Zwischencode (1)

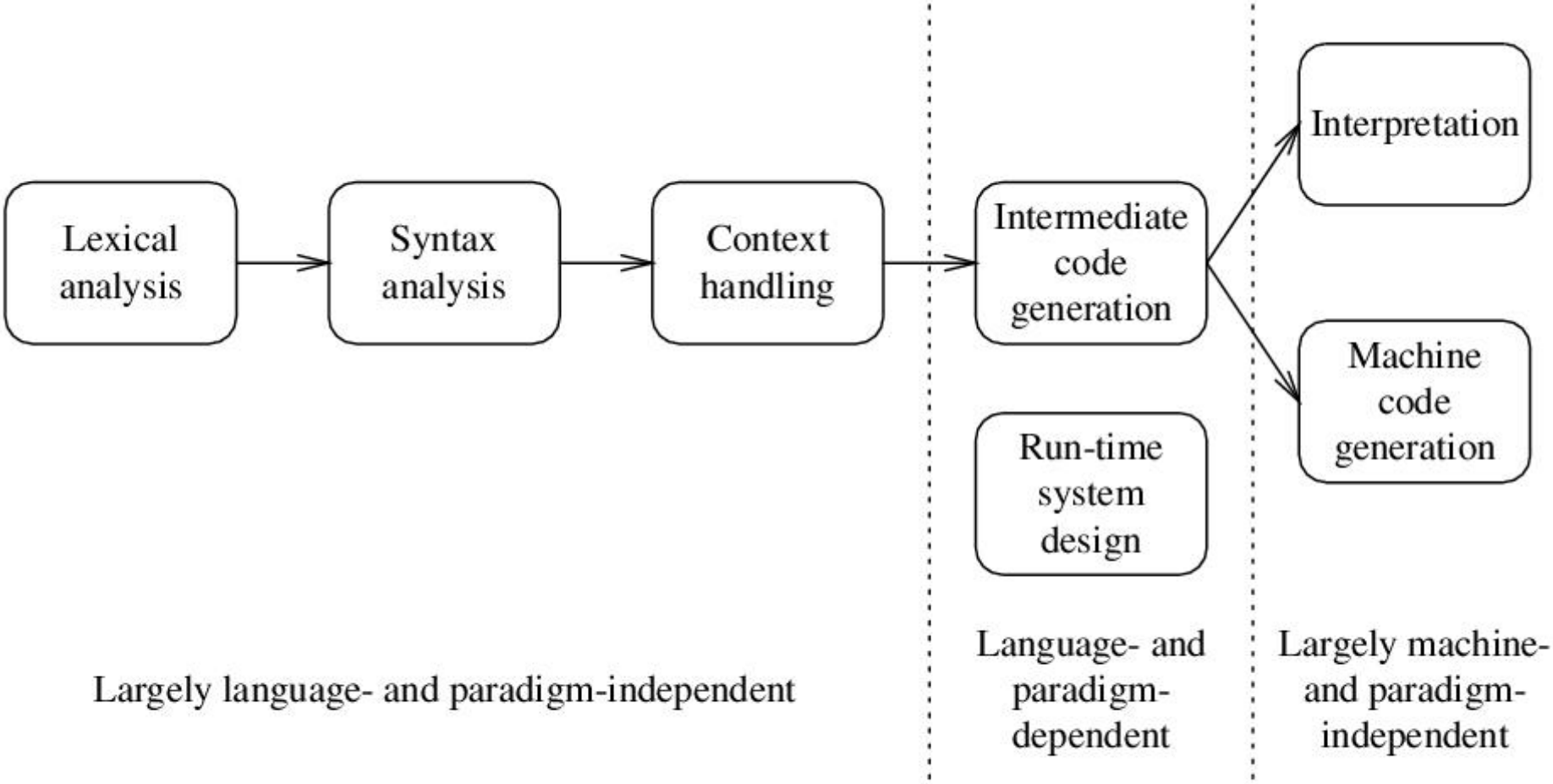

Figure 4.1 The status of the various modules in compiler construction.

Zwischencode (2)

- Form: zusammengesetzt aus elementaren Operationen
- Beispiel (Dreiadresscode):

```
100:   f := f*n
101:   n := n-1
102:   if n>0 goto 100
```

- Ausgangspunkt für
  - Codegenerierung
  - eine Form der Interpretation (iterative Interpretation)
- Erzeugung:
  - Threading des abstrakten Syntaxbaums
  - stark abhängig vom Programmierparadigma (behandelt in der Fortsetzungsvorlesung)

Zwischencode (3)

- Zwischencodebaum
  - mehr Knoten als zugehöriger abstrakter Syntaxbaum
  - weniger und einfachere Knotenarten
  - oft weniger Struktur (z.B. verzeigertes Array)
- Knotenarten
  - Anweisungen
    - Ausdrücke und Zuweisungen
    - Programmaufrufe, Prozedurköpfe und Return-Statements
    - bedingte und unbedingte Sprünge
  - Verwaltungsoperationen
    - Speicherverwaltung für globale Variablen
    - Formularverwaltung für Blöcke und Prozedurrümpfe
    - Informationen für das Linken von Programm-Modulen

