Implementierung von `case`-Verzweigungen (1)

- Syntax:

```
CASE case_expression IN
    I_1: statement_sequence_1
    ...
    I_n: statement_sequence_n
    ELSE else-statement_sequence
END_CASE;
```

- Direkte Implementierung: (Abb. 6.39)
  - Zeitaufwand der Auswahl linear in Anzahl n der Zweige
  - annehmbar für kleine Zweigzahl (z.B. $n \leq 10$)

Implementierung von `case`-Verzweigungen (2)

```
    tmp_case_value := case expression;
    IF tmp_case_value = I_1 THEN GOTO label_1;
    ...
    IF tmp_case_value = I_n THEN GOTO label_n;
    GOTO label_else;      // or insert the code at label_else
label_1:
    code for statement sequence_1
    GOTO label_next;
    ...
label_n:
    code for statement sequence_n
    GOTO label_next;
label_else:
    code for else-statement sequence
label_next:
```

Figure 6.39 A simple translation scheme for case statements.

Implementierung von `case`-Verzweigungen (3)

- Implementierung mit Sprungtabelle:
  - seien I_{low} kleinster, I_{high} größter Wert in $\{I_1,...,I_n\}$
  - erzeuge Tabelle mit $I_{high}-I_{low}+1$ Einträgen
  - Tabelleneintrag = Sprungmarke auf den Code eines Zweiges
  - zusätzlicher Zwischencode:

```
tmp_case := case_expression;
IF tmp_case < I_low THEN GOTO label_else;
IF tmp_case > I_high THEN GOTO label_else;
GOTO table[tmp_case – I_low];
```

  - Vorteil: Zeitaufwand der Zweigwahl konstant
  - Nachteil: möglicherweise viele Einträge mit `label_else`

Implementierung von `case`-Verzweigungen (4)

- Implementierung mit Binärbaum:
  - arrangiere die Marken $\{I_1,...,I_n\}$ in balanciertem Binärbaum
  - Nachfolger eines Knotens k: $left(k)$, $right(k)$
  - Code für Knoten k:

```
label_k:
    IF tmp_case < I_k THEN GOTO label_left(k);
    IF tmp_case > I_k THEN GOTO label_right(k);
    code for statement_sequence_k;
    GOTO label_next;
```

  - wenn ausgewählter Teilbaum leer, springe zu `label_else` (belege `label_`$left(k)$ bzw. `label_`$right(k)$ mit `label_else`)

Implementierung von `while`-Schleifen (1)

- Syntax:

  `WHILE` *Boolean_expression* `DO` *statement_sequence* `END_WHILE`

- Verwendung der Implementierung für Kontrollausdrücke

  `Boolean_control_code`

  `(`*Boolean_expression*`, true_label, false_label)`

- naïve Implementierung:

```
test_label:
    Boolean_control_code
      (Boolean_expression, No_label, end_label);
    code for statement_sequence;
    GOTO test_label;
end_label:
```

  - bei nichtleerer Schleife pro Iteration zwei zusätzliche Befehle:

    (1) ein Test auf Sprung im Kontrollcode (ohne Sprung)

    (2) ein unbedingter Sprung (GOTO)

Implementierung von `while`-Schleifen (2)

- alternative Implementierung:

```
        GOTO test_label;
sequence_label:
        code for statement_sequence;
test_label:
        Boolean_control_code
            (Boolean_expression, sequence_label, No_label);
```

  - bei nicht-leerer Schleife pro Iteration nur ein (bedingter) Sprung
  - allerdings, bei erster Iteration / leerer Schleife zwei Befehle:
    (1) ein unbedingter Sprung
    (2) ein Test auf Sprung mit/ohne Sprung

- Effizienz: Prozessor-abhängig
  - mit Pipelining: Test ohne Sprung billiger als mit Sprung
  - evtl. Hardware-Feature: Sprungvorhersage

Implementierung von `for`-Schleifen (1)

- Syntax:

```
FOR i IN lower_bound .. upper_bound DO
    statement_sequence
END_FOR;
```

- offensichtliche Implementierung:

```
i := lower_bound;
tmp_ub := upper_bound;
WHILE i <= tmp_ub DO
    code for statement_sequence;
    i := i+1;
END_WHILE;
```

