

# Vorlesung Compilerbau (SoSe2018)

## Teil 12: Laufzeitumgebungen – Teil II



### Vorlesung Compilerbau: Laufzeitumgebungen – Teil II

Vorlesung des BA-Studiums  
Prof. Johann Christoph Freytag, Ph.D.  
Institut für Informatik, Humboldt-Universität zu Berlin  
SoSe2018



### Überblick

Zugriff auf Variablen zur Laufzeit  
(Beziehung zwischen Namen und Datenobjekten)

- Speicherbehandlung und –verwaltung
- Aktivierungssegmente
- Adressberechnungen

# Vorlesung Compilerbau (SoSe2018)

## Teil 12: Laufzeitumgebungen – Teil II

### Speicherorganisation ...

... eines Maschinenprogramms zur Laufzeit (Laufzeitsystem)

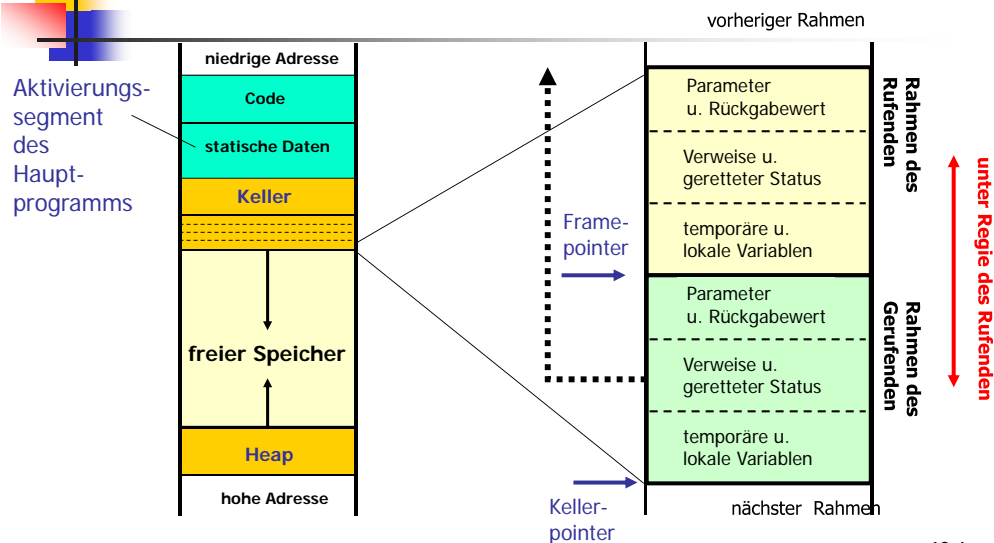
- Unterteilung des Speichers in Segmente zur besseren Verwaltung
- Segmente werden dynamisch (zur Laufzeit) angelegt und gelöscht
- die **Codegröße** eines Programms (Assembler oder Maschinencode) ist **konstant**
- dabei Prozeduraufrufe als calls mit konstantem Argument bei der Compilation erzeugt werden können (Sprungadresse ist fest)
- Vereinheitlichung (Hauptprogramm, Prozedur, Funktion)
- Größe des benötigten **Datenspeichers** ist **veränderlich**, sie lässt sich nicht zur Compilezeit bestimmen (Rekursivität)

→ es ist ein **dynamisches Speicherverwaltungsmodell** erforderlich

- beim Aufruf einer Prozedur wird ein eigenes Datensegment (Aktivierungssegment) erzeugt,
- beim Verlassen wird es wieder zerstört

12.3

### Speicherorganisation ... (Forts.)



12.4

# Vorlesung Compilerbau (SoSe2018)

## Teil 12: Laufzeitumgebungen – Teil II

### Adressierung von ...

... Variablen/Parametern innerhalb eines Datensegments

- Prinzip: immer relativ zum Segmentanfang (FP)

**absolute Adresse** := Anfangsadresse des Datensegments (Frame Pointer - FP)

**Basisregister zur Laufzeit**  
+

Verschiebung (Offset) im Datensegment

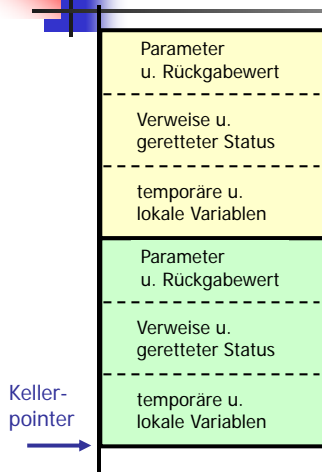
**zur Übersetzungszeit**

... Variablen anderer Datensegmente

- Prinzip: immer relativ zu deren Segmentanfängen (Frame Pointer - FP)

