UNIVERSITÄT PASSAU
Fakultät für Informatik und Mathematik

Struktur und Implementierung von Programmiersprachen II (Compilerbau)

WS 2006/2007

Vorlesung und Übung:
Dr. Christoph Herrmann

http://infosun.fmi.uni-passau.de/cl/lehre/sips2-ws0607/index.html

Einordnung im Studienplan

- Umfang: 2V+2Ü
- anrechenbar:

  (a) Bachelor (5 ECTS-Punkte)

  (b) Diplom

  - Säule I
  - Nebenfach Informatik
  - Vertiefungsgebiet: Methoden des Programmentwurfs

Inhalt der Übung

wöchentlicher Übungstermin: flexible Zweiteilung mit Pause

1. Teil: Vorbereitung auf die Klausur
   - Diskussion des Vorlesungsstoffs
   - Besprechung von Übungsaufgaben auf Papier
2. Teil: Programmierpraxis (kein offizielles Praktikum!)
   - Programmierung eines Mini-Compilers in Java
   - empfohlen: Arbeit in Teams zu drei Personen
   - bei Teilnahme bis zum Schluss: Bescheinigung
   - Eigeninitiative nötig

SIPS - Vorlesungszyklus

- SIPS I (SS 2006, 2SWS):

  Spracherkennung: Scanner/Parser, semantische Analyse; Interpretation, triviale Codegenerierung

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

  Implementierung imperativer Konstrukte, Context-Handling, Codegenerierung, Codeoptimierung, Laufzeitsystem

- SIPS III (voraussichtlich SS 2007, 2SWS)

  Implementierung funktionaler, logischer, Objekt-orientierter und paralleler Features

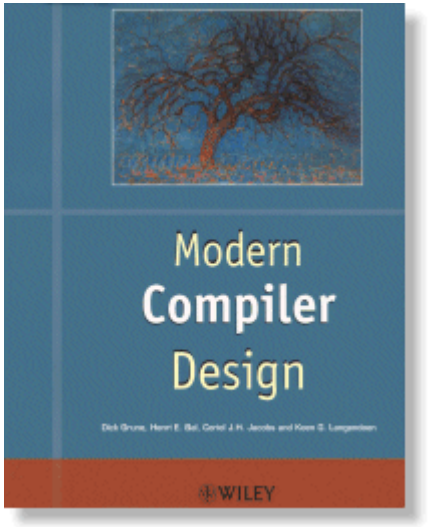
Literatur

Vorlesung:

Grune, Bal, Jacobs, Langendoen:

Modern Compiler Design

Wiley, 2002



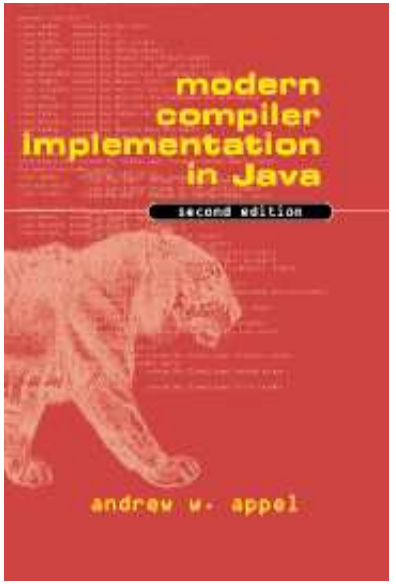
Übung:

Andrew Appel (mit Jens Palsberg):

Modern Compiler Implementation

in Java (second edition)

Cambridge University Press, 2002



Compilerbau: Sammlung von Technologien (1)

- 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)

Compilerbau: Sammlung von Technologien (2)

- 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

Compilerbau: Sammlung von Technologien (3)

- 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 ↔ Interpreter (1)

