GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Informatik II

SS 2005

Kapitel 7: Compilerbau

Teil 2: Konzepte von Programmiersprachen

Dr. Michael Ebner
Dipl.-Inf. René Soltwisch

Lehrstuhl für Telematik
Institut für Informatik

Überblick

- **Einführung Programmiersprachen**
- Namen, Bindungen und Gültigkeitsbereiche
- Speichermanagement und Implementierung
- Kontrollfluss

- Als Grundlage dient das Buch "*Programming Language Pragmatics*" von Michael L. Smith
  - Siehe u.a. Kapitel 3, 6 und 9

Abstraktionen...

- Eliminiere Details welche unnötig zum Lösen eines speziellen Problems sind
  - Komplexität wird versteckt
- Baue oft auf anderen auf
  - Erlaubt das Lösen von zunehmend komplexeren Problemen (teile und herrsche, divide and conquer)
- Komplexität moderner Software ist ohne Beispiel (Präzedenzfall)
  - Abstraktion ist ein grundlegender Bestandteil zum Handhaben von diesen komplexen Problemen

Abstraktion	**Abstraktum**
Digitale Logik	Transistoren
Computerarchitektur	Digitale Logik
Assemblersprache	Maschinensprache
Betriebssystem	Allokation von Ressourcen (Zeit, Speicher, etc.)
Computerkommunikation	(Physikalische) Netzwerke, Protokolle

Sprachen als Abstraktion

- Die menschliche Sprache ist ein Werkzeug für die Abstraktion von Gedanken
  - „Wenn es mir warm ist, dann schalte ich den Ventilator ein."
    - Eine einfache Absicht wird mitgeteilt, wobei aber die kognitiven und neurologischen Bedingungen, durch welche die Absicht aufkam, höchst wahrscheinlich für jeden zu komplex sind um sie zu Verstehen
    - Die Bedeutung dieser Aussage ist dem Verständnis des Individuums welches es äußert und den Individuen die es hören überlassen
- Programmiersprachen sind ein Werkzeug zum Abstrahieren von Berechnungen
  - if (temperatur() > 30.0) { schalte_ventilator_ein(); }
    - Beinhaltet eine komplexe aber konkrete Sequenz von Aktionen:
      - lese Thermostat; konvertiere den Ablesewert zu einer IEEE Fliesskomazahl nach der Celsiusskala; Vergleiche den Wert mit 30.0; wenn größer dann sende ein Signal an eine PCI Karte, welche ein Signal an ein Relais sendet, welches den Ventilator einschaltet
    - Die Bedeutung dieses Ausdrucks ist festgelegt durch die **formale Semantik** der Programmiersprache und der Implementierung der Funktionen temperatur() und schalte_ventilator_ein().

Wie abstrahieren Programmiersprachen Berechnungen? (1/4)

1. Biete eine Notation für den Ausdruck von Algorithmen welche
   - (meistens) unabhängig von der Maschine ist auf welcher der Algorithmus ausgeführt wird,
   - Fähigkeiten (features) auf höchster Ebene bietet und die Aufmerksamkeit des Programmierers mehr auf den Algorithmus fokussiert und weniger auf
     - wie der Algorithmus in einer Maschinensprache implementiert wird,
     - wie Hilfsalgorithmen, z.B. Hash-Tabellen, Listen, implementiert werden,
   - es dem Programmierer erlaubt seine eigene Abstraktion (Unterprogramme, Module, Bibliotheken, Klassen, etc.) zu bauen um die Weiterführung des Konzepts „Komplexitätsmanagement durch Schichtenbildung“ zu ermöglichen.

Wie abstrahieren Programmiersprachen Berechnungen? (2/4)

2. Verberge unterliegende (systemnahe) Details der Zielarchitektur
   - Befehlsnamen der Assemblersprache, Registernamen, Argumentordnung, etc.
   - Abbildung von Sprachelementen auf die Assemblersprache
     - Arithmetische Ausdrücke,
     - Bedingungen,
     - Konventionen für Prozeduraufrufe, etc.

```
if (a < b + 10) {
   do_1();
} else {
   do_2();
}
```

SPARC

```
      add   %l1,10,%l2
      cmp   %l0,%l2
      bge   .L1; nop
      call  do_1; nop
      ba    .L2; nop
.L1:  call  do_2; nop
.L2:  …
```

MIPS

```
      addi  $t2,$t1,10
      bge   $t0,$t2,L1
      call  do_1
      b     L2
L1:   call  do_2
L2:   …
```


Wie abstrahieren Programmiersprachen Berechnungen? (3/4)

3. Biete Grundbefehle (primitives), Unterprogramme und Laufzeitunterstützung für übliche (lästige) Programmierpflichten
   - Lesen und schreiben von Dateien
   - Handhabung von Zeichenfolgen (Vergleiche, Erkennung von Teilzeichenfolge, etc.)
   - Dynamische Allokation von Speicher (*new*, *malloc*, etc.)
   - Rückgewinnung von unbenutztem Speicher (garbage collection)
   - Sortieren
   - etc.

Wie abstrahieren Programmiersprachen Berechnungen? (4/4)

- Biete Merkmale welche eine besondere Art von Algorithmus oder Softwareentwicklung unterstützen (encourage) oder durchsetzen (enforce)

<u>Strukturiertes Programmieren</u>
- Unterprogramme
- Verschachtelte (Nested) Variablenbereiche (scopes)
- Schleifen
- Beschränkte Formen des „goto“ Befehls (statement??)

<u>Objekt-Orientierte Programmierung</u>
- Klassen
- Vererbung
- Polymorphismus

Kategorien von Programmiersprachen

Alle Sprachen fallen in eine der beiden folgenden Kategorien:

- **Imperative Sprachen** *erfordern* die *schrittweise* Beschreibung durch Programmierer *wie* ein Algorithmus seine Aufgabe erledigen soll.
  - Analogie aus der realen Welt: „Ein Rezept ist eine Art von imperativem Programm, welches einem Koch sagt wie ein Gericht zuzubereiten ist.“
- **Deklarative Sprachen** *erlauben* die Beschreibung durch Programmierer *was* ein Algorithmus erledigen soll *ohne* exakt zu beschreiben wie es getan werden soll.
  - Analogie aus der realen Welt: „Das Pfandgesetz ist ein deklaratives Programm welches Einzelhändlern mitteilt das sie ein Recyclingprogramm für Einwegflaschen und Dosen des eigenen Sortiments aufstellen müssen, ohne exakt mitzuteilen wie dies zu erfolgen hat.

Imperative Sprachen (1/2)

- Die von Neumann Sprachen
  - schließen Fortran, Pascal, Basic und C ein
  - stellen eine Reflektion der von Neumann Computerarchitektur dar, auf welcher die Programme laufen
  - Führen **Befehle** aus welche den **Zustand** des Programms (Variablen/Speicher) ändern
    - Manchmal auch Berechnung durch **Seiteneffekte** genannt
  - Beispiel: Aufsummieren der ersten n Ganzzahlen in C
    - `for(sum=0,i=1;i<=n;i++) { sum += i; }`

Imperative Sprachen (2/2)

- Die objekt-orientierten Sprachen
  - schließen Smalltalk, Eiffel, C++, Java und Sather ein
  - Sind ähnlich der von Neumann Sprachen mit der Erweiterung von **Objekten**
  - Objekte
    - enthalten ihren eigenen internen Zustand (**Klassenvariablen**, **member variables**) und Funktionen welche auf diesem Zustand operieren (**Methoden**)
    - Berechnung ist organisiert als Interaktion zwischen Objekten (ein Objekt ruft die Methoden eines anderen Objektes auf)
  - Die meisten objekt-orientierten Sprachen bieten Konstrukte (facilities) basierend auf Objekten welche **objekt-orientierte Programmierung** fördern
    - **Kapselung** (encapsulation), **Vererbung** (inheritance) und **Polymorphismus** (polymorphism)
    - Wir werden uns darüber später genauer unterhalten

Deklarative Sprachen (1/2)

- Die funktionalen Sprachen
  - schließen Lisp/Scheme, ML, Haskell (Gofer) ein
  - Sind eine Reflektion von Church‘s Theorie der rekursiven Funktionen (**lambda calculus**)
  - Berechnung werden ausgeführt als Rückgabewerte von Funktionen basierend auf der (möglicherweise rekursiven) evaluation von anderen Funktionen
    - Mechanismus ist als **Reduktion** bekannt
  - Keine Seiteneffekte
    - Erlaubt gleichungsbasiertes Problemlösen (equational reasoning), einfachere formale Beweise von Programmkorrektheit, etc.
  - Beispiel: Aufsummieren der ersten n Ganzzahlen in SML
    - `fun sum (n) = if n <= 1 then n else n + sum(n-1)`

Deklarative Sprachen (2/2)

- Die logischen Sprachen
  - schließen Prolog, SQL und Microsoft Excel/OpenOffice OpenCalc ein
  - Sind eine Reflektion von der Theorie der **Aussagenlogik** (**propositional logic)**
  - Berechnung ist ein Versuch einen Wert zu finden welcher eine Menge von logischen Beziehungen erfüllt
    - Der meist verwendete Mechanismus um diesen Wert zu finden ist bekannt als **Resolution** (resolution) und **Vereinheitlichung** (unification)
  - Beispiel: Aufsummieren der ersten n Ganzzahlen in Prolog
    - `sum(1,1).`
    - `sum(N,S) :-  N1 is N-1, sum(N1,S1), S is S1+N.`

Name	*Seit*	*Bemerkungen*	*Verbreitung*
FORTRAN	1957	Für mathematische und naturwissenschaftliche Anwendungen	Sehr groß
COBOL	1960	Für betriebswirtschaftliche Anwendungen	Sehr groß
ALGOL-60	1960	Für mathematische Berechnungen	Klein
LISP	1962	Wichtigste Sprache der Künstlichen Intelligenz (KI)	Mittel
BASIC	1963	Einfache Anfängersprache	Groß
PL/1	1965	Für technische und betriebswirtschaftliche Anwendungen, sehr umfangreich	Mittel
ALGOL-68	1968	Für mathematische und allgemeine Anwendungen; Einsatz fast nur im Hochschulbereich	Klein
PASCAL	1971	Sprache der strukturierten Programmierung; Grundlage für viele andere Sprachen	Mittel
C	1973	Für Systemprogrammierung und allgemeine Anwendungen; enger Bezug zu Unix	Sehr groß
SMALLTALK	1974	Erste objektorientierte Sprache	Klein
PROLOG	1977	KI-Sprache, logisches Schließen	Klein
MODULA	1978	Weiterentwicklung von PASCAL	Klein
ADA	1980	Sehr umfangreich; für allgemeine Anwendungen; basiert auf PASCAL und PL/1	Klein
C++	1982	Ergänzung von C um objektorientierte Sprachelemente, sog. Hybridsprache	Groß
Eiffel	1982	Rein objektorientierte Sprache; Einsatz fast nur im Hochschulbereich	Klein
Java	1994	Objektorientiert, plattformunabhängig, Internet-Programmierung	Klein