Arten der Interpretation

- rekursive Interpretation
  - durchläuft (rekursiv) den abstrakten Syntaxbaum
  - implementiert Sprachkonstrukte direkt
  - Datenobjekte werden in Graphstrukturen gehalten
- iterative Interpretation
  - durchläuft den Zwischencode, i.d.R. ein Array von Operationen für eine abstrakte Maschine
  - benötigt ein Threading des abstrakten Syntaxbaums
  - Datenobjekte werden in Stack und Heap gespeichert
  - spezielle Form: Bytecode (z.B. in Java, Ocaml)
    - fließende Übergänge zur kompilierten Ausführung
    - Operationen für Ein-/Ausgabe im Zwischencode
    - just-in-time Compiler (JIT) erzeugt und startet Maschinencode abschnittsweise zur Laufzeit

Rekursive Interpretation (1)

- Vorgehen
  - eine Funktion pro Knotentyp des abstrakten Syntaxbaums
  - Beispiel: arithmetische Ausdrücke (später)
- intuitive Datendarstellung
  - aufwändig, dynamische Datenstrukturen
  - Beispiel: komplexe Zahlen

Figure 4.2 A value of type `Complex_Number` as the programmer sees it.

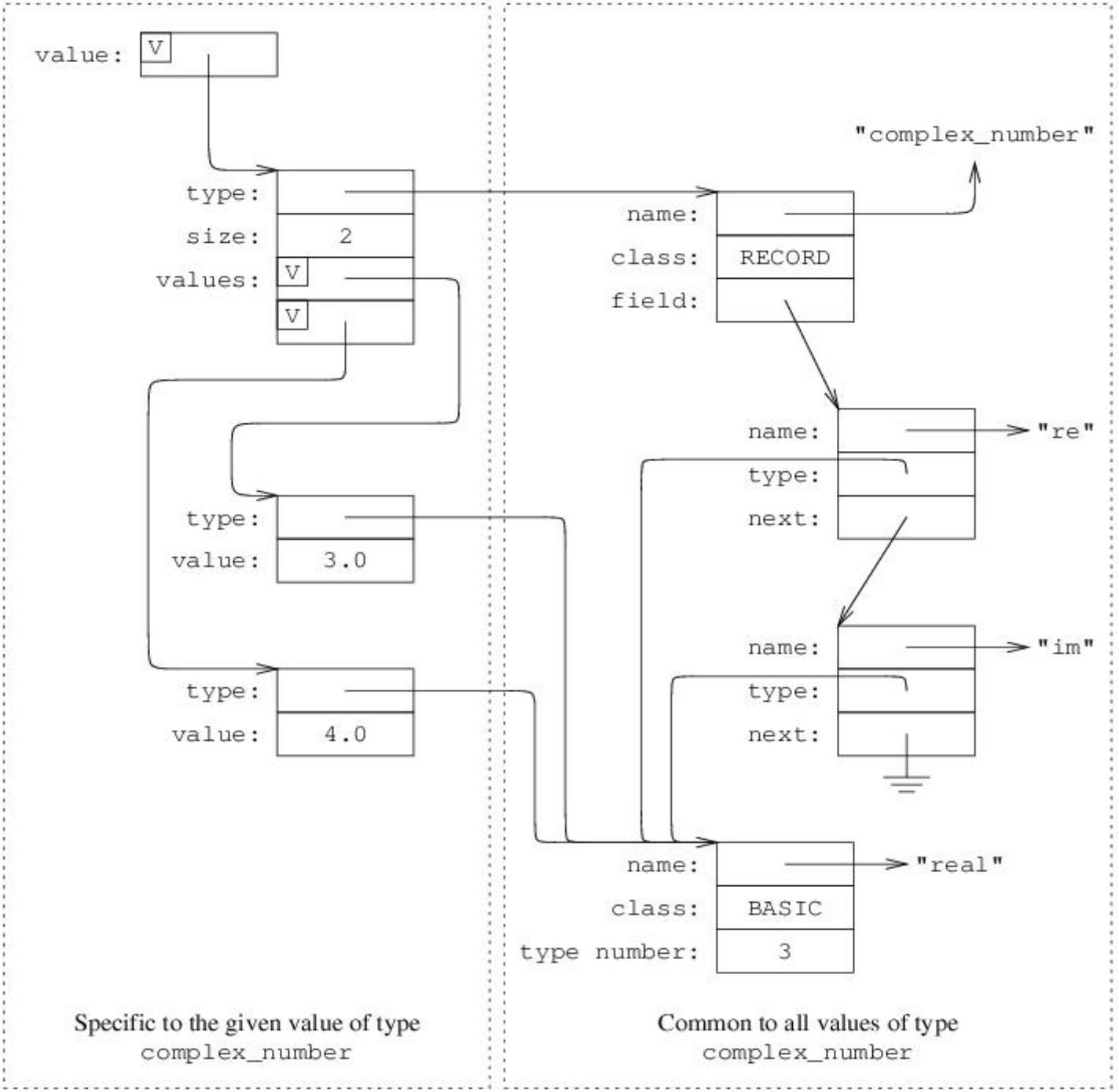

Figure 4.3 A representation of the value of Figure 4.2 in a recursive interpreter.

Rekursive Interpretation (2)

- intuitive Datendarstellung
  - aufwändig, dynamische Datenstrukturen
  - Beispiel: komplexe Zahlen
  - teils wertspezifisch, teils typspezifisch
  - Vereinfachung der Benutzung,

    (a) Vorschlag im Buch (ausgehend z.B. von C)
    - jeder Wert hat eigene Kopie aller Information

    (b) funktionale Implementierungssprache (Haskell, OCaml):
    - automatisches Sharing gemeinsam genutzter Information
    - automatische Kopie, wo nötig

Rekursive Interpretation (3)

- Steuerung des Interpreters durch Statusindikator
  - wird an bestimmten Stellen getestet
  - Komponenten
    - Mode: Normal, Jump, Exception, Return, diverse Fehler
    - Value: Sprungmarke, Ausnahmeinfo, Rückgabewert, ...
    - Debugginginformation: Name der Programmdatei, Zeilennummer, ...

Rekursive Interpretation (4)

- Steuerung des Interpreters durch Statusindikator
- Beispiel (Return with expression)
  - Interpretationsfunktion

```
PROCEDURE Elaborate return with expr_stmt (Rwe node):
    SET Result TO Evaluate_expression (Rwe node.expression)
    IF Status.mode /= Normal_mode: RETURN;
    SET Status.mode TO Return_mode;
    SET Status.value TO Result;
```

  - Status nicht normal ⇒ sofortiger Rücksprung

Rekursive Interpretation (5)

```
PROCEDURE Elaborate if statement (If node):
    SET Result TO Evaluate condition (If node .condition);
    IF Status .mode /= Normal mode: RETURN;
    IF Result .type /= Boolean:
        ERROR "Condition in if-statement is not of type Boolean";
        RETURN;
    IF Result .boolean .value = True:
        Elaborate statement (If node .then part);
    ELSE Result .boolean .value = False:
        // Check if there is an else-part at all:
        IF If node .else part /= No node:
            Elaborate statement (If node .else part);
        ELSE If node .else part = No node:
            SET Status .mode TO Normal mode;
```

Figure 4.4 Outline of a routine for recursively interpreting an if-statement.

Rek. Interpr. (6a), Informationsverwaltung

- Symboltabelle
  - Hauptsammelstelle für Informationen
  - oft Stack- und/oder Hashzugriff vorgeschaltet (Abb. 6.2, 6.4-5)