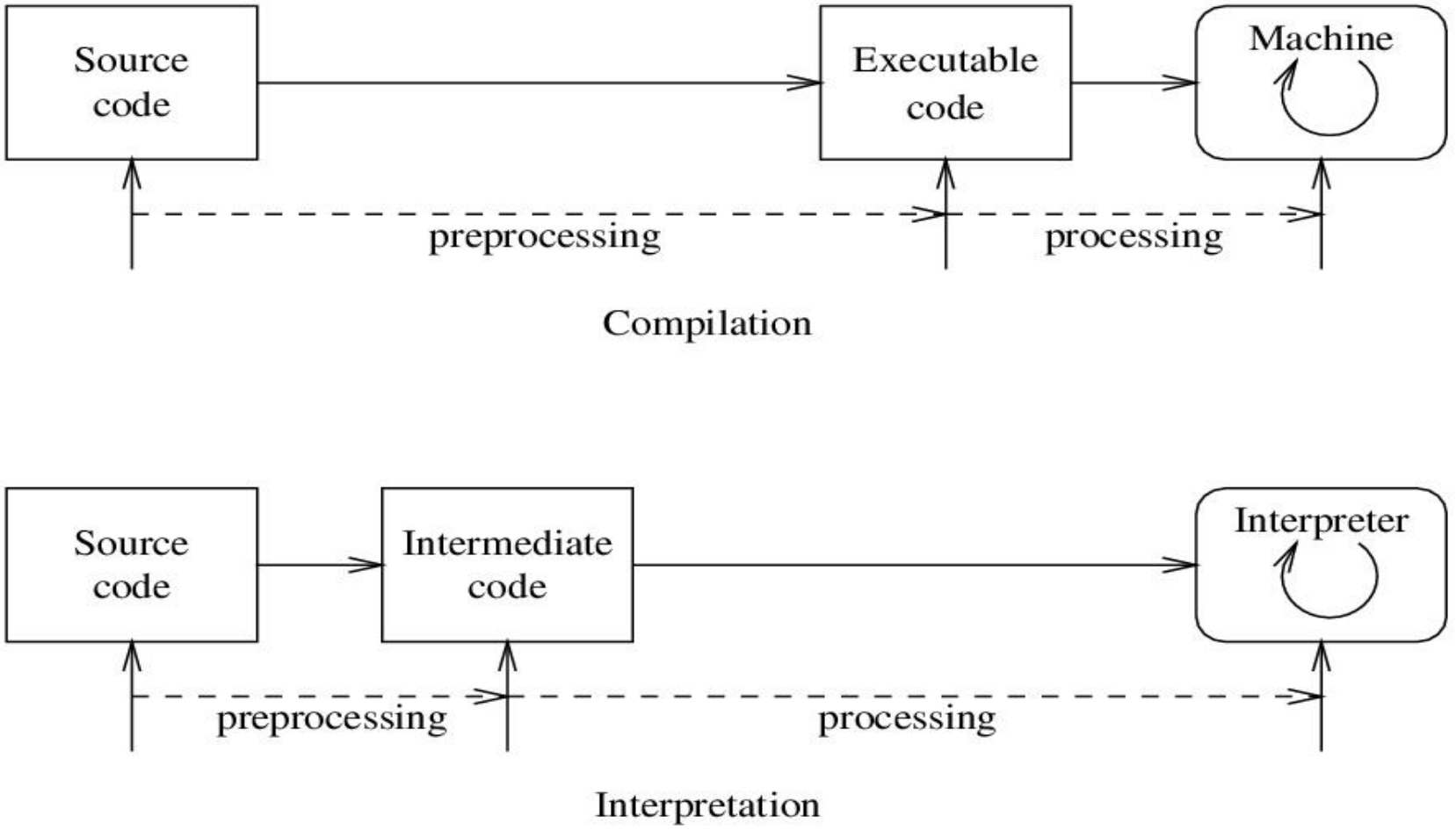


Figure 1.3 Comparison of a compiler and an interpreter.

Compiler ↔ Interpreter (2)

- 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

Grobaufbau eines Compilers

- Frontend: Analyse des Quellprogramms
- Backend: Generierung des Zielprogramms

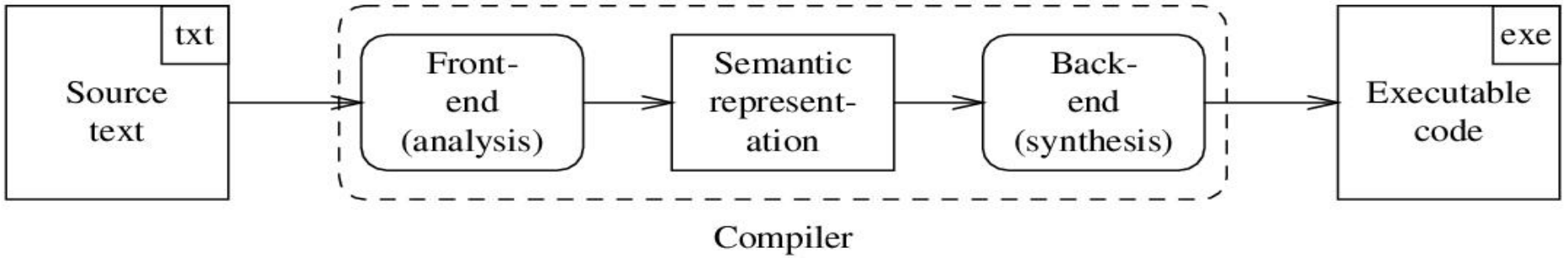


Figure 1.2 Conceptual structure of a compiler.

