TDDD55 Compilers and interpreters

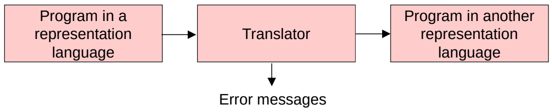
TDDB44 Compiler Construction

Compiler Construction Introduction

Peter Fritzson
IDA, Linköpings universitet, 2011

Introduction, Translators

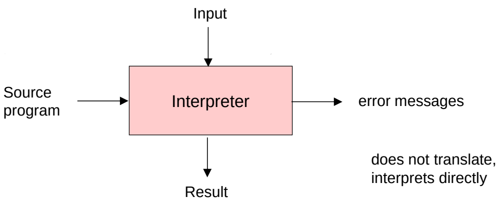
- **Compiler**



- High-level language → machine language or assembly language (Pascal, Ada, Fortran, Java, ..)
- Three phases of execution:
- "Compile time"
  1. Source program → object program (compiling)
  2. Linking, loading → absolute program
- "Run-time"
  3. Input → output

Interpreters

- High-level language → intermediate code – which is *interpreted,* e.g.
  - BASIC, LISP, APL
  - command languages, e.g. UNIX-shell
  - query languages for databases



Assembler

- Symbolic machine code → machine code

  e.g. MOVE R1,SUM → 01..101

Simulator, Emulator

- Machine code *is interpreted* → machine code
- e.g. Simulate a processor on an existing processor.

Preprocessor

- Extended ("sugared") high-level language → high-level language
- Example1: IF–THEN–ELSE in FORTRAN:
  Before preprocessing:

```
IF A < B THEN
    Z=A
ELSE
    Z=B
```

- After preprocessing:

```
        IF (A.LT.B) THEN GOTO 99
            Z=B
        GOTO 100
99      Z=A
100     CONTINUE
```

- Example 2: "File inclusion"
  #include "fil1.h"

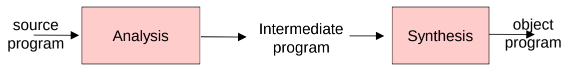
Natural Language – Translators

- e.g. Chinese → English
- Very difficult problem, especially to include context.
- Example 1: *Visiting relatives* can be hard work
  - To *go and visit* relatives ...
  - Relatives *who are visiting* ...
- Example 2: I saw a man with *a telescope*

Why High-Level Languages?

- Understandability (readability)
- Naturalness (languages for different applications)
- Portability (machine-independent)
- Efficient to use (development time) due to
  - separation of data and instructions
  - typing
  - data structures
  - blocks
  - program-flow primitives
  - subroutines

The Structure of the Compiler



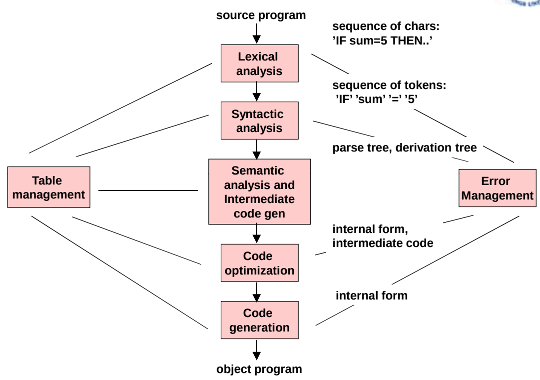
Logical organisation

- Analysis ("front-end"):

  Pull apart the text string (the program) to internal structures, reveal the structure and meaning of the source program.

- Synthesis ("back-end"):
  Construct an object program using information from the analysis.

The Phases of the Compiler



Compiler Passes and Phases

- **Pass:**
  - Physical organisation (phase to phase) dependent on language and compromises.
  - Available memory space, efficiency (time taken), forward references, portability- and modularity- requirements determine the number of passes.
- **The number of passes: (one-pass, multi-pass)**
  - The number of times the program is written into a file (or is read from a file).
  - Several phases can be gathered together in one pass.

Lexical Analysis (Scanner)

- **Input:**
  - Sequence of characters
- **Output:**
  - Tokens (basic symbols, groups of successive characters which belong together logically).

