

TDDDB44/TDDD55 Lecture 1: Compiler Construction Introduction

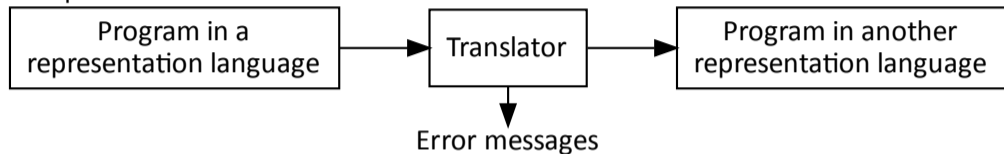
Adrian Pop and Martin Sjölund

Department of Computer and Information Science
Linköping University

2023-10-30

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

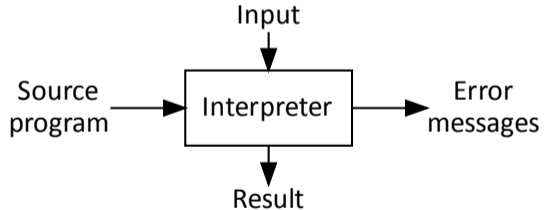
"Run-time"

3. Input → output

Interpreters

High-level language → intermediate code – which is *interpreted* directly, not translated, such as:

- ▶ BASIC, LISP, APL
- ▶ command languages such as UNIX-shell
- ▶ query languages for databases
- ▶ Early versions of JavaScript interpreters



Assembler

Symbolic machine code → machine code, for example:

MOVE R1,SUM → 01..101

Simulator, Emulator

Machine code *is interpreted* → machine code

Examples:

- ▶ Simulate a processor on an existing processor
- ▶ Running qemu on an amd64 laptop to run ARM Linux to test things
- ▶ Running old games on modern hardware

Generally, an emulator will try to mimic the behavior of the foreign architecture as best it can. A simulator will try to model the entire state of the foreign processor.

Preprocessor/Macro

Extended (“sugared”) high-level language → high-level language

Listing 1: “IF-THEN-ELSE in FORTRAN”

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

Listing 2: “FORTRAN after preprocessing”

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

Listing 3: “File inclusion in C”

Natural Language – Translators

For example 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 (if, switch, for, while, do-until, exceptions, goto)
 - ▶ subroutines

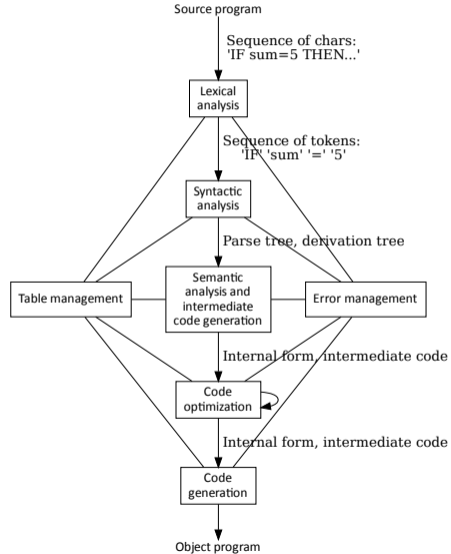
The Structure of the Compiler



Logical organization

- ▶ 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 organization (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:

<i>Tokens</i>	<i>Example</i>
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

Listing 4: FORTRAN

```
! A loop
  DO 10 I=1,15
! An assignment DO10I = 1.15
  DO 10 I=1.15
! Blanks have no meaning in FORTRAN.
```

Listing 5: Pascal

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

Scanner Return Values

Regular expressions are used to describe tokens, which the scanner returns values in the form: `<type, value>`

Listing 6: Example: `IF sum < 15 THEN z :`

```

< 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
  
```

Table: Symbol Table

Index	Symbol	Data
⋮		
9	z	...
⋮		
14	sum	...

Syntax Analysis (parsing) 1 – Checking

- ▶ Input: Sequence of tokens + symbol table
- ▶ 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.

Listing 7: OK

```
'IF' 'X' '=' '1' 'THEN' 'X' ':=' '1'
```

Listing 8: Not OK

```
'IFF' 'X' '=' '1' 'THEN' 'X' ':=' '1'
```

which produces the sequence of tokens:

```
< 7, 23 >
```

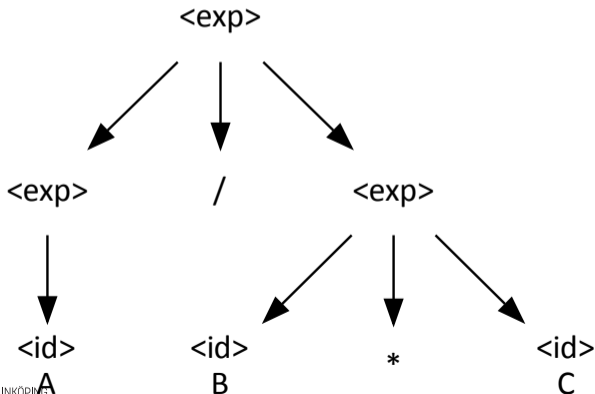
```
< 7, 16 > {Two identifiers in a row is wrong}
```

```
< 9, 0 >
```

Syntax Analysis (parsing) 2 – Build Trees

Function: (2) Group tokens into syntactic units and construct parse trees which exhibit the structure.

Figure: 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:
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: $A + B * C$

Listing 9: Reverse Polish notation

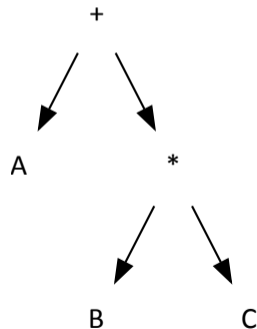
A B C * +

Listing 10: Three-address code

T1 := B * C

T2 := A + T1

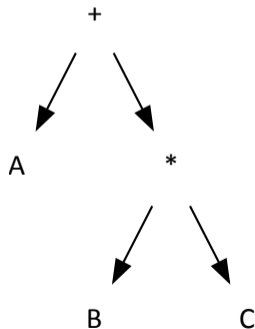
Figure: Abstract syntax tree



Semantic Analysis and Intermediate Code Generation 3 - Intermediate Code

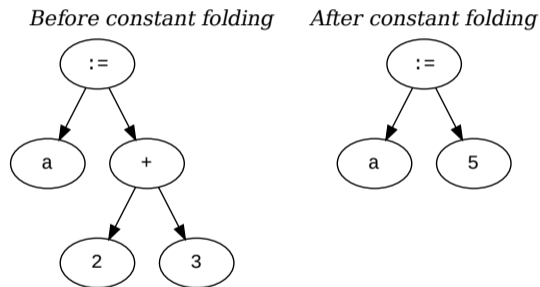
- ▶ The intermediate form is used because it is:
 - ▶ Simpler than the high-level language (fewer and simpler operations).
 - ▶ Not profiled for a given machine (portability).
 - ▶ Suitable for optimization.
- ▶ Syntax-directed translation schemes are used to attach semantic routines (rules) to syntactic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

Figure: Abstract syntax tree



Code Optimization (CO) - more appropriately: “Code Improvement”

- ▶ Input: Internal form
- ▶ Output: Internal form (hopefully improved)
- ▶ Machine-independent code optimization:
 - ▶ 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 optimization (so-called “peephole optimization”).

Listing 11: $Z := A + B * C$ is translated to assembly code

```
MOVE   R1 , B
IMUL   R1 , C
ADD    R1 , A
MOVEM  R1 , Z
```

Compiler evolution in the last 20 years - I

- ▶ Language diversity - more flexible compilers
- ▶ Better optimization techniques - loop optimizations, automatic vectorization and JIT
- ▶ Parallel and Multicore Support - compiler adaptation was needed - automatic parallelization and parallel languages
- ▶ Just-in-Time Compilation: JIT compilers for dynamic languages
- ▶ Better Intermediate Representation - to enable more optimization and better code generation
- ▶ Better Compiler Frameworks - LLVM, GCC, rust, C, Go, TS, WebAssembly

Compiler evolution in the last 20 years - II

- ▶ Language-Independent Tools - for various compiler phases parsing, type checking, code generation - greater flexibility and IDE improvements (LSP, etc)
- ▶ Better Error Reporting - much easier to fix errors
- ▶ Improved Security - stack checks (canaries), address sanitizer, control-flow integrity checks. Different computation paradigms for safety (rust).
- ▶ Compiler Verification via formal methods - advancements, can be used on non-toy examples to guarantee compiler correctness for safety critical systems.
- ▶ Support of new architectures - mobile, IoT, edge, GPUs, TPUs, FPGAs, etc.
- ▶ ML & AI for compiler construction to automate CO and CG.

www.liu.se