Eine historische Perspektive: Maschinensprachen

- Die ersten Maschinen wurden direkt in einer Maschinensprache oder Maschinencode programmiert
- Langweilig, aber Maschinenzeit war teurer als Programmiererzeit

```
27bdffd0 afbf0014 0c1002a8 00000000 0c1002a8 afa2001c 8fa4001c
00401825 10820008 0064082a 10200003 00000000 10000002 00832023
00641823 1483fffa 0064082a 0c1002b2 00000000 8fbf0014 27bd0020
03e00008 00001025
```

MIPS Maschinencode für ein Programm zum Berechnen des GGT von zwei Ganzzahlen

Eine historische Perspektive: Assemblersprachen (1/2)

- Programme wurden immer Komplexer
  - zu schwierig, zeitintensiv und teuer um Programme in Maschinencode zu schreiben
- Assemblersprachen wurden entwickelt
  - Für den Menschen lesbar ☺
  - Ursprünglich wurde eine eins-zu-eins Beziehung zwischen Instruktionen der Maschinensprache und Instruktionen der Assemblersprache bereitgestellt
  - Schließlich wurden „makro“ Einrichtungen hinzugefügt um Softwareentwicklung durch anbieten von primitiven Formen von Code-Wiederverwendung weiter zu beschleunigen
  - Der **Assembler** war das Programm welches ein Assemblerprogramm in Maschinencode übersetzte mit welchem die Maschine laufen konnte

Eine historische Perspektive: Assemblersprachen (2/2)

- Beispiel:

```
    addiu   sp,sp,-32
    sw      ra,20(sp)
    jal     getint
    nop
    jal     getint
    sw      v0,28(sp)
    lw      a0,28(sp)
    move    v1,v0
    beq     a0,v0,D
    slt     at,v1,a0
A:  beq     at,zero,B
    nop

            b       C
            subu    a0,a0,v1
        B:  subu    v1,v1,a0
        C:  bne     a0,v1,A
            slt     at,v1,a0
        D:  jal     putint
            nop
            lw      ra,20(sp)
            addiu   sp,sp,32
            jr      ra
            move    v0,zero
```

Assembler

```
27bdffd0 afbf0014 0c1002a8 00000000 0c1002a8 afa2001c 8fa4001c
00401825 10820008 0064082a 10200003 00000000 10000002 00832023
00641823 1483fff8 0064082a 0c1002b2 00000000 8fbf0014 27bd0020
03e00008 00001025
```


Eine historische Perspektive: höhere Sprachen (1/2)

- Programme wurden immer Komplexer
  - es war zu schwierig, zeitintensiv und teuer um Programme in Assemblersprache zu schreiben
  - es war zu schwierig von einer Maschine zu einer anderen zu wechseln, welche eine andere Assemblersprache hatte
- Es wurden höhere Programmiersprachen entwickelt
  - Mitte der 1950er wurde Fortran entworfen und implementiert
    - es erlaubte numerische Berechnungen in einer Form ähnlich von mathematischen Formeln auszudrücken
  - Der **Compiler** war das Programm welches ein höheres Quellprogramm in ein Assemblerprogramm oder Maschinenprogramm übersetzte.
    - Ursprünglich konnten gute Programmierer schnellere Assemblerprogramme schreiben als der Compiler
  - Andere höhere Programmiersprachen folgen Fortran in den späten 50er und frühen 60er
    - Lisp: erste funktionale Sprache, basierte auf der Theorie der rekursiven Funktionen
    - Algol: erste block-strukturierte Sprache

Eine historische Perspektive: höhere Sprachen (2/2)

- Beispiel:

```
int gcd (int i, int j) {
  while (i != j) {
    if (i > j)
      i = i - j;
    else
      j = j - i;
  }
  printf("%d\n",i);
}
```

Compiler

```
addiu   sp,sp,-32
sw      ra,20(sp)
jal     getint
nop
jal     getint
sw      v0,28(sp)
lw      a0,28(sp)
move    v1,v0
beq     a0,v0,D
slt     at,v1,a0
A:  beq     at,zero,B
    nop

        b       C
        subu    a0,a0,v1
    B:  subu    v1,v1,a0
    C:  bne     a0,v1,A
        slt     at,v1,a0
    D:  jal     putint
        nop
        lw      ra,20(sp)
        addiu   sp,sp,32
        jr      ra
        move    v0,zero
```


Ausführung von Programmen höherer Sprachen

- Kompilation
  - Programm wird in Assemblersprache oder direkt in Maschinensprache übersetzt
  - Kompilierte Programme können so erstellt werden, dass sie relativ schnell in der Ausführung sind
  - Fortran, C, C++
- Interpretation
  - Programm wird von einem anderem Programm gelesen und Elemente der Quellsprache werden einzeln ausgeführt
  - Ist langsamer als kompilierte Programme
  - Interpreter sind (normalerweise) einfacher zu implementieren als Compiler, sind flexibler und können exzellent Fehlersuche (debugging) und Diagnose unterstützen
  - Java, Pyhton, Perl, etc.

Überblick

- Einführung Programmiersprachen
- **Namen, Bindungen und Gültigkeitsbereiche**
- Speichermanagement und Implementierung
- Kontrollfluss

Namen, Bindungen und Gültigkeitsbereiche (1/3)

- Namen
  - Ein mnemonischer Zeichenname wird verwendet um irgendetwas anderes zu repräsentieren oder zu benennen (Mnemonic ist die Bezeichnung für Ausdrücke deren Bedeutung vorwiegend durch die verwendete Buchstabenfolge leicht behalten werden kann.)
    - Normalerweise Bezeichner (identifier)
  - Ist essentiell für Abstraktion
    - Erlaubt es Programmierern Werte zu bezeichnen damit die Notwendigkeit zur direkten Manipulation von Adressen, Registernamen, etc. vermieden wird
      - Beispiel: Es ist nicht notwendig zu wissen ob die Variable foo im Register $t0 oder an der Speicherstelle 10000 gespeichert wird
    - Erlaubt es Programmierern einen einfachen Namen für ein potenziell komplexes Programmstück stehen zu lassen
      - Beispiel: foo = a*a + b*b + c*c + d*d;
    - Beide Fälle verringern die konzeptuelle Komplexität

Namen, Bindungen und Gültigkeitsbereiche (2/3)

- Bindungen
  - Ist eine Assoziation zwischen zwei Dingen
    - Ein Variablenname zu einem Wert
    - Ein Variablenname zu einer spezifischen Speicherstelle
    - Ein Typ und seine Repräsentation oder Layout im Speicher
  - Die **Bindezeit** ist die Zeit zu der eine solche Assoziation gemacht wird

Namen, Bindungen und Gültigkeitsbereiche (3/3)

- Gültigkeitsbereiche (Scope)
  - Ist der Textbereich eines Programms in welchem eine Bindung aktiv ist
  - Java Beispiel:

```
public void foo (int a) {
    int b;
    while(a < n) {
        int c;
        c = a + a;
        b = a * c;
        a++;
    }
}
```

Global Scope
Method Scope
Block Scope

  - Verbessert die Abstraktion durch die Kontrolle über die Sichtbarkeit von Bindungen

Bindezeitpunkte (1/8)

- Wir unterscheiden 7 Zeitpunkte für die Entscheidung über eine Bindung
- Sprache
  - Entwurf
  - Implementierung (mit Compilern)
- Programm
  - Programmierung (Schreiben des Programms)
  - Kompilation
  - Linken
  - Laden
  - Ausführen

Bindezeitpunkte (2/8)

- Zeitpunkt des Sprachentwurfs
  - Entscheidungen welche vom Designer der Programmiersprache gemacht wurden
  - Typische Beispiele:
    - Binden von Kontrollstrukturen (Bedingungen, Schleifen, etc.) zu Ihrer abstrakten Bedeutung
      - „Befehle in einem while Schleifenblock werden ausgeführt bis die Bedingung nicht mehr länger wahr ist“
    - Binden von primitiven Datentypnamen (int, float, char, etc.) zu Ihrer geforderten Repräsentation
      - „Variablen vom Typ int beinhalten vorzeichenbehaftete Ganzzahlen“
      - „Variablen vom Typ int beinhalten vorzeichenbehaftete 32-bit Werte“

Bindezeitpunkte (3/8)

- Zeitpunkt der Sprachimplementierung
  - Entscheidungen welche vom Compiler bzw. vom Compilerprogrammierer gemacht wurden
    - Mit anderen Worten, Angelegenheiten der Sprachimplementierung die nicht spezifisch während des Sprachentwurfs definiert wurden
  - Typische Beispiele:
    - Binden von primitiven Datentypen zu deren Repräsentationsgenauigkeit (Anzahl der Bits)
      - „bytes sind 8-bits, shorts sind 16-bits, ints sind 32-bits, longs sind 64-bits“
    - Binden von Dateioperationen zur betriebssystemspezifischen Implementierung dieser Operationen
      - open() ist mit einem SYS_open Systemaufruf implementiert

Bindezeitpunkte (4/8)

- Zeitpunkt des Programmierens
  - Entscheidungen des Programmierers
  - Typische Beispiele:
    - Binden eines Algorithmus zu den Befehlen der Programmiersprache mit denen der Algorithmus implementiert ist
    - Binden von Namen zu Variablen, welche für einen Algorithmus erforderlich sind
    - Binden von Datenstrukturen zu Sprachdatentypen
- Zeitpunkt der Kompilation
  - Entscheidungen des Compilers
  - Typische Beispiele:
    - Binden von höheren Konstrukten zu Maschinencode (Optimierung eingeschlossen)
    - Binden von statisch definierten Datenstrukturen zu einem spezifischen Speicherlayout
    - Binden von benutzerdefinierten Datentypen zu einem spezifischen Speicherlayout