  - Problem falls `tmp_ub` höchster Wert seines Typs (z.B. `maxint`)
    - Overflow-Erkennung ⇒ abnormale Termination (Exception)
    - keine Overflow-Erkennung ⇒ Nichttermination
  - Lösung: nächste Folie

Implementierung von `for`-Schleifen (2)

- robustere Implementierung:

```
    i := lower_bound;
    tmp_ub := upper_bound;
    IF i > tmp_ub THEN GOTO end_label;
loop_label:
    code for statement_sequence;
    If i = tmp_ub THEN GOTO end_label;
    i := i+1;
    GOTO loop_label;
end_label:
```

  - Gleichheitstest *nach* Iteration, aber *vor* Inkrement von `i`
  - Extratest auf leere Schleife vor Eintritt in Schleifenrumpf
  - `i` wird niemals größer als *upper_bound* ⇒ kein Overflow

Implementierung von `for`-Schleifen (3)

- `for`-Schleifen mit Schrittangabe:
  - Syntax: `FOR i IN` *lb*`..`*ub* `STEP` *step* `DO ... END_FOR;`
  - *step* darf im Rumpf der Schleife nicht geändert werden
  - *step* kann negativ sein (dann i.d.R. "*ub*" < "*lb*")
- Problem (bei Test auf Gleichheit): "verpasste" Obergrenze
  - Bsp.: `FOR i IN 1..6 STEP 2 DO ... END FOR;`
- Lösung: (Abb. 6.40)
  - kein verpassen der Grenze: trenne Schleifenkontrolle von `i`
  - Schrittzähler (`tmp_loop_count`) wird dekrementiert
    - voller Bereich (`i`$\in$`[minint..maxint]`) $\Rightarrow$ `unsigned int`
    - Sprung falls (nicht) Null (häufig vorhandener Maschinenbefehl)
  - berechne Anzahl der Iterationen zur Laufzeit
    - Fallunterscheidung für positive / negative Schrittweite

Implementierung von `for`-Schleifen (4)

```
    i := lower bound;
    tmp_ub := upper bound;
    tmp_step_size := step_size;
    IF tmp_step_size = 0 THEN
        ... probably illegal; cause run-time error ...
    IF tmp_step_size < 0 THEN GOTO neg_step;
    IF i > tmp_ub THEN GOTO end_label;
    // the next statement uses tmp_ub - i
    //     to evaluate tmp_loop_count to its correct, unsigned value
    tmp_loop_count := (tmp_ub - i) DIV tmp_step_size + 1;
    GOTO loop_label;
neg_step:
    IF i < tmp_ub THEN GOTO end_label;
    // the next statement uses i - tmp_ub
    //     to evaluate tmp_loop_count to its correct, unsigned value
    tmp_loop_count := (i - tmp_ub) DIV (-tmp_step_size) + 1;
loop_label:
    code for statement sequence
    tmp_loop_count := tmp_loop_count - 1;
    IF tmp_loop_count = 0 THEN GOTO end_label;
    i := i + tmp_step_size;
    GOTO loop_label;
end_label:
```

Figure 6.40 Code generation for a general for-statement.

Implementierung von `for`-Schleifen (5)

- Optimierungspotenzial:
  - falls vorhanden, nutze Maschinenbefehl, der
    1. Variable dekrementiert und
    2. Sprung veranlasst, wenn Ergebnis größer Null

```
loop_label:
    code for statement_sequence;
    i := i + tmp_step_size;
    DECREMENT tmp_loop_count
       AND   IF GREATER_ZERO
             THEN GOTO loop_label;
end_label:
```

"`for`"–Schleifen in C

```
for(init;cond;incr) {
        body;
}
```

- ist keine `for`-Schleife im vorigen Sinne (Anzahl der Iterationen steht vor Eintritt nicht fest)
- sondern entspricht *im wesentlichen* einer `while`-Schleife:

```
init;
while (cond) {
        body;
        incr;
}
```

(!) funktioniert nicht, falls *body* `continue`–Anweisung enthält

Loop Unrolling (1)

```
FOR i := 1 TO n DO
    sum := sum + a[i];
END FOR;
```

```
FOR i := 1 TO n-3 STEP 4 DO
    // The first loop takes care of the indices 1 .. (n div 4) * 4
    sum := sum + a[i];
    sum := sum + a[i+1];
    sum := sum + a[i+2];
    sum := sum + a[i+3];
END FOR;

FOR i := (n div 4) * 4 + 1 TO n DO
    // This loop takes care of the remaining indices
    sum := sum + a[i];
END FOR;
```

eine feste Zahl von Iterationen (hier 4) wird in linearen Code expandiert

Figure 6.43 Two for-loops resulting from unrolling a for-loop.