1. In the source text isolate and classify the basic elements that form the language:

Tokens	*Example*
Identifiers	Sum, A, id2
Constants	556, 1.5E-5
Strings	"Provide a number"
Keywords,reserved words	while, if
Operators	* / + -
Others	, ;

2. Construct tables (symbol table, constant table, string table etc.).

Scanner Lookahead for Tricky Tokens

- Example1: FORTRAN:

  DO 10 I=1,15 is a loop, but
  DO 10 I=1.15 is an assignment DO10I = 1.15

  NB! This is since blanks have no meaning in FORTRAN.

- Example 2: Pascal

  VAR i: 15..25; (15. is a real 15.. 15 is an integer)

Scanner Return Values

The scanner returns values in the form
<type, value>

Example: IF sum < 15 THEN z := 153

< 5, 0 > 5 = IF, 0 = lacks value
< 7, 14 > 7 = code for identifier,
14 = entry to symbol table
< 9, 1 > 9 = relational operator, 1 = '<'
< 1, 15> 1 = code for constant, 15 = value
< 2, 0 > 2 = THEN, 0 = lacks value
< 7, 9 > 7 = code for identifier,
9 = entry to symbol table
< 3, 0 > 3 = ':=', 0 = lacks value
< 1,153 > 1 = code for constant, 153 = value

Index	Symbol table		
9	z		
.			
.			
14	sum		

Regular expressions are used to describe tokens!

Syntax Analysis (parsing) 1 – Checking

- **Input:** Sequence of tokens
- **Output:** Parse tree, error messages
- **Function:**

1. Determine whether the input sequence forms a structure which is legal according to the definition of the language.

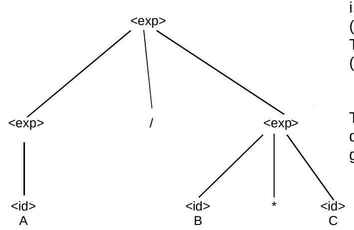
Example1: OK.
'IF' 'X' '=' '1' 'THEN' 'X' ':=' '1'

Example 2: Not OK.
'IFF' 'X' '=' '1' 'THEN' 'X' ':=' '1'
which produces the sequence of tokens:
< 7, 23 >
< 7, 16 > {Two identifiers in a row → wrong! }
< 9, 0 >
...

Syntax analysis (parsing) 2 – Build Trees

2. Group tokens into syntactic units and construct parse trees which exhibit the structure.

Example: A/B*C



This represents A/(B*C)
i.e. right-associative
(is this desirable?)
The alternative would be:
(A/B)*C – not the same!

The syntax of a language is described using a context-free grammar.

Semantic Analysis and Intermediate Code Generation 1 – More Checking.

- **Input:**
  - Parse tree + symbol table
- **Output:**
  - intermediate code + symbol table temp.variables, information on their type ...
- **Function:**
  - 1. Semantic analysis checks items which a grammar can not describe, e.g.
  - type compatibility a := i * 1.5
  - correct number and type of parameters in calls to procedures as specified in the procedure declaration.

Semantic Analysis and Intermediate Code Generation 2 - Generate Intermediate Code

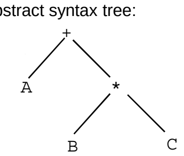
Example: A + B * C in the form of a parse tree

- Produces in reverse Polish notation:

  A B C * +
- Or three-address code:

  T1 := B * C

  T2 := A + T1
- Or abstract syntax tree:



- The intermediate form is used because it is:
  - Simpler than the high-level language (fewer and simpler operations).
  - Not profiled for a given machine (portability).
  - Suitable for optimisation.
- Syntax-directed translation schemes are used to attach semantic routines (rules) to syntactic constructions.

Code Optimization
(more appropriately: ‘‘Code Improvement’’)

- **Input:** Internal form
- **Output:** Internal form, hopefully improved.
- Machine-independent code optimisation:
  - In some way make the machine code faster or more compact by transforming the internal form.

Code Generation

- **Input:** Internal form
- **Output:** Machine code/assembly code
- **Function:**

1. Register allocation and machine code generation (or assembly code).
2. Instruction scheduling (specially important for RISC)
3. Machine-dependent code optimisation
   (so-called ‘‘peephole optimisation’’).

- Example: Z := A+B*C is translated to:

```
MOVE  1, B
IMUL  1, C
ADD   1, A
MOVEM 1, Z
```