

**Dep: Computer science**

**Subject: Compiler**

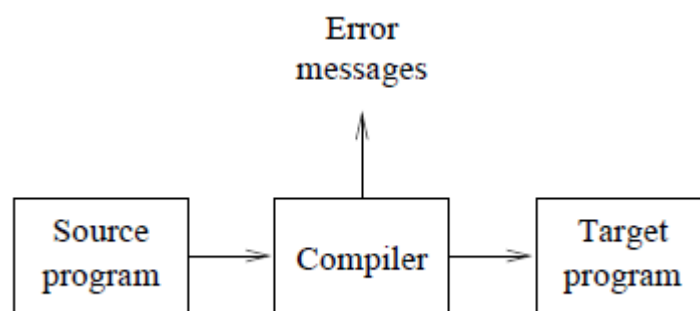
**Third Class**

## Lecturer (1)

### 1. Introduction

#### 1.1 Compilation

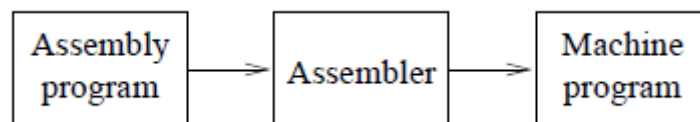
Definition. Compilation is a process that translates a program in one language (the source language) into an equivalent program in another language (the object or target language).



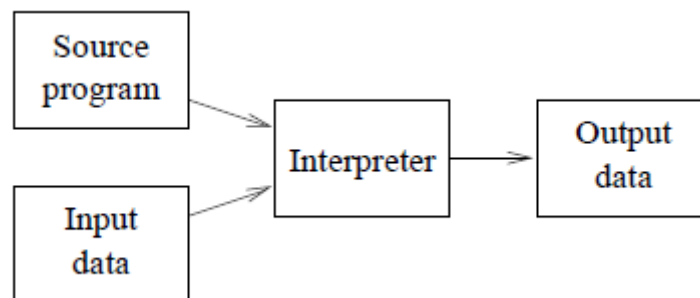
An important part of any compiler is the detection and reporting of errors; this will be discussed in more detail later in the introduction. Commonly, the source language is a high-level programming language (i.e. a problem-oriented language), and the target

language is a machine language or assembly language (i.e. a machine-oriented language). Thus compilation is a fundamental concept in the production of software: it is the link between the (abstract) world of application development and the low-level world of application execution on machines.

Types of translators. An assembler is also a type of translator:



An interpreter is closely related to a compiler, but takes both source program and input data. The translation and execution phases of the source program are one and the same.



Although the above types of translator are the most well-known, we also need knowledge of compilation techniques to deal with the recognition and translation of many other types of languages including:

- Command-line interface languages;
- Typesetting / word processing languages (e.g. TEX);
- Natural languages;
- Hardware description languages;
- Page description languages (e.g. PostScript);
- Set-up or parameter files.

## **Early Development of Compilers.**

1940's. Early stored-program computers were programmed in machine language. Later, assembly languages were developed where machine instructions and memory locations were given symbolic forms.

1950's. Early high-level languages were developed, for example FORTRAN. Although more problem-oriented than assembly languages, the first versions of FORTRAN still had many machine-dependent features. Techniques and processes involved in compilation were not well-understood at this time, and compiler-writing was a huge task: e.g. the first FORTRAN compiler took 18 man years of effort to write. Chomsky's study of the structure of natural languages led to a classification of languages according to the complexity of their grammars. The context-free languages proved to be useful in describing the syntax of programming languages.

1960's onwards. The study of the parsing problem for context-free languages during the 1960's and 1970's has led to efficient algorithms for the recognition of context-free languages. These algorithms, and associated software tools, are central to compiler construction today. Similarly, the theory of finite state machines and regular expressions (which correspond to Chomsky's regular languages) have proven useful for describing the lexical structure of programming languages.