Loop Unrolling (2)

- Loop Unrolling (Abb. 6.43)
  - eine feste Anzahl von Scheifenschritten (Unrolling-Faktor) wird in Codesequenz expandiert
  - reduziert Laufzeitaufwand für die Iterationsverwaltung
  - erleichtert Iterationen-übergreifende Optimierungen
  - Effizienz durch Ausnutzung des Prozessor-Pipelinings
  - Optimierungpotenzial Unrolling-Faktor
    - zur Übersetzungszeit bekannt: Konstante c, sodass Iterationszahl $n = c \cdot n'$
    - Wahl von c als Unrolling-Faktor vermeidet zweite Schleife (in Abb. 6.43)

Routinenaufrufe (1)

- was aufrufen?
  - oft ist der Routinenname statisch bekannt; Overloading wird vom Compiler aufgelöst
  - Ausnahmen:
    - Routinenvariablen (in C: Funktionszeiger)
      (1) Bestimmung der tatsächlichen Routine zur Laufzeit
      (2) Aufruf durch indirekten Sprung
    - dynamische Bindung von Methoden
      - Zugriff über Dispatch-Table

Routinenaufrufe (2)

- wie aufrufen?
  - Belegung des Stackframes
    - Parameter
    - Verwaltungsinformation
    - lokale Variablen
    - Arbeitsstack
    - dynamischer Bereich
  - Übergabe der Kontrolle
  - Umsetzen Framepointer
  - Umsetzen Stackpointer

