

VO Übersetzerbau

E. Mehofer

Institut für Scientific Computing

Universität Wien

WS 2005/2006

Compiler Design, WS 2005/2006

Course Description

Contents:

- Introduction to compiler design.

Prerequisite:

- Programming experience, theory.

Textbook for course:

- K.C. Louden. *Compiler Construction: Principles and Practice*. Course Technology (PWS Publishing Company), 1997.

Important facts:

- Tue 16-17:30 HS 23
- Course page:
<http://www.par.univie.ac.at/~mehofer/teach/CC/VO.html>
- Contact: Wed 12:30-13:30, UZA 4, Room C314
Email: mehofer@par.univie.ac.at

Compiler Design, WS 2005/2006

Overview Literature

3

Basic compiler books:

- A.V. Aho, R. Sethi, J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986. ("Dragon Book").
(In preparation: A.V. Aho, R. Sethi, J.D. Ullman, M. Lam. *21st Century Compilers*. Addison-Wesley, 2005/2006/2007.)
- K.D. Cooper, L. Torczon: *Engineering a Compiler*. Morgan Kaufmann, 2004.
- A.W. Appel: *Modern Compiler Implementation in C/ML/ Java*.

Advanced compiler books:

- S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
- F. Nielson, H.R. Nielson, C. Hankin: *Principles of Program Analysis*. Springer, 1999.
- Y.N. Srikant, P. Shankar: *The Compiler Design Handbook: Optimizations & Machine Code Generation*. CRC Press, 2002.

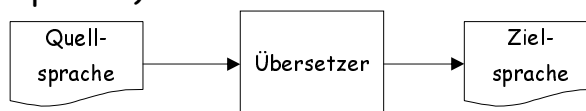
Compiler books for parallel computing:

- H. Zima, B. Chapman: *Supercompilers for Parallel and Vector Computers*. ACM Press, 1991.
- M. Wolfe: *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

Compiler Design, WS 2005/2006

Übersetzer (Compiler): Programm das eine Quellsprache in eine Zielsprache übersetzt (Compiler: Zielsprache ist eine Maschinsprache)

4



- Input: Quellsprache
- Output: (semantische äquivalente) Zielsprache

Beispiele:

Input	Output	Name
C	Maschinencode	cc
Fortran	Maschinencode	g77
PDF	Postscript	pdf2ps
Assembler	Maschinencode	as
Maschinencode	Assembler	dis

Compiler Design, WS 2005/2006

Warum Compilerbau?

5

- Compiler schließen Kluft zwischen höheren Programmiersprachen und Maschinensprachen:
 - ⇒ Compilerbau führt zu besserem Verständnis von beiden.
 - ⇒ Wichtig für Effizienzüberlegungen.
- Compilerbau ist ein wichtiges Anwendungsgebiet der theoretischen Informatik.
- Compilerbau ist ein Spezialgebiet von Software Engineering.
- Compilerbau ist ein klassisches Gebiet der Informatik.

Compiler Design, WS 2005/2006

Informelle Einführung in Sprachen (1)

6

Eine Sprachbeschreibung erfolgt i.a. in 3 Ebenen:

1. Lexikalische Ebene
 - Schreibweise: Rechtschreibung.
 - Deutsch: Duden-Rechtschreibung (alle Worte),
C++: Liste der Keywords, Zahlenformate, etc.
2. Syntaktische Ebene
 - Wohlgeformte Sätze: Grammatik.
 - Englisch: Wordorder "Subject Predicate Object",
C++: Syntaxregeln für Schleifen, etc.
3. Semantische Ebene
 - Bedeutung von wohlgeformten Sätze (richtig?, falsch?).

