Aufrufsequenz (1)

1. Belege das Aktivierungssegment (AS) des Callees auf dem Laufzeitstack
2. Berechne die Werte der aktuellen Parameter und übertrage sie in das AS.
   Die Übertragung ist vermeidbar; überlappe Working Stack des Callers mit Parameterregion des Callees (Abb. 6.45):

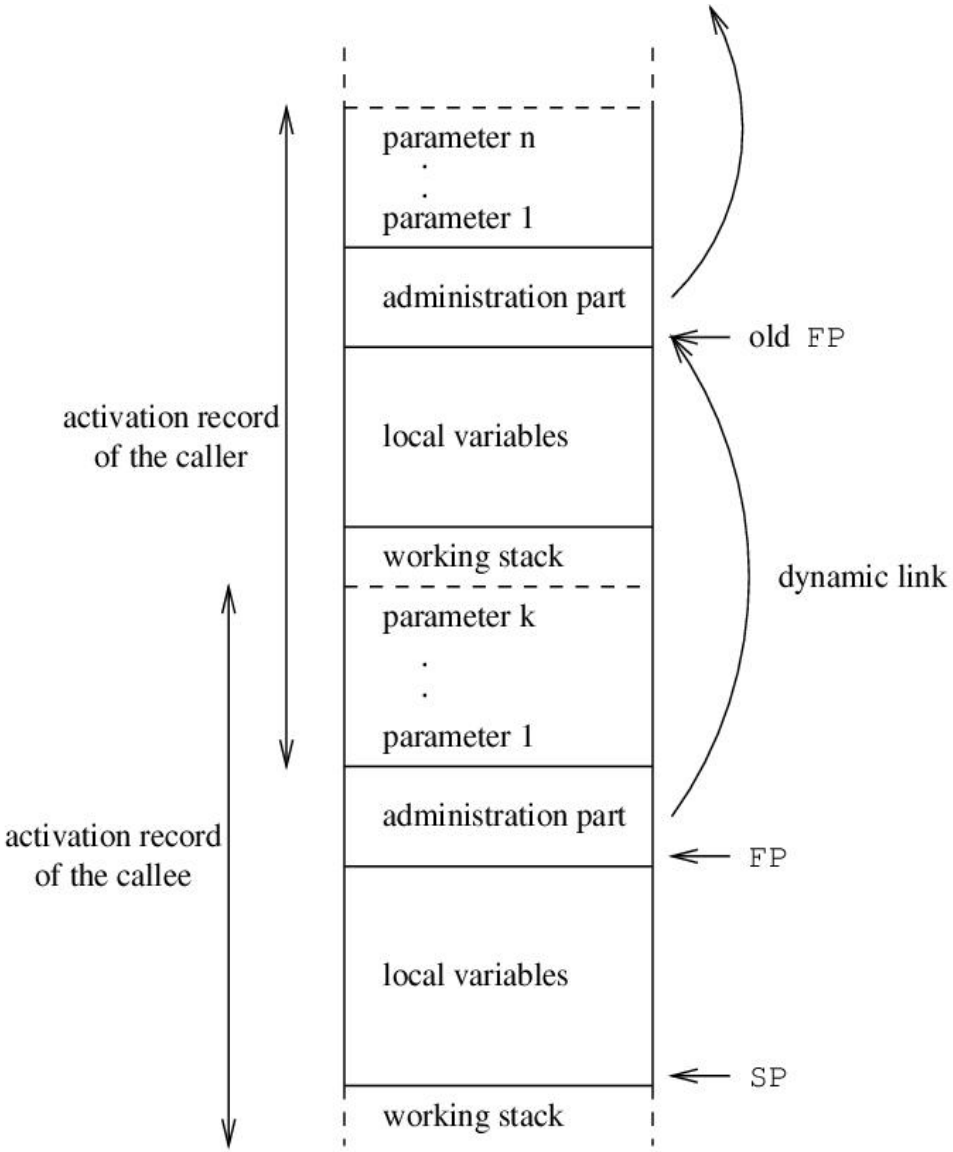


Figure 6.45 Two activation records on a stack.

Aufrufsequenz (2)

1. Belege das Aktivierungssegment (AS) des Callees auf dem Laufzeitstack
2. Berechne die Werte der aktuellen Parameter und übertrage sie in das AS.
3. Trage die administrative Information in das AS ein (teils vor, teils nach Übergabe der Kontrolle an den Callee)
   - Framepointer des Callers (dynamischer Link)
   - Rücksprungadresse (Sicherung Teil des `call`-Befehls des Pentium-Prozessors)
   - Framepointer der textuell umgebenden Routine (statischer Link)
   - alter Wert des Stackpointers (vor Allokation des AS)
   - alte Registerinhalte (die vom Caller nach Beendigung der Routine wieder gebraucht werden)

Aufrufsequenz (3)