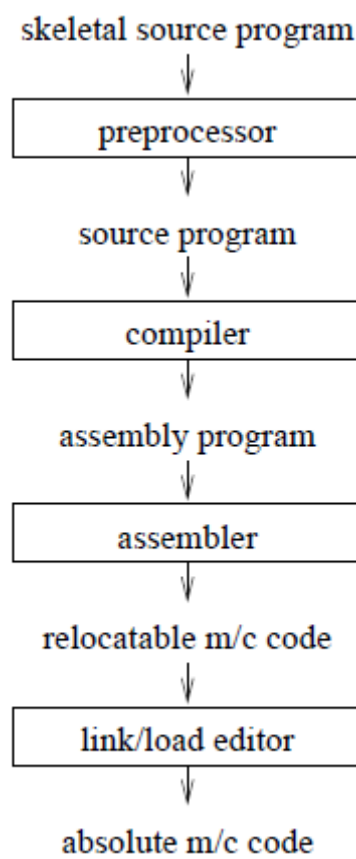
From Algol 60, high-level languages have become more problem-oriented and machine independent, with features much removed from the machine languages into which they are compiled.

The theory and tools available today make compiler construction a manageable task, even for complex languages. For example, your compiler assignment will take only a few weeks (hopefully) and will only be about 1000 lines of code (although, admittedly, the source language is small).

## Lecturer (2)

### 1.2 The Context of a Compiler

The complete process of compilation is illustrated as:



### 1.2.1 Preprocessors

Preprocessing performs (usually simple) operations on the source file(s) prior to compilation. Typical preprocessing operations include:

(a) Expanding macros (shorthand notations for longer constructs). For example, in C,

```
#define foo(x,y) (3*x+y*(2+x))
```

defines a macro foo, that when used in later in the program, is expanded by the preprocessor. For example, `a = foo(a,b)` becomes

```
a = (3*a+b*(2+a))
```

(b) Inserting named files. For example, in C,

```
#include "header.h"
```

is replaced by the contents of the file header.h

### 1.2.2 Linkers

A linker combines object code (machine code that has not yet been linked) produced from compiling and assembling many source programs, as well as standard library functions and resources supplied by the operating system. This involves resolving references in each object file to external variables and procedures declared in other files.