Trennung von Frontend und Backend

Idee: L+M Module statt L*M Compiler

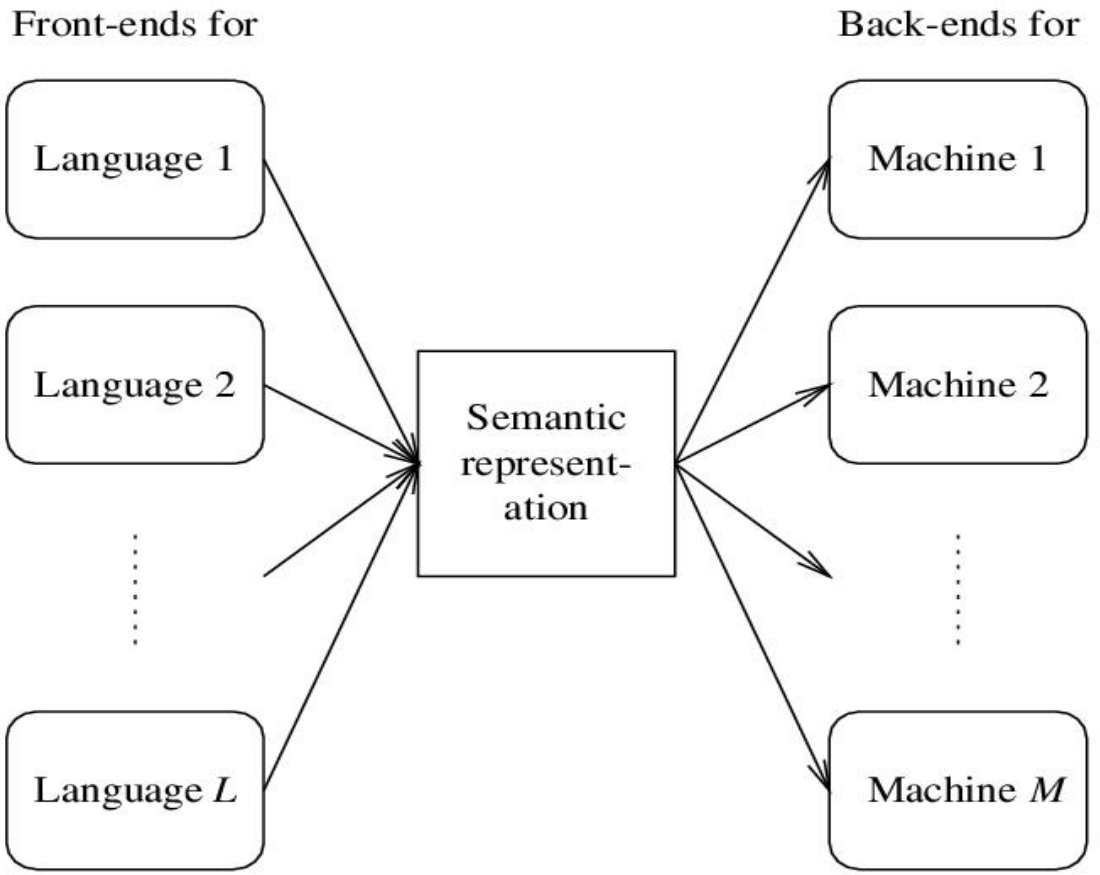


Figure 1.4 Creating compilers for L languages and M machines.