Bindezeitpunkte (5/8)

- Zeitpunkt des Verbindens (linken)
  - Entscheidungen des Linkers
    - Linker binden Programmmodule zusammen
  - Typische Beispiele:
    - Binden von Objekten (Unterprogramme und Daten) zu einem spezifischen Platz in einer ausführbaren Datei
    - Binden von Namen, welche Objekte in anderen Modulen referenzieren, zu deren tatsächlichen Ortsreferenz
- Zeitpunkt des Ladens
  - Entscheidungen des Laders
    - Lader holen ausführbare Dateien in den Speicher
  - Typische Beispiele:
    - Binden von virtuellen Adressen in der ausführbaren Datei zu den physikalischen Speicheradressen
      - In modernen Betriebssystemen nicht mehr wirklich notwendig, da das Betriebssystem virtuelle Adresse zu physikalischen Adressen bindet in dem es virtuellen Speicher (einschließlich der notwendiger Hardware) verwendet

Bindezeitpunkte (6/8)

- Zeitpunkt der Programmausführung
  - Entscheidungen welche während der Programmausführung gemacht werden
  - Typische Beispiele:
    - Binden von konkreten Werten zu Programmvariablen
      - „a = a + 1;“
    - Binden von Referenzen von dynamisch zugeteilten Objekten zu Speicheradressen
    - Binden von Namen zu Objekten in Sprachen mit dynamischen Gültigkeitsbereichen

Bindezeitpunkte (7/8)

- **Ähnliche** Bindeentscheidungen können zu **mehreren** Bindezeitpunkten durchgeführt werden
  - **Linkentscheidungen** können zur Linkzeit, Ladezeit (ein Typ des dynamischen Linken) oder Laufzeit (ein anderer Typ des dynamischen Linkens) auftreten
  - **Optimierungsentscheidungen** können zur Compilezeit, Linkzeit, Ladezeit und sogar zur Laufzeit (dynamische Optimierung) auftreten
- Bindeentscheidungen können zu **verschiedenen** Bindezeitpunkten in verschiedenen Sprachen getroffen werden
  - C bindet Variablennamen an die referenzierten Objekte zur Compilezeit
    - Wenn wir in C sagen „foo=bar;“, dann wissen wir genau ob foo und bar global oder lokal sind oder nicht, von welchem Typ sie sind, ob ihre Typen für Zuweisungen kompatibel sind oder nicht, etc.
  - Perl (und gilt eigentlich für alle interpretierten Sprachen) bindet Variablennamen an die referenzierten Objekte zur Laufzeit
    - Wir können in Perl sagen „$foo=$bar;“, und wenn der Name $bar nicht schon an ein Objekt gebunden ist, dann wird eines erzeugt, und dann zu $foo zugewiesen

Bindezeitpunkte (8/8)

- **Statisches** Binden (static binding)
  - Bezieht sich auf alle Bindeentscheidungen die **vor** der Laufzeit gemacht werden
- **Dynamisches** Binden (dynamic binding)
  - Bezieht sich auf alle Bindeentscheidungen die **zur** Laufzeit gemacht werden
- **Frühe** Bindezeitpunkte
  - Verbunden mit größerer Effizienz
    - Kompilierte Sprachen laufen typischerweise viel schneller, weil die meisten Bindeentscheidungen zur Compilezeit getroffen wurden
    - Ist eine Bindeentscheidung aber erst einmal getroffen worden, dann verlieren wir auch einiges an Flexibilität
- **Späte** Bindezeitpunkte
  - Verbunden mit größerer Flexibilität
    - Interpretierte Sprachen erlauben es die meisten Bindeentscheidungen zur Laufzeit zu treffen, was eine größere Flexibilität erlaubt
      - Zum Beispiel, die meisten interpretierten Sprachen erlauben es einem Programm Fragmente eines anderen Programms dynamisch zu generieren und auszuführen
    - Da Bindeentscheidungen zur Laufzeit getroffen werden, können interpretierte Sprachen langsam sein

Zusammenfassung Namen und Bindungen

- Namen
  - Ein mnemonischer Zeichenname wird verwendet um irgendetwas anderes zu repräsentieren oder zu benennen
    - Beispiel: Variable foo kann die Speicherstelle 10000 referenzieren
- Bindungen
  - Ist eine Assoziation zwischen zwei Dingen
    - Ein Name und das was er referenziert
- Bindezeit
  - Die **Bindezeit** ist die Zeit zu der die Entscheidung über eine solche Assoziation gemacht wird
    - Beispiel: Ist der Wert von foo in einem Register oder im Speicher? (Entscheidung wird zur Compilezeit gemacht)

Gültigkeitsbereiche

- Textueller Bereich eines Programms in welchem eine Bindung aktiv ist
- Es gibt grundsätzlich zwei Varianten:
  - Statische Gültigkeitsbereiche (static scopes)
    - Es kann zur Compilezeit genau festgestellt werden welcher Name welches Objekt an welchen Punkten im Programm referenziert
  - Dynamische Gültigkeitsbereiche (dynamic scopes)
    - Bindungen zwischen Namen und Objekten hängen vom Programmfluss zur Laufzeit ab
- Nähere Ausführung
  - Der Prozess durch den eine Menge von Bindungen aktiv wird, wenn die Kontrolle in einen Gültigkeitsbereich eintritt
    - Zum Beispiel die Allokation von Speicher um Objekte darin zu halten

Gültigkeitsbereiche

- Referenzierende Umgebung (referencing environment)
  - Die Menge von aktiven Bindungen zu einem gegebenen Zeitpunkt in der Programmausführung
  - Wird durch die **Regeln für Gültigkeitsbereiche** einer Programmiersprache festgelegt
- **Statische** Gültigkeitsbereiche
  - Bindungen zwischen Namen und Objekten könne zur Compilezeit festgestellt werden
  - Einfache Varianten
    - Frühe Versionen von BASIC hatten einen, globalen Gültigkeitsbereich
  - Komplexe Varianten
    - Moderne Programmiersprachen erlauben verschachtelte Unterprogramme und Module weshalb kompliziertere Regeln für Gültigkeitsbereiche erforderlich sind

Static scope: Verschachtelte Unterprogramme

- Frage:
  - Welches Objekt wird von X im Funktionsblock von F1 referenziert?
- Regel über den nächsten Gültigkeitsbereich (closest nested scope)
  - Referenzen von Variablen referenzieren das Objekt im **naheliegendsten** Gültigkeitsbereich

```
procedure P1 (A1 : T1);
var X : real;
    ...
    procedure P2 (A2 : T2);
        ...
        procedure P3 (A3 : T3);
        ...
        begin
            ...     (* body of P3 *)
        end;
        ...
    begin
        ...         (* body of P2 *)
    end;
    ...
    procedure P4 (A4 : T4);
        function F1 (A5 : T5) : T6;
        var X : integer;
        ...
        begin
            ...     (* body of F1 *)
        end;
        ...
    begin
        ...         (* body of P4 *)
    end;
    ...
begin
    ...             (* body of P1 *)
end
```

Ein Problem welches nicht von verschachtelten Unterprogrammen behandelt wird (1/2)

- Geheimnisprinzip (information hiding) für komplexe abstrakte Datentypen (ADT)
- Für einfache ADTs könnten Funktionen mit statischen lokalen Variablen funktionieren

- Siehe Beispiel auf nächster Folie:
  - Die Variable `name_nums` behält seinen Wert über Aufrufe von `gen_new_name` bei
  - Dies ist zu einfach für ADTs mit mehreren Funktionen die sich einen globalen Zustand teilen müssen

Ein Problem welches nicht von verschachtelten Unterprogrammen behandelt wird (2/2)

```
/*
    Place into *s a new name beginning with the letter l and continuing
    with the ascii representation of an integer guaranteed to be distinct
    in each separate call.  s is assumed to point to space large enough
    to hold any such name; for the short ints used here, seven characters
    suffice.  l is assumed to be an upper or lower-case letter.
    sprintf 'prints' formatted output to a string.
*/
void gen_new_name (char *s, char l) {
    static short int name_nums[52];
        /* C guarantees that static local variables are initialized
        to zeros */
    int index = (l >= 'a' && l <= 'z') ? l-'a' : 26 + l-'A';
    name_nums[index]++;
    sprintf (s, "%c%d\0", l, name_nums[index]);
}
```


Module

- Ein Modul erlaubt es eine Sammlung von Objekten zu kapseln, so dass
  - Objekte innerhalb eines Moduls sich **gegenseitig** sehen können
  - Objekte innerhalb des Moduls nach außen nicht sichtbar sind, es sei denn sie werden explizit **exportiert**
  - Objekte von außerhalb nach innen nicht sichtbar sind, es sei denn sie werden explizit **importiert**

- Module mit geschlossenem Gültigkeitsbereich
  - Betrifft Module in welche Namen explizit importiert werden müssen um innerhalb des Moduls sichtbar zu sein
- Module mit offenem Gültigkeitsbereich
  - Betrifft Module für die es nicht explizit erforderlich ist Namen von außerhalb zu importieren um sichtbar zu sein

Ein Modul Beispiel

- Abstraktion eines Stacks in Modula
- Wir exportieren die push und pop Funktionen
- Außerhalb des Moduls nicht sichtbar
  - Top des Stackzeigers „top“
  - Stack array „s“

```
CONST stack_size = ...
TYPE element = ...
...
MODULE stack;
IMPORT element, stack_size;
EXPORT push, pop;
TYPE
    stack_index = [1..stack_size];
VAR
    s   : ARRAY stack_index OF element;
    top : stack_index;        (* first unused slot *)

PROCEDURE error; ...

PROCEDURE push (elem : element);
BEGIN
    IF top = stack_size THEN
        error;
    ELSE
        s[top] := elem;
        top := top + 1;
    END;
END push;

PROCEDURE pop () : element;   (* A Modula-2 function is just a *)
BEGIN                         (* procedure with a return type. *)
    IF top = 1 THEN
        error;
    ELSE
        top := top - 1;
        RETURN s[top];
    END;
END pop;

BEGIN
    top := 1;
END stack;
```

```
VAR x, y : element;
...
push (x);
...
y := pop;
```