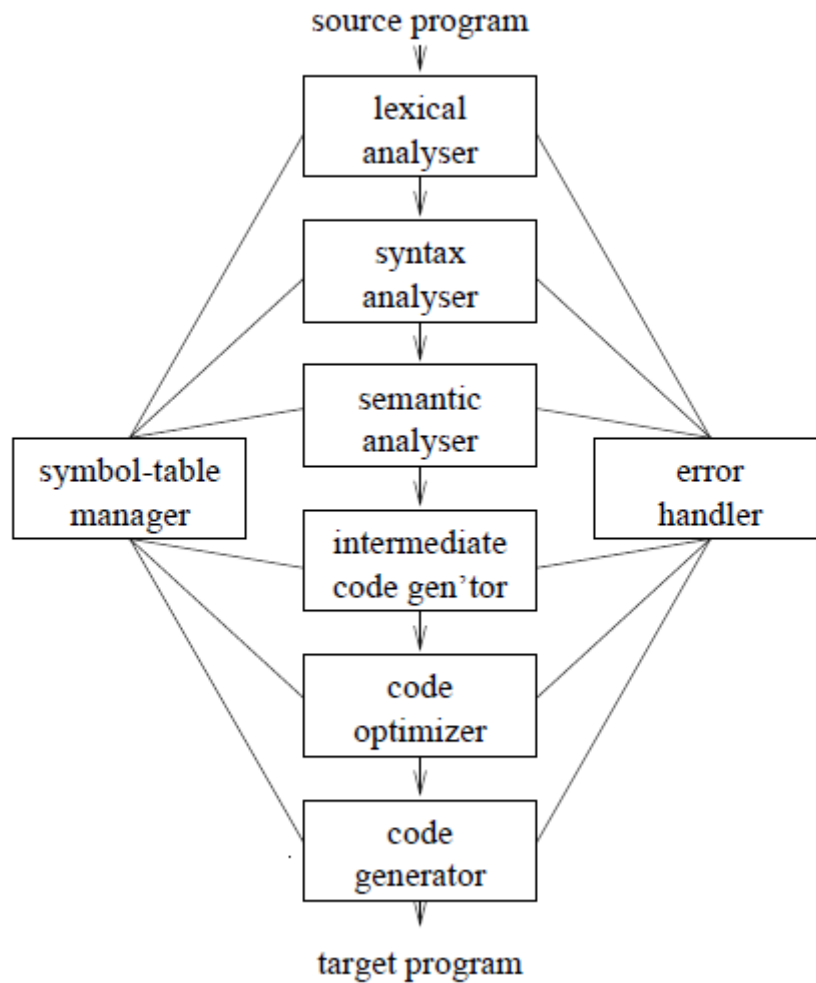
### 1.2.3 Loaders

Compilers, assemblers and linkers usually produce code whose memory references are made relative to an undetermined starting location that can be anywhere in memory (relocatable machine code). A loader calculates appropriate absolute addresses for these memory locations and amends the code to use these addresses.

## **Lecturer (3)**

### **1.3 The Phases of a Compiler**

The process of compilation is split up into six phases, each of which interacts with a symbol table manager and an error handler. This is called the analysis/synthesis model of compilation. There are many variants on this model, but the essential elements are the same.



### 1.3.1 Lexical Analysis

A lexical analyser or scanner is a program that groups sequences of characters into lexemes, and outputs (to the syntax analyser) a sequence of tokens. Here:

- (a) Tokens are symbolic names for the entities that make up the text of the program; e.g. if for the keyword if, and id for any identifier. These make up the output of the lexical analyser.
- (b) A pattern is a rule that specifies when a sequence of characters from the input constitutes a token; e.g the sequence i, f for the token if, and any sequence of alphanumeric starting with a letter for the token id.



- (c) A lexeme is a sequence of characters from the input that match a pattern (and hence constitute an instance of a token); for example `if` matches the pattern for `if`, and `foo123bar` matches the pattern for `id`.

For example, the following code might result in the table given below.

<i>Lexeme</i>	<i>Token</i>	<i>Pattern</i>
program	<i>program</i>	p, r, o, g, r, a, m newlines, spaces, tabs
foo	<i>id (foo)</i>	letter followed by seq. of alphanumerics
(	<i>leftpar</i>	a left parenthesis
input	<i>input</i>	i, n, p, u, t
,	<i>comma</i>	a comma
output	<i>output</i>	o, u, t, p, u, t
)	<i>rightpar</i>	a right parenthesis
;	<i>semicolon</i>	a semi-colon
var	<i>var</i>	v, a, r
x	<i>id (x)</i>	letter followed by seq. of alphanumerics
:	<i>colon</i>	a colon
integer	<i>integer</i>	i, n, t, e, g, e, r
;	<i>semicolon</i>	a semi-colon
begin	<i>begin</i>	b, e, g, i, n newlines, spaces, tabs
readln	<i>readln</i>	r, e, a, d, l, n
(	<i>leftpar</i>	a left parenthesis
x	<i>id (x)</i>	letter followed by seq. of alphanumerics
)	<i>rightpar</i>	a right parenthesis
;	<i>semicolon</i>	a semi-colon
writeln	<i>writeln</i>	w, r, i, t, e, l, n
(	<i>leftpar</i>	a left parenthesis
'value read ='	<i>literal ('value read =')</i>	seq. of chars enclosed in quotes
,	<i>comma</i>	a comma
x	<i>id (x)</i>	letter followed by seq. of alphanumerics
)	<i>rightpar</i>	a right parenthesis newlines, spaces, tabs
end	<i>end</i>	e, n, d
.	<i>fullstop</i>	a fullstop

It is the sequence of tokens in the middle column that are passed as output to the syntax analyser. This token sequence represents almost all the important information from the input program required by the syntax analyser. Whitespace (newlines, spaces and tabs), although often important in separating lexemes, is usually not returned as a token. Also, when outputting an id or literal token, the lexical analyser must also return the value of the matched lexeme (shown in parentheses) or else this information would be lost.