Feinaufbau eines komplexen Compilers

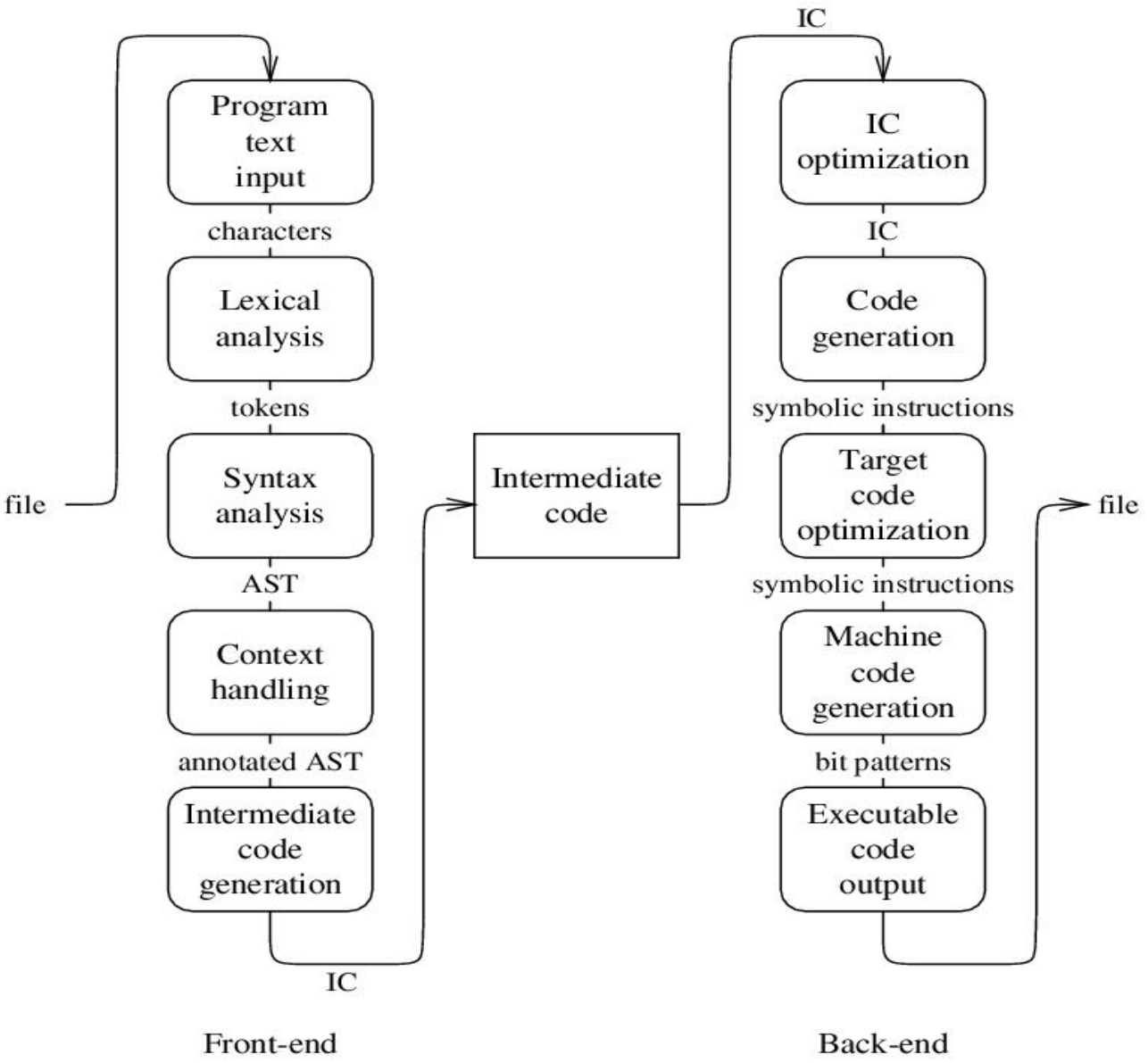


Figure 1.21 Structure of a compiler.

Rückblick auf SIPS I, Einflüsse auf SIPS II

- Scanner (endliche Automaten)
  - Erkennung von Namen, Schlüsselwörtern, Operatoren
- Parsing (deterministische Kellerautomaten)
  - Strukturanalyse, Erzeugung eines Parsebaums / Syntaxbaums

  (a) Top-Down (vorgegebener Parser in SIPS II - Übung)

  (b) Bottom-Up
- Attributierung (Kontextsensitivität)
  - Typprüfung in der SIPS II - Vorlesung/Übung ist Spezialfall davon
- Datenflussanalyse
  - kommt in einfacher Form in SIPS II - Übung
- Interpreter + partielle Auswertung = triviale Codegenerierung

Inhalte der SIPS II - Vorlesung

- Context-Handling (Symboltabelle, Gültigkeitsbereiche, Typprüfung)
- Repräsentation von Daten, Objekten, Routinen
- Kontrollflussorganisation (Bedingungen, Schleifen)
- verschiedene Arten der Codegenerierung
- Code-Optimierung
- Assembler, Linker, Loader
- Laufzeitsystem und Speicherverwaltung