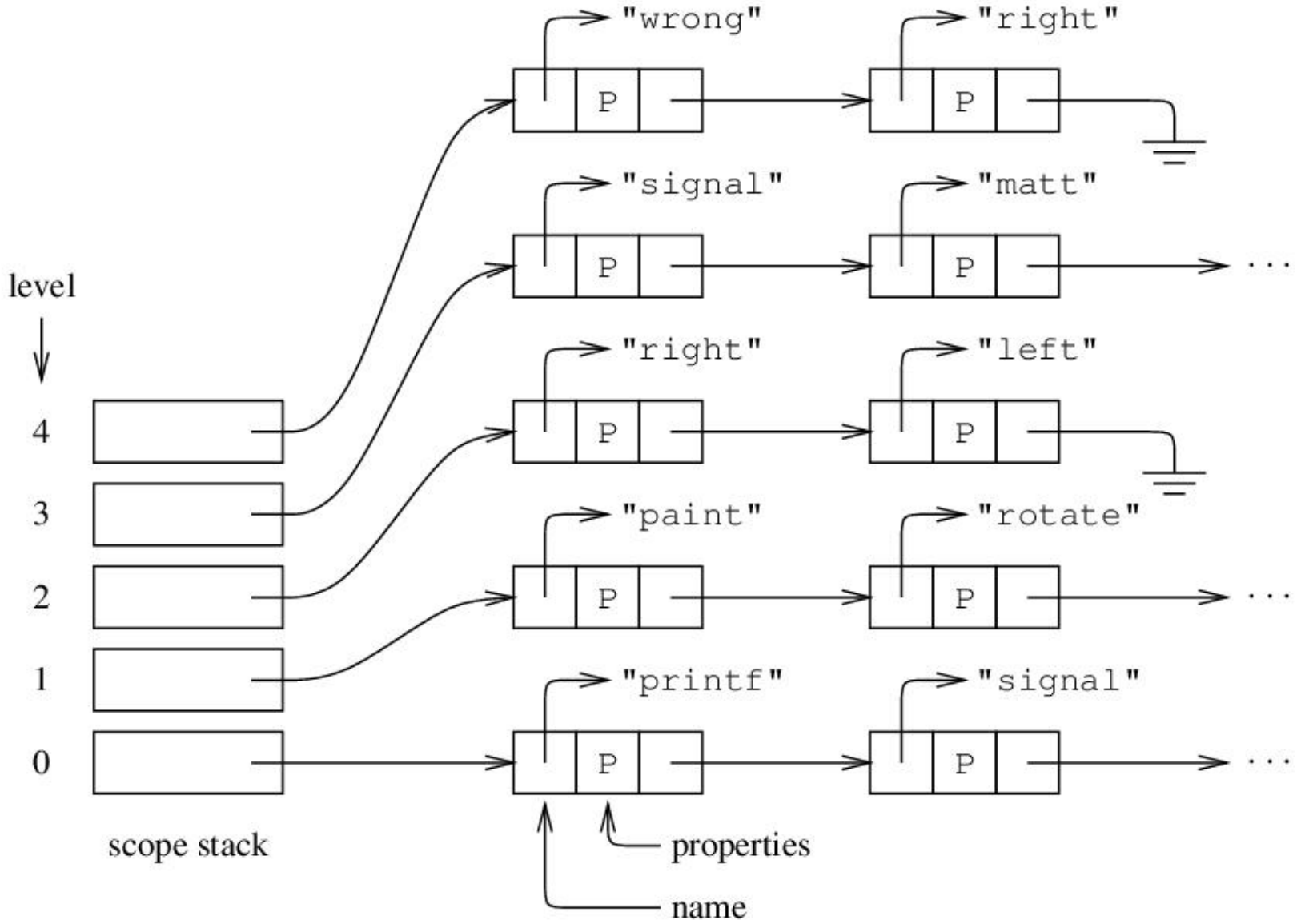

Figure 6.2 A naive scope-structured symbol table.

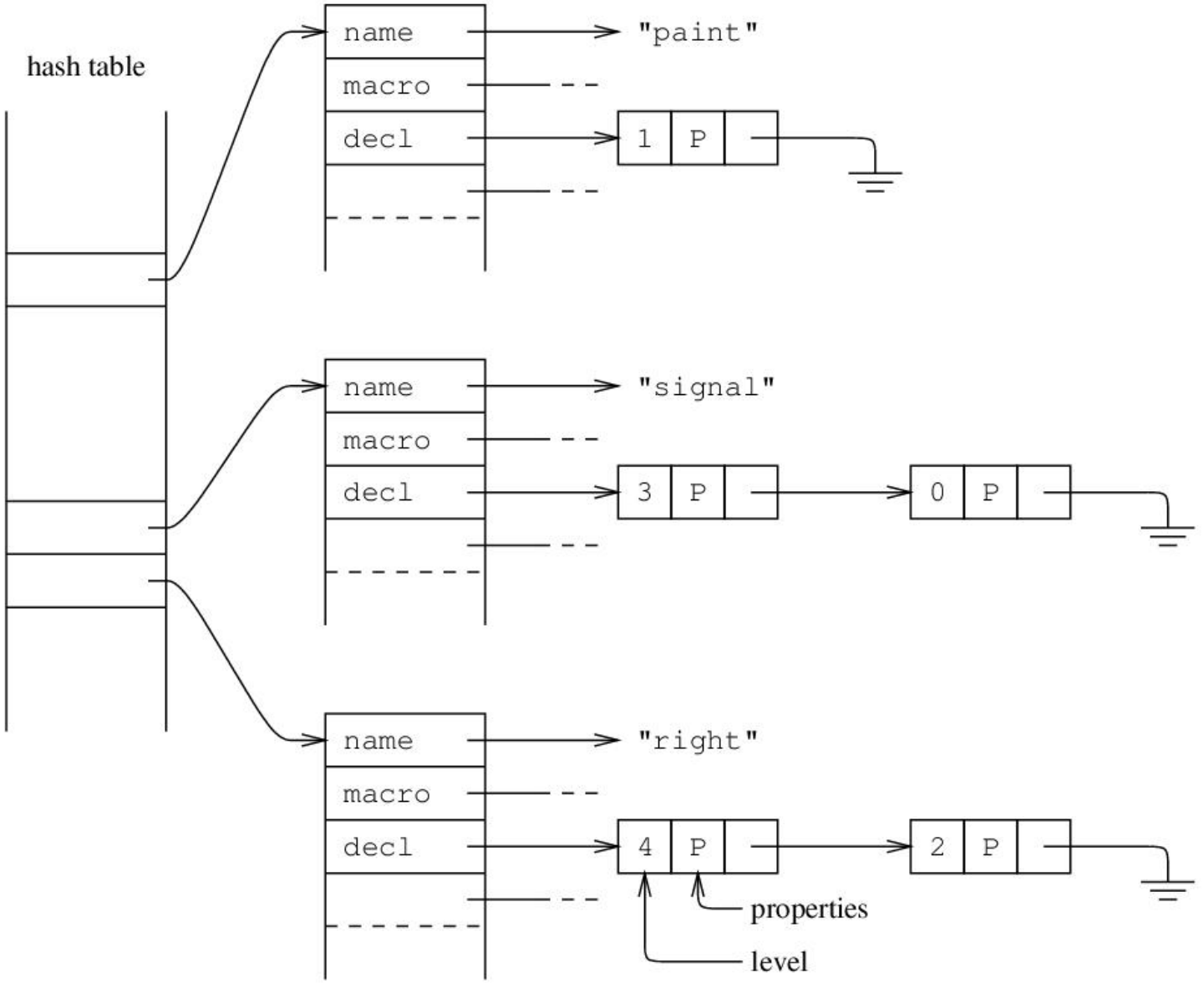

Figure 6.4 A hash-table based symbol table.

