# Grundlagen des Compilerbau Zusammenfassung

Thomas Mohr

# Contents

# 1 Introduction

## 1.1 Language Processors

- Compilers are programs that
  - read programs in source language
  - produce equivalent programs in target language
  - report errors encountered during translation

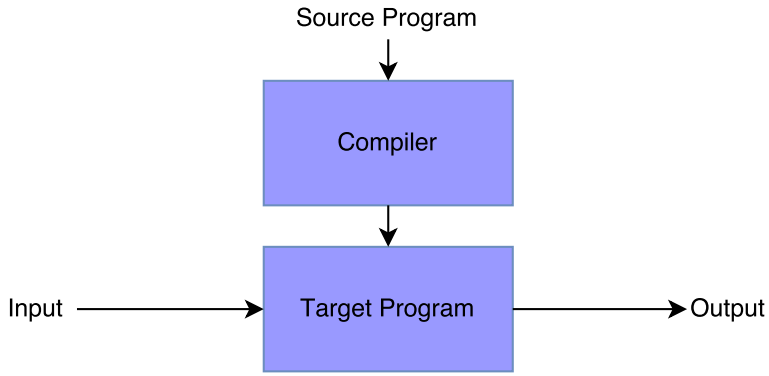


Figure 1: Compiler

- Interpreters are programs that
  - read programs in source language
  - read input
  - directly perform operations of source programs

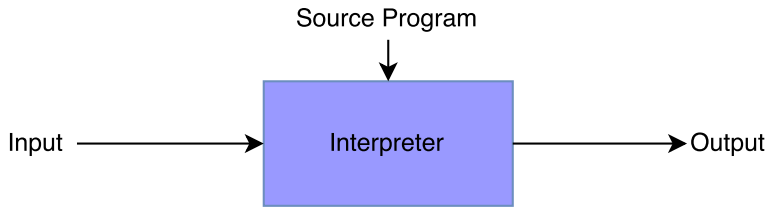


Figure 2: Interpreter

- In hybrid approaches
  - compiler first generates intermediate code, not directly executable on physical machine
  - interpreter processes intermediate code

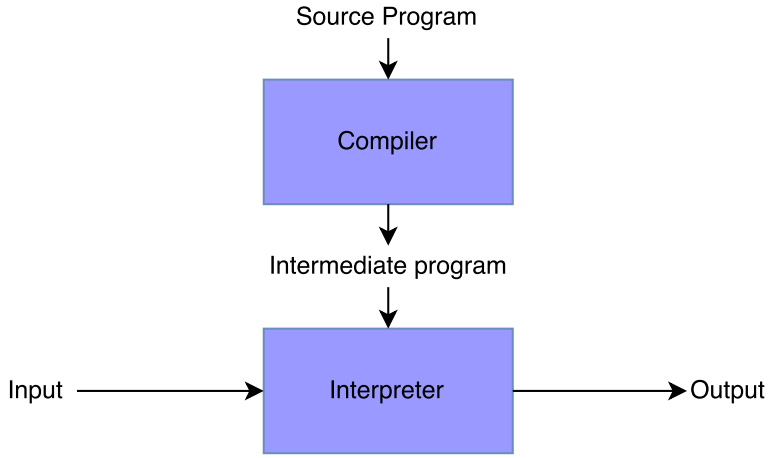


Figure 3: Hybrid

## 1.2 Compiler Phases

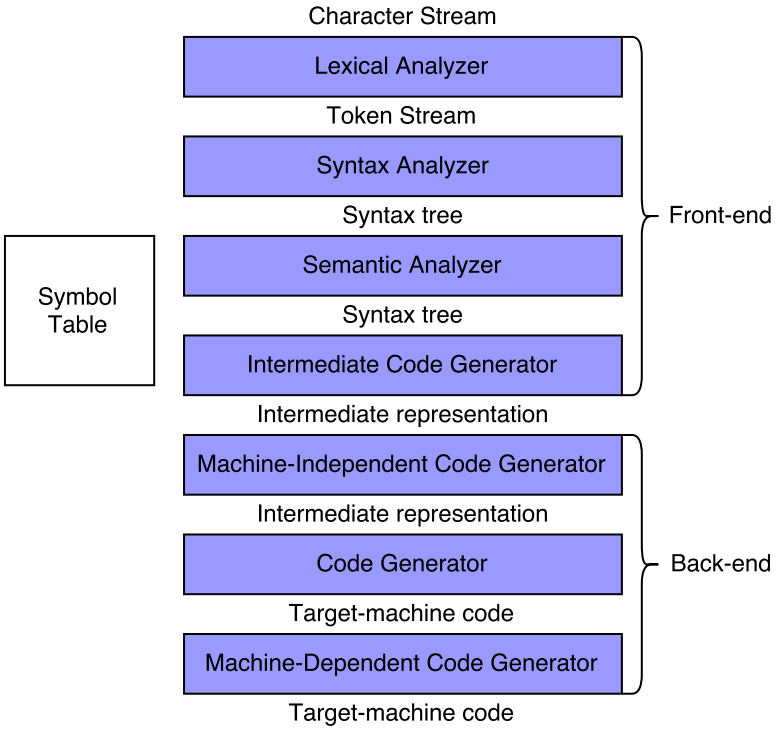


Figure 4: Anatomy of a Compiler

### 1.2.1 Lexical Analysis or Scanning

- Read character stream
- Group characters into lexemes
  - Meaningful to compiler
  - Has token name and attribute values
- Examples
  - $60 : \langle \text{integer}, 60 \rangle$
  - new : $\langle$keyword$\rangle$
  - setValue : $\langle$identifier, 'setValue'$\rangle$