Compiler Design, WS 2005/2006

Informelle Einführung in Sprachen (2)

7

Beispiele:

- Deutsch: "Hörsal." - lexikalisch falsch.
- Deutsch: "Hund Mauer schnell." - lexikalisch richtig aber syntaktisch falsch.
- C++: "while; if(;;) for @fred" - lexikalisch falsch.
- C++: "while; if(;;) ffor fred" - lexikalisch richtig aber syntaktisch falsch.
- Deutsch: "Ein Baum ist ein Säugetier." - syntaktisch richtig aber semantisch falsch.
- C++: "float I; I="Hello"; I=3.14; I++;" - syntaktisch richtig aber semantisch falsch.
- Deutsch: "Ein Hund ist ein Säugetier." - semantisch korrekt.

Compiler Design, WS 2005/2006

Informelle Einführung in Sprachen (3)

8

Lexikalische Ebene:

- Beschreibung mittels regulärer Ausdrücke.

Syntaktische Ebene:

- Beschreibung mit Grammatiken.
- Linguist Noam Chomsky definierte eine 4-stufige Hierarchie von Klassen von Grammatiken (1956,1959).
- Zur Beschreibung von Programmiersprachen sind vor allem kontextfreie Grammatiken von Bedeutung.

Semantische Ebene:

- Beschreibung in Prosa.
- axiomatische Semantik.
- wp-Semantik.

Compiler Design, WS 2005/2006

Phases of a Compiler

11

1. Lexical analyzer (Scanner): input characters \Rightarrow tokens
2. Syntax analyzer (Parser): tokens \Rightarrow syntax tree
3. Semantic analyzer: type checking etc.
4. Intermediate code generator: annotated tree \Rightarrow intermediate representation IR
5. Code optimizer: optimizes intermediate code
6. Code generator: Optimized IR \Rightarrow assembly code / machine code

Compiler Design, WS 2005/2006

Lexical Analysis (Scanning)

12

- Input: stream of characters (source code file)
- Output: stream of tokens

Examples:

	source code	token
operator	>=	OP_GE
constants	34	INTEGER:34
identifier	fred	ID:fred
keyword	while	WHILE

Compiler Design, WS 2005/2006

Syntax Analysis (Parsing)

13

- Input: stream of tokens
- Output:
 - syntax tree (represents syntactic structure of the program)
 - syntax errors to the user

Example:

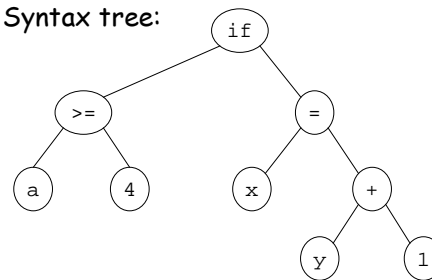
```
if (a>=4) x=y+1;
```

Scanner

```
(IF) (LPAREN) (ID:a) (OP_GE)  
(INTEGER:4) (RPAREN) (ID:x)  
(OP_AS) (ID:y) (OP_PLUS)  
(INTEGER:1) (SC)
```

Parser

Syntax tree:



Compiler Design, WS 2005/2006

Semantic Analysis

14

- Gathers semantic information, e.g. symbol table for subsequent phases.
- Checks semantic rules (type checking):
 - Is variable x a scalar or an array?
 - Is an expression type-consistent?
 - Does the dimension of an array reference match the declaration?
 - Is an array reference in bounds?

Compiler Design, WS 2005/2006

Intermediate Code Generation

15

- Input: syntax tree, symbol table
- Output: intermediate representation (IR)

Intermediate representation is machine independent representation of the program.

Why intermediate representation?

- Appropriate representation for subsequent phases.
- Break compiler into manageable pieces.
- Simplifies retargeting new host.
- Multiple passes.