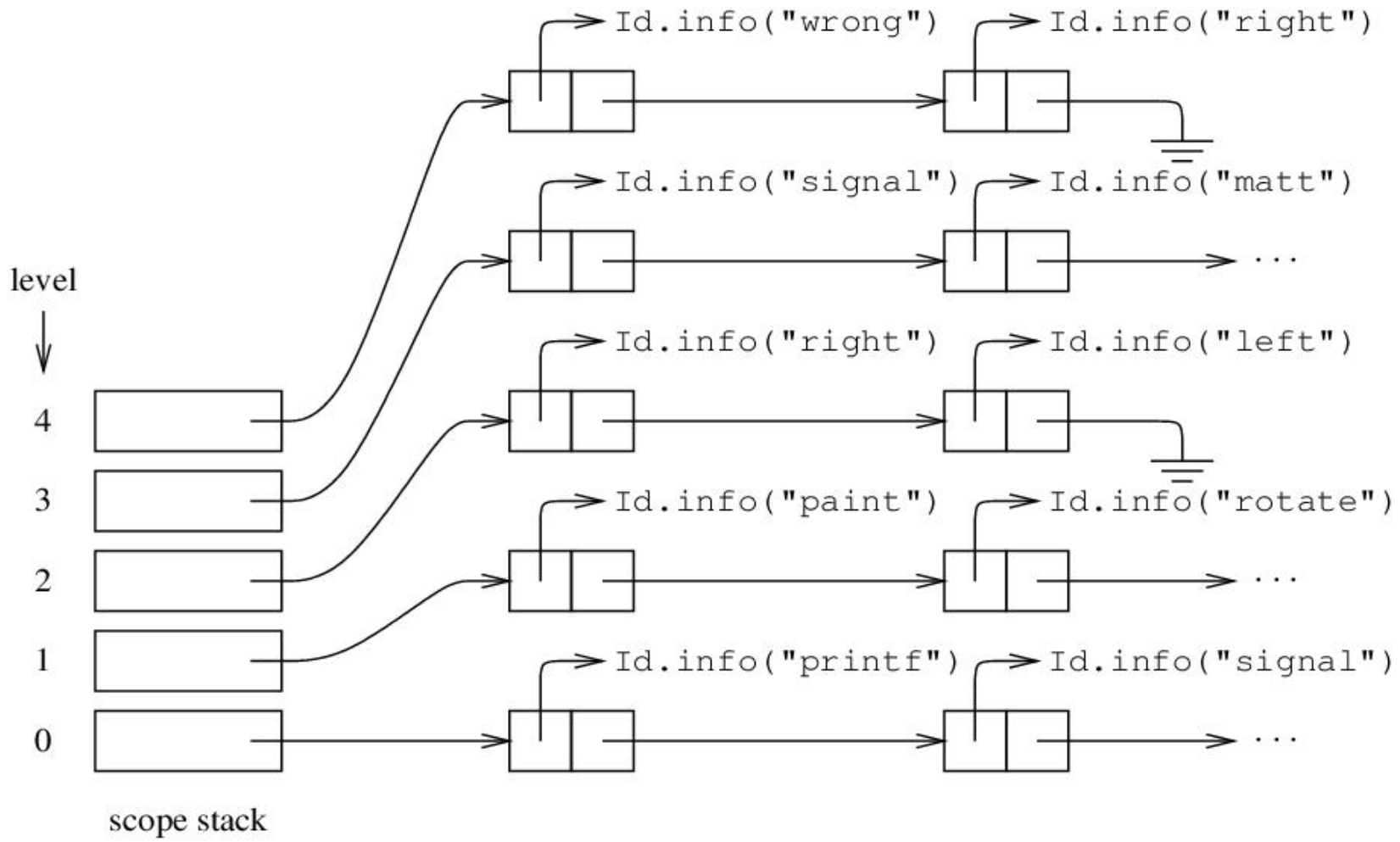

Figure 6.5 Scope table for the hash-table based symbol table.

Rek. Interpr. (6b), Informationsverwaltung

- Symboltabelle
  - Hauptsammelstelle für Informationen
  - oft Stack- und/oder Hashzugriff vorgeschaltet (Abb. 6.2, 6.4-5)
- Beispiel: Deklaration einer Variablen V vom Typ T
  - Sprachbeschreibung: Allokation der Variablen auf dem Stack
  - Interpreter:
    - Allokation auf dem Heap
    - Zugriff auf Variable über Symboltabelle

Rek. Interpr. (6c), Bsp. Deklaration

Eintrag in der Symboltabelle

- Zeiger auf den Namen V
- Dateiname und Zeilennummer der Deklaration
- Art des deklarierten Objektes

  (Variable, Konstante, Recordkomponente, ...)
- Zeiger auf den Namen T
- Zeiger auf den für V reservierten Speicherplatz
- Bitanzeige, ob V initialisiert ist
- diverse Scope- oder Stack-bezogene Zeiger
- andere sprachspezifische Information

Rek. Interpr. (7), Optimierung

Memoization

- Funktion “merkt sich” Ergebnisse früherer Berechnungen in einem Array
- weit verbreitete Verwendungen:
  - binär rekursive Funktionen: Fibonacci, Binomialkoeffizienten
  - Dynamic Programming: CYK-Algorithmus
  - Graphalgorithmen: markieren bereits besuchter Knoten
- bei der rekursiven Interpretation gebraucht für
  - merken einmal berechneter Typinformation für einen Bezeichner in einem Ausdruck
  - statischer Kontextcheck mit weitgehender Attributauswertung vor dem Start der eigentlichen Interpretation

Iterative Interpretation (1)