### 1.2.2 Syntax Analysis or Parsing

- Arrange tokens in trees
- Representing grammatical structure of program
- Disambiguates possible meanings (e.g. operator precedence)

### 1.2.3 Semantic Analysis

- The grammar generally allows writing more programs than are meaningful
  - Using stricter grammar would exclude too many meaningful programs
  - Semantic analysis checks whether program can be executed
- For example:
  - Are value types of operands compatible?
  - Are identifiers used that refer to applicable program element?
- Gather type information in symbol table

### 1.2.4 Intermediate Code Generation

- Optionally generate intermediate code
- Properties
  - Easy to generate
  - Easy to translate to target machine code
  - Easy to optimize

### 1.2.5 Code Optimization

- Improve intermediate code
  - Faster
  - Shorter
  - Consuming less energy
- E.g., statically evaluate sub expressions

### 1.2.6 Code Generation

- Only truly machine-specific phase
- Map intermediate code operations to physical machine operations
  - Possibly exploit special-purpose instructions
  - Map variables to registers or other locations

## 1.3 Symbol Table Management

- Maintain information on visible symbols
  - Name
  - Type
  - Signature of procedures
- Variables have a scope
  - Global
  - Local
  - Nested scopes
- Symbol table shared among all phases

## 1.4 Environment

- Rules for determining location corresponding to name
- Resolution depends on different factors
  - Kind of program element referenced by name
  - Programming language design
- Generally determined by two policies
  - Binding (find location for name)
  - Scoping (find correct decleration for name)

### 1.4.1 Binding

- Bind name to location
- Static (location is always the same for each appearance of name)
- Dynamic (location may be different each time name is used)
- Binding most commonly is dynamic

### 1.4.2 Scoping

- Range of validity for name decleration
  - Where can a declared name be used?
  - Upon usage of name: Which is the decleration it refers to?
- Strategies
  - Static scoping
    - Sequence of top-level declerations of variables and functions
      - Valid until end of program
    - Functions start with sequence of local variable declerations
      - Valid until end of function
      - Shadow global declerations with same name
  - Block structure
    - Block is specific kind of instruction
    - Start by sequence of declerations
    - Followed by statements
    - Decleration shadows declerations in outer blocks/function/global scope
  - Dynamic scoping
    - Scope determined at runtime
    - Decleration is valid from the moment it is encountered at runtime

## 1.5 Parameter Passing

- Call-by-value
  - Value of variable is copied before procedure call
  - Expressions are evaluated into temporary variable first
  - Procedure gets own storage location for each parameter
  - Can be costly for large data types

  - Example (Main): $2 * 2 - 4 * 1 * 3 = -8$

- Call-by-reference
  - Reference to storage location is passed to procedure
  - Can modify value of variable from caller

- Call-by-name
  - Pass full expressions to procedure
  - Evaluate in place where formal parameter is used
  - Example (Main): $1 * 2 - 4 * 3 * 4 = -46$

```
class Main {
    int index = 0;
    int[] arr = {0,1,2,3,4,5};

    int discr(int a, int b, int c) {
        int value = b * b - 4 * a * c;
        return value;
    }

    int arrNext() {
        index++;
        return arr[index];
    }

    public static void main(String... args) {
        System.out.print(discr(
            arrNext(),
            arrNext(),
            arrNext()
        ));
    }
}
```

## 1.6 Aliasing

- Call-by-reference semantic can lead to aliases

- Aliases prevent optimizations
  - Assume local variable has single assignment
  - Optimization could replace occurence of variable with constant

– If aliases exist for variable, undetected changes may occur and therefore optimization may be incorrect

# 2 Lexical Analysis

## 2.1 Scanner

- Definitions
  - $\Sigma$: Input alphabet
  - $p \in \Sigma^*$: Input
  - $L \in \mathcal{P}$: Language
- Tasks
  - Split source program into sequence of lexemes
  - Transform sequence of lexemes to sequence of tokens
- Many different lexemes play the same role
  - Group into classes of lexemes: tokens
  - Token: $\langle$token-name, attribute(s)$\rangle$

## 2.2 Typical kinds of Tokens

- Identifiers
  - Sequence of letters and digits
  - Start with letter
- Numbers
  - Sequence of digits
  - Possibly with a sign
- Keywords
- Special characters
  - Operators: $+, *, \ldots$
  - Parantheses: $(, ), \{, \}$
  - $\ldots$
- Compund symbols
  - Operators: $==, <=, ++, \ldots$
- Whitespace
  - Space
  - Tab
  - Newline

  - ...
- Special symbols
  - Comments
  - Pragmas (Compiler Anweisungen)

## 2.3 Regular Expressions

- $RE(\Sigma)$: Regular Expression
  - $\epsilon \in RE(\Sigma)$: The empty expression
  - $\forall a \in \Sigma . a \in RE(\Sigma)$: All characters of $\Sigma$
  - Given $\alpha, \beta \in RE(\Sigma)$
    - $(\alpha|\beta) \in RE(\Sigma)$
    - $(\alpha \cdot \beta) \in RE(\Sigma)$
    - $(\alpha^*) \in RE(\Sigma)$
  - Closed under regular composition
  - Order of precedence: $*, \cdot, |$
- $[[RE(\Sigma)]]$: Language generated by regular expression
  - $[[\epsilon]] := \emptyset$
  - $[[\alpha]] := \{\{a\} | \forall a \in \Sigma\}$
  - $[[\alpha|\beta]] := [[\alpha]] \cup [[\beta]]$
  - $[[\alpha \cdot \beta]] := [[\alpha]] \cdot [[\beta]]$ (element-wise concatenation)
  - $[[\alpha^*]] := [[\alpha]]^*$ (closure)
  - $[[\epsilon^*]] := \{\epsilon\}$

## 2.4 Non-Deterministic Finite Automaton