1. Belege das Aktivierungssegment (AS) des Callees auf dem Laufzeitstack
2. Bestimme die aktuellen Parameter und übertrage sie in das AS
3. Trage die administrative Information in das AS ein / Übergabe der Kontrolle an den Callee
4. Sichern der Callee-saves Register
5. Callee setzt Framepointer FP auf sein AS (Rand Verwaltungsbereich)
6. Callee setzt Stackpointer SP an das Ende des AS

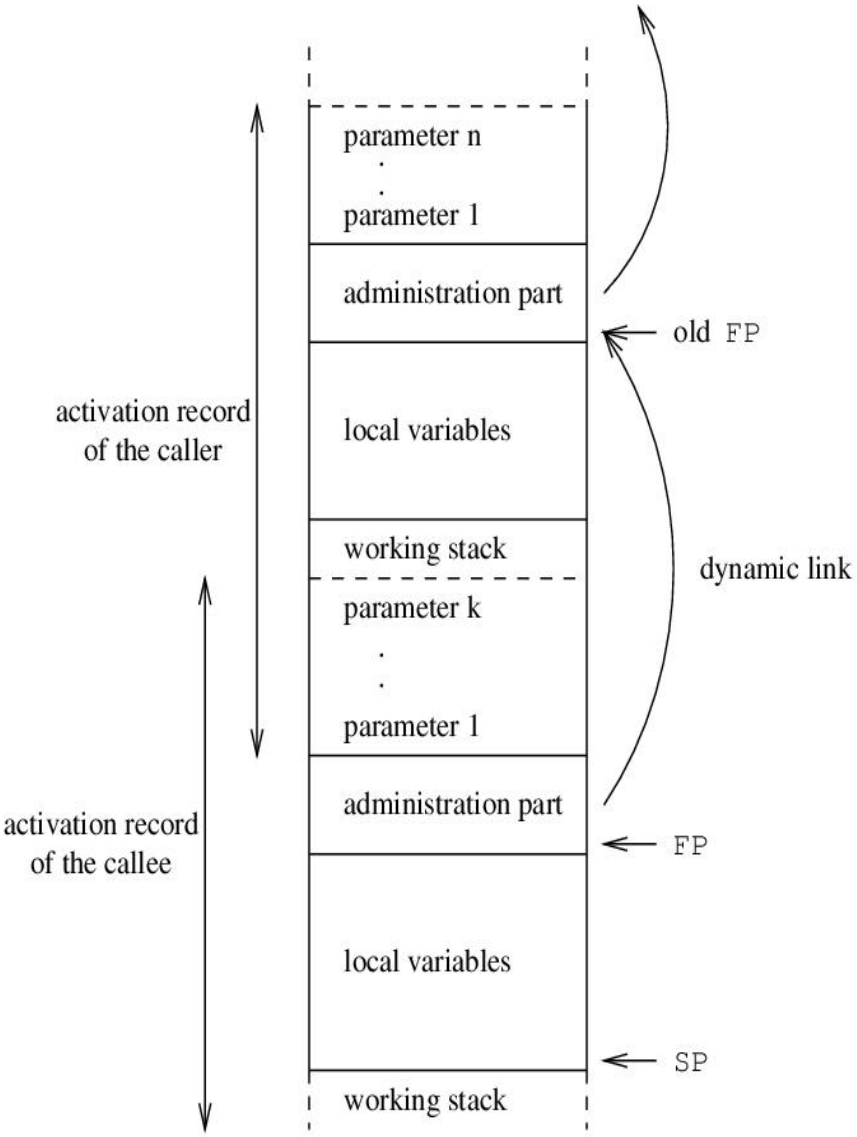


Figure 6.45 Two activation records on a stack.

Rückkehrsequenz

1. Speichere den Rückgabewert an dem dafür vorgesehenen Platz
2. Setze den Stackpointer SP auf die Stelle des Framepointers FP (Rand des Verwaltungsbereichs)
3. Stelle den alten Zustand der Callee-saves-Register wieder her (z.B. setze FP auf das Aktivierungssegment des Callers)
4. Übergib die Kontrolle an den Caller (Pentium-Rückkehrbefehl `ret` nimmt Sprungadresse aus `mem[SP]`)
5. Korrektur des Stackpointers (Frame des Callees wird frei)