Dynamische Gültigkeitsbereiche

- Bindungen zwischen Namen und Objekten hängen vom Programmfluss zur Laufzeit ab
  - Reihenfolge in welcher Unterprogramme aufgerufen werden ist wichtig
- Die Regeln für dynamische Gültigkeitsbereiche sind normalerweise einfach
  - Die aktuelle Bindung zwischen Name und Objekt ist diejenige die während der Ausführung als letzte angetroffen wurde (sprich diejenige die am **kürzlichsten** gesetzt wurde)

Statische vs. dynamische Gültigkeitsbereiche

- Statischer Gültigkeitsbereich
  - Programm gibt 1 aus
- Dynamischer Gültigkeitsbereich
  - Programm gibt 1 oder 2 aus in Abhängigkeit des gelesenen Wertes in Zeile 8
- Warum?
  - Ist die Zuweisung zu „a“ in Zeile 3 eine Zuweisung zu dem globalen „a“ von Zeile 1 oder dem Lokalen „a“ von Zeile 5?

```
1:  a : integer        -- global declaration

2:  procedure first
3:      a := 1

4:  procedure second
5:      a : integer    -- local declaration
6:      first ()

7:  a := 2
8:  if read_integer () > 0
9:      second ()
10: else
11:     first ()
12: write_integer (a)
```


Vorteile von dynamischen Gültigkeitsbereichen

- Was sind die Vorteile von dynamischen Gültigkeitsbereichen?
  - Es erleichtert die Anpassung von Unterprogrammen
  - Beispiel:

```
begin --nested block
    print_base: integer := 16
    print_integer(n)
```

- Die Variable `print_base` kontrolliert die Basis welche `print_integer` für die Ausgabe von Zahlen verwendet
- `print_integer` kann früh in einem globalen Gültigkeitsbereich mit einem **Standardwert** belegt werden und kann **temporär** in einem globalen Gültigkeitsbereich auf eine andere Basis überschrieben werden

Ein Problem von dynamischen Gültigkeitsbereichen

- Problem: unvorhersagbare referenzierende Umgebungen

```
max_score : integer        -- maximum possible score

function scaled_score (raw_score : integer) : real
    return raw_score / max_score * 100
...
procedure foo
    max_score : real := 0      -- highest percentage seen so far
    ...
    foreach student in class
        student.percent := scaled_score (student.points)
        if student.percent > max_score
            max_score := student.percent
```

- Globale Variable `max_score` wird verwendet von `scaled_score()`
- `max_score` wird umdefiniert in `foo()`
- `scaled_score()` wird von `foo()` aufgerufen
- Ahhhhh

Vermischte Gültigkeitsbereiche (mixed scoping)

- Perl unterstützt beide Arten, dynamische wie statische Gültigkeitsbereiche

Dynamic Scoping

```
$i = 1;
sub f {
  local($i) = 2;
  return g();
}
sub g { return $i; }
print g(), f();
```

```
Ausgabe:
1 2
```

Static Scoping

```
$i = 1;
sub f {
  my($i) = 2;
  return g();
}
sub g { return $i; }
print g(), f();
```

```
Ausgabe:
1 1
```


Zusammenfassung für Gültigkeitsbereiche

- Statische Gültigkeitsbereiche
  - Wird von den meisten, kompilierten Hochsprachen verwendet
    - C, C++, Java, Modula, etc.
  - Bindungen von Namen zu Variablen können zur Compilezeit festgestellt werden
  - Effizient
- Dynamische Gültigkeitsbereiche
  - Wird von vielen interpretierten Sprachen verwendet
    - Ursprüngliches LISP, APL, Snobol und Perl
  - Bindungen von Namen zu Variablen können eine Feststellung zur Laufzeit benötigen
  - Flexibel

Überblick

- Einführung Programmiersprachen
- Namen, Gültigkeitsbereiche und Bindungen
- **Speichermanagement und Implementierung**
- Kontrollfluss

Speichermanagement und Implementierung

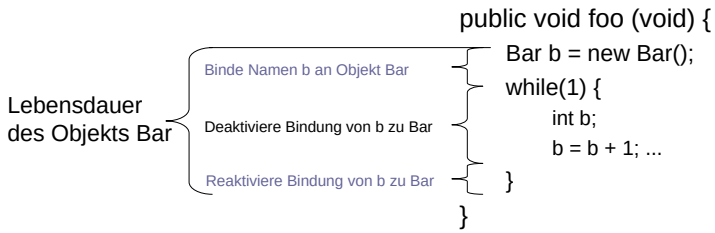
- Lebensdauer von Objekten
- Speichermanagement
- Weiterführende Spracheigenschaften und Bindungen
  - Implementierung von statischen Gültigkeitsbereichen für verschachtelte Unterprogramme
  - Implementierung von Unterprogrammreferenzen

Objektlebensdauer (1/3)

- Schlüsselereignisse während der Lebensdauer eines Objektes
  - Objekt wird erzeugt
  - Bindungen zum Objekt werden erzeugt
  - Referenzen zu den Variablen, Unterprogrammen und Typen werden durch Bindungen gemacht
  - Deaktivierung und Reaktivierung von temporär nicht verwendbaren Bindungen
  - Vernichtung von Bindungen
  - Vernichtung des Objekts
- Lebensdauer von Bindungen: Zeit zwischen Erzeugung und Vernichtung einer Bindung
  - Beispiel: Zeit während der eine Java Referenz ein Objekt im Speicher referenziert
- Lebensdauer von Objekten: Zeit zwischen Erzeugung und Vernichtung eines Objekts
  - Beispiel: Zeit während der ein Objekt im Speicher „lebt“

Objektlebensdauer (2/3)

- Beispiel:



- Anmerkung: Das durch foo() erzeugte Objekt Bar ist nicht unbedingt nach der Rückkehr aus dem Unterprogramm zerstört. Es kann nach der Rückkehr von foo() nicht länger referenziert werden, aber es wird wahrscheinlich erst bei der „garbage collection“ zu einem späteren Zeitpunkt in der Programmausführung zerstört werden.

Objektlebensdauer (3/3)

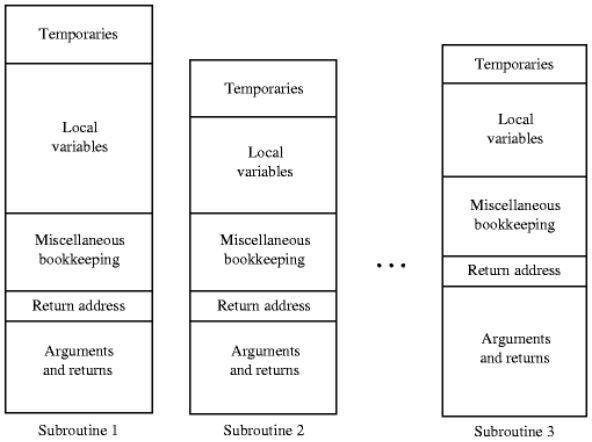
- Die Lebenszeit eines Objekts korrespondiert mit einem von drei Hauptmechanismen der Speicherallokation
  - Statische Allokation
    - Objekte werden zur Compilezeit oder Laufzeit zu festen Speicherplätzen allokiert
  - Stack Allokation
    - Objekte werden zur Laufzeit wie benötigt allokiert, wobei eine last-in, first-out Ordnung gilt
      - Beispiel: locales, wie z.B. lokale Variablen
  - Heap Allokation
    - Objekte werden zur Laufzeit in einer beliebigen Reihenfolge allokiert und freigegeben
      - Beispiel: dynamisch allokierte Objekte

Statische Allokation

- Was wird statisch allokiert?
  - Globale Variablen
    - Beispiel: Statische Klassenvariablen
  - Konstante Variablen welche während der Programmausführung sich nicht ändern sollten
    - Beispiel: “i=%d\n” ist konstant in printf(“i=%d\n”,i);
  - Der Programmcode (Unterprogramme, etc.)
    - Ausnahme: dynamisch verbundene Unterprogramme
  - Vom Compiler produzierte Informationen zum Debuggen
  - Falls eine Sprache rekursive Unterprogrammaufrufe nicht untersützt, dann auch lokale Variablen, Unterprogrammargumente, Rückgabewerte, vom Compiler generierte temporäre Daten, etc.

Beispiel: Statische Allokation von Lokalem

- Keine Rekursion bedeutet das es zu einer Zeit nur eine aktive Instanz bzw. **Aktivierung** einer Instanz von jedem gegebenen Unterprogramm geben kann
- Daher können wir für eine einzelne, mögliche Aktivierung für jedes Unterprogramm eines Programms den Speicher statisch reservieren



Allokation über einen Stack (1/2)

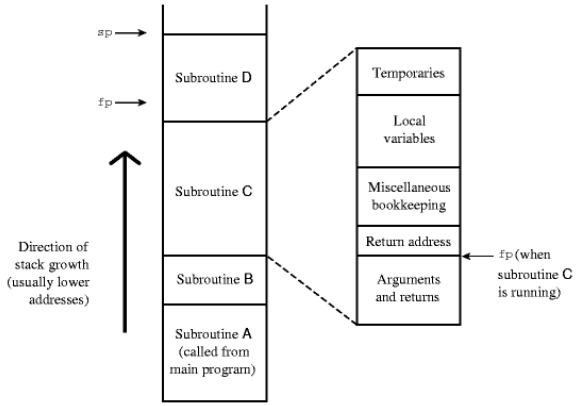
- Falls eine Sprache Rekursion erlaubt, dann kann jedes Unterprogramm mehrere simultane Aktivierungen haben
- Beispiel:

```
public int foo (int n) {
    int a, b, c;
    a = random(); b = random();
    if (n > 0) { c = foo(n-1); }
    c = c * (a + b);
}
```

- Wir können **n** Aktivierungen von foo() haben und jede benötigt Speicherplatz um die eigenen Kopien von a, b, c, Übergabeargument n und jeden temporären, vom Compiler generierten Wert zu speichern

Allokation über einen Stack (2/2)

- Die Lösung für das Rekursionsproblem ist die **Allokation** über einen **Stack** (**stack allocation**)
- „Push“ Stack um Platz für Lokales (locals) für Unterprogrammaufrufe zu reservieren
- „Pop“ Stack um Platz für Lokales (locals) bei der Rückkehr von einem Unterprogramm wieder freizugeben

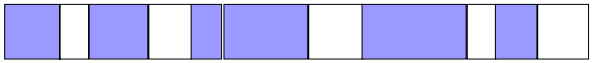


Allokation über einen Heap

- Wir wissen wie Code für Globales, Lokales, Konstanten, Temporäres, etc. allokiert wird
- Was bleibt übrig:
  - Dynamisch allokierte Objekte
  - Warum können diese nicht statisch allokiert werden?
    - Weil sie dynamisch erzeugt werden
  - Warum können diese nicht auf einem Stack allokiert werden?
    - Ein Objekt, welches dynamisch von einem Unterprogramm erzeugt wurde, könnte die Aktivierung des Unterprogramms überleben
      - Beispiel: Objekt wird einem Globalen zugewiesen oder von dem Unterprogramm zurückgegeben
- Heaps lösen dieses Problem
  - Ein Heap ist eine Speicherregion in welcher Speicherblöcke jederzeit willkürlich allokiert und wieder freigegeben werden können
- Wie sind Heaps implementiert?
  - Wie handhaben wir Anfragen zur Allokation und Freigabe?