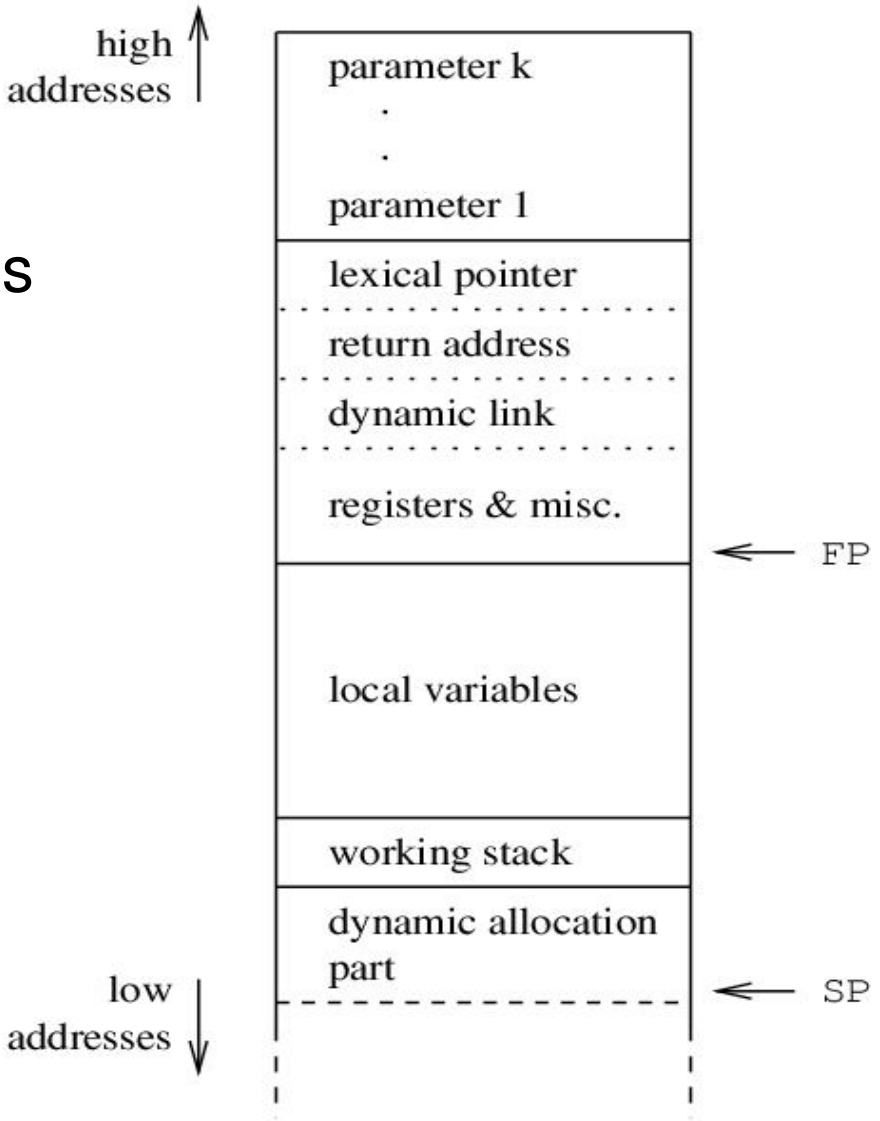

Figure 6.44 An activation record with a dynamic allocation part.

Arten der Parameterübergabe

Art	Bindungszeitpunkt L-Wert	Bindungszeitpunkt R-Wert	Sprachen
value	Aufruf	Aufruf	Ada, Pascal, Java, C, OCaml
result	Aufruf	Rücksprung	Ada, Algol W
value-result	Aufruf	Aufruf & Rücksprung	Ada, Algol W
reference	Aufruf	Zugriff	Pascal, Fortran, Java(Objekte)
name	Zugriff	Zugriff	Haskell (*), (Makros), Algol 60

- call-by-value: Kopierkosten können hoch sein
- call-by-reference:
  - Gefahr von Aliasing (mehrere Namen für das gleiche Objekt)
  - aktueller Parameter muss L-Wert sein
- call-by-name (* Variante call-by-need: ohne Mehrfachauswertung)
  - Semantik: Substitution (formal:=aktuell), Umbenennung lokaler Variablen (fkt. Sprachen: β-Reduktion mit spezieller Strategie)

Übergabearten im Vergleich

value-result vs. reference

```
{  int i = 0;


   void p(int x) {
      i = i+1;
      print(x);  }

   p(i);
}
```

value-result: Ausgabe 0
reference: Ausgabe 1

reference: Aliasing von `x` und `i` (gleiche L-Werte)

reference vs. name

```
{  int   i = 0;
   int[] a = {0,1};

   void p(int x) {
      i = i+1;
      print(x);  }

   p(a[i]);
}
```

reference: Ausgabe 0
name: Ausgabe 1

name: Substitution `x:=a[i]`

Parameter und Registerinhalte

- Parameter:
  - für variable Anzahl: letzten zuerst auf den Stack
  - effizienter: Übergabe via Register (bei RISC-Prozessoren)
  - extra Feld im Stackframe für Rückgabewert
  - Parameter dynamischer Größe: Zeiger auf Heap-Objekt
- Sicherung / Wiederherstellung von Registerinhalten:
  - *callee-saves*: Aufgerufener rettet alle Inhalte, die er verändert
    - Vorteil: Codeaufwand nur am Routinenanfang/-ende
    - Nachteil: Laufzeitaufwand, viele Inhalte zu retten
  - *caller-saves*: Rufer rettet alle Inhalte, die er noch braucht
    - Vorteil: oft weniger Laufzeitaufwand als bei callee-saves
    - Nachteil: Codeaufwand bei jedem Aufruf