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 ´)´

...
```

Syntaxanalyse (Parsing)

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

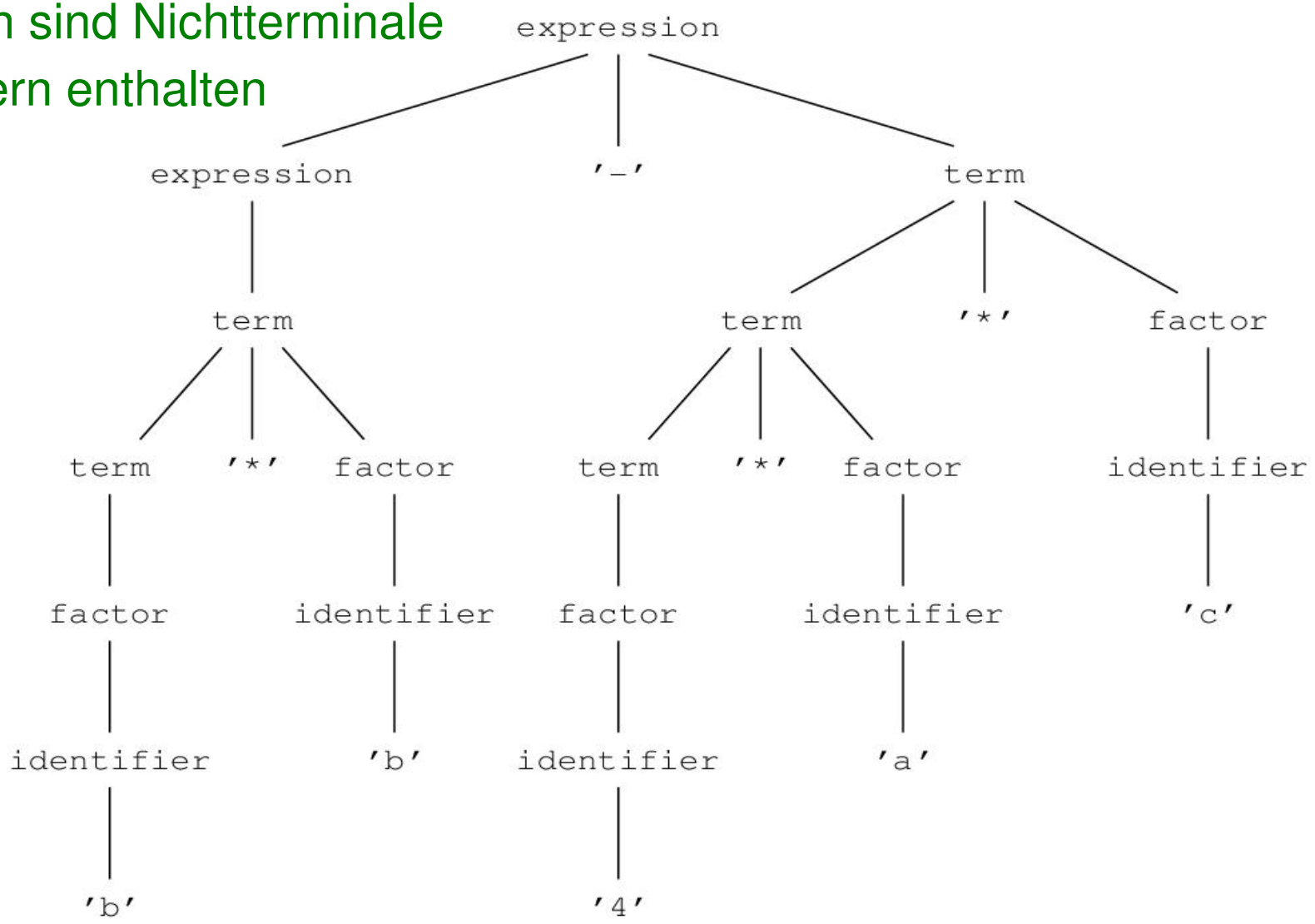


Figure 1.5 The expression `b*b - 4*a*c` as a parse tree.

Strategien der Syntaxanalyse

- Top-Down Baumerzeugung (LL(k) mit Eingabe-Lookahead k)
  - Vorhersage der richtigen Produktion nötig (k evtl. groß)
  - Nachteil: keine linksrekursive Produktion möglich, Optionen:

    (a) Grammatik umschreiben, Parsebaum rücktransformieren
    (b) auf syntaktischen Komfort verzichten, Bsp.: `x=(a*b)+c`