Compiler Design, WS 2005/2006

Code Optimization

16

An optimization is a transformation that

1. improves the runtime of a program, and/or
2. decreases the size of a program, and/or
3. decreases power consumption, etc.

i.e. ***optimization goal*** important

Machine independent optimizations:

- eliminate dead code
- perform constant folding
- move loop-invariant code out of loops

Machine dependent optimizations:

- replace costly operation with a cheaper one
- data prefetching

Compiler Design, WS 2005/2006

Code Generation

17

- Input: intermediate representation (IR)
- Output: assembly code / machine code

Issues:

- Memory management
- Instruction selection
- Register allocation
- Instruction scheduling

Compiler Design, WS 2005/2006

Grouping of Phases

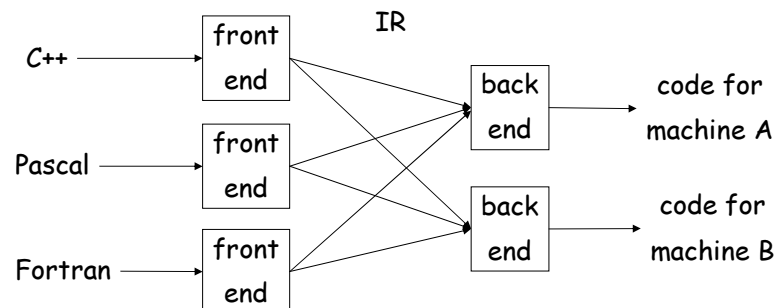
18

- **Front end:** machine independent phase
 - lexical analysis
 - syntax analysis
 - semantic analysis
 - intermediate code generation
 - some code optimization
- **Back end:**
 - code optimization
 - code generation

Compiler Design, WS 2005/2006

Interesting Approach

19



$m \times n$ compilers with $m + n$ components:

- all features must be represented in one IR

Compiler Design, WS 2005/2006

Compiler Passes

20

One pass compiler:

- All phases performed in a single pass.
- Possible for C, Pascal, however not for Modula-2.

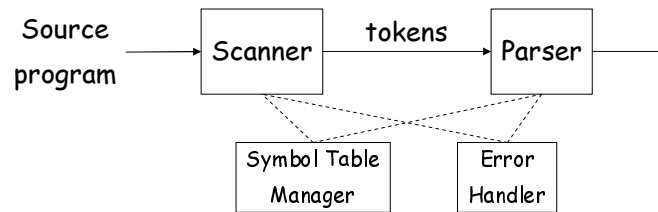
Typically multiple passes:

- Scanning, parsing, and semantic analysis.
- Optimization (several passes?)
- Code generation.

Compiler Design, WS 2005/2006

Lexical Analysis (Scanner)

21



Lexical analysis: Read input characters and map them into sequence of tokens.

Token:

- Scanner view: logical cohesive sequence of characters.
- Parser view: basic unit of syntax (terminal of grammar).
Therefore: no comment token - usually recognized and discarded.

Compiler Design, WS 2005/2006

Token-Lexeme Pair

22

- **Categories of tokens:**

- Keywords: if, while, float, char, ...
- Identifier: fred, x, foo, ...
- Literals: numeric constants, string literals, characters.
- Special symbols: arithmetic operators, parenthesis, assignment, equality, ...

- **Lexeme or string value:** character sequence forming a token.

- Example:

<u>x</u>	<u>=</u>	<u>1</u>	<u>;</u>
token: ID	token: OP_AS	token: INTEGER	token: SC
lexeme: x	lexeme: =	lexeme: 1	lexeme: ;

- Let tuple $\langle \text{token}, \text{lexeme} \rangle$ denote a token and its attribute, then we get the following sequence:

$\langle \text{ID}, x \rangle \langle \text{OP_AS}, = \rangle \langle \text{INTEGER}, 1 \rangle \langle \text{SC}, ; \rangle$

(Note: Lexeme considered as attribute of token)

Compiler Design, WS 2005/2006

Specification and Recognition of Tokens

23

1. Specification of tokens:
 - **Regular expressions (RE)** - regular grammars.
 - A RE generates a language:
The RE r generates the language $L(r)$.
2. Recognition of tokens:
 - **Deterministic finite state automata (DFA)**.
 - A DFA accepts a language:
The DFA M accepts the language $L(M)$.
3. Requirement: $L(M) = L(r)$