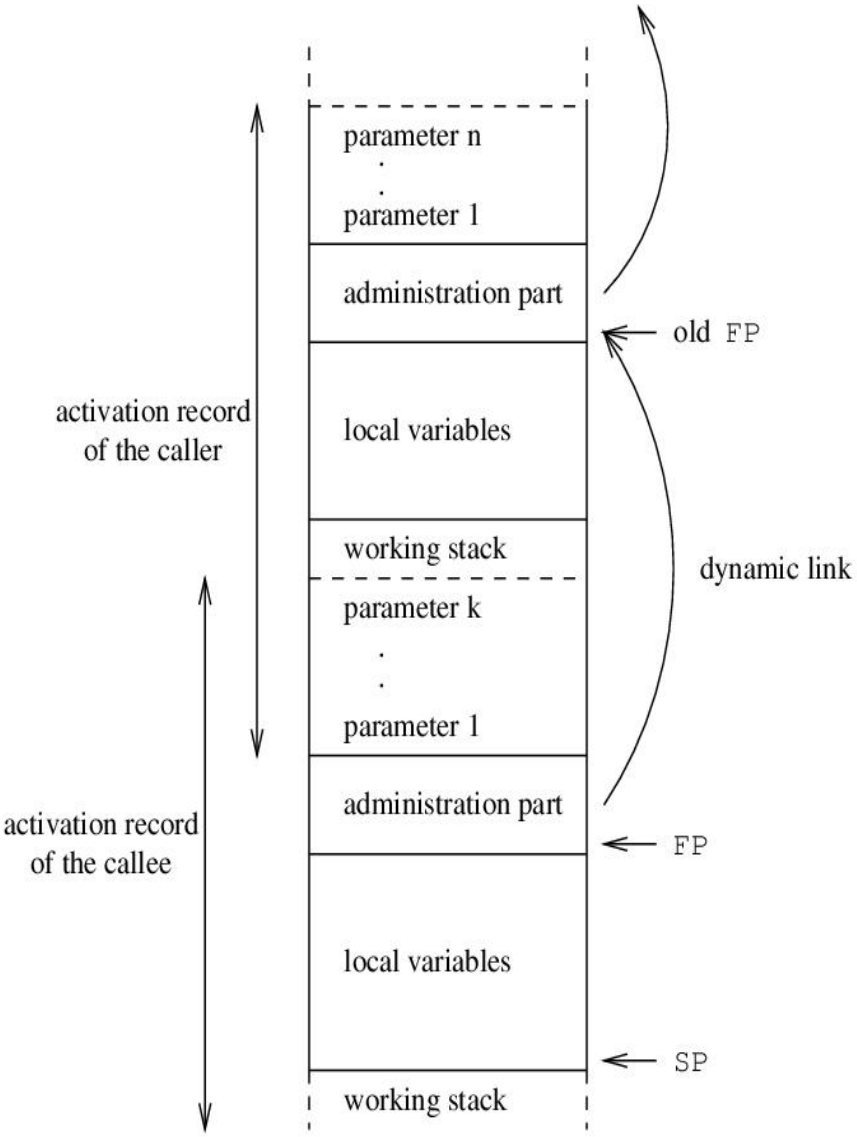


Figure 6.45 Two activation records on a stack.

Heraussprünge (Deep Exits)

- Problem: welche Register müssen in den durch den Heraussprung verschwindenden Aktivierungssegmenten wieder hergestellt werden?
- Lösungen
  1. Option: rette in jedem Segment einen festen Registersatz (u.U. von der Hardware unterstützt)
  2. Option: notiere im Segment, welche Register gerettet wurden
  3. Option: benutze keine Register in einer Routine *R*, die Ziel eines Heraussprungs ist
     - nötig: *R* muss zu Beginn selbst alle Register retten und am Ende wiederherstellen
     - Vorteil: Keine Wiederherstellung von Registerinhalten für alle zwischendurch aufgerufenen Routinen nötig

Laufzeitfehlerbehandlung

- Fehlerquellen: sehr unterschiedlicher Art
  - Integerüberlauf
  - Arrayzugriff ausserhalb der Grenzen
  - versuchte Speicherreservierung kann nicht erfüllt werden
  - Hardwarefehler (z.B. durch Spannungsabfall)
- Fehlererkennung durch
  - erzeugten Code
  - Laufzeitsystem
  - Betriebssystem
  - Hardware
  - ... keine ...
- Behandlung
  - Erkennung
  - Bearbeitung

Fehlererkennung

- Betriebssystem: Interrupts
  - Hardware-verursacht
    - Division durch Null, Dereferenzierung eines Nullzeigers, etc.
    - automatische Bearbeitung durch System-Interrupthandler
  - Anwender-verursacht
    - CTRL-C, Mausklicks, etc.
    - Schnittstellen zur Bearbeitung durch das Anwendungsprogramm
- Laufzeitsystem: Tests
  - verursachen oft erheblichen Overhead
  - Diagnostik z.T. über Statusregister
  - ansonsten muss der Compiler Code für die Tests generieren, Bsp.:

```
if ((val<lowerbound) || (value>upperbound))
        throw(range_error);
```

Bsp.: Signal-Handler mit Heraussprung

```
#include <setjmp.h>

jmp_buf jmpbuf;           /* type defined in setjmp.h */

void handler(int signo) {
    printf("ERROR, signo = %d\n", signo);
    longjmp(jmpbuf, 1);
}

int main(void) {
    signal(6, handler);          /* install the handler ... */
    signal(12, handler);         /* ... for some signals */
    if (setjmp(jmpbuf) == 0) {
        /* setting up the label for longjmp() lands here */
        /* normal code: */
        ...
    } else {
        /* returning from a call of longjmp() lands here */
        /* exception code: */
        ...
    }
}
```

Figure 6.46 An example program using `setjmp/longjmp` in a signal handler.