  - Vorteil: Steuerung des Parsens durch Kontextinformation
- Bottom-Up Baumerzeugung (LR(k))
  - Erkennung und Reduktion rechter Seiten von Produktionen
  - Vorteil: Linksrekursion (und auch Rechtsrekursion) möglich
  - große Mächtigkeit: $\forall k$:LL(k)$\subset$LR(k), aber $\neg\exists k$:LR(0)$\subseteq$LL(k)

Abstrakter Syntaxbaum (AST)

- Ausgangspunkt für die SIPS II -Vorlesung und -Übung
- 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

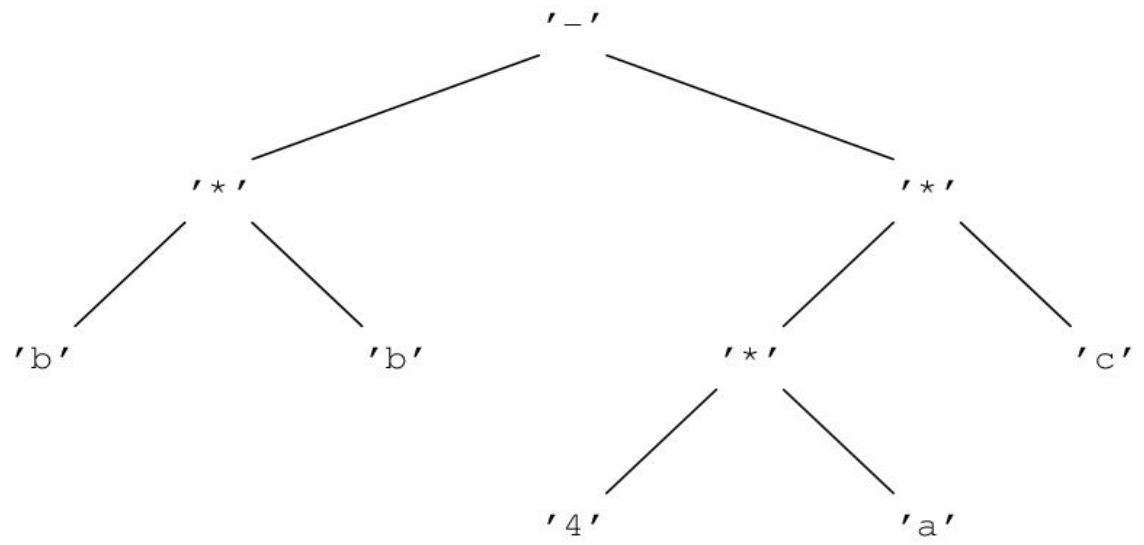


Figure 1.6 The expression `b*b - 4*a*c` as an AST.