Heap-Management (1/4)

Heap



Allokationsanforderung 

- Wir bekommen eine Allokationsanforderung für n Bytes vom Speicher
- Der Heap hat verwendete (dunkle) und freie (helle) Bereiche
- Wie wählen wir einen freien Bereich um eine Allokationsanforderung zu befriedigen?
- Einfache Antwort:
  - Finde den ersten freien Bereich welcher groß genug ist der Allokation zu entsprechen (wird **first fit** genannt)
  - Problem: interne Fragmentierung
    - Wenn wir n Bytes anfordern und der erste, verfügbare freie bereich hat n+k Bytes, dann verschwenden wir k Bytes durch die Allokation im ersten Bereich

Heap-Management (2/4)

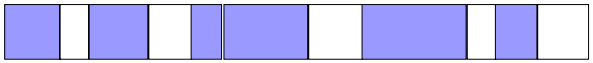
Heap

Allokationsanforderung 

- Eine bessere Antwort (vielleicht):
  - Finde den ersten Bereich, welcher von der Größe am nächsten zur Allokationsanforderung ist (wird **best fit** genannt)
  - Problem: Zeitaufwendiger
    - Muss alle freie Blöcke finden um den Besten zu finden
    - Es kann immer noch interne Fragmentierung geben, aber hoffentlich weniger als bei first fit

Heap-Management (3/4)

Heap



Allokationsanforderung 

- Ein anderes Problem: externe Fragmentierung
  - Wir bekommen eine Allokationsanforderung für n Bytes und der Heap hat mehr als n Bytes frei, aber
    - **kein einzelner freier Bereich ist n Bytes groß oder größer**
  - Wir haben genügend freien Speicher, aber wir können die Allokationsanforderung nicht befriedigen
- Mögliche Lösungen:
  - Vereinigung von Bereichen: wenn zwei benachbarte Bereiche mit j und k Bytes frei sind, dann vereinige diese zu einem einzelnen, freien Bereich mit j+k Bytes
  - Ist eine Verbesserung, aber kann nicht alle externen Fragmentierungen eliminieren

Heap-Management (4/4)

Heap

Allokationsanforderung 

- Wie handhaben wir die Freigabe?
  - Explizit:
    - Der Programmierer muss dem Heap-Manager mitteilen dass ein Bereich nicht länger vom Programm verwendet wird
      - Beispiel: verwende in C free(p) um den Bereich freizugeben auf den p zeigt
  - Automatisch:
    - Das Laufzeitsystem bestimmt welche Objekte auf dem Heap lebendig (sprich immer noch an Namen gebunden sind) oder tot (sprich nicht länger zu irgendeinem Namen gebunden sind) sind und gibt die toten Objekte automatisch frei
    - Wird **Garbage Collection** (Speicherbereinigung) genannt

Speicherbereinigung (Garbage Collection) (1/2)

- Warum verwenden wir Speicherbereinigung?
  - Verhindert:
    - **Speicherlöcher (memory leak):** Programmierer können die Freigabe von dynamisch allokiertem Speicher vergessen
    - **Dangling pointers:** Programmierer können ein Objekt freigeben bevor alle Referenzen darauf zerstört sind
  - Reduzieren den Programmieraufwand und resultiert in zuverlässigeren Programmen

Speicherbereinigung (Garbage Collection) (2/2)

- Warum Speicherbereinigung nicht verwenden?
  - **Teuer:** Festzustellen welche Objekte lebendig und welche tot sind kostet Zeit
  - **Schlecht für Echtzeitsysteme**: Können normalerweise nicht garantieren wann die Speicherbereinigung laufen wird und wie lange es dauern wird
  - **Schwierig zu implementieren**: Das Schreiben einer Speicherbereinigung ist schwierig und macht den Compiler und die Laufzeit einer Sprache komplizierter
  - **Sprachdesign**: Einige Sprachen wurden nicht mit dem Gedanken an eine Speicherbereinigung entworfen, was es schwierig machte eine Speicherbereinigung zuzufügen

Weiterführende Spracheigenschaften und Bindungen

- Wie implementieren wir statische Gültigkeitsbereiche in Sprachen die verschachtelte Unterprogramme erlauben?
- Wie implementieren wir Referenzen auf Unterprogramme?

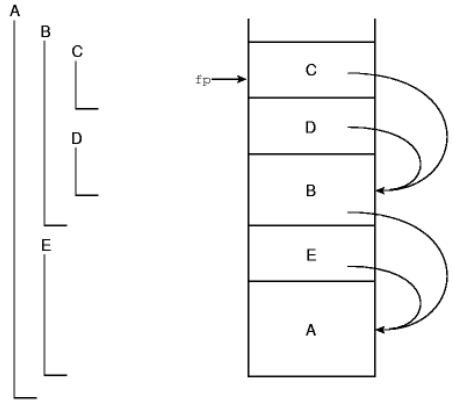
Implementierung von statischen Gültigkeitsbereichen (1/2)

- Problem:
  - Von c(), wie referenzieren wir nichtlokale Variablen in a() und b()?

```
void a (void) {
   int foo1;
    void b (void) {
        int foo2;
        void c (void) {
            int foo3 = foo1 + foo2;
        }
        void d (void) {
        }
    }
    void e (void) {
    }
}
```

Implementierung von statischen Gültigkeitsbereichen (2/2)

- Lösung: statische Links
  - Jede Aktivierung eines Unterprogramms speichert einen Zeiger zum nächsten, umgebenden Gültigkeitsbereich in seinem Stackrahmen
  - c() bekommt seine eigenen Lokalen von seinem eigenen Stackrahmen
  - c() bekommt die Nichtlokalen in b() durch die einmalige Dereferenzierung seines statischen Links
  - c() bekommt die Nichtlokalen in a() durch die Dereferenzierung seines statischen Links zu b() und dann b()'s statischen Link zu a()



Implementierung von Referenzen auf Unterprogramme - 1

- Große Frage:
  - Wie wenden wir Gültigkeitsbereichsregeln in Sprachen an wo Unterprogramme als Wert übergeben werden können?
- Zwei Antworten:
  - Flache Bindung (shallow binding):
    - Referenzierende Umgebung wird sofort festgestellt bevor das referenzierte Unterprogramm aufgerufen wird
    - Ist normalerweise der Standard in Sprachen mit dynamischen Gültigkeitsbereichen
  - Tiefe Bindung (deep binding):
    - Referenzierende Umgebung wird festgestellt wenn die Referenz auf das Unterprogramm erzeugt wird

Tiefe vs. flache Bindung

- Flache Bindung (shallow)
  - Nötig für die **line_length** Zuweisung in **print_selected_records** um das **print_person** Unterprogramm zu erreichen
- Tiefe Bindung (deep)
  - Nötig für die **threshold** Zuweisung im Hauptprogramm um das **older_than** Unterpogramm zu erreichen

```
type person = record
    ...
    age : integer
    ...
threshold : integer
people : database

function older_than (p : person) : boolean
    return p.age ≥ threshold

procedure print_person (p : person)
    -- Call appropriate I/O routines to print record on standard
    -- output. Make use of non-local variable line_length to format
    -- data in columns.
    ...

procedure print_selected_records (
        db : database
        predicate, print_routine : procedure)
    line_length : integer

    if device_type (stdout) = terminal
        line_length := 80
    else    -- Standard output is a file or printer.
        line_length := 132
    foreach record r in db
        -- Iterating over these may actually be
        -- a lot more complicated than a 'for' loop.
        if predicate (r)
            print_routine (r)

-- main program
...
threshold := 35
print_selected_records (people, older_than, print_person)
```


Implementierung von Referenzen auf Unterprogramme - 2

- Zum Abschluss
  - Ein Zeiger auf den Code des Unterprogramms und ein Zeiger auf die referenzierende Umgebung des Unterprogramms
- Warum benötigen wir einen Zeiger auf die referenzierende Umgebung?
  - Wir müssen einen Weg für das Unterprogramm haben mit welcher es Zugriff auf seine nichtlokalen, nichtglobalen Variablen hat
    - Für Sprachen mit dynamischen Gültigkeitsbereichen schließt dies Variablen in den umschließenden, dynamisch verschachtelten Unterprogrammen ein (Unterprogramme die andere aufrufen)
    - Für Sprachen mit statischen Gültigkeitsbereichen schließt dies Variablen in den umschließenden, statisch verschachtelten Unterprogrammen ein

Zusammenfassung

- Lebensdauer von Objekten
  - Die Zeitperiode während der ein Name an ein Objekt (Variable, Unterprogramm, etc.) gebunden ist
- Speichermanagement
  - Statische Allokation, Stacks und Heaps
- Weiterführende Spracheigenschaften und Bindungen
  - Implementierung von statischen Gültigkeitsbereichen für verschachtelte Unterprogramme
    - Links auf Stackrahmen ablegen
  - Implementierung von Unterprogrammreferenzen
    - Ein Zeiger auf den Code des Unterprogramms und ein Zeiger auf die referenzierende Umgebung des Unterprogramms (flache und tiefe Bindung)

Überblick

- Einführung Programmiersprachen
- Namen, Bindungen und Gültigkeitsbereiche
- Speichermanagement und Implementierung
- **Kontrollfluss**

Was fehlt uns noch?