Compiler Design, WS 2005/2006

Def. Regular Expressions over Alphabet Σ

24

1. ϵ is a RE: denotes $\{\epsilon\}$ (*set with empty string*)
2. if $u \in \Sigma$ then u is a RE: denotes $\{u\}$ (*set with string u*)
3. Suppose r and s are REs denoting languages $L(r)$ and $L(s)$:
 - a) $r | s$ is a RE denoting $L(r) \cup L(s)$. (*choice*)
 - b) rs is a RE denoting $L(r)L(s)$. (*concatenation*)
 - c) r^* is a RE denoting $L(r)^*$. (*repetition, Kleene closure*)
 - d) (r) is a RE denoting $L(r)$. (*parenthesis allowed*)

Compiler Design, WS 2005/2006

Extensions to REs (Notational Shorthands)

25

- One or more repetitions: r^+ ($r^+ = r r^*$).
- Zero or one instance (optional subexpression): $r^?$
i.e. $r^? = (r | \epsilon)$
- Range of characters: $[0-9]$, $[a-z]$, $[A-Z]$.
E.g. reasonable assumption: uppercase characters are between "A" and "Z" (true for ASCII character set)
- Any character: period "."
- Any character not in a given set: $\sim(u | v)$ i.e. any character that is not either u or v .
- Names for REs:
 - digit = $[0-9]$
 - letter = $[a-zA-Z]$

Compiler Design, WS 2005/2006

(Nondeterministic) Finite Automaton NFA

26

Definition:

A (nondeterministic) finite automaton M is a 5-tuple:

$M = (Q, \Sigma, \delta, q_0, F)$ with

- Q : finite set of states
- Σ : finite input alphabet
- δ : transition function $Q \times \Sigma \rightarrow 2^Q$ (Überföhrungsfkt.)
- $q_0 \in Q$: initial state
- $F \subseteq Q$: set of final states

Deterministic Finite Automaton DFA

- Restriction: $|\delta(q,x)| = 1$

Compiler Design, WS 2005/2006

Example DFA

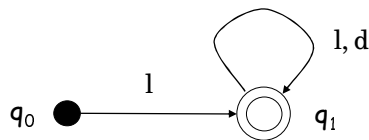
27

$M = (\{q_0, q_1\}, \{l, d\}, \delta, q_0, \{q_1\})$ with $\delta: \delta(q_0, l) = \{q_1\}, \delta(q_1, l) = \{q_1\}, \delta(q_1, d) = \{q_1\}$

- Transition table (Überführungstabelle):

input states \	l	d
q ₀	q ₁	-
q ₁	q ₁	q ₁

- Transition diagram (Überführungsdiagramm):

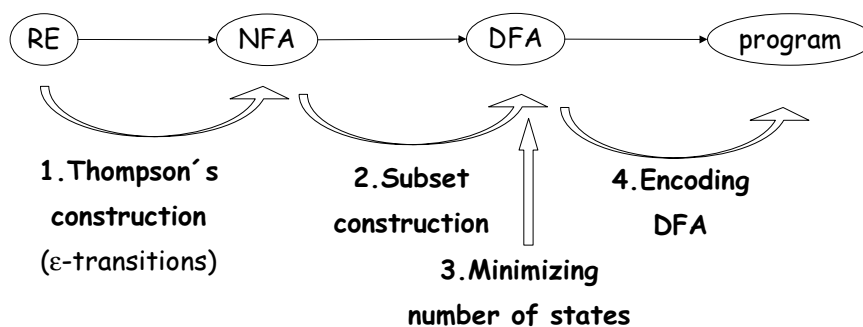


Compiler Design, WS 2005/2006

Scanner: From Regular Expression to Executable

28

Main task: From RE to DFA and encoding DFA.



- Simplest algorithm: first intermediate construction of an NFA.
- Also possible: from RE directly to DFA.
- For each DFA exists a unique minimum-state DFA.

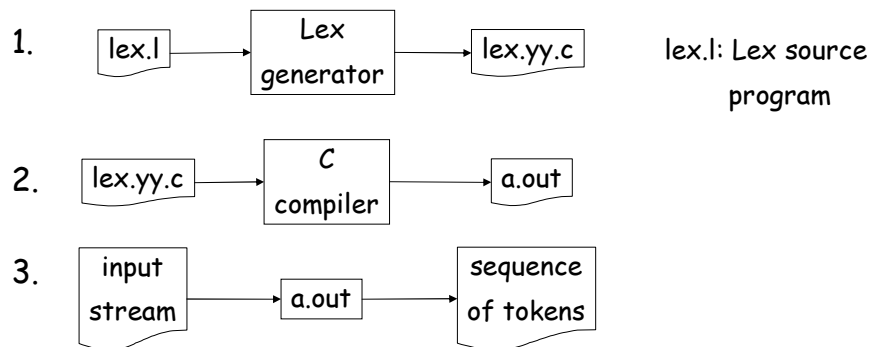
Compiler Design, WS 2005/2006

Scanner Generators

29

- All 4 steps are done automatically.
- Well-known scanner generators: lex, flex (fast lex, Gnu compiler package)

Creating a Scanner with Lex



Compiler Design, WS 2005/2006

Lex Specifications

30

A Lex program consists of 3 sections:

```
declarations
%%
translation rules
%%
auxiliary procedures
```

- Declaration section: contains regular definitions (special purpose notations based on regular expressions) to describe tokens to be matched.
- Translation rules: specify actions to be taken when a regular definition is matched.

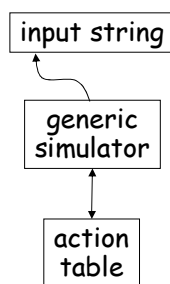
```
P1    { action1 }
...
Pn    { actionn }
```
- Auxiliary procedures: procedures needed by actions.

Compiler Design, WS 2005/2006

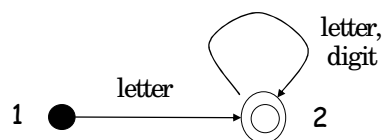
Encoding DFA

31

1. DFA "hardwired" into code:
 - for each state a segment of code.
 - entering state: input head contains current character.
 - leaving state: input head is moved to next character and control is transferred to next state.
2. DFA realized table-driven with "generic" simulator.



Example: Identifier



Compiler Design, WS 2005/2006

Encoding DFA: Directly Programmed V1 - Ex. ID

32

```
/*state 1*/
if (head==letter) {
  read-next();
  /*state 2*/
  while (head==letter || head==digit) {
    read-next();
  }
  return (ID); /*accept*/
}
else
  ... error or other ...
```

- Disadvantage:
 - ad-hoc approach with little systematic.
 - code very complex for non trivial automata.
 - hard to maintain (*hardwired approach*).

Compiler Design, WS 2005/2006

Encoding DFA: Directly Programmed V2 - Ex. ID

33

```
state=1;
while (state==1 || state==2) {
  switch (state) {
    case 1: switch (head) {
              case letter: read-next(); state=2;break;
              default: ...error or other...
            }
    case 2: switch (head) {
              case letter, digit: read-next();break;
              default: state=3;
            }
  }
  if (state==3) return(ID); /*accept*/
}
```

Doubly nested case statement inside loop:

first case statement: tests state,

second case statement: tests input character.

Version 2 advantageous to version 1.

Compiler Design, WS 2005/2006

Encoding DFA: Table-driven with Simulator (1)-ID

34

Data structure, indexed by states and input characters, is

derived from **transition table** and specifies actions:

- (read input and) go to state k .
- accept token and go to start state to wait for a request for another token.
- error occurred.