Fehlerbearbeitung

- Signale
  - primitiver Mechanismus bei Auftreten einer Fehlerbedingung
  - mögliche Fehler werden in Klassen kategorisiert
  - Installation des Handlers: signal (Klasse, Handler)
  - Aufruf des Handlers: raise(Klasse)
  - nach Behandlung Rück- oder Heraussprung (Abb. 6.46)
- Ausnahmen
  - Handler hat Zugriff auf Kontext der Fehlerstelle
  - Kontext wird vom Anwenderprogramm definiert
  - ein Kontext kann mehrere Handler für verschiedene Fehlerklassen haben

Implementierung einer Ausnahmebehandlung (1)

- Maßnahmen: für Block, in dem Ausnahmen auftreten können
  - Ausnahmetabelle mit Einträgen (Klasse, Handler)
  - Zeiger auf die Tabelle im Aktivierungssegment des Blocks
  - Zielcode für jeden Handler, abgeschlossen mit Heraussprung zum Ende des assoziierten Blocks
- beim Auftreten einer Ausnahme
  - Aufrufkette wird zurückverfolgt bis passender Handler gefunden
  - Fortsetzung mit dem Code des Handlers

Implementierung einer Ausnahmebehandlung (2)

- Handlersuche: bei Auftreten einer Ausnahme *E*
  1. finde die Handler-Tabelle für das aktuelle Aktivierungssegment
  2. suche nach Handler für *E*:
     - wenn vorhanden (Handler sei *H*), setze in Schritt 3 fort
     - wenn nicht vorhanden, gib aktuelles Segment frei und starte neu in Schritt 1 mit dem Segment des Callers
     - wenn kein Caller-Segment existiert, übergib an das Betriebssytem oder terminiere das Programm mit einer Fehlermeldung
  3. nicht-lokales goto zum Handler *H*, beendet alle Segmente, die keinen Handler für *E* haben
- Nachteil: Tabellenkonstruktion verteuert Blockeintritt

Implementierung einer Ausnahmebehandlung (3)

- Alternative: eine *globale* Handlertabelle
  - Einträge bestehen aus Quadrupeln
    - Name der Ausnahme
    - Sprungmarke des Handlers
    - Anfangsadresse des assoziierten Blocks
    - Endadresse des assoziierten Blocks
  - Handlersuche bei Auftreten einer Ausnahme

    1. vergleiche PC mit den Start- und Endadressen in der Tabelle
    2. wenn passendes Intervall gefunden,
       - aktiviere den Handler mit nicht-lokalem goto, sonst
       - suche bzgl. der Rücksprungadresse (PC im Caller beim Aufruf), usw.
  - keine zusätzlichen Kosten bei Blockeintritt
  - teurer in der *Behandlung* einer Ausnahme

Codegenerierung für Module (1)

- Spezielle Eigenschaften
  - Zusammenfassung von Programmteilen zu
    - Programmbibliotheken
    - abstrakten Datentypen
  - separat kompilierbar
  - Ähnlichkeiten zu Objekten, aber z.B. nicht dynamisch erzeugbar
  - Namensgenerierung
    - gib dem i.d.R. flachen Namensraum der *Zielsprache* eine Struktur
    - Zielsprache erlaubt i.d.R. zusätzliche Symbole in Bezeichnern (z.B. $)
    - benutze diese zur Bildung eindeutiger Namen in einem flachen Raum

Codegenerierung für Module (2)

- jedes Modul muss genau einmal initialisiert werden
  - Bsp.: A benutzt B und C, B benutzt C, aber B darf nicht zweimal initialisiert werden
  - teste während der Initialisierung
- es darf keine Zyklen im Abhängigkeitsgraphen geben

(a) Compiler sucht nach Zyklen in Spezifikationen (über die Importlisten der Module)

(b) bei separater Kompilation

- Implementierungen können Spezifikationen anderer Module importieren
- dadurch entstehende Zyklen sind schwer zu finden

1. Lösung: Tests bei der Modulinitialisierung
2. Lösung: zusätzliche Compilerphase, die am Ende nach zyklischen Abhängigkeiten sucht

Codegenerierung für Generics (1)

- Generics: parametrisierte Programmteile
  - i.d.R. Routinen, Module oder Klassen
  - generische Parameter i.d.R. Typen
- Instanzierung durch Expansion
  - Code für den generischen Programmteil existiert nur als AST
  - für jede Instanzierung wird nach einer Kopierregel ein eigener Zielcode erzeugt
  - Namensgenerierung: z.B. Konkatenation von generischem und Versionsnamen
  - Eigenschaften
    - Vorteil: kein Laufzeitoverhead
    - Gefahr: Codeexplosion (exponenziell in Parameteranzahl)
    - nicht möglich bei Sprachen, bei denen sich potenziell unbeschränkt viele Instanzen zur Laufzeit entfalten können (Bsp. Haskell, polymorphe Rekursion in Datentypen)