12.5

### Adressierungsunterstützung



**Deskriptor** enthält u.a.:

- Registerwerte** (viele Maschinen: 32 Register)  
Befehlszähler  
Basisadressregister, ...
- Rücksprungadresse** (zum Rufer)
- dynamischer Link** (für Aufrufverkettung)  
Verweis auf Segment der rufenden Prozedur  
(einfaches Löschen des Datensegments nach Rückkehr der gerufenen Prozedur möglich)
- statischer Link** (für textuelle Verschachtelung)  
Verweis auf Segment der umgebenden Prozedur im Quelltext (einfacher Zugriff auf nichtlokale Variablen, möglicherweise über mehrere Stufen)

12.6

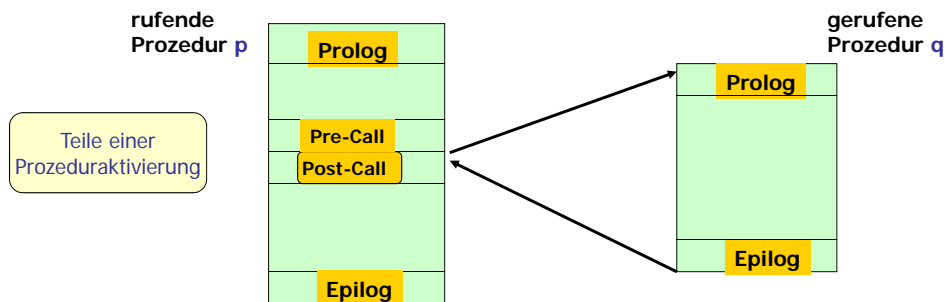
# Vorlesung Compilerbau (SoSe2018)

## Teil 12: Laufzeitumgebungen – Teil II



### Dynamischer Aufbau der Aktivierungssegmente

... durch Aktivierung (Aufruf) der Prozeduren,  
wobei der (eigentliche) Code um spezifische Passagen erweitert wird:  
Prolog, Epilog und Pre-Call, Post-Call



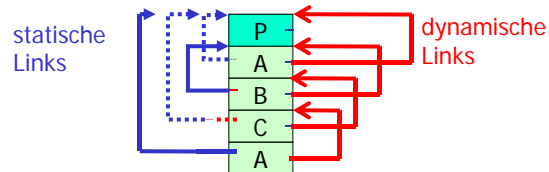
12.7



### Zugriff auf Daten der Aktivierungssegmente

```

program P (input, output)
  procedure A()
    procedure B()
      begin {B}
        C(): ...
      end {B};
    begin {A}
      B(): ...
    end {A}
  procedure C()
    begin
      A(): ...
    end {C};
  begin {P}
    A();
    ...
  end {P}.
  
```



#### Unterscheidung von Zugriffen auf

- globale Variablen (im oberstes Segment)
- lokale Variablen (im aktuellen Segment)
- nichtlokale Variablen (in einem der Zwischensegmente)


#### Methode:

Steigen zur Adressermittlung entlang der statischen Links ab,  
**Frage:** aber wie weit? **Gesucht** ist absolute Adresse!

12.8

# Vorlesung Compilerbau (SoSe2018)

## Teil 12: Laufzeitumgebungen – Teil II



### Zugriff auf nicht-lokale Daten (Forts.)

zur Übersetzungszeit können bestimmt werden

- Deklarationsniveau eines Bezeichners und
- das Offset für den Speicherplatz eines beliebig erzeugtes Aktivierungssegmentes der entsprechenden Prozedur

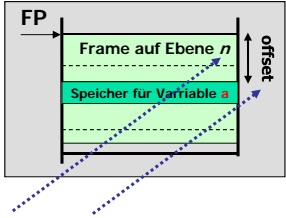
→ jeder Bezeichner (z.B. *a*) ist assoziiert mit einem (level, offset)-Paar

Behandlung von Blockstrukturen (z.B. in C)

- nach gleichem Prinzip: Aktivierungssegmente je aufgerufene Block-Instanz

Bestimmung der Speicheradresse nach zwei unterschiedlichen Verfahren

- Access-Link-Methode
- Display-Tabellen-Methode



12.9



### Zugriff auf nicht-lokale Daten (Forts.)

Adressbestimmung einer Variable *a* mittels der

#### Access-Link-Methode

- bestimme das Niveau *k* der aktiven Prozedur (auf dem Keller), dann ist das Anwendungsniveau  $V_{\text{apply}}$  des Bezeichners *a*, also
$$V_{\text{apply}}(a) := k$$
- bestimme das Deklarationsniveau  $V_{\text{decl}}$  und den *offset* des Bezeichners *a* aus der Symboltabelle
- Bestimme *diff*
$$\text{diff} := V_{\text{apply}}(a) - V_{\text{decl}}(a) \geq 0$$
- Gehe *diff* Schritte in der Kette der *statischen Links* zurück, man erhält FP (Framepointer) des Aktivierungselements (Basisadresse) für Variable *a*