Überlappung benachbarter Stackframes

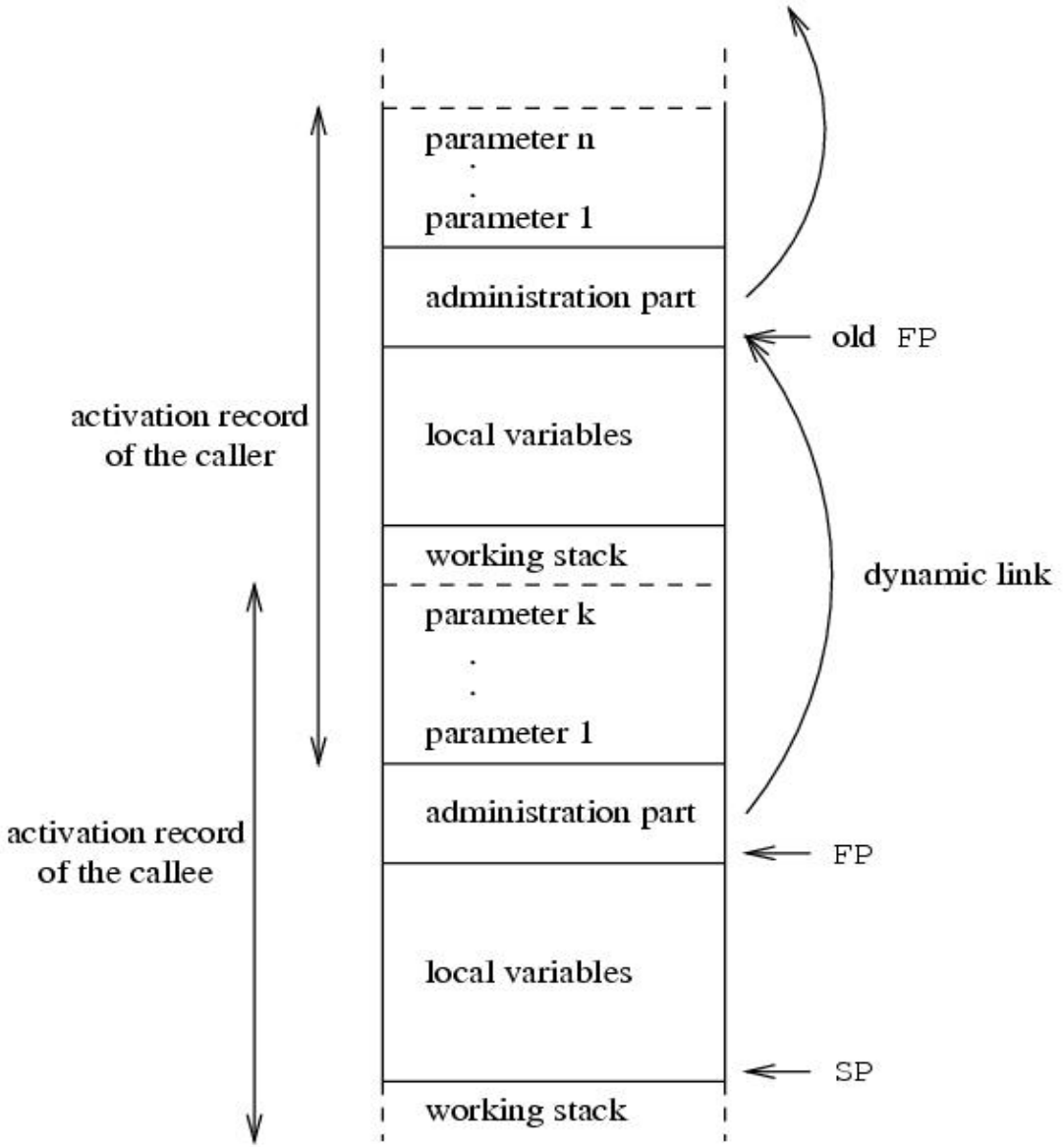

Figure 6.45 Two activation records on a stack.

Variablen und Parameter im Stackframe

- lokale Variablen:
  - zur Lagerung benannter Zwischenwerte
  - Bereichsgröße und -struktur zur Compilezeit bekannt
- Working Stack:
  - zur Lagerung namenloser Zwischenwerte
  - maximale Ausdehnung zur Compilezeit bekannt
- dynamischer Bereich (notwendig: am Ende des Stacks)
  - Vorteile:
    - keine Heapallokation für Parameter dynamischer Größe
    - schnelle Reservierung (C99: `alloca`)
    - Freigabe automatisch mit Freigabe des Stackframes
  - Nachteil:
    - Working Stack braucht separaten Stackpointer
  - nicht möglich, wenn Framegröße bei Aufruf fixiert werden muss (SPARC-Prozessor)

Rückgabe von Werten

- Wert von Wortgröße: über Register oder Parameterbereich
- Alternativen bei größerem Wert
  - Compiler reserviert Platz im caller-Datenbereich und übergibt Adresse dieses Bereichs an callee
  - callee allokiert Platz auf dem Heap und gibt Adresse zurück
  - !gefährlich: callee allokiert Platz in seinem Stackframe, caller muss den Rückgabewert sichern bevor Stackframe überschrieben wird