- benötigt ein Threading des abstrakten Syntaxbaums
- Vorgehen
  - gesamte Interpretation durch eine einzige while-Schleife
  - Fallunterscheidung für alle Knotentypen
  - Active-Node-Pointer zeigt auf aktuellen Knoten (vergleichbar mit Programmzähler bei der Codeausführung)
- Beispiele
  - if-Anweisung (Abb. 4.5)
  - Demo-Compiler für einfache arithmetische Ausdrücke (Abb. 4.6)

Iterative Interpretation (2), Bsp.: `if`

1. Berechnung der Bedingung vorher, Wert liegt auf dem Stack
2. Selektion des Zweiges durch Active-Node-Pointer
3. Präsenz des **else**-Zweiges muss weiterhin getestet werden

```
WHILE Active node .type /= End of program type:
    SELECT Active node .type:
        CASE ...
        CASE If type:
            // We arrive here after the condition has been evaluated;
            // the Boolean result is on the working stack.
            SET Value TO Pop working stack ();
            IF Value .boolean .value = True:
                SET Active node TO Active node .true successor;
            ELSE Value .boolean .value = False:
                IF Active node .false successor /= No node:
                    SET Active node TO Active node .false successor;
                ELSE Active node .false successor = No node:
                    SET Active node TO Active node .successor;
        CASE ...
```

Figure 4.5 Sketch of the main loop of an iterative interpreter, showing the code for an if-statement.

Iter. Interpr. (3), Bsp.: Demo-Compiler

```
#include    "parser.h"      /* for types AST_node and Expression */
#include    "thread.h"      /* for Thread_AST() and Thread_start */
#include    "stack.h"       /* for Push() and Pop() */
#include    "backend.h"     /* for self check */
                                        /* PRIVATE */
static AST_node *Active_node_pointer;

static void Interpret_iteratively(void) {
    while (Active_node_pointer != 0) {
        /* there is only one node type, Expression: */
        Expression *expr = Active_node_pointer;
        switch (expr->type) {
        case 'D':
            Push(expr->value);
            break;
        case 'P': {
            int e_left = Pop(); int e_right = Pop();
            switch (expr->oper) {
            case '+': Push(e_left + e_right); break;
            case '*': Push(e_left * e_right); break;
            }}
            break;
        }
        Active_node_pointer = Active_node_pointer->successor;
    }
    printf("%d\n", Pop());      /* print the result */
}
                                        /* PUBLIC */
void Process(AST_node *icode) {
    Thread_AST(icode); Active_node_pointer = Thread_start;
    Interpret_iteratively();
}
```

Figure 4.6 An iterative interpreter for the demo compiler of Section 1.2.

Iterative Interpretation (4)

Informationsverwaltung

- Stack
  - für lokale Daten (implementiert als dynamisches Array)
  - für Information über Scope
- Array
  - für globale Daten
  - Variablen des Quellprogramms werden durch Arrayadressen repräsentiert
- Symboltabelle
  - im wesentlichen nur noch für aussagekräftige Fehlermeldungen gebraucht

Iterative Interpretation (5)

Schattenspeicher

- gespiegelte Arrays
- jedes Byte des Schattenspeichers enthält Info über das entsprechende Byte des Originalarrays, z.B.
  - Byte nicht initialisiert
  - nicht-führendes Byte eines Zeigers
  - Byte Teil eines read-only Arrays
- höchstens 256 Informations-Kombinationen pro Byte
- Anwendungen
  - Benutzung nicht initialisierter Variablen
  - inkorrekter Zugriff auf eine Datenstruktur (Misalignment)
  - Überprüfung von Zugriffsbeschränkungen
- kann zur Effizienzsteigerung ausgeschaltet werden

Baumdarstellungen des Zwischencodes (1)

- als gestreut gespeicherte Struktur, verzeigert (Abb. 4.7)
- als verzeigertes Array (Abb. 4.8(a))
  - Vorteil: einfacher in Datei abzuspeichern
- als Array mit Pseudoanweisungen (Abb. 4.8(b))
  - Vorteil: spart Speicher für Zeiger auf Nachfolgeinstruktion
  - wird vornehmlich in iterativer Interpretation eingesetzt

  (Nähe zum von-Neumann-Modell)

Baumdarstellungen des Zwischencodes (2)