12.10

# Vorlesung Compilerbau (SoSe2018)

## Teil 12: Laufzeitumgebungen – Teil II

### Beispiel 1

```

program () {1}
  procedure foo1() {2}
    begin {foo1}; ...
  end {foo1};
  procedure foo2() {2}
    procedure foo21() {3}
      begin {foo21}
        foo1() ...
      end {foo21};
    begin {foo2}
      foo21(); ...
    end {foo2};
  begin {program}
    foo2(); ...
  end {main}.
  
```

Keller

12.11

### Beispiel 2

```

program foo (input, output); {1}
  var global : integer;
  procedure b (function h (i: integer): integer); {2}
    begin { b } writeln (h(2)) end { b };

  procedure c; {2}
    var m : integer;
    function f (n : integer): integer; {3}
      begin { f } f := m + n - global
    end { f };

    begin { c } m := 0; b(f) end { c };

  begin { foo }
    global = 0; c;
  end { foo }.
  
```

Keller

Zugriff auf Variable global

3-1 → 2

V(apply) V(dec)

12.12

# Vorlesung Compilerbau (SoSe2018)

## Teil 12: Laufzeitumgebungen – Teil II



### Alternative Methode: Display-Tabelle

#### Nachteil der Access-Link-Methode

- Durchlauf langer Ketten bedeutet schlechte Performanz

#### Alternative: Display-Tabelle

- wird **nicht** auf dem Keller angelegt
- Tabelle mit Access-Links für Adresswerte niedrigerer Niveaus
- Auffinden durch Verweise auf Basisadresse in der Tabelle plus Offset für das Niveau
- konstanter Aufwand, d.h. **unabhängig** von momentaner Aufruftiefe!

12.13




### Display-Realisierung (cont.)

- Gegeben: Globale Tabelle *display*
- Aufruf von Prozedur def. auf Ebene  $l$  ( $V_{\text{decl}}$ ) von Ebene  $k$  ( $V_{\text{appl}}$ )
  - $l = k + 1$ : füge einen **neuen Eintrag** für Ebene  $k$  in die Display-Tabelle hinzu
  - $l = k$ : dann muss die Tabelle (Anzahl Einträge) nicht geändert werden
    - Nicht Anzahl der Elemente, aber oberster Eintrag muss geändert werden, um den Zugriff auf lokale Variablen zu ermöglichen
    - Oberster Eintrag muss im lokalen Rahmen der aufrufenden Prozedur gesichert werden
    - Bei Rückkehr ändere oberen Eintrag
  - $l < k$ :
    - **Speichere** die Einträge der Displaytabelle von Ebene  $l + 1$  bis  $k$  **im lokalen Rahmen der aufrufenden Prozedur**
    - Bei **Rückkehr** füge die gesicherten Display-Tabelleneinträge wieder ein (also nicht physisch löschen!)
- Nachteil: großer Verwaltungsaufwand

12.14

# Vorlesung Compilerbau (SoSe2018)

## Teil 12: Laufzeitumgebungen – Teil II

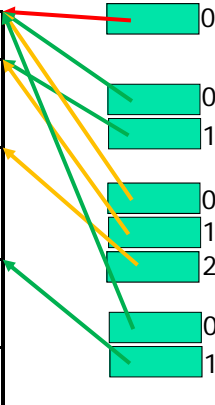


### Beispiel


```
main() {level 0}
  procedure foo1() ... end; {level 1}
  procedure foo2() {level 1}
    procedure foo21() {level 2}
      begin {foo21}
        foo1()
        ...
      end {foo21};
    begin {foo2}
      ...
      foo21();
      ...
    end {foo2};
  begin {main}
    foo2();
    ...
  end {main}.
```

#### Display Tabelle

frame main	0
frame foo2	0
frame foo21	1
frame foo1	0
	1



12.15



### Dynamischer Gültigkeitsbereich

Herausforderung:

- Überführung eines Namens auf eine Speicheradresse erst zur Laufzeit
  - ergibt sich aus aktueller Aufrufhierarchie
  - Referenz kann sich bei unterschiedlichen Aufrufen ändern
- Typüberprüfung zur Laufzeit
  - Typverträglichkeit muss bei jeder Referenz sichergestellt werden