Code fragment of a simulator:

```
state=1;
head=read-next;
while (table[state,head]!=accept and
      table[state,head]!=error) {
  newstate=table[state,head];
  if (table[state,head]==advance) head=read-next();
  state=newstate;
}
if (table[state,head]==accept) {state=1; return ACCEPT;}
if (table[state,head]==error) {error-recovery; state=1;}
```

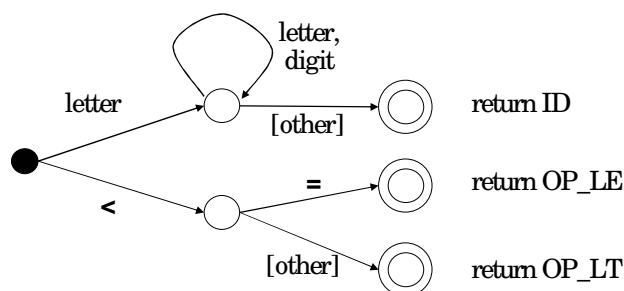
Compiler Design, WS 2005/2006

Encoding DFA: Table-driven with Simulator (2)-ID ³⁵

- Observation: simulator more powerful than a DFA acceptor.
- DFA does not represent everything, but gives an outline of the pattern matching process:
 - What to do when a character is matched?
Typical action: char from input string to lexeme.
 - What to do in accepting states?
Typical action: return token with associated attribute.
 - What to do in case of errors?
Typical action: backtracking or error-recovery.

Compiler Design, WS 2005/2006

Modified DFA ³⁶



- identifier: rule of longest match explicit.
- **Look-ahead character:**
 - [other]: returned to the input string and not consumed.
- and error handling

Compiler Design, WS 2005/2006

Table-driven Approach

37

- Advantages table-driven:
 - size of code reduced.
 - easier to maintain.
- Disadvantage table-driven:
 - table-driven approach slower than directly programmed automata.
 - tables can become very large.

Compiler Design, WS 2005/2006

Handling Keywords & Reducing States

38

Approach 1:

- separate DFA for each keyword.
- large number of states.

Approach 2:

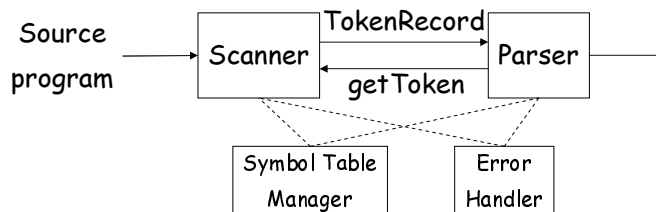
- put keywords in table, search table when an identifier is matched, return keyword.
- fewer states, smaller code.
- (also interesting for some identifier classes to simplify grammar of parser, e.g. typedef-names in C)

Compiler Design, WS 2005/2006

Scanning Process (1)

39

Lexical analyzer: subroutine of the parser.



```
typedef enum {  
    IF, THEN, ELSE, OP_PLUS, ...  
} TokenType  
typedef struct {  
    TokenType tokenval;  
    union {  
        char *stringval;  
        int numval;  
    } attribute;  
} TokenRecord;
```

Attribute: any value associated with a token.

- required if token matches more than one lexeme.

Compiler Design, WS 2005/2006

Scanning Process (2)

40

- Lexical Errors:
 - only few errors at lexical level (in modern languages).
 - error-recovery strategies:
 - » deleting characters until well-formed token found.
 - » inserting missing characters.
 - » replacing characters.
- Efficiency:
 - Large amount of compilation time spent in scanning.
 - Efficient I/O ("buffer pairs") and efficient handling of characters required (operation per char must be efficient).
 - Minimal states of automaton.

Compiler Design, WS 2005/2006