## Lecturer (4)

### 1.3.2 Symbol Table Management

A symbol table is a data structure containing all the identifiers (i.e. names of variables, procedures etc.) of a source program together with all the attributes of each identifier. For variables, typical attributes include:

- its type,
- how much memory it occupies,
- its scope.

For procedures and functions, typical attributes include:

- the number and type of each argument (if any),
- the method of passing each argument, and
- the type of value returned (if any).

The purpose of the symbol table is to provide quick and uniform access to identifier attributes throughout the compilation process. Information is usually put into the symbol table during the lexical analysis and/or syntax analysis phases.

### 1.3.3 Syntax Analysis

A syntax analyser or parser is a program that groups sequences of tokens from the lexical analysis phase into phrases each with an associated phrase type.

A phrase is a logical unit with respect to the rules of the source language. For example, consider:

$$a := x * y + z$$

After lexical analysis, this statement has the structure

$$\text{id1 assign id2 binop1 id3 binop2 id4}$$

Now, a syntactic rule of Pascal is that there are objects called ‘expressions’ for which the rules are (essentially):

(1) Any constant or identifier is an expression.

(2) If  $\text{exp}_1$  and  $\text{exp}_2$  are expressions then so is  $\text{exp}_1 \text{ binop } \text{exp}_2$ .

Taking all the identifiers to be variable names for simplicity, we have:

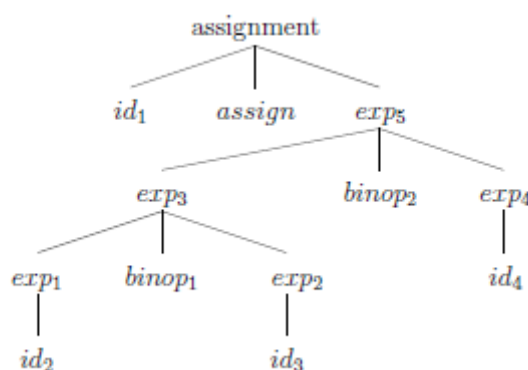
- By rule (1)  $\text{exp}_1 = \text{id}_2$  and  $\text{exp}_2 = \text{id}_3$  are both phrases with phrase type ‘expression’;
- by rule (2)  $\text{exp}_3 = \text{exp}_1 \text{ binop}_1 \text{exp}_2$  is also a phrase with phrase type ‘expression’;
- by rule (1)  $\text{exp}_4 = \text{id}_4$  is a phrase with type ‘expression’;
- by rule (2),  $\text{exp}_5 = \text{exp}_3 \text{ binop}_2 \text{exp}_4$  is a phrase with phrase type ‘expression’.

Of course, Pascal also has a rule that says:

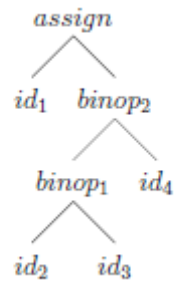
$\text{id assign exp}$

is a phrase with phrase type ‘assignment’, and so the Pascal statement above is a phrase of type ‘assignment’.

**Parse Trees and Syntax Trees.** The structure of a phrase is best thought of as a parse tree or a syntax tree. A parse tree is tree that illustrates the grouping of tokens into phrases. A syntax tree is a compacted form of parse tree in which the operators appear as the interior nodes. The construction of a parse tree is a basic activity in compiler-writing. A parse tree for the example Pascal statement is:



and a syntax tree is:



Comment. The distinction between lexical and syntactical analysis sometimes seems arbitrary.

The main criterion is whether the analyser needs recursion or not:

- lexical analysers hardly ever use recursion; they are sometimes called linear analysers since they scan the input in a ‘straight line’ (from left to right).
- syntax analysers almost always use recursion; this is because phrase types are often defined in terms of themselves (cf. the phrase type ‘expression’ above).

## Lecturer (5)

### 1.3.4 Semantic Analysis

A semantic analyser takes its input from the syntax analysis phase in the form of a parse tree and a symbol table. Its purpose is to determine if the input has a well-defined meaning; in practice semantic analysers are mainly concerned with type checking and type coercion based on type rules. Typical type rules for expressions and assignments are:

Expression Type Rules. Let  $\text{exp}$  be an expression.

