UNIVERSITÄT PASSAU

Struktur und Implementierung von Programmiersprachen I (Compilerbau)

SS 2006

http://infosun.fmi.uni-passau.de/cl/passau/sips2006/index.html

Vorlesung und Übung:
Dr. Christoph Herrmann

Einordnung im Studienplan

- Umfang: 2V+2Ü
- anrechenbar:
  - Bachelor (5 ECTS-Punkte), alt und neu
  - Diplom
    - Säule I
    - Nebenfach Informatik
    - Vertiefungsgebiet: Methoden des Programmentwurfs

SIPS - Vorlesungszyklus

- SIPS I (SS 2006, 2SWS)

  Scanner/Parser, semantische Analyse, Interpretation, einfache Codegenerierung

- SIPS II (WS 2006/2007, 2SWS)

  Codeoptimierung, Codegenerierung, Laufzeitsystem

- SIPS III (SS 2007, 2SWS)

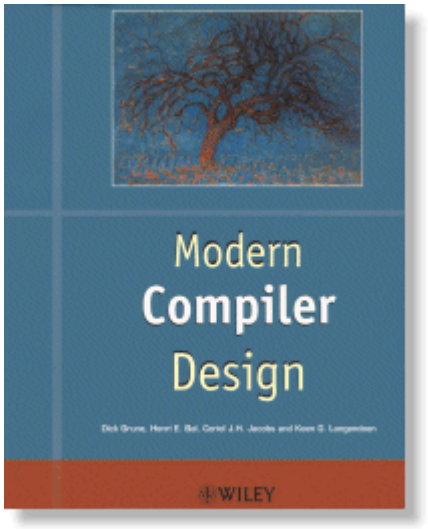
  Implementierung funktionaler, logischer, Objekt-orientierter und paralleler Features

Zentrale Literatur

Grune, Bal, Jacobs, Langendoen:

Modern Compiler Design

Wiley, 2002



Warum ist Compilerbau (noch) relevant?

... eine der reifsten Technologien in der Informatik ...

- Programme als mächtige Ein-/Ausgabeelemente

  Model-driven Architectures, Tcl/Tk, Bytecode, HTML, XML, Latex, Postscript, PDF, JPEG, MPEG, SQL, VHDL, Spezifikationssprachen, Domänen-spezifische Sprachen (Bildverarbeitung, Simulation)

Warum ist Compilerbau (noch) relevant?

... eine der reifsten Technologien in der Informatik ...

- Programme als mächtige Ein-/Ausgabeelemente
- Anwendungsgebiet für Algorithmen und Datenstrukturen

  Automatentheorie, Sprachen/Grammatiken, Programmanalyse, Symbolverwaltung, semantische Repräsentation, Ressourcenmanagement und -Optimierung (Graphalgorithmen, P/NP-Probleme)

  - Funktionale Programmierung: Typsysteme, Termersetzung, Funktionen höherer Ordnung
  - Logikprogrammierung: Unifikation, Backtracking, deduktive Datenbanken
  - Parallelität/Nebenläufigkeit: Grid-Computing, Bytecode, Scheduling, Allokation, Marshaling

Warum ist Compilerbau (noch) relevant?

... eine der reifsten Technologien in der Informatik ...

- Programme als mächtige Ein-/Ausgabeelemente
- Anwendungsgebiet für Algorithmen und Datenstrukturen
- Programmgeneratoren für hohe Performanz
  - reguläre Ausdrücke/Grammatik → Scanner/Parser
  - Ersetzungsregeln → Optimierer
  - Maschinenspezifikation → Codegenerator

  domänenspezifisch:
  - SQL-Spezifikation → Datenbankoperationen
  - Systemmodell → Simulator
  - DSP-Spezifikation → Signalprozessor

Compiler oder Interpreter?

	Compiler	Interpreter
Quellcode	wird übersetzt	wird ausgeführt
Portabilität	eingeschränkt	uneingeschränkt
Fehlermeldungen	schlecht verständl.	gut verständlich
Entwicklungsaufwand	hoch	niedrig
Analysemöglichkeiten	geringer	höher
Sicherheit	geringer	höher
Programmverarbeitung	kompliziert	einfach
Zwischencode	low-level	high/medium level
Eingabedaten	nicht vorhanden	vorhanden
ausführende Maschine	real	virtuell
Ausführungs-geschwindigkeit	hoch	schlecht

Compiler ↔ Interpreter

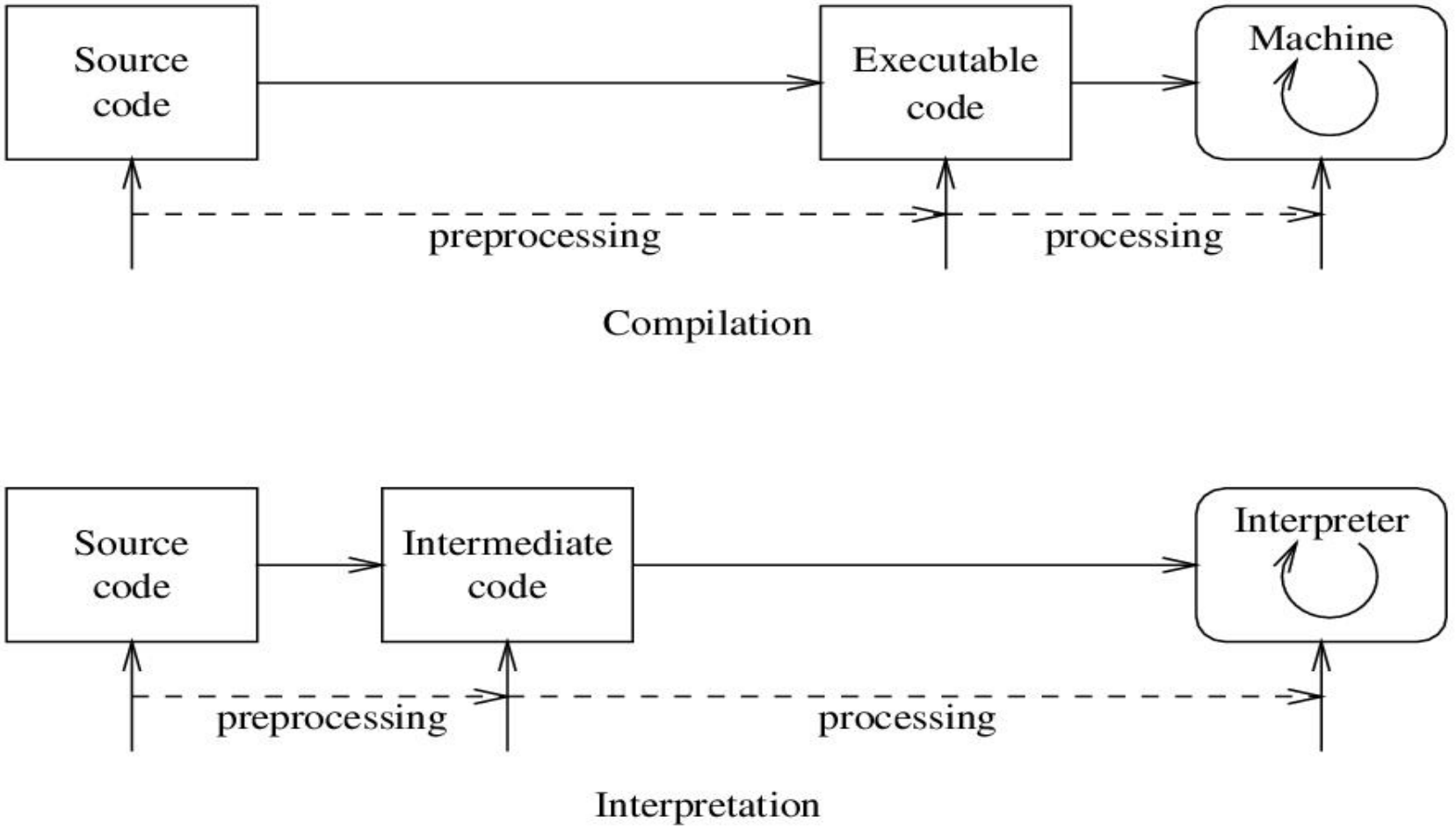


Figure 1.3 Comparison of a compiler and an interpreter.

Compiler ↔ Interpreter

- Übergänge Compiler/Interpreter sind fließend
  - Prozessor interpretiert Maschinenbefehle
  - es gibt: Bytecode-Interpreter, Just-in-time Compiler

Compiler ↔ Interpreter

- Übergänge Compiler/Interpreter sind fließend
- man kann Interpreter in einen Compiler transformieren
  - speziell: z.B. arithmetische Ausdrücke in Stack-Code
    - Quellcode: `(a+b)*c`
    - Postfix: `ab+c*`
    - Zielcode: `PUSH a;PUSH b;ADD;PUSH c;MUL`
  - allgemein: partielle Auswertung des Interpreters
    - funktionale Programmierung (Currying):

      Abstraktion von der Programmeingabe
    - spezielle Tools für imperative Sprachen

Compiler ↔ Interpreter

- Übergänge Compiler/Interpreter sind fließend
- man kann Interpreter in einen Compiler transformieren
- Verwendung eines Compilers allein garantiert noch keine effiziente Ausführung, es gibt viele Effizienzaspekte!
  - allgemein: semantische Transformationen, Verringerung der Anzahl von Operationen, Optimierung des Programmflusses, Speicherverwaltung, Inlining, Schleifentransformationen
  - maschinenabhängig: Registervergabe, Cache-Optimierung, partielles Loop-Unrolling, Pipelining, Sprungoptimierung, Befehlsauswahl

Bootstrapping

(partielle *Selbstgenerierung* eines Compilers)

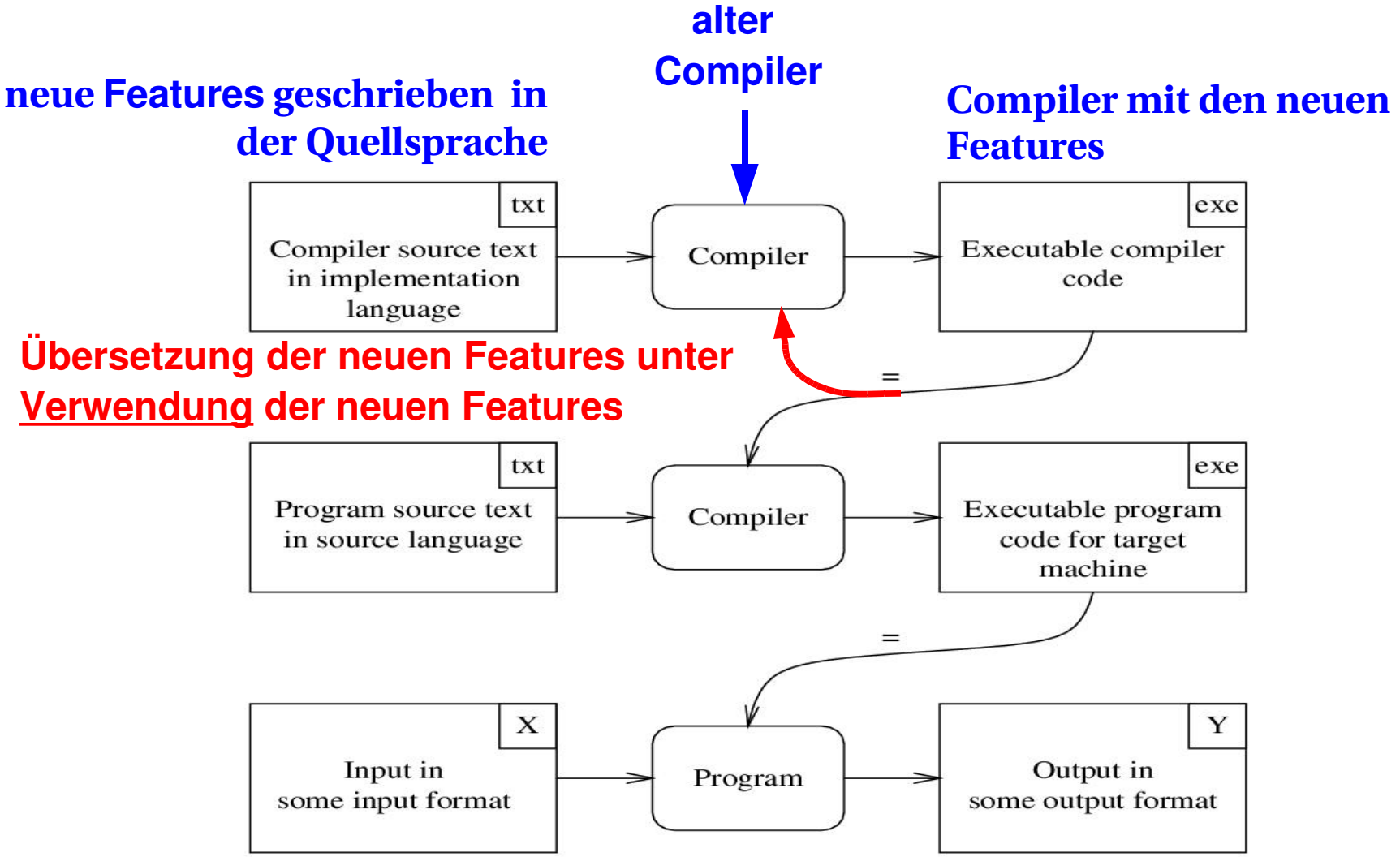


Figure 1.1 Compiling and running a compiler.

Trennung Frontend/Backend

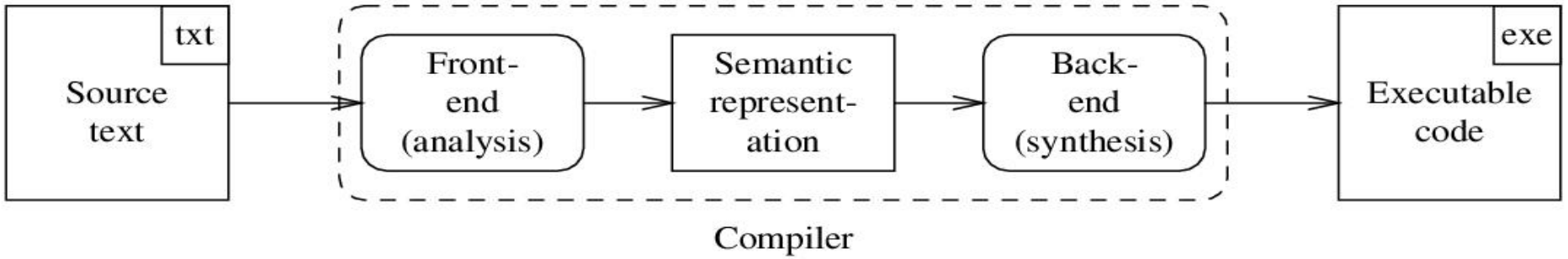


Figure 1.2 Conceptual structure of a compiler.

Motivation der Trennung

(L+M Module statt L*M Compiler)

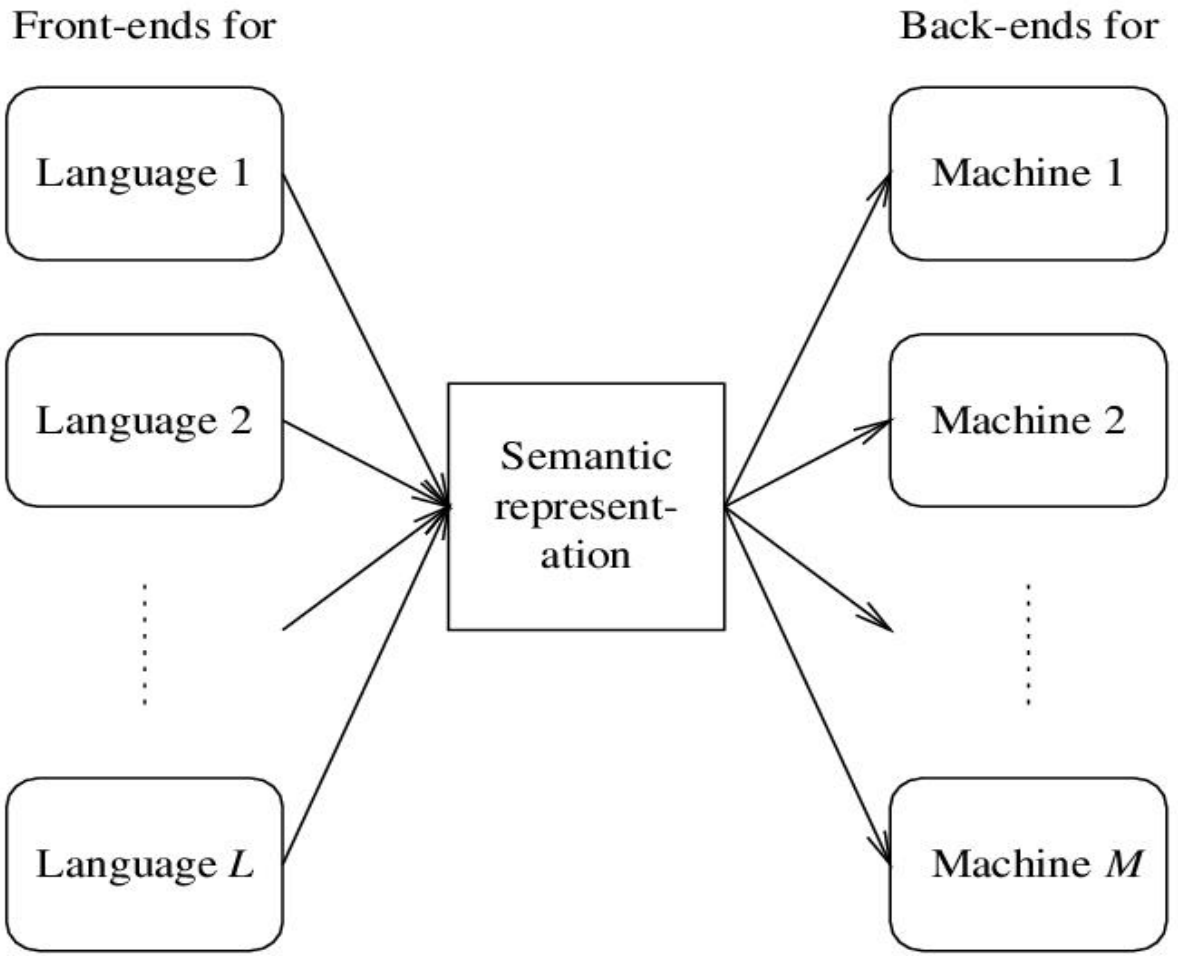


Figure 1.4 Creating compilers for L languages and M machines.

Beispiel zur Syntax

Quellsprache: arithmetische Ausdrücke mit Klammerung

Beispielausdruck: `b*b-4*a*c`

- Nichtterminale: { `expression`, `term`, `factor`, `identifier`, `constant` }
- Terminale: { `+`, `-`, `*`, `/`, `(`, `)` }
- Startsymbol: `expression`
- Produktionen:

```
expression → expression ´+´ term
           | expression ´-´ term
           | term

term       → term ´*´ factor
           | term ´/´ factor
           | factor

factor     → identifier
           | constant
           | ´(´ expression ´)´

...
```

Parsebaum

- reflektiert die Grammatik
- innere Knoten sind Nichtterminale
- kann Klammern enthalten

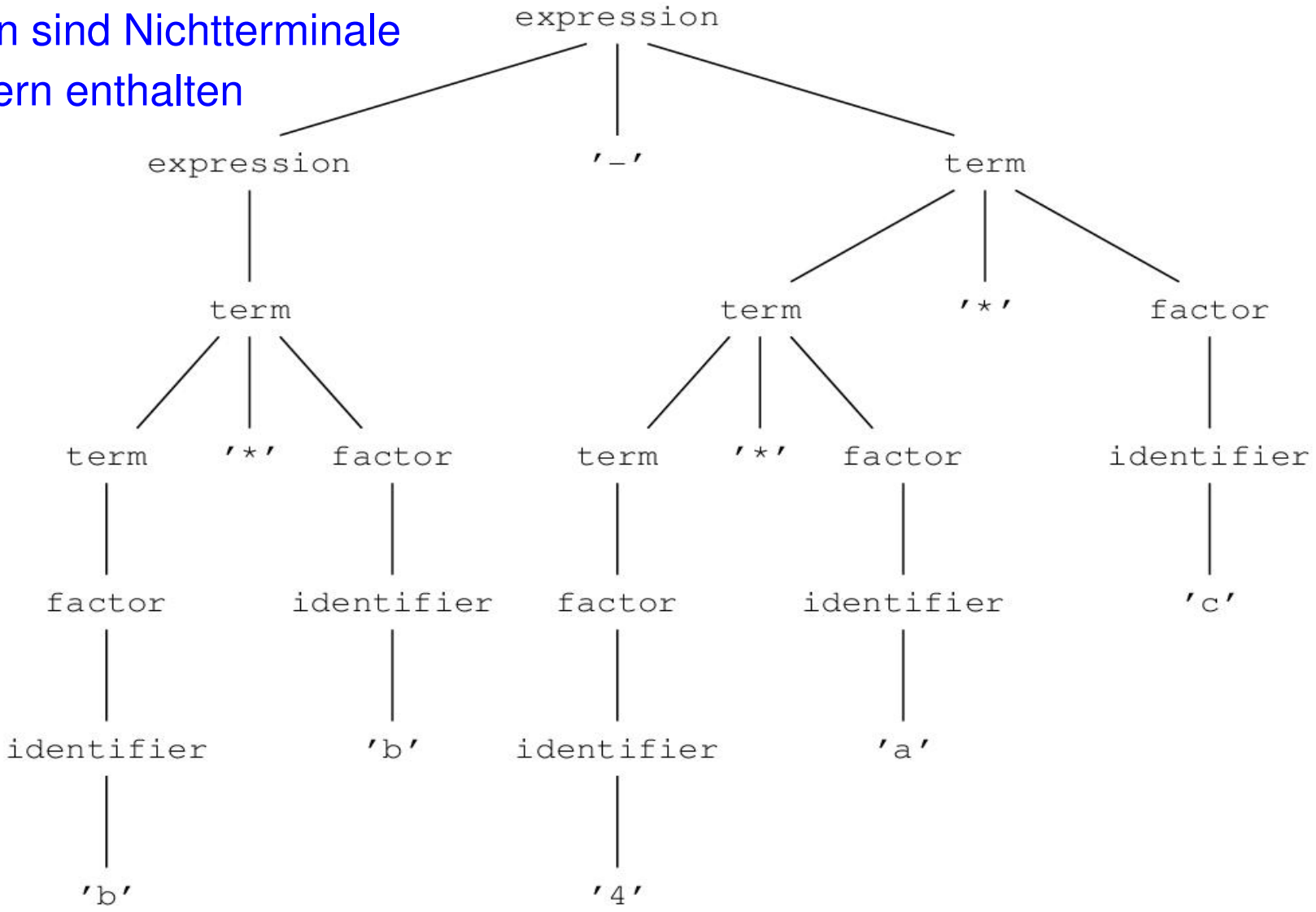


Figure 1.5 The expression `b*b - 4*a*c` as a parse tree.

Abstrakter Syntaxbaum (AST)

- ableitbar aus dem Parsebaum, meist aber direkt erzeugt
- Ausdrücke als Funktionen auf Termen
- Aufbau/Priorität durch Baumdarstellung gegeben
  (keine Klammern, keine Nichtterminale)
- innere Knoten sind Operatoren/Funktionssymbole

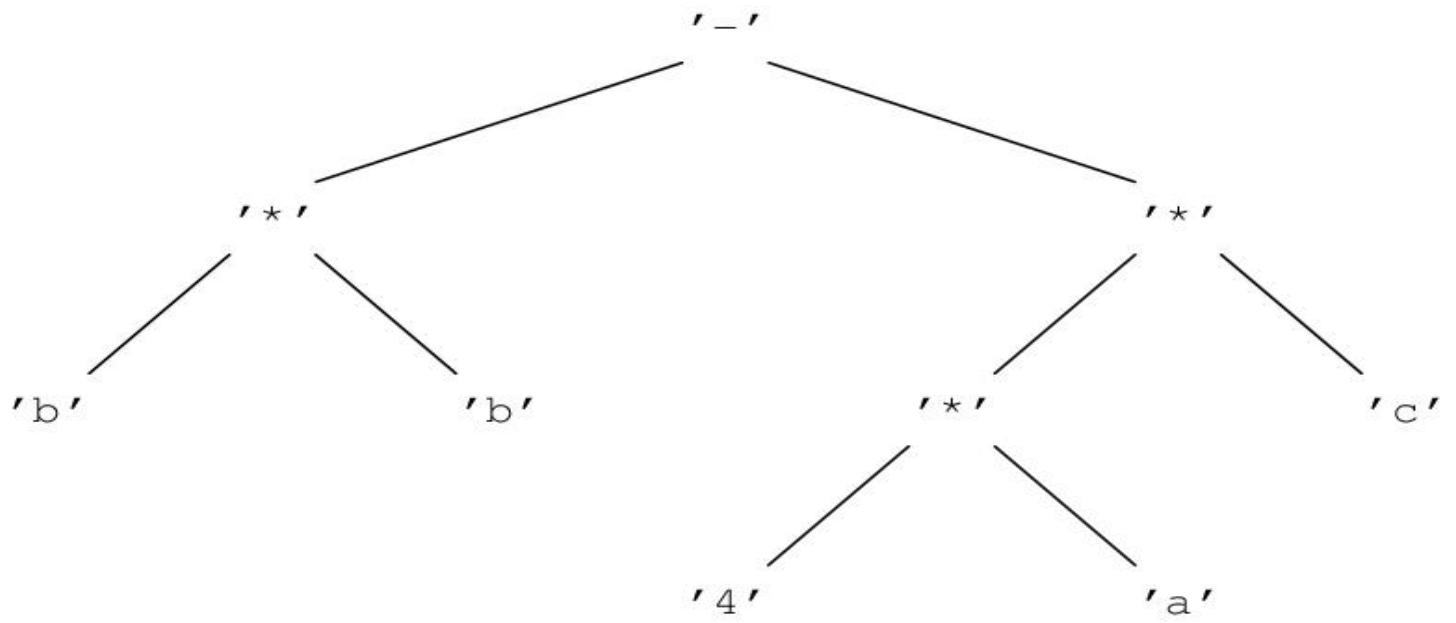


Figure 1.6 The expression `b*b - 4*a*c` as an AST.

Annotierter AST

Abstrakter Syntaxbaum plus

- semantische Information (hier: Typen)
- Implementierungsinformation (hier: Speicherstelle)

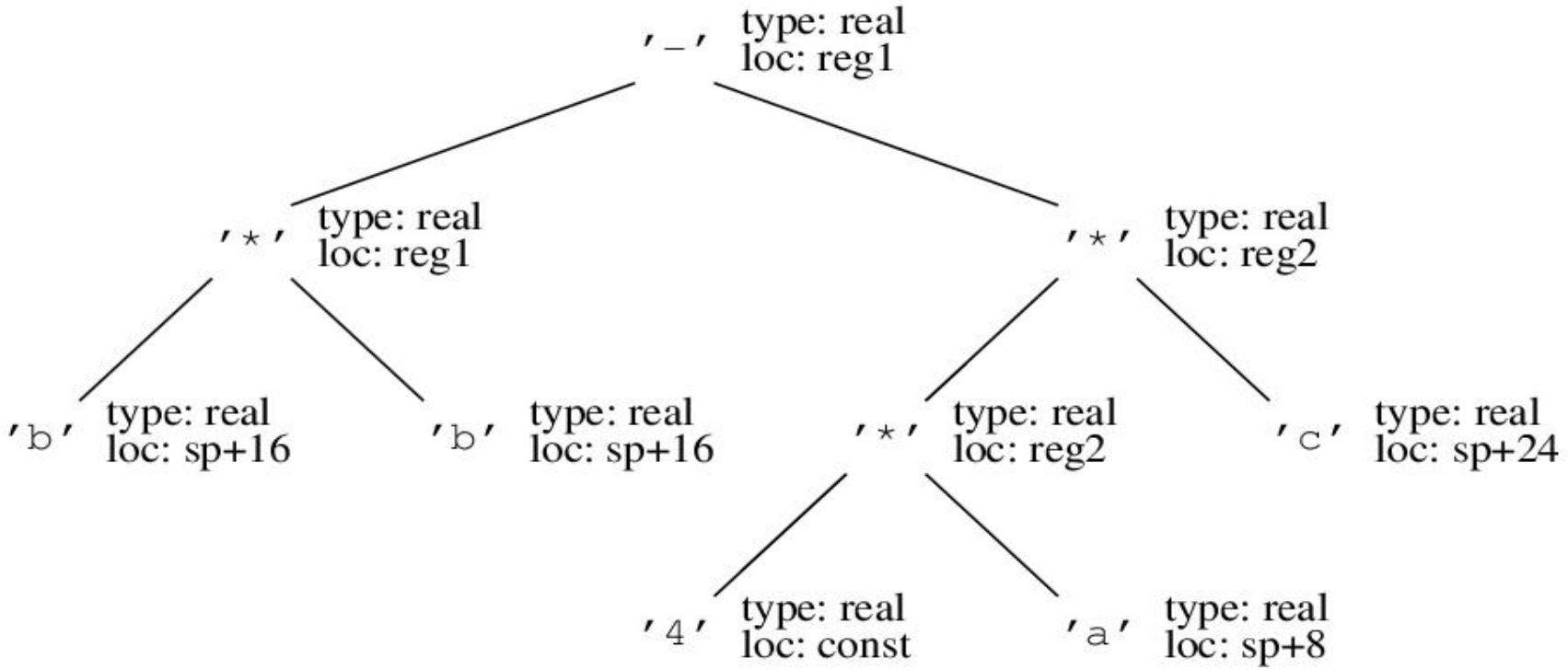


Figure 1.7 The expression `b*b - 4*a*c` as an annotated AST.

AST als Zwischencode

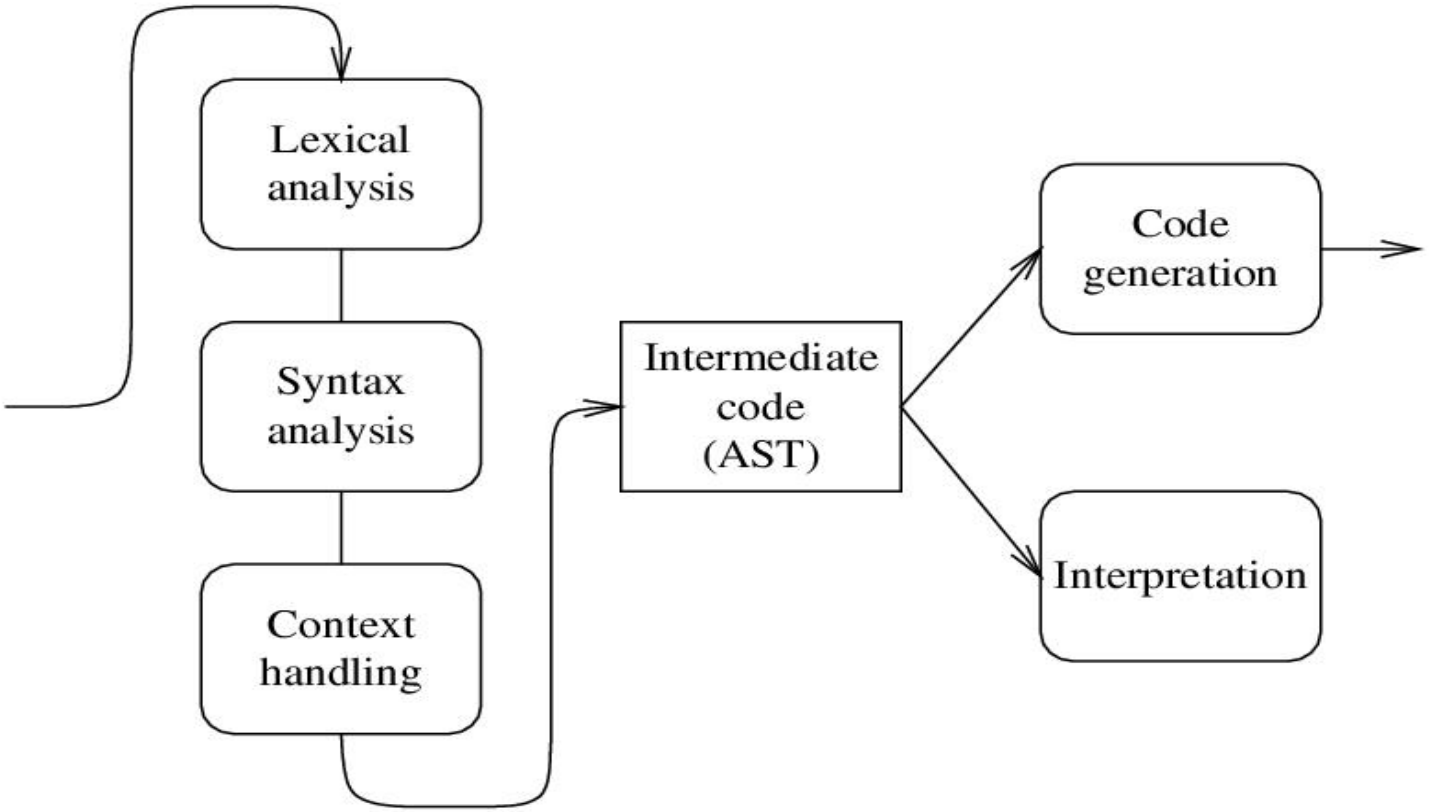


Figure 1.8 Structure of the demo compiler/interpreter.

Demo-Compiler/Interpreter (Buch)

- Lexikalische Analyse: generiert Tokenstrom
- Syntaxanalyse:
  - generiert abstrakten Syntaxbaum
  - gibt detaillierte Meldungen bei Syntaxfehlern
- Semantische Analyse: (beim Demo-System keine)
  - Propagation von Typinformation
  - Verbindung von Sprunganweisungen mit Sprungmarken
  - Erkennung eines Aufrufs als lokal oder entfernt
- Codegenerierung: Zielcode (hier: für Stackmaschine)
- Interpretation: Auswertung von Ausdrücken

Demo-Compiler (Buch)

- Quellsprache: arithmetische Ausdrücke
  - vollständig geklammert
  - keine Prioritäten
- Implementierungssprache: C
- später (Übungen)
  - Scanner-/Parsergeneratoren
  - Implementierungssprache: Objective CAML

```
expression → digit | '(' expression operator expression ')'
operator → '+' | '*'
digit → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Figure 1.9 Grammar for simple fully parenthesized expressions.

```
#include    "parser.h"      /* for type AST_node */
#include    "backend.h"     /* for Process() */
#include    "error.h"       /* for Error() */

int main(void) {
    AST_node *icode;

    if (!Parse_program(&icode)) Error("No top-level expression");
    Process(icode);

    return 0;
}
```

Figure 1.10 Driver for the demo compiler.

```
/* Define class constants */
/* Values 0-255 are reserved for ASCII characters */
#define    EOF       256
#define    DIGIT     257

typedef struct {int class; char repr;} Token_type;

extern Token_type Token;
extern void get_next_token(void);
```

Figure 1.11 Header file `lex.h` for the demo lexical analyzer.

```
#include    "lex.h"          /* for self check */
                                          /* PRIVATE */
static int Layout_char(int ch) {
    switch (ch) {
    case ' ': case '\t': case '\n': return 1;
    default:                        return 0;
    }
}
                                          /* PUBLIC */
Token_type Token;

void get_next_token(void) {
    int ch;

    /* get a non-layout character: */
    do {
        ch = getchar();
        if (ch < 0) {
            Token.class = EOF; Token.repr = '#';
            return;
        }
    } while (Layout_char(ch));

    /* classify it: */
    if ('0' <= ch && ch <= '9') {Token.class = DIGIT;}
    else {Token.class = ch;}

    Token.repr = ch;
}
```

Figure 1.12 Lexical analyzer for the demo compiler.

```
static int Parse_operator(Operator *oper) {
    if (Token.class == '+') {
        *oper = '+'; get_next_token(); return 1;
    }
    if (Token.class == '*') {
        *oper = '*'; get_next_token(); return 1;
    }
    return 0;
}

static int Parse_expression(Expression **expr_p) {
    Expression *expr = *expr_p = new_expression();

    /* try to parse a digit: */
    if (Token.class == DIGIT) {
        expr->type = 'D'; expr->value = Token.repr - '0';
        get_next_token();
        return 1;
    }
```

Figure 1.13 (a) Parsing routines for the demo compiler

```
    /* try to parse a parenthesized expression: */
    if (Token.class == '(') {
        expr->type = 'P';
        get_next_token();
        if (!Parse_expression(&expr->left)) {
            Error("Missing expression");
        }
        if (!Parse_operator(&expr->oper)) {
            Error("Missing operator");
        }
        if (!Parse_expression(&expr->right)) {
            Error("Missing expression");
        }
        if (Token.class != ')') {
            Error("Missing )");
        }
        get_next_token();
        return 1;
    }

    /* failed on both attempts */
    free_expression(expr); return 0;
}
```

Figure 1.13 (b) Parsing routines for the demo compiler

```
#include    "lex.h"
#include    "error.h"       /* for Error() */
#include    "parser.h"      /* for self check */
                                        /* PRIVATE */
static Expression *new_expression(void) {
    return (Expression *)malloc(sizeof (Expression));
}
static void free_expression(Expression *expr) {free((void *)expr);}
static int Parse_operator(Operator *oper_p);
static int Parse_expression(Expression **expr_p);

                                        /* PUBLIC */
int Parse_program(AST_node **icode_p) {
    Expression *expr;

    get_next_token();           /* start the lexical analyzer */
    if (Parse_expression(&expr)) {
        if (Token.class != EOF) {
            Error("Garbage after end of program");
        }
        *icode_p = expr;
        return 1;
    }
    return 0;
}
```

Figure 1.14 Parser environment for the demo compiler.

Parsing von Alternativen

$$P \rightarrow A_1 A_2 \ldots A_n \mid B_1 B_2 \ldots \mid \ldots$$

```
int P(...) {
    /* try to parse the alternative A1 A2 ... An */
    if (A1(...)) {
        if (!A2(...)) Error("Missing A2");
        ...
        if (!An(...)) Error("Missing An");
        return 1;
    }
    /* try to parse the alternative B1 B2 ... */
    if (B1(...)) {
        if (!B2(...)) Error("Missing B2");
        ...
        return 1;
    }
    ...
    /* failed to find any alternative of P */
    return 0;
}
```

Figure 1.15 A C template for a grammar rule.

```
typedef int Operator;

typedef struct _expression {
    char type;                          /* 'D' or 'P' */
    int value;                          /* for 'D' */
    struct _expression *left, *right; /* for 'P' */
    Operator oper;                      /* for 'P' */
} Expression;

typedef Expression AST_node;            /* the top node is an Expression */

extern int Parse_program(AST_node **);
```

Figure 1.16 Parser header file for the demo compiler.

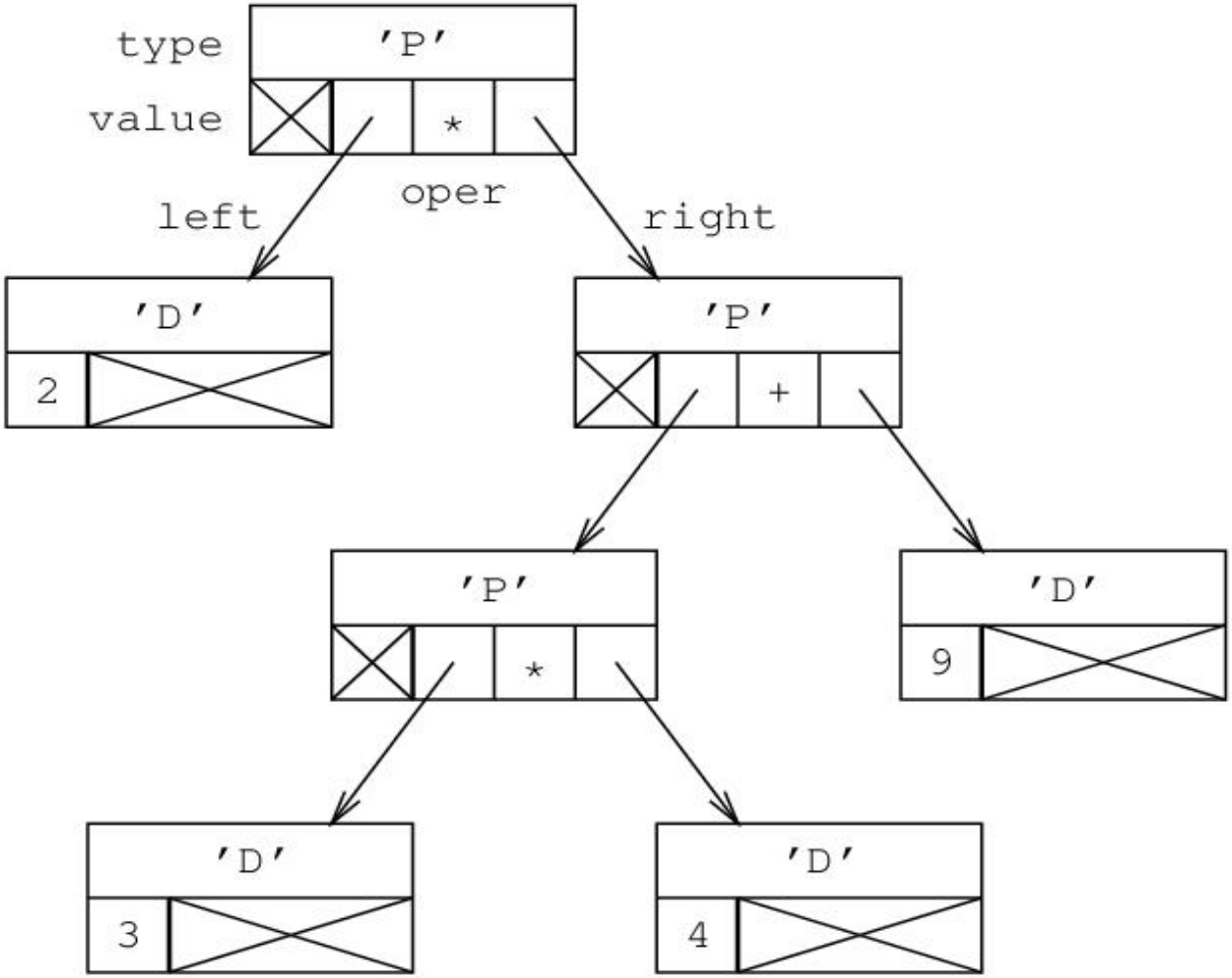


Figure 1.17 An AST for the expression `(2*((3*4)+9))`.

Codegenerator