Codegenerierung für Generics (2)

- Instanzierung mit Dopevektoren
  - der generische Programmteil wird kompiliert
  - ihm werden zur Laufzeit die aktuellen Parameter übergeben
  - ein Typparameter wird durch einen Dopevektor repräsentiert, der Verweise auf den Code jeder mit dem Typ assoziierten Operation enthält (Abbildung 6.47)
  - Eigenschaften
    - Vorteil: keine Code-Explosion
    - Nachteil: Laufzeit-Overhead
      - dynamische Allokation, wo sonst statische ginge
      - Inlining nicht möglich (Operation steht erst zur Laufzeit fest)

Codegenerierung für Generics (3)

- Typische Operationen
  - Allokation und Freigabe
  - Initialisierung
  - Zuweisung
  - evtl. Vergleich zweier Elemente des Typs

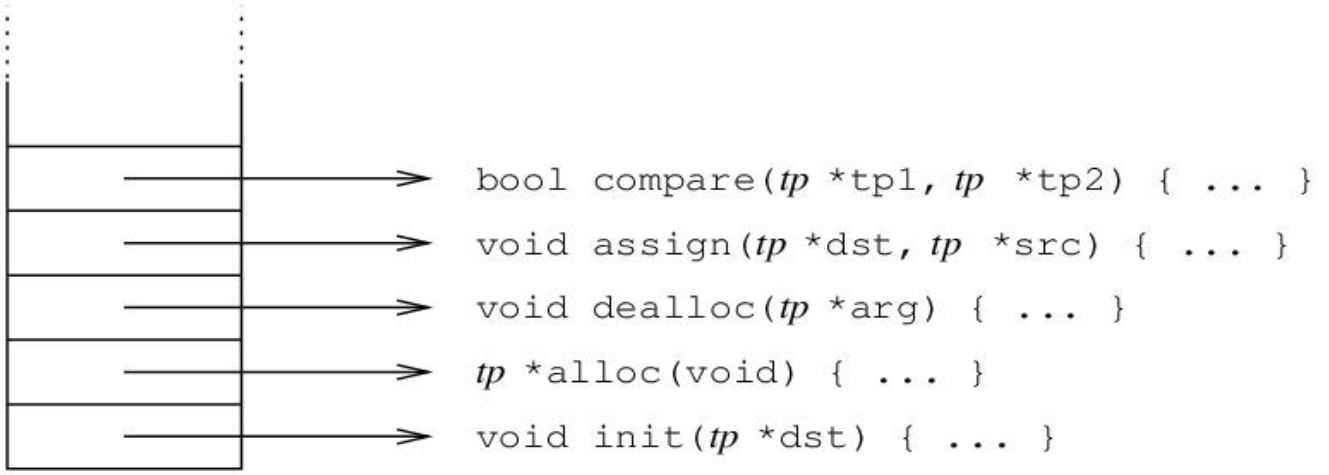


Figure 6.47 A dope vector for generic type *tp*.

Codegenerierung für Generics (4)

- verwandtes, extremes Beispiel: Haskell-Zwischencode (Core)
  - Funktionen im Core parametrisiert in polymorphen Typen
    - primäres Ziel: Typconstraints für Optimierungen erhalten
    - formales Modell: "*second-order lambda calculus*"
    - nicht zu verwechseln mit Polytypic/Generic Haskell
  - Inlining im Spezialfall möglich
  - zusätzliche Schwierigkeit bei Implementierung:
    Einschränkung von polymorphen Typen auf Typklassen,
    pro Typklasse:
    - Programmierer-definierte überladene Funktionen, vom Zweck vergleichbar den Funktionen in einem Dopevektor

Kapitel 4: Paradigma-unabhängige Codesynthese

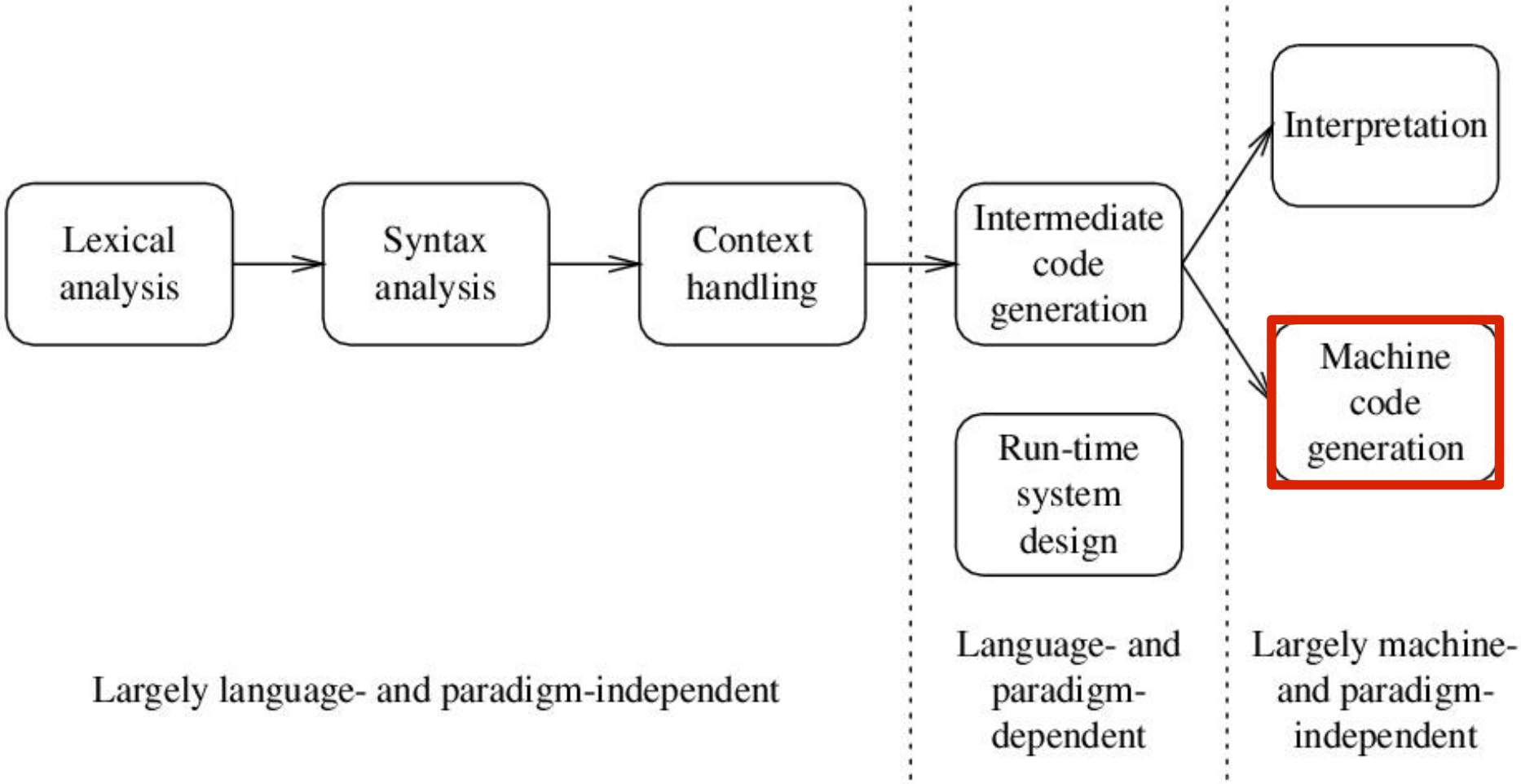


Figure 4.1 The status of the various modules in compiler construction.

Aufwandsebenen für Codegenerierung

- keine Codegenerierung:
  - Interpretervorlauf auf dem AST bis zur ersten Laufzeitabhängigkeit
- triviale Codegenerierung (in SIPS I bereits behandelt):
  - modif. Interpreter: ersetzt jede Auswertung durch Code dafür
  - Optimierung durch Separation von Compilezeit/-Laufzeitwerten (Staging) und partielle Auswertung
    - SIPS I: Native-Code für Stufe 1 mit MetaOCaml-Compiler
- einfache Codegenerierung:
  - kleine Auswahl an Maschinenbefehlen / Adressierungsarten
  - Heuristiken für Registervergabe
- fortgeschrittene Codegenerierung
  - optimale Befehlsauswahl durch Dynamic Programming (BURS)
  - optimale Registervergabe durch Graph Coloring

Varianten von Zwischencoderepräsentationen

- frühes Stadium: (abstrakter) Syntax-Baum (Appel-Buch, Kap. 4)
  - Informationsspeicherung (Typen, Werte) in Knoten
  - Kontrollfluss-Information (für imperative Sprachen):
    - Threading (Verzeigerung quer zur Baumstruktur), daraus ableitbar: Kontrollflussgraph (für Registervergabe)
  - Optimierung: gerichtete ayklische Graphen für gemeinsame Teilausdrücke
- mittleres Stadium: Interm. Repr. Trees (Appel-Buch, Kap. 7)
  - Knoten: Statements, Expressions, Labels, (un)bedingte Sprünge
- spätes Stadium: operationale Repräsentationen:

  (a) abstrakte Stackmaschine: `push`, `pop`, `add` (ohne Argumente)

  (b) abstrakte Registermaschine:
    - Dreiadresscode: Operator, 2 Operandenregister, 1 Zielregister
    - Jouette-Assembler (Appel-Buch, Kap. 9)