- Kontrollfluss
  - Spezifikation der Reihenfolge in welcher Elemente einer höheren Programmiersprache ausgeführt werden
- Kontrollflussmechanismen
  - Anweisungen (statements), Schleifen, Unterprogrammaufrufe, Rekursion, etc.
  - Übersetzung zu ausführbarem Code

Kontrollfluss (control flow)

- **Das Offensichtliche**: Programme machen ihre Arbeit durch ausführen von Berechnungen
- **Kontrollfluss** (**control flow**) spezifiziert die Reihenfolge in welcher Berechnungen ausgeführt werden
- Sprachen stellen eine große Vielfalt von **Kontrollflussmechanismen** (**control flow mechanism**) bereit welche es dem Programmierer erlauben den Kontrollfluss zu spezifizieren

Kontrollfluss und Kontrollflussmechanismen

- Iterationen (Wiederholungen)
  - Führt eine Gruppe von Berechnungen mehrmals aus (repeatedly)
  - Eine von sieben Kategorien von Kontrollflussmechanismen
- Wie implementieren Sprachen Iterationen?

Schleifen: C, C++, Java

```
for(i=0;i<N;i++) {
  do_something(i);
}
```

Iteratoren: Icon, CLU

```
every do_something(0 to N-1)
```

Einige Sprachen verwenden andere Mechanismen um das gleiche Ziel zu erreichen

Rekursion: ML

```
fun foo i N =
    if i < N then
        do_something i
        foo i + 1 N
foo 0 N
```


Wichtige Fragen

- Wie lauten die Kategorien von Kontrollflussmechanismen
- Welche Eigenschaften stellen Sprachen zur Verfügung um die Kontrollflussmechanismen einer gegebenen Kategorie zu implementieren?
- Wie übersetzen Compiler Kontrollflussmechanismen in ausführbaren Code?

Kategorien von Kontrollflussmechanismen

- **Sequentialität** (sequencing)
  - Bestimmt für ein gegebenes Paar von Berechnungen welches zuerst ausgeführt wird
- **Auswahl** (selection)
  - Wähle, basierend auf einer Laufzeitbedingung, welche von mehreren Berechnungen ausgeführt werden soll
- **Iteration** (iteration)
  - Führt eine Gruppe von Berechnungen mehrmals aus
- **Rekursion** (recursion)
  - Erlaubt es Berechnungen durch sich selbst zu definieren (Allows computations to be defined in terms of themselves)
- **Prozedurale Abstraktion** (procedural abstraction)
  - Erlaubt es einer Gruppe von Berechnungen zu bennen, möglicherweise zu parametrisieren und auszuführen wann immer der Name referenziert wird
- **Nebenläufigkeit** (concurrency)
  - Erlaubt es Berechnungen „zur gleichen Zeit“ auszuführen
- **Keine Festlegung** (Nondeterminancy)
  - Reihenfolge zwischen Berechnungen wird unspezifiziert gelassen

Sequentialität (sequencing)

- Bestimmt für ein gegebenes Paar von Berechnungen welches zuerst ausgeführt wird
- Zwei Typen von Berechnungen für welche wir die Reihenfolge betrachten wollen
  - Ausdrücke (expressions)
    - Berechnungen die einen Wert erzeugen
    - Beispiel:
      - zweistellige Ausdrücke: foo + bar
      - Evaluieren wir zuerst die Unterausdrücke foo oder bar bevor wir die Addition ausführen?
  - Zuweisungen (assignments)
    - Berechnungen welche die Werte von Variablen ändern
    - Beispiel:
      - foo = bar + 1;
      - bar = foo + 1;
    - Welche von diesen beiden Zuweisungen evaluieren wir zuerst?

Was genau ist eine Ausdruck (expression)?

- Berechnungen welche einen Wert erzeugen
  - Variablenreferenzen
    - Holt den Wert der Variablen aus dem Speicher
  - Konstantenreferenzen
    - 1, 2, 3, ‘a’, ‘b’, ‘c’, …
  - Operatoren oder Funktionen angewandt auf eine Sammlung von Unterausdrücken
    - Funktionsaufrufe
      - foo(a,b+10)
    - Arithmetische Operationen
      - foo + bar

Evaluation von Ausdrücken und Seiteneffekte

- Evaluation von Ausdrücken kann zusätzlich zu den **Werten** auch **Seiteneffekte** erzeugen
- Beispiele:
  - int foo (void) { a = 10; return a; }
  - … foo() + 20 …
  - Evaluation von „foo()+20“ ergibt den Wert 30 UND als einen Seiteneffekt die Zuweisung des Wertes 10 an die Globale a
- Ausdrücke ohne Seiteneffekte werden „**referentially transparent**” genannt
  - Solche Ausdrücke können als mathematische Objekte behandelt werden und entsprechend durchdacht

Priorität und Assoziativität

- Zwei Wege um die Reihenfolge zu spezifizieren in welcher Unterausdrücke von komplexen Ausdrücken evaluiert werden
- Prioritätsregel
  - Spezifiziert wie Operationen gruppieren wenn Klammern abwesend sind
- Assoziativität
  - Ähnlich dem mathematischen Konzept mit dem gleichen Namen
  - Spezifiziert ob Operatoren von der gleichen Prioritätsgruppe zur Linken oder Rechten zuerst evaluieren

Beispiel zur Motivation (1/3)

```
// kleines Bsp. für die  Verwendung von *= und ++
#include <iostream>
using namespace std;

int main()
{
  int x=3, y=2;
  cout << "x=3, y=2“ << endl;

  x *= ( (x++) - (++y) + x );

  cout << "x*=((x++)-(++y)+x): " <<x<<endl;

return 0;
}
```

Beispiel zur Motivation (2/3)

- Das Programm liefert auf folgenden Plattformen und Compilern unterschiedliche Ergebnisse, z.B.:
  - Linux Intel g++ 3.3.3; Ergebnis = **10**
  - Linux Intel g++ 2.95.4; Ergebnis = **10**
  - Solaris SPARC g++ 2.95.3; Ergebnis = **4**
  - Solaris SPARC Sun WorkShop 6 update 1 C++ 5.2; Ergebnis =**16**

- Das Problem liegt **nicht** an der Priorität der Operatoren oder der Assoziativität, da alles geklammert ist.

  Das Problem liegt an der fehlenden Spezifikation im C Standard über den Zeitpunkt der **tatsächlichen Ausführung** der Addition um 1 beim ++-Operator.

→ Unterschiedliche Interpretation der Operatoren durch C(++) Compiler

Beispiel zur Motivation (3/3)

```
// kleines Bsp. für die  Verwendung
   von *= und ++
#include <iostream>
using namespace std;

int main()
{
  int x=3, y=2;
  cout << "x=3, y=2“ << endl;

  x *= ( (x++) - (++y) + x );

  cout << "x*=((x++)-(++y)+x): "
   <<x<<endl;

return 0;
}
```

1. Ergebnis **x=16** ergibt sich durch
   ((x++)-(++y) + x) =
   ((3+1)-(2+1) + 3) = 4 => x=16

→ x++ und y++ werden **vor** der Zuweisung *= ausgeführt

2. Ergebnis **x=4** ergibt sich durch
   ((**x++**)-(++y) + x) =
   (3 -(2+1) + 3) = 3

   *= für altes x=3 => x=9
   x++ für **altes** x=3 => x=4

→ x++ wird erst **nach** der Zuweisung *= ausgeführt (mit Bezug auf das Alte)!!!

3. Ergebnis **x=10** ergibt sich durch
   ((**x++**)-(++y) + x) =
   (3 -(2+1) + 3) = 3

   *= für altes x=3 => x=9
   x++ für **neues** x=9 => x=10

→ x++ wird erst **nach** der Zuweisung *= ausgeführt (mit Bezug auf das Neue)!!!

Priorität (Precedence)

- Beispiel:
  - -a * c
  - Wie ist die Reihenfolge der Evaluierung?
    - (-a) * c oder –(a * c)???
  - In C, -, unäre Negierung (unary negation), hat höhere Priorität als * (Multiplikation), weshalb die erste Klammerung korrekt ist
- Prioritätsregeln variieren stark von Sprache zu Sprache

- Frage:
  - Wie wird in C `int i = 0; int *ip = &i; **++*ip++;**` ausgewertet?
  - ++( ) auf den Wert von *ip++. Der Wert ist der Inhalt der Variablen, auf die ip verweist. Wenn ip abgerufen wurde, wird ip im nachhinein um eins erhöht (Zeigerarithmetik).
  - Daher ist am Ende i = 1 und der Zeiger ip wurde auch um eins erhöht (und zeigt somit höchstwahrscheinlich ins Nirvana, da kein Bezug zu dieser Speicherstelle besteht)

Assoziativität

- Die Assoziativität ist in Sprachen einheitlicher
- Elementare arithmetische Operatoren assoziieren von links nach rechts
  - Operatoren werden von links nach rechts gruppiert und evaluiert
  - Beispiel (Subtraktion):
    - 9 – 3 – 2 evaluiert eher zu (9 – 3) -2 als 9 - (3 - 2)
- Einige arithmetische Operatoren assoziieren aber von rechts nach links
  - Beispiel (Potenzieren):
    - 4**3**2 evaluiert eher zu 4**(3**2) als (4**3)**2
- In Sprachen die Zuweisungen in Ausdrücken erlauben, assoziieren Zuweisungen von rechts nach links
  - Beispiel:
    - a = b = a + c evaluiert zu a = (b = a + c)
    - a + c wird berechnet und b zugewiesen, danach wird b zu a zugewiesen

Arbeiten mit Prioritäten und Assoziativität

- Regeln für Prioritäten und Assoziativität variieren stark von Sprache zu Sprache
  - In Pascal:
    - „if A < B und C < D then ...“
    - Könnte zu „if A < (B and C) < D then...“ evaluiert werden
    - Upps!
  - **Leitfaden/Richtlinie:**
    - **Im Zweifelsfall lieber Klammern verwenden, speziell wenn jemand oft zwischen verschiedenen Sprachen wechselt!**

Priorität/Assoziativität von Operatoren in C

Priorität	*Operator*		*Assoziativität*
15	( ) [ ] ->	Funktionsaufruf Arrayindex Memberzugriff	von links nach rechts
14	! ~ ++ -- + - * & (typ) sizeof	Negation (logisch, bitweise) Inkrement, Dekrement Vorzeichen Dereferenzierung, Adresse Typecast	von rechts nach links
13	* / %	Multiplikatiom, Division Modulo	von links nach rechts
12	+ -	Addition, Subtraktion	von links nach rechts
11	<< >>	bitweises Schieben	von links nach rechts
10	< <= > >=	Vergleich gößer/kleiner	von links nach rechts
9	== !=	gleich/ungleich	von links nach rechts
8	&	bitweises und	von links nach rechts
7	^	bitweises exklusiv-oder	von links nach rechts
6	|	bitweises oder	von links nach rechts
5	&&	logisches und	von links nach rechts
4	||	logisches oder	von links nach rechts
3	?:	bedingte Auswertung	von rechts nach links
2	= += -= *= /= %= &= ^= |= <<= >>=	Wertzuweisung kombinierte Zuweisung	von rechts nach links
1	,	Kommaoperator	von links nach rechts

Mehr zur Evaluierungsordnung von Ausdrücken (1/2)

- Prioritäten und Assoziativität können nicht immer eine Evaluierungsreihenfolge festlegen
  - Beispiel:
    - a – f(b) – c * d
    - Mit Prioritäten ergibt sich a – f(b) – (c * d) und weiter mit Assoziativität (a – f(b)) - (c * d)
    - Aber welches wird zuerst ausgeführt: a-f(b) oder (c*d)?
    - Warum kümmert uns das?
- Seiteneffekte
  - Wenn f(b) die Variablen c oder d modifiziert, dann hängt der Wert des ganzen Ausdruckes davon ab, ob f(b) zuerst ausgeführt wird oder nicht
- Verbesserung des Codes
  - Wir wollen (c*d) zu erst berechnen, so dass wir das berechnete Ergebnis nicht speichern und wiederherstellen müssen bevor und nachdem f(b) aufgerufen wird
- Nochmals: **Im Zweifelsfall Klammern setzen!**

Mehr zur Evaluierungsordnung von Ausdrücken (2/2)

- Boolesche Ausdrücke
  - Ausdrücke die eine logische Operation ausführen und entweder zu wahr (true) oder falsch (false) evaluieren
  - Beispiele:
    - a < b
    - a && b || c oder a & b | c
- Die Evaluation von booleschen Ausdrücken kann durch die „**short circuiting**“ Technik optimiert werden
  - Beispiel:
    - (a < b) && (c < d)
    - Wenn (a < b) zu falsch evaluiert wird dann gibt es keinen Bedarf mehr (c <d) auszuwerten
    - Siehe Regel R1 aus Kapitel 2, Boolesche Algebra und Gatter
- Dies wird wichtig bei Code nach folgendem Muster:
  - if (unwahrscheinliche Bedingung && teure Funktion()) ...
  - Wir wollen die Auswertung einer teuren Funktion() verhindern, wenn eine unwahrscheinliche Bedingung zu falsch ausgewertet wird

Zuweisungen

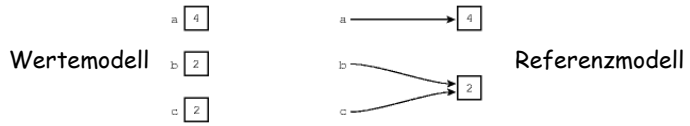
- Berechnungen welche den Wert einer Variablen beeinflussen
  - Beispiel: c = a + b
    - Weisst den Wert, der durch den Ausdruck „ a +b“ berechnet wird, der Variablen c zu
- Zuweisungen sind der fundamentale Mechanismus um **Seiteneffekte** zu erzeugen
  - Werte die Variablen zugewiesen werden können zukünftige Berechnungen beeinflussen
- Zuweisungen sind unerlässlich für das imperative Programmiermodel

Was genau ist eine Variable? (1/2)

- Die Interpretation eines Variablennamens hängt von dem Kontext ab in welchem er auftritt
- d = a
  - Die Referenz zu der Variablen „a“ auf der „**rechten Seite**“ einer Zuweisung benötigt einen Wert
    - Wird ein **r-value-context** genannt und das „a“ wird **r-value** genannt
- a = b + c
  - Die Referenz zu der Variablen „a“ auf der „**linken Seite**“ einer Zuweisung bezieht sich auf a‘s Speicherort
    - Wird ein **l-value-context** genannt und das „a“ wird l-**value** genannt

Was genau ist eine Variable? (2/2)

- Zwei Wege um eine Variable zu behandeln
  - Wertemodell
    - Variablen sind benannte Container für Werte
    - Beide Interpretationen als l-value und r-value von Variablen sind möglich
    - Modell wird verwendet von Pascal, Ada, C, etc.
  - Referenzmodell
    - Variablen sind benannte Referenzen für Werte
    - Nur die Interpretation als l-value ist möglich
      - Variablen in einem r-value-context müssen „dereferenziert“ werden um einen Wert zu erzeugen
    - Modell wird verwendet von Clu



Zuweisungsfolgen

- Zuweisungsfolgen (assignment sequencing) sind in den meisten Sprachen einfach
  - Zuweisungen werden in der Reihenfolge ausgeführt in der sie im Programmtext auftreten
    - Beispiel:
      - a = 10; b = a; ← Es wird zuerst die 10 dem a zugewiesen und danach wird a dem b zugewiesen
- Ausnahmen
  - Zuweisungsausdrücke
    - Beispiel: a = b = a * c
    - Erinnere dich an die Auswertereihenfolge bei Ausdrücken: In den meisten Programmiersprachen gilt die Assoziation von rechts nach links
  - Initialisierung
  - Kombinationen mit dem Zuweisungsoperator

Zuweisungsfolgen: Initialisierung (1/2)

- Wie initialisieren oder wie weisen wir initiale Werte Variablen zu?
- Verwende den Zuweisungsoperator um initiale Werte zur Laufzeit zuzuweisen
  - Probleme:
    - **Ineffizienz**: Wenn wir den initialen Wert einer Variablen zur Compilezeit kennen, dann kann der Compiler diesen Wert dem Speicher zuweisen ohne eine Initialisierung zur Laufzeit zu benötigen
    - **Programmierfehler**: Wenn Variablen kein initialer Wert bei der Deklaration zugewiesen wird, dann könnte ein Programm eine Variable verwenden bevor sie irgendeinen (sinnvollen) Wert enthält

Zuweisungsfolgen: Initialisierung (2/2)

- Verwende den Zuweisungsoperator um initiale Werte zur Laufzeit zuzuweisen
  - Lösungen:
    - **Statische Initialisierung**:
      - Beispiel für C: static char s[] = “foo”
      - Der Compiler kann den Elementen des Arrays s die Werte zur Compilezeit zuweisen, was uns 4 Zuweisungen zur Laufzeit erspart (eine für jedes Zeichen plus dem NULL (String-)Terminator)
      - siehe auch C++ Konstruktorinitialisierung
    - **Defaultwerte** (Standardwerte/Ausgangswerte/...)
      - Sprachen können einen Defaultwert für jede Deklaration eines eingebauten (built-in) Typs spezifizieren
    - **Dynamische Wertkontrollen**
      - Mache es zu einem Laufzeitfehler, wenn eine Variable verwendet wird bevor ihr ein Wert zugewiesen wurde
    - **Statische Wertkontrollen**
      - Mache es zu einem Compilerfehler wenn eine Variable verwendet werden **könnte** bevor ihr ein Wert zugewiesen wurde

Zuweisungsfolgen: Kombinationen

- Gibt es einen effizienteren Weg um komplexe Zuweisungen zu handhaben?
  - Beispiele:
    - a = a + 1;
    - b.c[3].d = b.c[3].d * e;
- Probleme:
  - Ist fehleranfällig und schwierig zu schreiben, da Text wiederholt wird
  - Kann zu ineffizientem kompilierten Code führen
- Lösungen: Kombinationen mit dem Zuweisungsoperator
  - Beispiele:
    - a += 1;
    - b.c[3].d *= e;
  - Zuweisungen werden mit Operatoren für Ausdrücke kombiniert
    - Vermeidet sich wiederholenden Code
    - Erlaubt es dem Compiler auf einfacherem Wege effizienteren Code zu generieren

Sequentialität: Zusammenfassung

- Ausdrücke
  - Prioritäten und Assoziativität kontrollieren die Reihenfolge
  - Es müssen **Seiteneffekte** bei Unterausdrücken beachtet werden
  - Logische Ausdrücke können von der short-circuit Auswertung profitieren
- Zuweisungen
  - Die Folge des Auftretens im Programmtext bestimmt die Reihenfolge
  - Zuweisungen zur Compilezeit erlauben die Vermeidung von Zuweisungen zur Laufzeit
  - Zusammengesetzte Zuweisungsoperatoren kombinieren Ausdrucksauswertung und Zuweisung um effizienter sein zu können

Auswahl (selection) (1/4)

- Wähle, basierend auf einer Laufzeitbedingung, welche von mehreren Berechnungen ausgeführt werden soll
- Am häufigsten Ausgeführt vom `if..then..else` Sprachkonstrukt
  - Beispiel:
    - if ((a < b) && (c < d)) { do_something(); }
    - Wenn beide bedingte Anweisungen „a<b“ und „c<d“ zu wahr ausgewertet werden, dann führe die Berechnung, welche durch das Unterprogramm do_something() referenziert wird, aus

Auswahl (selection) (2/4)

- Short circuiting
  - Zur Erinnerung: Kann verwendet werden um unnötige Ausdrucksauswertungen zu verhinden
  - Beispiel:
    - if ((a < b) && (c < d)) { do_something(); }
    - Wenn (a < b) falsch ist, dann kann do_something() nicht ausgeführt werden, egal was (c < d) ergibt
    - Siehe Regel R1 aus Kapitel 2, Boolesche Algebra und Gatter
    - Short circuiting wird Code generieren der die Auswertung von (c < d) genauso ausläßt (skipped) wie die Ausführung von do_something(), wenn (a < b) falsch ist