```
#include    "parser.h"      /* for types AST_node and Expression */
#include    "backend.h"     /* for self check */
                                        /* PRIVATE */
static void Code_gen_expression(Expression *expr) {
    switch (expr->type) {
    case 'D':
        printf("PUSH %d\n", expr->value);
        break;
    case 'P':
        Code_gen_expression(expr->left);
        Code_gen_expression(expr->right);
        switch (expr->oper) {
        case '+': printf("ADD\n"); break;
        case '*': printf("MULT\n"); break;
        }
        break;
    }
}
                                        /* PUBLIC */
void Process(AST_node *icode) {
    Code_gen_expression(icode); printf("PRINT\n");
}
```

Figure 1.18 Code generation back-end for the demo compiler.

Interpreter

```
#include    "parser.h"      /* for types AST_node and Expression */
#include    "backend.h"     /* for self check */
                                        /* PRIVATE */
static int Interpret_expression(Expression *expr) {
    switch (expr->type) {
    case 'D':
        return expr->value;
        break;
    case 'P': {
        int e_left = Interpret_expression(expr->left);
        int e_right = Interpret_expression(expr->right);
        switch (expr->oper) {
        case '+': return e_left + e_right;
        case '*': return e_left * e_right;
        }}
        break;
    }
}
                                        /* PUBLIC */
void Process(AST_node *icode) {
    printf("%d\n", Interpret_expression(icode));
}
```

Figure 1.19 Interpreter back-end for the demo compiler.

Compiler ↔ Interpreter

beide verwenden dieselbe Schnittstelle

```
extern void Process(AST_node *);
```

Figure 1.20 Common back-end header for code generator and interpreter.

Compiler: narrow ↔ broad

```
WHILE NOT Finished:
    Read some data D from the source code;
    Process D and produce the corresponding object code, if any;
```

Figure 1.23 Flow-of-control structure of a narrow compiler.

```
SET Object code TO
    Assembly(
        Code generation(
            Context check(
                Parse(
                    Tokenize(
                        Source code
                    )
                )
            )
        )
    );
```

Figure 1.22 Flow-of-control structure of a broad compiler.

Aufbau eines komplexen Compilers

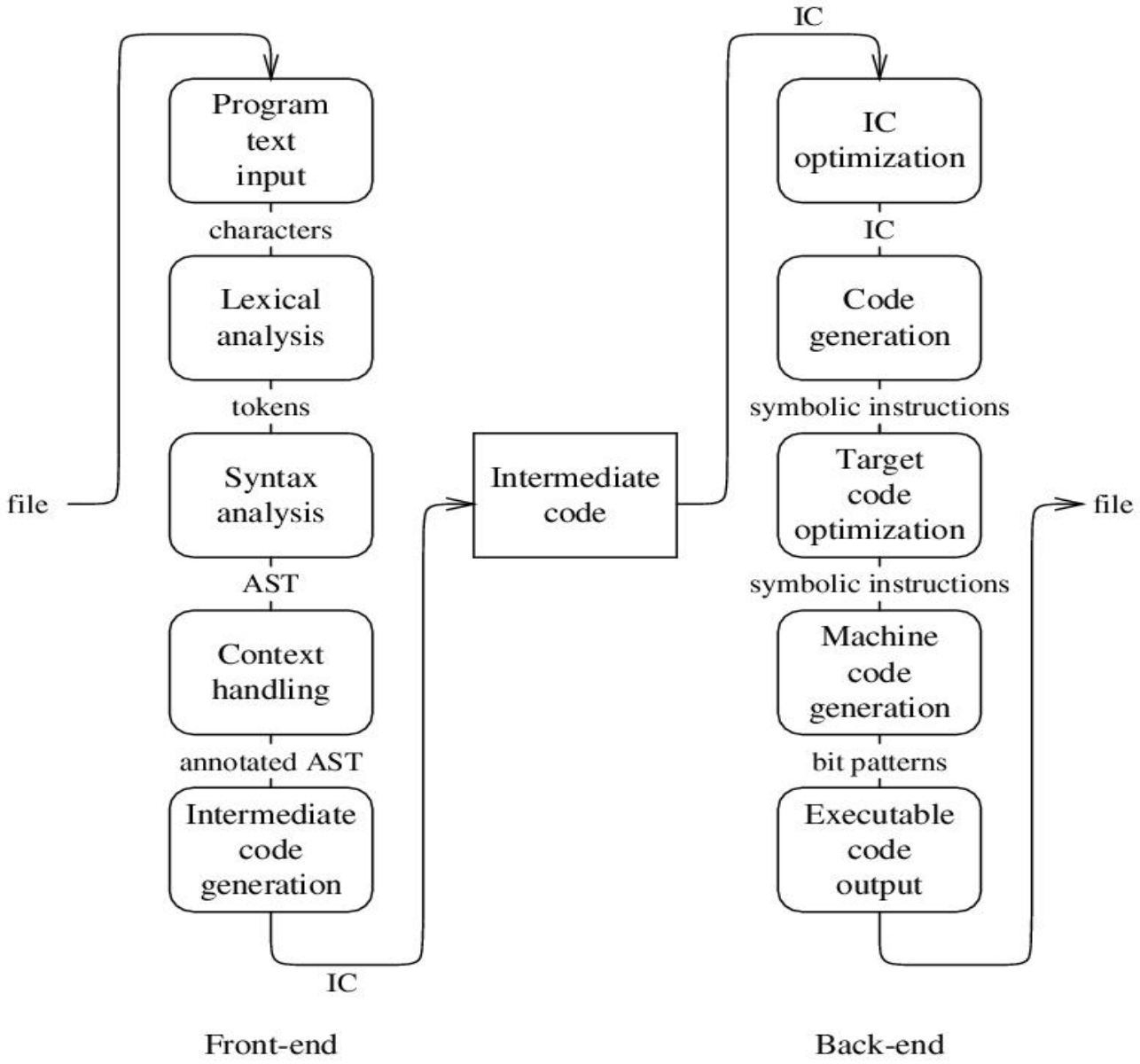


Figure 1.21 Structure of a compiler.

Laufzeitsystem

- (Gewöhnlich) vorkompilierte Programmstücke:
  - `printf(), malloc()`
  - Speicherallokation für Arrays
  - Manipulation des Laufzeitstacks
  - Methodenauswahl bei dynamischer Bindung
  - Parameterunifikation
  - entfernter Methodenaufruf
  - Task Scheduling