Aspekte der Codesynthese

- Befehlsauswahl
  - Ersetzung von Syntaxbaumteilen durch Maschinenbefehle
  - CISC-Prozessor: Bestimmung der Befehlsparameter, Variabilität durch komplexe Adressierungsarten
  - RISC-Prozessor: wenige und einfache, aber schnelle Befehle
- Registervergabe
  - welche Temporärwerte in Registern speichern, in welchen?
  - Register sind oft eine knappe Ressource
  - gekoppelt mit Befehlsauswahl
    - jeder Befehl stellt Bedingungen an verwendete Register
    - komplexer Befehl könnte Hilfsregister sparen
- Befehlsanordnung (topologische Sortierung)
  - Vermeidung von Umkopieren
  - Ausnutzung von Spezialregistern (z.B. Bedingungscoderegister)

Baumersetzung (1)

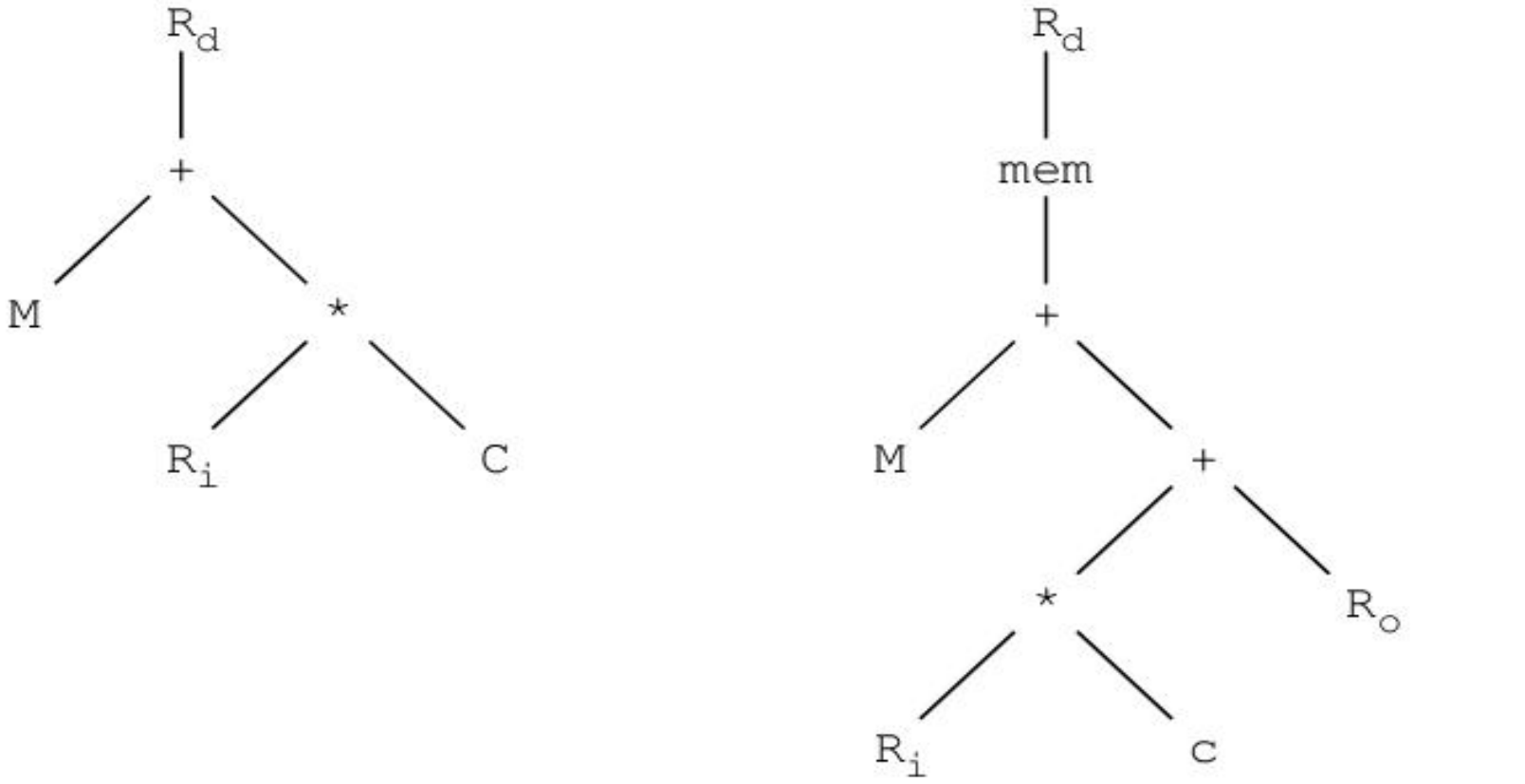
Beispiel: `a :=(b[4*c+d]*2)+9`

- Speicherreferenzen einsetzen
- Teilbaum mit Baum für Baum für Maschinenbefehl matchen: sei `M` ein Array mit Elementen der Bytegröße `c` (Abb 4.10)
  - `Load_Addr M[`R_i`],C,`R_d (Pentium: `leal`)

    lade Adresse des R_i-ten Elements von M in R_d
  - `Load_Byte(M+`R_O`)[`R_i`],C,`R_d (Pentium: `movsbl`)

    lade den Inhalt des R_i-ten Elements von `M` plus Offset R_O in R_d
- Teilbaum zu Maschinenbefehl reduzieren (Abb. 4.9)