Auswahl (selection) (3/4)

- Angenommen wir haben Code der ähnlich wie dieser aussieht:

```
j := … (* something complicated *)
IF j = 1 THEN
   clause_A
ELSIF j IN 2, 7 THEN
   clause_B
ELSIF j in 3..5 THEN
   clause_C
ELSE
   clause_D
END
```

- Problem:
  - Ist kompliziert zu schreiben

Auswahl (selection) (4/4)

- Lösung: case/switch Befehle
- Beispiel:

```
CASE …
      1:        clause_A
    | 2,7:      clause_B
    | 3..5:     clause_C
      ELSE      clause_D
END
```

- Ist einfacher zu schreiben und zu verstehen

Iteration

- Führt eine Gruppe von Berechnungen mehrmals aus
- Sprachen bieten für Iterationen Schleifenkonstrukte an
  - Aufzählungsgesteuerte Schleifen
    - Die Schleife wird für jeden Wert aus einer endlichen Menge (Aufzählung) ausgeführt
  - Logikgesteuerte Schleifen
    - Die Schleife wird solange ausgeführt bis sich eine boolesche Bedingung ändert
- Anmerkung:
  - Der imperative Stil der Programmierung favorisiert Schleifen gegenüber Rekursion
  - Iteration und Rekursion können beide verwendet werden um eine Gruppe von Berechnungen wiederholt auszuführen
  - Alle Arten von Schleifen können auf die beiden Konstrukte Selektion (if..then..end) und Sprung (goto) zurückgeführt werden. Wird oft in Zwischensprachen zur Codegenerierung verwendet.

Aufzählungsgesteuerte Schleifen: Probleme (1/3)

- Beispiel von FORTRAN I, II und IV

```
do 10 i = 1, 10, 2
...
10: continue
```

  - Die Variable i wird die **Indexvariable** genannt und nimmt hier die Werte 1, 3, 5, 7, 9 an
  - Die Statements zwischen der ersten und letzte Zeile werden der **Schleifenrumpf** (**loop body**) genannt und wird hier für jeden der fünf Werte von i einmal ausgeführt
- Probleme:
  - Der Schleifenrumpf wird immer mindestens einmal ausgeführt
  - Statements im Schleifenrumpf könnten i ändern und somit würde auch das Verhalten der Schleife verändert werden
  - Goto Statements können in oder aus der Schleife springen
    - Goto-Sprünge in die Schleife hinein, wobei i nicht vernünftig initialisiert wird, werden wahrscheinlich in einem Laufzeitfehler enden
  - Die Schleife wird beendet wenn der Wert von i die obere Grenze **überschreitet**. Dies könnte einen (unnötigen) arithmetischen Überlauf veranlassen, wenn die obere Grenze in der Nähe des größten, zulässigen Wertes von i liegt

Aufzählungsgesteuerte Schleifen: Fragen (2/3)

- Allgemeine Form von Schleifen:

```
FOR j := first TO last BY step DO
    …
END
```

- Fragen die zu dieser Schleifenart zu stellen sind:
  1. Kann `j`, `first` und/oder `last` im Schleifenrumpf verändert werden? Wenn ja, welchen Einfluss hat dies auf die Steuerung?
  2. Was passiert wenn `first` größer als `last` ist?
  3. Welchen Wert hat `j` wenn die Schleife beendet wurde?
  4. Kann die Kontrolle von außerhalb in die Schleife hineinspringen?

Aufzählungsgesteuerte Schleifen: Antworten (3/3)

- Können der Schleifenindex oder die Grenzen verändert werden?
  - Ist bei den meisten Sprachen verboten
    - Algol 68, Pascal, Ada, Fortran 77 und 90, Modula-3
- Was passiert wenn `first` größer als `last` ist?
  - Die meisten Sprachen werten `first` und `last` aus bevor die Schleife ausgeführt wird
  - Falls `first` größer als `last` ist, dann wird der Schleifenrumpf niemals ausgeführt
- Welchen Wert hat der Schleifenindex `j` am Ende der Schleife?
  - Ist in einigen Sprachen nicht definiert, z.B. Fortran IV oder Pascal
  - In den meisten Sprachen gilt der zuletzt definierte Wert
  - Einige Sprachen machen die Indexvariable zu einer lokalen Variablen der Schleife, weswegen die Variable außerhalb des Gültigkeitsbereiches der Schleife nicht sichtbar ist
- Kann die Kontrolle von außerhalb in die Schleife hineinspringen?
  - Nein, für die meisten Sprachen
  - Viele Sprachen erlauben aber den Sprung von innen nach außen
  - Viele Sprachen bieten Anweisungen (break; exit; last, etc,) an, die einen frühzeitige Abbruch der Schleife ohne expliziten Sprung erlauben

Iteratoren

- Alle bisher betrachteten Schleifen iterieren über eine arithmetische Folge (1,3,5,7)
- Aber wie iterieren wir über eine frei wählbare Menge von Objekten?
- Antwort: Iteratoren
  - Die “FOR j := first TO last BY step DO … END” Schleife kann auch geschrieben werden als
    “every j := first to last by step do { … }” in ICON
- Beispiel:
  - “every write (1 + upto(‘ ‘, s))” in ICON schreibt jede Position in den String s welcher ein Leerzeichen folgt
  - upto(‘ ‘,s) erzeugt jede Position in s gefolgt von einem Leerzeichen
- Iteratoren sind Ausdrücke die mehrere Werte generieren und welche Ihre enthaltende Ausdrücke und Statements bestimmen können um diese mehrmals auszuwerten oder auszuführen
- Ein anderes Beispiel:
  - “write(10 + 1 to 20)” schreibt 10 + j für jedes j zwischen 1 und 20
  - 1 bis 20 erzeugt die Folge 1 bis einschließlich 20
- Siehe auch Standard Template Library (STL) von C++

Logikgesteuerte Schleifen

- Beispiele:
  - Vorprüfende Schleife
    - „while Bedingung do Anweisung“
    - Die C for-Schleife ist eine vorprüfende, logikgesteuerte Schleife und nicht eine aufzählungsgesteuerte Schleife!
      - for(Vor-Anweisung; Bedingung; Nach-Anweisung) Block
  - Nachprüfende Schleife
    - „repeat Anweisung until Bedingung“
    - „do Anweisung while Bedingung“
    - Diese Art von Schleife werden oft falsch verwendet und sollten daher vermieden werden
      (da mindest ein Durchlauf erfolgt!)
  - Prüfung in der Mitte einer Schleife
    - „loop Anweisung **when Bedingung exit** Anweisung end“

Rekursion

- Erlaubt es Berechnungen durch sich selbst zu definieren
  - Mit anderen Worten, komplexe Berechnungen werden mit Begriffen von einfacheren Berechnungen ausgedrückt
  - Ähnlich dem Prinzip der mathematischen Induktion
- Ist die bevorzugte Methode der wiederholenden Programmausführung in funktionalen Sprachen
  - Eigentlich ist es die meiste Zeit der einzigste Weg um Berechnungen in funktionalen Sprachen wiederholt auszuführen
  - Warum?
    - Weil für die Alternative, nämlich Iteration (Schleifen), die Ausführung von Anweisungen mit Seiteneffekten (wie Zuweisungen) verbunden ist, was in rein funktionalen Sprachen verboten ist
  - Dies ist nicht wirklich eine Einschränkung da Rekursion streng genommen genauso Mächtig ist wie Iteration und umgekehrt

Rekursion und Iteration

- Informell gesprochen, die Schleife
  “`FOR j := first TO last BY step DO … END`”
  wird durch Rekursion zu:

```
fun loop j step last =
    if j <= last then
        …
        loop j + step step last
```

- Es gibt mit der rekursiven Emulation von Iteration aber ein Problem: sie ist **langsam**!

Verbesserte Durchführung von Rekursion

- Gegebene Schleife

```
fun loop j step last =
    if j <= last then
        …
        loop j + step step last
```

- Der rekursive Aufruf der Schleife „loop“ ist die letzte ausgeführte Berechnung der Funktion „loop“
  - Wird **Endrekursion** (**tail recursion**) genannt
- Geschickte Compiler werden Endrekursionen durch eine Schleife ersetzen
  - Argumente für den Aufruf der Endrekursion werden ausgewertet und in den entsprechenden Lokalen platziert und ein Sprung zurück zum Anfang der Funktion wird gemacht anstatt einen rekursiven Aufruf zu machen
  - Dies kann zu einer signifikanten Steigerung der Leistung führen

Evaluationskonzepte von Funktionsargumenten

- Wie werden Argumente für Funktionen ausgewertet?
  - Die **Argumente** einer Funktion werden ausgewertet **bevor** die Funktion die Steuerung erhält?
    - Wird Applicative-Order Evaluation (eager evaluation, call-by-value) bezeichnet
    - Ist der Standard in den meisten Sprachen
  - Die **Argumente** einer Funktion werden nur dann ausgewertet, wenn sie auch **benötigt** werden?
    - Wird Normal-Order Evaluation (lazy evaluation, call by need, delayed evaluation) bezeichnet
    - Kann zu einer besseren Leistung führen wenn an eine Funktion übergebene Argumente manchmal nicht gebraucht werden
    - Beispiel:

```
void foo (int a, int b, int c) {
    if (a + b < N) { return c; } else { return a + b; }
        foo(a,b,expensive_function())
```

      - In einigen Fällen wird „c“ nicht benötigt, und der Wert von c könnte von einerteuer zu berechnenden Funktion bestimmt werden

Was haben Sie in diesem Kapitel gelernt?

- Programmiersprachen sind Abstraktionsmechanismen, welche:
  - ein Rahmenwerk zum Lösen von Problemen und erzeugen neuer Abstraktionen bieten
  - Schirmen den Programmierer von niederen Detailebenen (low-level details) der Zielmaschine (Assemblersprache, Verbinden (linken), etc.) ab
- Compiler sind komplexe Programme welche eine höhere Programmiersprache in eine Assemblersprache umwandeln um danach ausgeführt zu werden
- Compilerprogrammierer handhaben die Komplexität des Kompilationsprozesses durch:
  - aufteilen des Compilers in unterschiedliche Phasen
  - verwenden, ausgehend von der Spezifikation, eine Theorie für den Bau von Compilerkomponenten

Ausblick