12.16



# Vorlesung Compilerbau (SoSe2018)

## Teil 12: Laufzeitumgebungen – Teil II

### Dynamischer Gültigkeitsbereich (Forts.)

```
program dynamic (input, output);
  var r, z : integer;
  procedure show;
  begin
    write (r); write (z); writeln
  end { show };

  procedure small;
  var r: integer;
  begin
    r := 2; show
  end { small };
begin { dynamic}
  r := 20; z := 50;
  show; small
end { dynamic} .
```

#### Lexikalischer Gültigkeitsbereich

20	50
20	50

wird von Pascal benutzt

#### Dynamischer Gültigkeitsbereich

20	50
2	50

12.17

### Dynamischer Gültigkeitsbereich (Forts.)

#### Realisierung

- Aktivierungssegment muss *für lokale Variablen* auch *Namen* mitführen
- zur Laufzeit: dynamische Überprüfung für Bindung
- Anmerkung
  - Semantik des Programms ist nur noch schwierig zu verstehen
  - große Belastung zur Laufzeit
  - hohe Flexibilität

12.18

# Vorlesung Compilerbau (SoSe2018)

## Teil 12: Laufzeitumgebungen – Teil II



### Organisation der Parameterübergabe

#### Parameterübergabe in Programmiersprachen

(Verbindung von aktuellen und formalen Parametern)

##### 1. Call-by-Value (Wertübergabe)

- einfachste Methode (Algol, Simula, Pascal, C, C++, Java, ...)
- Übernahme des Wertes des Parameters, und nicht der Adresse des Wertes!
- Änderungen des Parameters in der Prozedur haben keine Wirkung nach außen
- Arrays, Strukturen und Strings müssten bei dieser Übergabe kopieren werden (da aufwendig werden diese Typen für Call-by-Value ausgenommen)

##### 2. Call-by-Reference (Adressübergabe)

- Übergabe der Adresse einer Speicherzelle (Simula, Pascal, C, C++, Java, ...)
- Verwendung (Zugriff auf formalen Parameter) wird umgesetzt in eine indirekte Referenz auf den aktuellen Parameter
- Änderung des Wertes der adressierten Speicherstelle hat Außenwirkung

12.19



### Organisation der Parameterübergabe (Forts.)

#### Parameterübergabe in Programmiersprachen

(Verbindung von aktuellen und formalen Parametern)

##### 3. Call-by-Name (Namensübergabe)

- es findet eine textuelle Ersetzung des Bezeichners des formalen Parameter durch die Zeichenkette des aktuellen Parameters im Prozedur-Code statt (Inline-Expansion)  
**ausdrucks mächtig**, wenn aktueller Parameter kein einfacher Name ist, sondern ein (komplexer) Ausdruck, der Bezeichner aus dem lexikalischen Kontext kombiniert, dieser wird dann bei jeder Verwendung im Prozedur-Code neu berechnet
- Sprachen: Algol-60, Algol-68, Simula

- haben Einfluss auf die Implementierung von Pre-Call und Post-Call

12.20

# Vorlesung Compilerbau (SoSe2018)

## Teil 12: Laufzeitumgebungen – Teil II



### Organisation der Parameterübergabe (Forts.)

#### Parameterübergabe in Programmiersprachen

##### 4. Copy-Restore

- Übergabe der Parameter erfolgt nach Call-by-Value, jedoch werden (implizit) zusätzlich die Adressen der aktuellen Parameter bestimmt und mit übergeben
- bei Rückkehr der Prozedur werden die (letzten) Werte der formalen Parameter über die (übergebenen) Adressen der aktuellen Parameter kopiert
- Kombination von Call-by-Value und Call-by-Reference (heißt auch: »Copy-In Copy-Out« oder »Value-Result« in einigen Fortran-Varianten)

12.21



### Organisation der Parameterübergabe (Forts.)

#### Parameterlisten **variabler** Länge (C)

- falls die gerufene Prozedur weiß, dass die rufende Prozedur die Anzahl der Parameter übergeben hat,
  - übergibt die rufende Proz. die Anzahl als nullten Parameter
  - so kann die gerufene Prozedur die Anzahl direkt ablesen
- falls die gerufene Prozedur keine Information bekommt,
  - muss die gerufene Prozedur die Anzahl bestimmen können
  - muss der erste Parameter dem FP am nächsten sein

#### Beispiel printf:

- Anzahl der Parameter wird durch den Formatstring bestimmt
- Annahme: Zahl der Parameter entspricht der Angabe im String

12.22

# Vorlesung Compilerbau (SoSe2018)

## Teil 12: Laufzeitumgebungen – Teil II



### Dynamische Speicherverwaltung

- Speicherzuweisung aus einem Speicherbereich (Heap)
- es gibt Sprachen mit
  - (1) **expliziten Zuweisungen** (Pascal, Ada, Java): **new**
  - (2) **impliziten Zuweisungen** (Snobol, Lisp): **Str= Str 'XYZ'**
  - (3) **ohne explizite und implizite** Zuweisungsmöglichkeiten (C hat dies in die Standardbibliothek verlagert): **malloc(), free()**  
**Vorteil:** erlaubt die angepasste Ersetzung der Mechanismen durch die Bibliothekshersteller  
**Nachteil:** kein Test auf korrekte Anwendung für Compiler und Laufzeitsystem möglich
  - (4) **Kompromisslösung** in C++: explizite Operatoren **new, delete** aber diese können für spezifische Klassen überladen werden
- verschiedene Möglichkeiten zur *Speicherfreigabe*

12.23



### Dynamische Speicherverwaltung (Forts.)

- Laufzeitsystem unterstützt **keine** Speicherfreigabe d.h.: ist kein Speicher mehr verfügbar, stoppt das Programm
  - anwendbar bei Implementationen mit virtuellem Speicher: eigentliche Hauptspeicher wird durch nicht mehr verwendete Datenobjekte tatsächlich auch nicht belastet
  - Anwendung kann für jeden Objekttyp zwei Prozeduren bereitstellen
    1. DisposeObject-Prozedur, die freie Objekte in einer typspezifischen Liste zusammenfasst
    2. AllocateObject-Prozedur, die aus der entsprechenden Liste ein freies Objekt entnimmt und nur wenn keins vorhanden ist auf dem Heap Speicher allokiert

12.24

# Vorlesung Compilerbau (SoSe2018)

## Teil 12: Laufzeitumgebungen – Teil II



### Dynamische Speicherverwaltung (Forts.)

- Sprachen (Laufzeitsysteme) mit **expliziter** Speichervergabe
  - unchecked\_deallocation (Ada)
  - **free** ( C )
  - **dispose** (Pascal)

#### Bemerkung:

- hier wird die (schwierige) Entscheidung, wann Speicherplatz tatsächlich freizugeben ist, in die Anwendung verlagert
- mit dem Preis: Gefahr von **Dangling-Pointer**

12.25



### Dynamische Speicherverwaltung (Forts.)

- Sprachen (Laufzeitsysteme) mit **impliziter** Speicherfreigabe (**garbage collection**)  
Beseitigung vom (Abfall-)Speicherbereich, der von der Anwendung nicht mehr erreicht werden kann, weil Referenzen umgesetzt oder Gültigkeitsbereiche verlassen wurden

#### (1 ) Single-Reference-Methode

**Forderung:** für jedes Objekt auf dem Heap gilt:  
es kann nicht mehr als eine Referenz auf dieses Objekt geben (**schlecht möglich für komplexe Datenstrukturen:**  
**Graphen, Listen – aber gut für Zeichenketten)**

**Methode:** wenn immer eine Zuweisung erfolgt ist,  
oder ein Gültigkeitsbereich verlassen wird,  
werden Objekte ohne Referenz freigegeben

12.26

# Vorlesung Compilerbau (SoSe2018)

## Teil 12: Laufzeitumgebungen – Teil II

### Dynamische Speicherverwaltung (Forts.)

#### (2) Reference-Counting-Methode

**Forderung:** für jedes Objekt auf dem Heap wird ein Zähler geführt, der Anzahl von bestehenden Referenzen auf das Objekt speichert

Referenz-zähler
Datenobjekt

**Methode:** erreicht Referenzzähler Null, wird das Objekt freigegeben

**Preis:** bei jeder Erzeugung, Kopieerstellung und Freigabe muss Zähler aktualisiert werden

**Problem:** 

12.27

### Dynamische Speicherverwaltung (Forts.)

#### (3) Markierungs-Methode

**Forderung:** alle verwendeten Zeiger für Objekte auf dem Heap sind bekannt

**Methode:**

- Benutzerprogramm wird unterbrochen und geprüft, welche Speicherblöcke erreicht werden (diese werden markiert)
- man beginnt mit globalen Zeigervariablen, Variablen von Aktivierungselementen
- man muss auch Datenstrukturen durchmustern
- nichtmarkierte Blöcke werden freigegeben

12.28

# Vorlesung Compilerbau (SoSe2018)

## Teil 12: Laufzeitumgebungen – Teil II

