Imperative Sprachen

Wichtiges Konzept für Variablen und ihre Attribute:

Binding

- Verknüpfung (zwischen Entity und Attribut, zwischen Operation und Symbol,….
- Binding time: Zeitpunkt der Verknüpfung
  - Sprachentwurfszeit: binde `*` an Multiplokationsoperation
  - Compiler-Implementierungszeit: binde `int` an bestimmten erlaubten Wertebereich
  - Compilezeit: binde Variable an bestimmten Typ
  - Ladezeit des Programms: binde globale Variable an Speicherzelle
  - Laufzeit: binde lokale Variable an Speicherzelle
- zum Programmverständnis wichtig: wissen
  - wann eine Variable gebunden wird
  - wie z.B. aktuelle Parameter an formale gebunden werden

Imperative Sprachen

Statische Bindung vs. Dynamische Bindung:

statisch:

- Bindung vor Laufzeit
- bleibt bis Programmende bestehen

dynamisch:

- Bindung erst während der Laufzeit
- kann sich während Programmausführung ändern

Imperative Sprachen

Statische Typ-Bindung:

- durch explizite Typ-Deklaration: `int a, b, c;`
- durch implizite Typ-Deklaration (in Sprachdesign festgelegt):
- Beispiel 1: Fortran

  Variable nicht explizit deklariert und

  Name startet mit i,I,j,J,…,n,N → Variable implizit von Typ Integer

  Name startet mit anderem Buchstaben → Typ Real

  `Implicit none` schaltet implizite Typbindung ab
- Beispiel 2: Perl

  Name beginnt mit $: Typ scalar

  Name beginnt mit @: Typ array => $nenner ≠ @nenner

  Name beginnt mit %: Hash-Struktur
- Typ-Inferenz (siehe ML)

Imperative Sprachen

Dynamische Typ-Bindung:

- keine Deklarations-Anweisung
- keine implizite Typ-Bindung (z.B. über Name)
- Bindung durch Zuweisungsstatement
- gebunden wird an Typ des Wertes der rechten Zuweisungsseite
- Vorteile: mehr Flexibilität beim Programmieren, generische Funktionen/Methoden einfach realisierbar
- Nachteile: keine Typfehler –Findung möglich, Programm weniger zuverlässig, Typ-Bindung und Typ-Überprüfung zur Laufzeit ist teuer
- Sprachen mit dynamischer Typ-Bindung haben i.d.R. Interpreter (nicht Compiler)

Imperative Sprachen

Typen:

- Sammlung berechenbarer Einheiten mit gleichen Eigenschaften
- Beispiele: Integer, Real, String, …

  komplexe Typen: Person, Student, Konto,…

  Lisp-Liste, Lisp-Atom
- Vorteile:
  - verständlichere Programme
  - Typ checking
  - Typinformationen zur Optimierung durch Compiler/Interpreter nutzbar

Imperative Sprachen

Typ checking:

- Typfehler: Benutzung einer Funktion/eines Wertes inkonsistent zum Typ
- Beispiele:
  - Operation auf Operanden vom falschen Typ: int-add(3,2.5)
  - Integerwert als Funktion aufrufen: int x = 256; … x(..);

- bei Typüberlappung manchmal automatische Konvertierung (→ siehe C)
- Wunsch/Ziel: viele Typfehler zur Übersetzungszeit finden

Imperative Sprachen

Typ-sicher:

- kein Programm einer PS darf Typunterscheidung verletzen
- manche Typunterscheidungen in (fast) allen PS Pflicht:
  - Funktion ist anderer Typ als Standardtypen (int, float, char,…)
  - Speicherzugriff auf nicht allokierten Speicher verboten
- Sprachen, die dies erlauben: typ-unsicher

Sicherheitsgrad	Sprache	Grund
typ-unsicher	C, C++	Pointer-Arithmetik
Ziemlich sicher	Pascal	Expliz. Deallokierung, dangling pointer
typ-sicher	Lisp, SMALLTALK, Java	komplettes Typ-checking

Imperative Sprachen

Typ-Überprüfung

- zur Übersetzungszeit:
  - häufigste Form in PS
  - Vorteile: typkorrekte Software wird ausgeliefert, Code kann effizienter sein, Programm zur Laufzeit schneller
  - konservative Form: Warnung vor Typfehlern, die zur Laufzeit evtl. keine sind
- zur Laufzeit:
  - vor jeder einzelnen Anweisung wird Typkorrektheit geprüft
  - fängt alle Typfehler ab
  - Programmlaufzeit länger
- oftmals Kombination von beidem (Java)