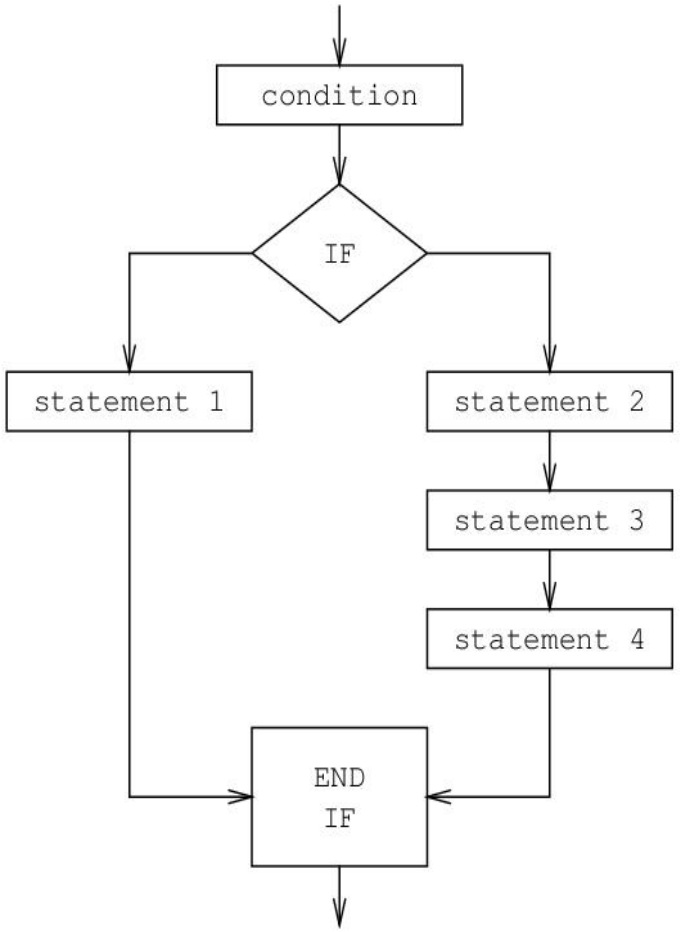

Figure 4.7 An AST stored as a graph.

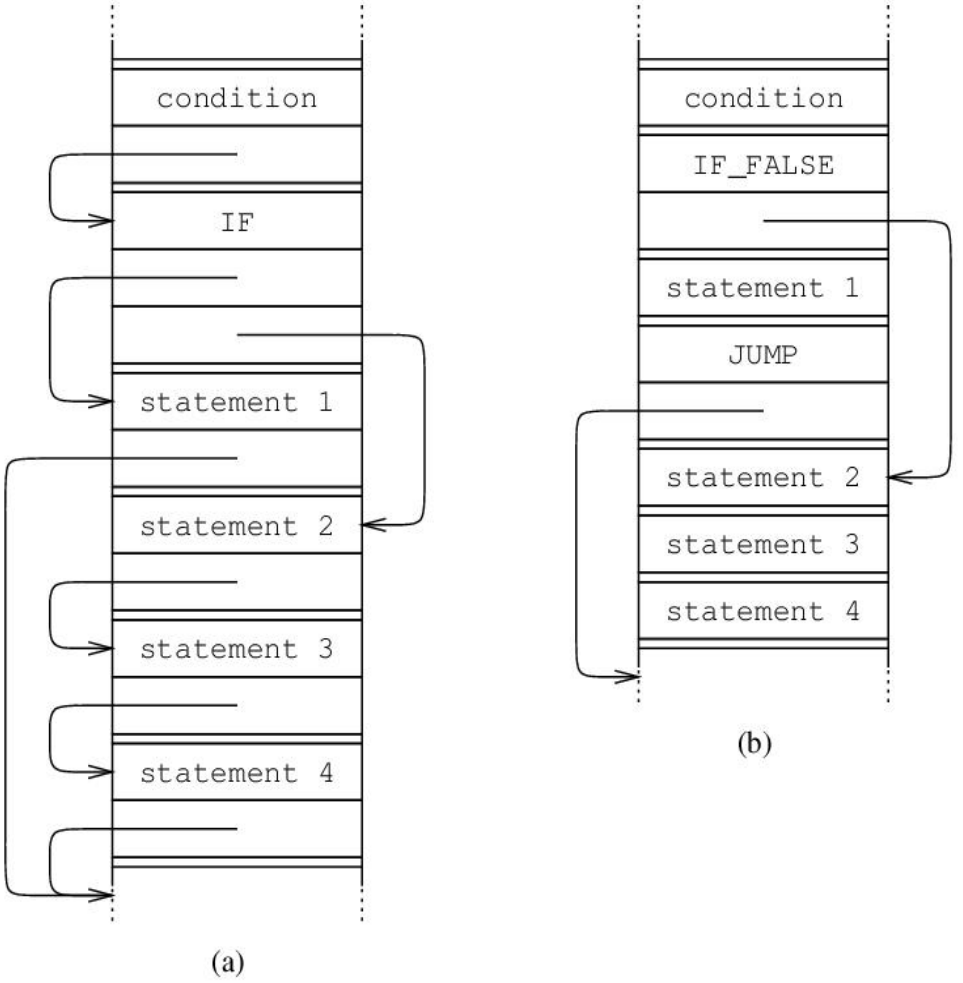

Figure 4.8 Storing the AST in an array (a) and as pseudo-instructions (b).

Effizienz (rekursive vs. iterative Interpretation)

- rekursive Interpretation
  + relativ schnell zu schreiben
  + gut beim Entwurf einer neuen Sprache, Testen neuer Features
  - keine statische Analyse: nur ausgeführter Code wird getestet
  - Slowdown-Faktor von 1000 und mehr gegenüber Kompilat
- iterative Interpretation
  + Interpreter selbst noch einfacher zu schreiben, aber
  - Threading des abstrakten Syntaxbaums zu implementieren

Effizienz iterativer Interpretation

- Laufzeitprüfungen
  - + weniger als bei rekursiver Interpretation
    (besonders ohne Schattenspeicher)
  - - erheblich mehr als bei Kompilat

- Slowdown gegenüber (optimiertem) Maschinencode
  - ohne Optimierungen: Faktor ca. 100-1000
  - mit Optimierungen: Faktor ca. 30

Compilerphasen, Orientierung (1)

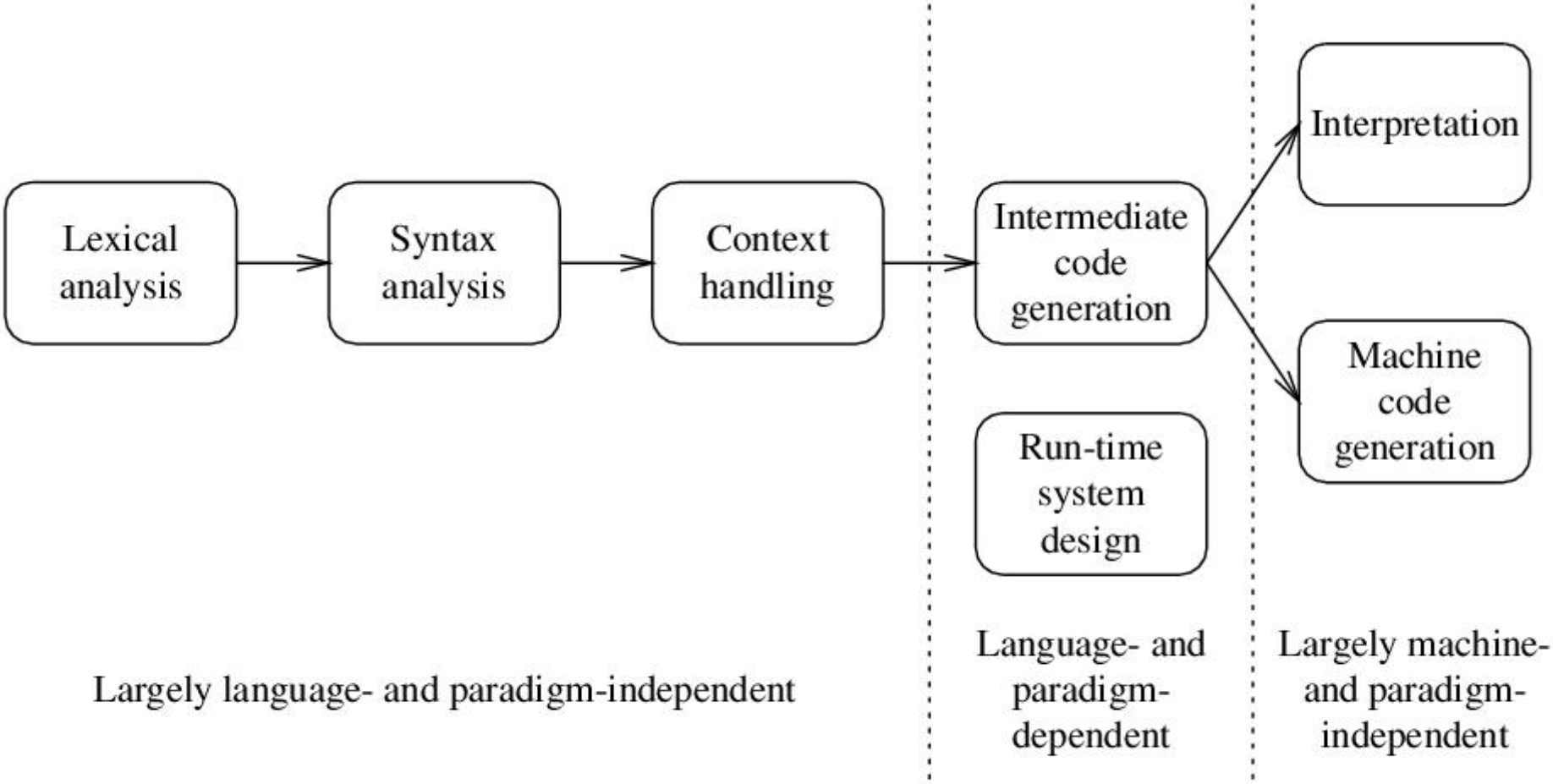

Figure 4.1 The status of the various modules in compiler construction.

Compilerphasen, Orientierung (2)

- die Wahl des Zwischencodes ist sehr variabel, Möglichkeiten:
  - (a) annotierter abstrakter Syntax-Baum oder -Graph
    - (a1) mit Threading
    - (a2) ohne Threading
  - (b) lineare Darstellungen wie Dreiadresscode
  - (c) Befehle für eine abstrakte Stackmaschine (push/pop)
- Festlegung auf eine bestimmte Form
  - notwendig für Aussagen über eine konkrete Implementierung
  - Entwurfsentscheidung: sollte keinen Einfluss auf das Vorstellungsvermögen oder Entwurfsalternativen haben

Compilerphasen, Orientierung (3)

- die Wahl des Zwischencodes ist sehr variabel
- eine Festlegung von vornherein ist nicht sinnvoll
- Konsequenzen
  - bestimmte Aufgaben (z.B. Linearisierung) sind nicht eindeutig einer Compilerphase zuordbar
  - vorgestellte Compilerphasen sind nur als Grobform zu verstehen

    Compiler, die auf Graph/Term-Transformationen basieren, haben viele kleine Phasen

    Bsp.: Glasgow Haskell-Compiler