Annotierter AST

abstrakter Syntaxbaum plus

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

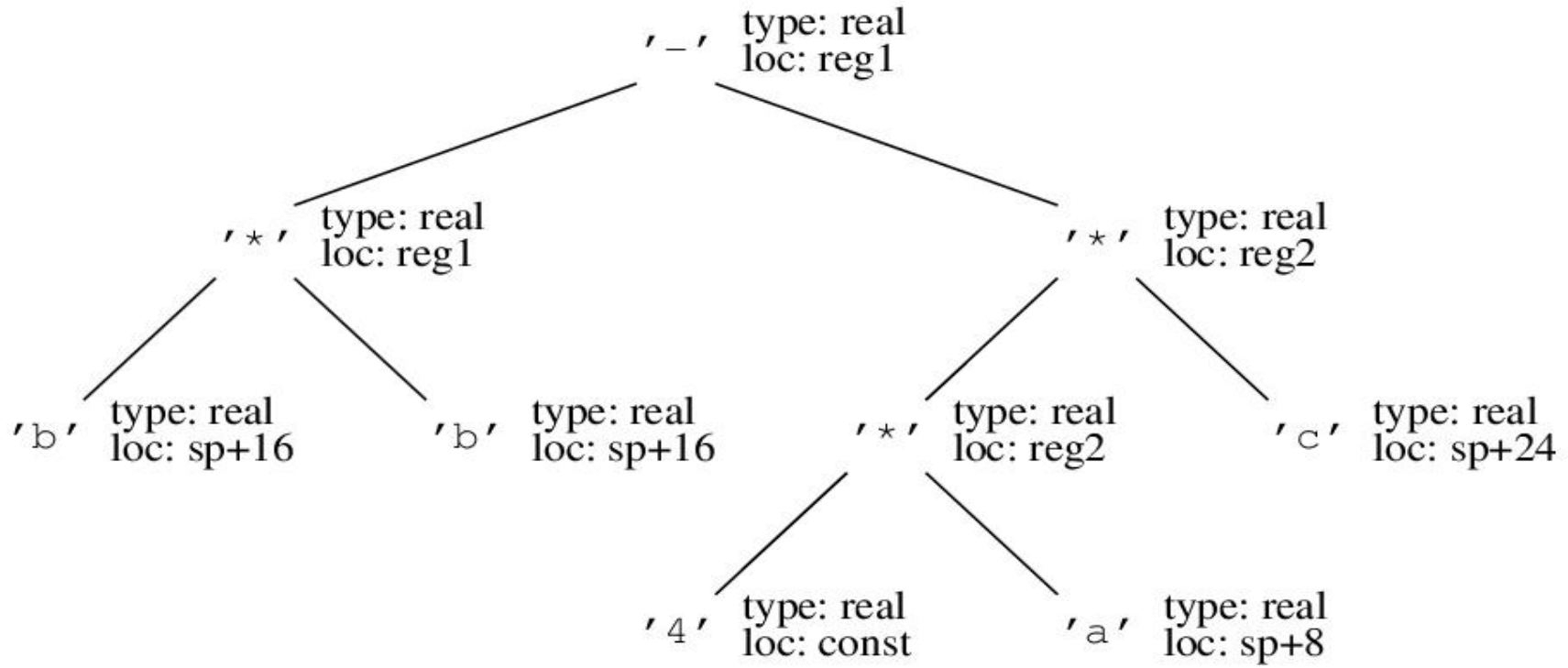


Figure 1.7 The expression `b*b - 4*a*c` as an annotated AST.

AST als Ausgangspunkt für Interpreter und Compiler

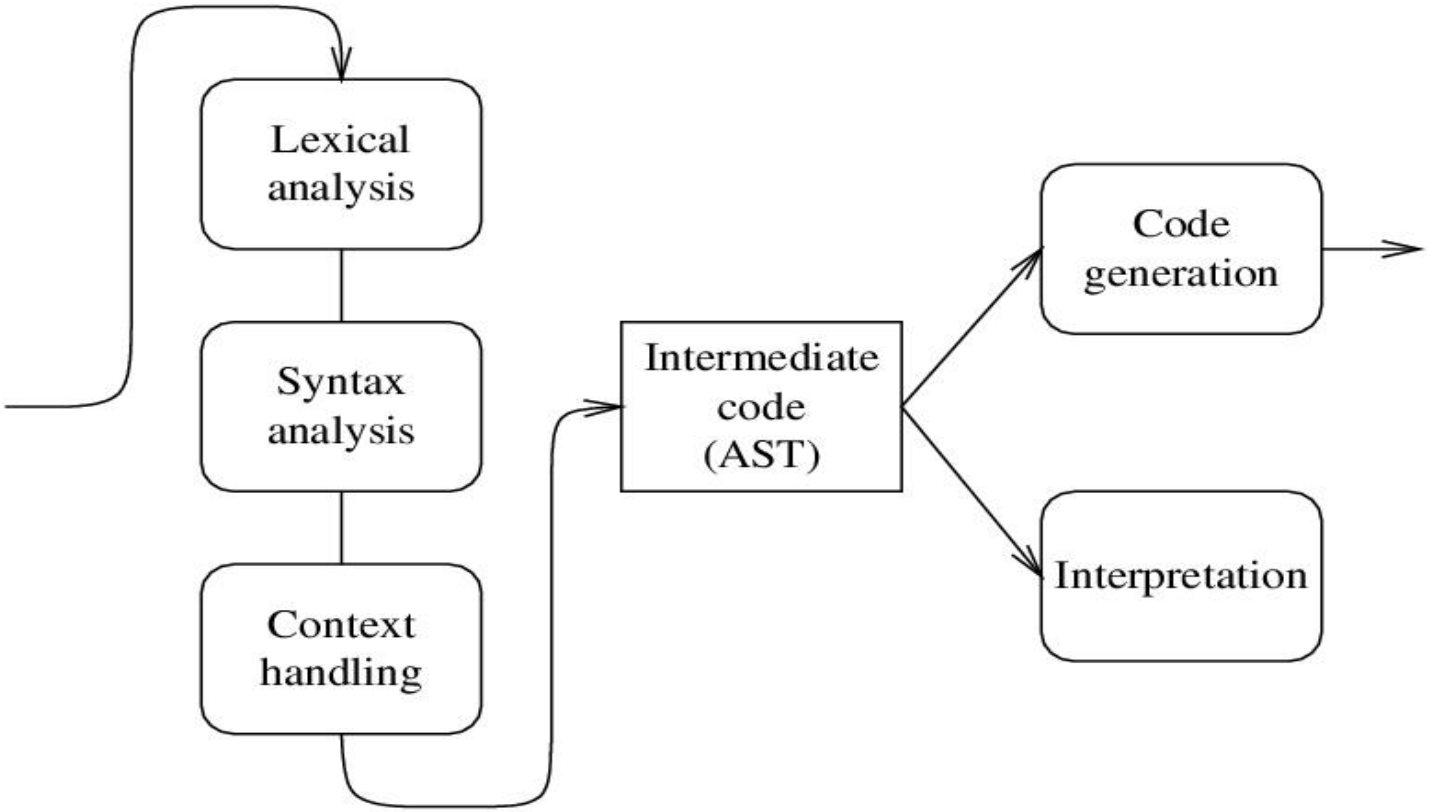


Figure 1.8 Structure of the demo compiler/interpreter.

Repräsentation eines AST als verzeigerte Struktur

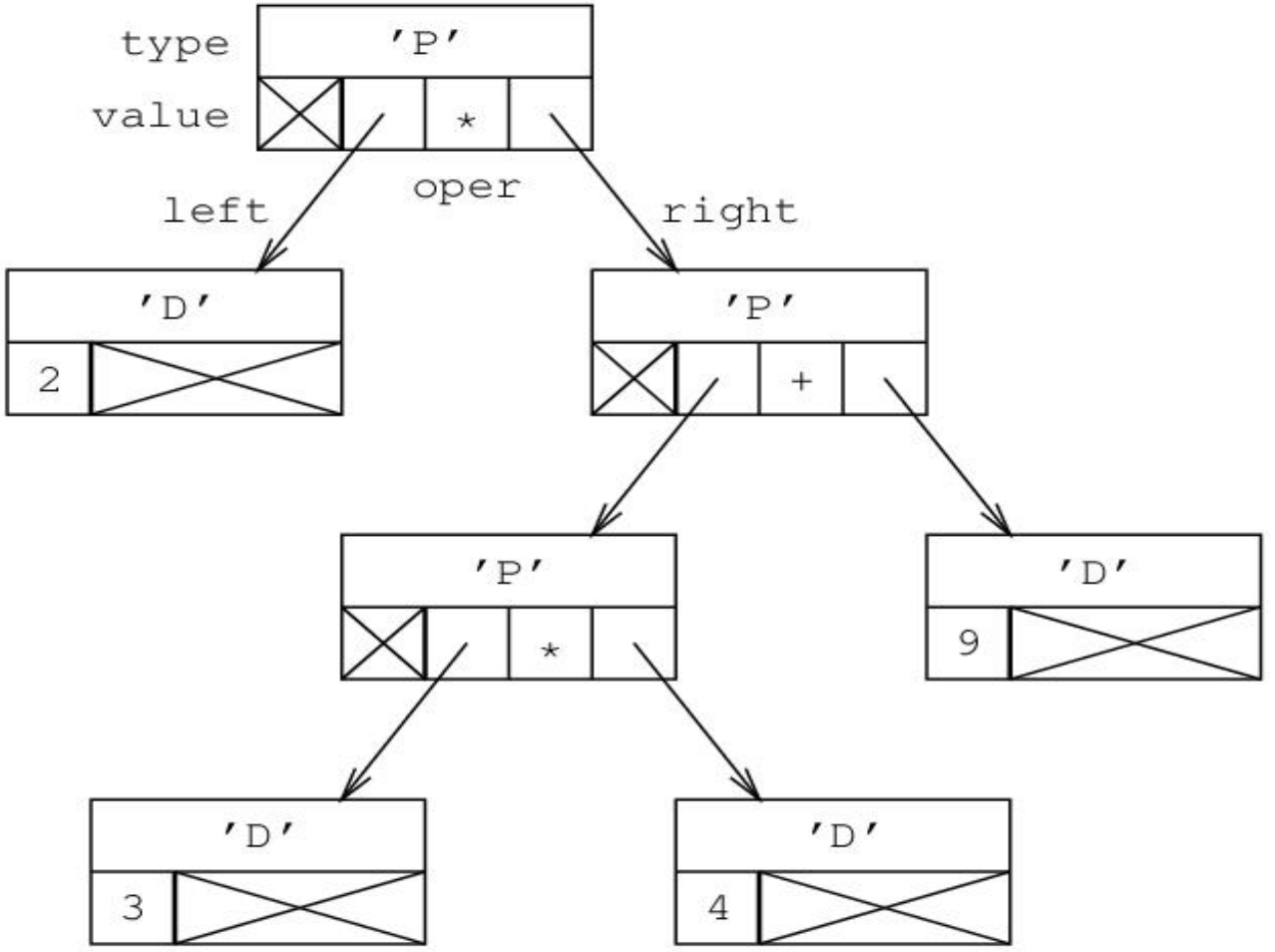


Figure 1.17 An AST for the expression `(2*((3*4)+9))`.

Einfache Codeerzeugung aus einem AST (für Stackmaschine)

```
#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.

Alternativen für den Zwischencode

- die Wahl des Zwischencodes ist variabel, Möglichkeiten:
  - (a) AST: annotierter abstrakter Syntax-Baum oder -Graph
  - (b) lineare Darstellungen wie Dreiadresscode

    Bsp.: `x=a*b; y=x+1`
  - (c) Befehle für eine abstrakte Stackmaschine

    Bsp.: `push a; push b; mult; push 1; add; pop y`
- Festlegung auf eine bestimmte Form
  - notwendig für Aussagen über eine konkrete Implementierung
  - nicht sinnvoll für eine Übersichtsvorlesung