- (a) If  $\text{exp}$  is a constant then  $\text{exp}$  is well-typed and its type is the type of the constant.
- (b) If  $\text{exp}$  is a variable then  $\text{exp}$  is well-typed and its type is the type of the variable.
- (c) If  $\text{exp}$  is an operator applied to further subexpressions such that:
  - (i) the operator is applied to the correct number of subexpressions,
  - (ii) each subexpression is well-typed and
  - (iii) each subexpression is of an appropriate type,then  $\text{exp}$  is well-typed and its type is the result type of the operator.

Assignment Type Rules. Let  $\text{var}$  be a variable of type  $T1$  and let  $\text{exp}$  be a well-typed expression of type  $T2$ . If

- (a)  $T1 = T2$  and
- (b)  $T1$  is an assignable type

then  $\text{var assign exp}$  is a well-typed assignment.

For example, consider the following code fragment:

```
intvar := intvar + realarray
```

where `intvar` is stored in the symbol table as being an integer variable, and `realarray` as an array of reals. In Pascal this assignment is syntactically correct, but semantically incorrect since `+` is only defined on numbers, whereas its second argument is an array. The semantic analyser checks for such type errors using the parse tree, the symbol table and type rules.

### 1.3.5 Error Handling

Each of the six phases (but mainly the analysis phases) of a compiler can encounter errors. On detecting an error the compiler must:

- report the error in a helpful way,
- correct the error if possible, and
- continue processing (if possible) after the error to look for further errors.

**Types of Error.** Errors are either syntactic or semantic:

Syntax errors are errors in the program text; they may be either lexical or grammatical:

- (a) A lexical error is a mistake in a lexeme, for examples, typing `tehn` instead of `then`, or missing off one of the quotes in a literal.
- (b) A grammatical error is a one that violates the (grammatical) rules of the language, for example if `x = 7 y := 4` (missing `then`).

Semantic errors are mistakes concerning the meaning of a program construct; they may be either type errors, logical errors or run-time errors:

- (a) Type errors occur when an operator is applied to an argument of the wrong type, or to the wrong number of arguments.
- (b) Logical errors occur when a badly conceived program is executed, for example:  
while `x = y` do ... when `x` and `y` initially have the same value and the body of loop need not change the value of either `x` or `y`.

- (c) Run-time errors are errors that can be detected only when the program is executed, for example:

```
var x : real; readln(x); writeln(1/x)
```

which would produce a run time error if the user input 0.

Syntax errors must be detected by a compiler and at least reported to the user (in a helpful way). If possible, the compiler should make the appropriate correction(s). Semantic errors are much harder and sometimes impossible for a computer to detect.

### **1.3.6 Intermediate Code Generation**

After the analysis phases of the compiler have been completed, a source program has been decomposed into a symbol table and a parse tree both of which may have been modified by the semantic analyser. From this information we begin the process of generating object code according to either of two approaches:

- (1) generate code for a specific machine, or
- (2) generate code for a 'general' or abstract machine, then use further translators to turn the abstract code into code for specific machines.

Approach (2) is more modular and efficient provided the abstract machine language is simple enough to:

- (a) produce and analyse (in the optimisation phase), and
- (b) easily translated into the required language(s).

One of the most widely used intermediate languages is Three-Address Code (TAC).

TAC Programs. A TAC program is a sequence of optionally labelled instructions. Some common TAC instructions include:

- (i)  $\text{var1} := \text{var2 binop var3}$
- (ii)  $\text{var1} := \text{unop var2}$
- (iii)  $\text{var1} := \text{num}$



(iv) goto label

(v) if var1 relop var2 goto label

There are also TAC instructions for addresses and pointers, arrays and procedure calls, but will use only the above for the following discussion.

**Syntax-Directed Code Generation.** In essence, code is generated by recursively walking through a parse (or syntax) tree, and hence the process is referred to as syntax-directed code generation. For example, consider the code fragment:

