöglichen Compiler für eine Turing-vollständige Sprache" zu schreiben.
- Abgeleitet von P'' (Corrado Boehm, 1964)
- Acht Kommandos, Turing-vollständig
- Kryptisch und von keinem praktischen Nutzen

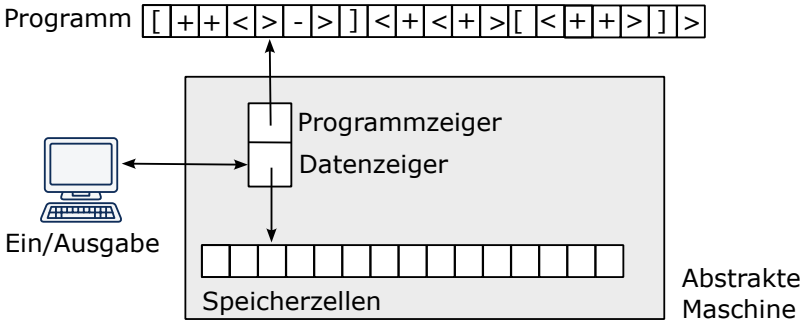
Ein Beispielprogramm

- Hello, world:

```
++++++++
[>++++[>++>+++>+++>+<<<<-]>+>+>->>+[<]<-]
>>.>---.+++++++..+++.>>.<-.
<.+++.------.--------.>>+.>++.
```

Die Sprache

- Syntax: acht Kommandozeichen
- Lexikalik: alles andere ist Kommentar
- Ausführungsmodell:



Kommandos

<	Datenzeiger erhöhen
>	Datenzeiger erniedrigen
+	Wert der aktuellen Zelle erhöhen
-	Wert der aktuellen Zelle erniedrigen
.	Wert der aktuellen Zelle ausgeben
,	Wert der aktuellen Zelle einlesen
[	Springt hinter das entsprechende] wenn Wert der aktuellen Zellen 0 ist
]	Springt hinter das entsprechende [wenn Wert der aktuellen Zellen nicht 0 ist

- [P] ist Iteration von P solange aktueller Zellenwert ungleich 0 ist.

Einfache Programme

- ,[. ,]

Einfache Programme

- ,[. ,] Echo
- [>+<−]

Einfache Programme

- ,[. ,] Echo
- [>+<−] Addition p[i+1]= p[i+1]+p[i]
- [>−<−]

Einfache Programme

- ,[. ,] Echo
- [>+<−] Addition p[i+1]= p[i+1]+p[i]
- [>−<−] Subtraktion p[i+1]= p[i+1]−p[i]
- >[−]<[>+<−]

Einfache Programme

- `,[.,]` Echo
- `[>+<-]` Addition p[i+1]= p[i+1]+p[i]
- `[>-<-]` Subtraktion p[i+1]= p[i+1]−p[i]
- `>[-]<[>+<-]` Verschieben p[i+1]= p[i]; p[i]= 0

Größere Programme

- Kopieren:

```
>[-]>[-]<<          Initialisierung
[>+>+<<-]           Verschiebe p[i] nach p[i+1],p[i+2]
>>[<<+>>-]          Verschiebe p[i+2] nach p[i]
<<
```

Zusammenfassung

- Brainfuck ist:
  - Turing-vollständig
  - extrem kompliziert zu benutzen
  - extrem einfach zu implementieren
  - in der Praxis unbrauchbar
- Weitere skurrile Programmiersprachen (siehe `https://esolangs.org/`):
  - Intercal: eine der ersten dieser Art.
  - Whitespace: Token sind TAB, LF, Space. Programme sind ‘unsichtbar‘ und können in andere Programmiersprachen (oder Texte) eingebettet werden.


Frohe Weihnachten und einen Guten Rutsch!

Programmiersprachen
Vorlesung 13 vom 01.02.24
Abschließende Bemerkungen

Christoph Lüth

Universität Bremen

Wintersemester 2023/24

Vorbemerkung

Bitte die Veranstaltung **bewerten**! (s. stud.ip)

Die Prüfungen

- Daten: 15.02.2024 und 14.03.2024
- Anmeldung über stud.ip
- Was wird geprüft?
  - Verständnis, weniger enzyklopädisches Wissen
  - Hauptsächlich Stoff aus Vorlesung, Übungen, weniger Referate
- Ablauf: Einzelprüfung, 25 Minuten
- Per default: **Kein Beisitzer**. Wenn gewünscht, bitte vorher Nachricht.

Bewertung

- Alles und noch mehr gewußt: 1
- Fast alles gewußt: 2
- Kleine Lücken: 3
- Große Lücken: 4
- Hauptsächlich Lücken: 5

Grundlagen

- Was ist eine Programmiersprachen?
- Was ist Turing-Mächigkeit?
- Operationale Semantik
  - Was ist das?
  - Warum haben wir das hier gemacht?
  - Wie funktioniert das?
- Was sind Werte? Was sind Typen?

Sprachen

- C
- Rust
- Java
- Python
- Haskell

Variablen und Speichermodelle

- Was macht eine Variablendeklaration?
- Lebenszyklus, Speichermodelle, Stack, Heap, Garbage Collection, Ownership
- Sichtbarkeit, Binding, Scope (statisch oder dynamisch)
- Lokale Namen (Bindung), Substitution, name capture

Typen

- Basistypen
- Aggregierende Typen: Produkte und Tupel, Arrays, Listen, Maps, disjunkte Vereinigungen und rekursive Typen
  - Welche Sprache hat welche Typen?
- Gleichheit
- Copy Semantics
- Referenzen

Fehler und Ausnahmen

- Ausnahmen vs. reifizierte Fehler
- Semantik: $\Sigma \rightharpoonup (V \times \Sigma) + E$ vs. $\Sigma \rightharpoonup (V + e) \times \Sigma$
- “Ask forgiveness, not permission.”

Prozeduren und Funktionen

- Arten von Parametern
- Arten der Parameterübergabe
- Welche Sprache kann was?
- Wie wird es semantisch modelliert?

Typsysteme

- Typäquivalenz
- Polymorphie: Arten der Polymorphie, in verschiedenen Sprachen
- Subtyping und Inheritance, Subtyping und parametrische Polymorphie
- Abhängige Typen (Dependent Types)

Datenabstraktion

- Was ist das? Abstrakte Datentypen
- Sprachmittel zur Unterstützung der Datenabstraktion
- Module, mix-ins, Packages, ...
- Semantisch: Grundprinzip

Programmierparadigmen

- Was ist ein Programmierparadigma? Unterschied zu einer Programmiersprache?
- Woraus besteht ein Programmierparadigma?
- Was sind bekannte Programmierparadigmen?