Baumersetzung (2)



Load_Address M[R_i],C,R_d

Load_Byte (M+R_o)[R_i],C,R_d

Figure 4.10 Two sample instructions with their ASTs.

Baumersetzung (3)

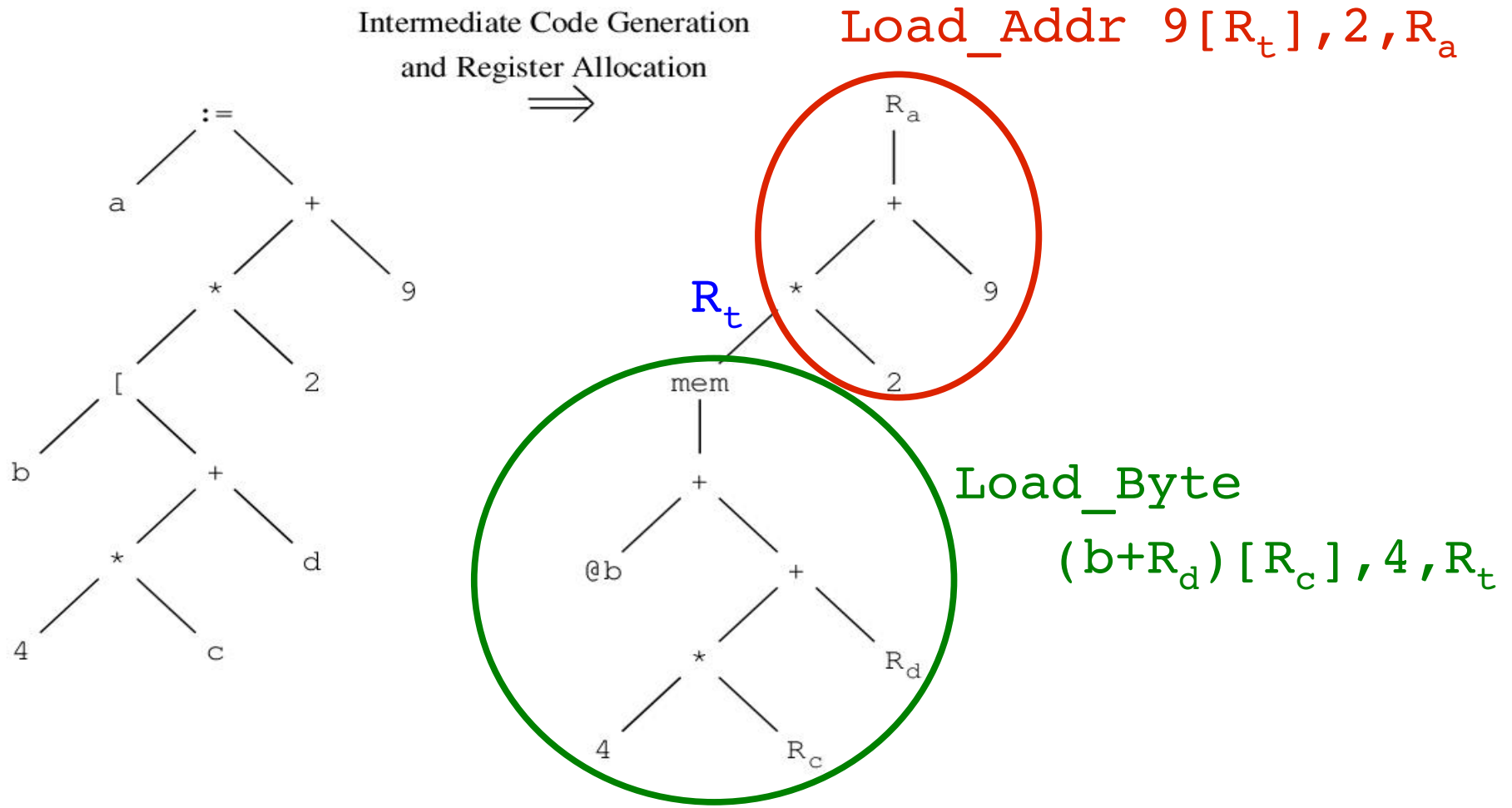


Figure 4.9 Two ASTs for the expression `a := (b[4*c + d] * 2) + 9`.