Language Processors

Lesson 1

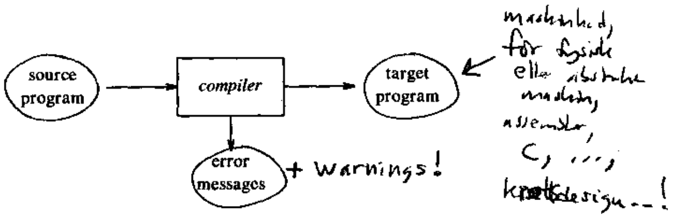
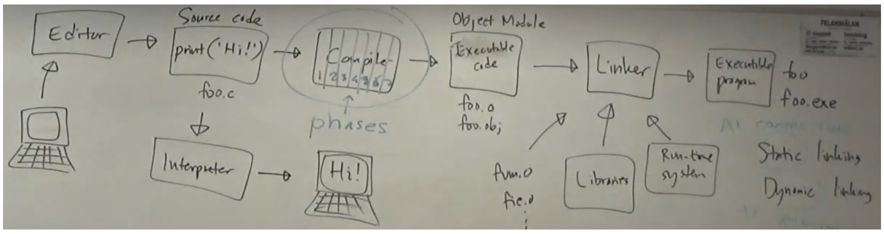
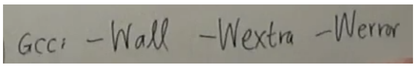


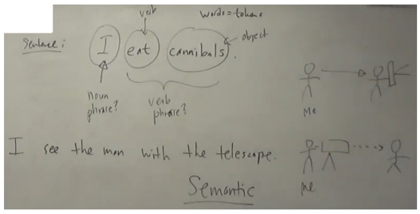
Fig. 1.1. A compiler.

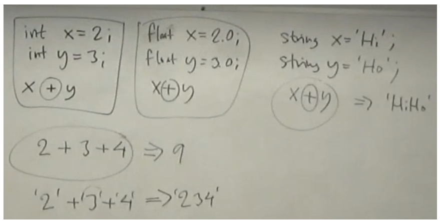
input: source program
output: target program, error messages, warnings



gcc warnings

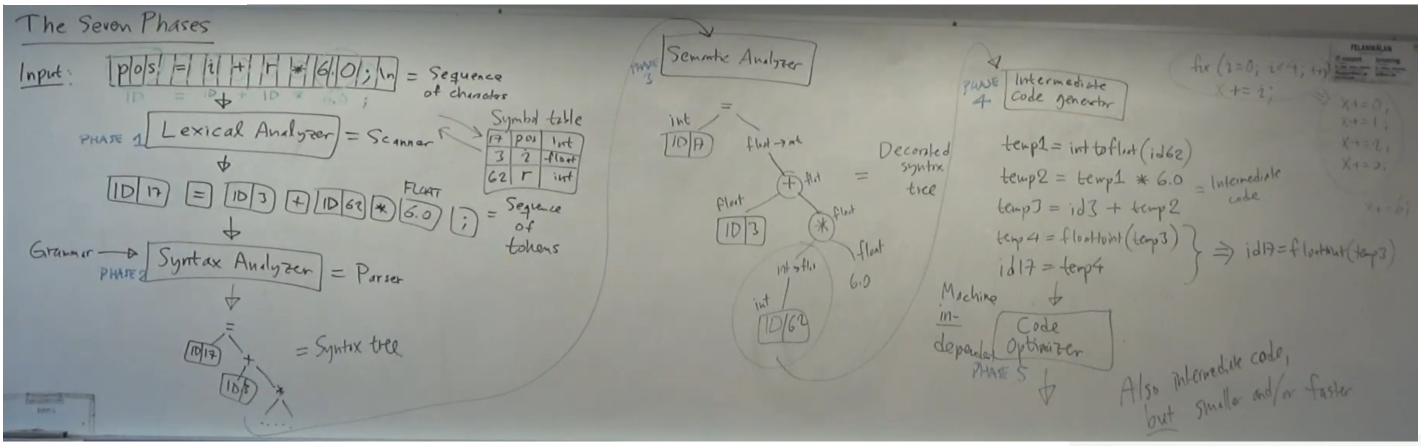


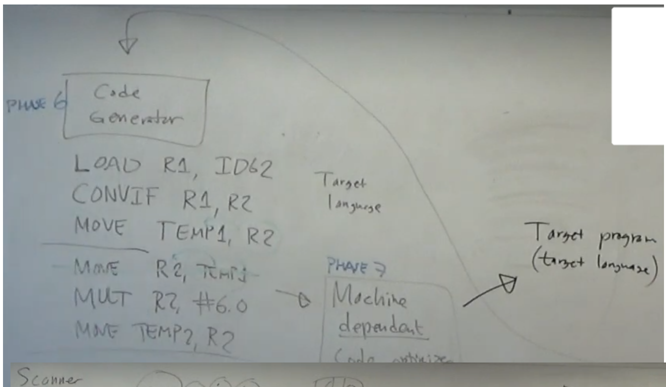


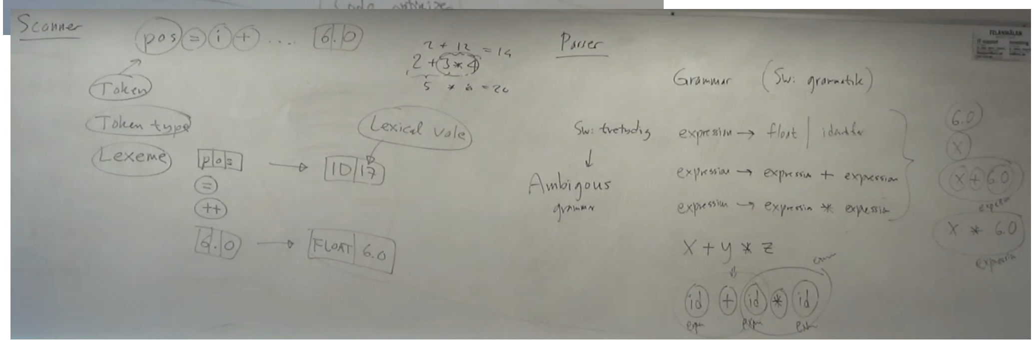


The Structure of a Compiler

All the phases







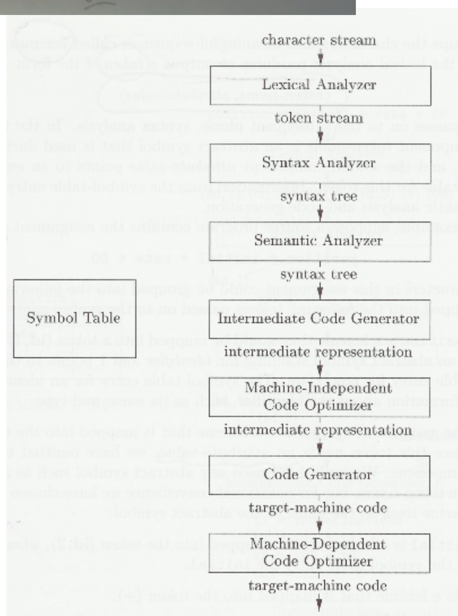


Figure 1.6: Phases of a compiler

Compiler-Construction Tools

- Scanner generators (ex: Lex, Flex)
- Parser generators (ex: Yacc, Bison)

Grammars for computer languages

A simple language: greetings

- hello
- good morning
- good afternoon
- good evening

Simple grammar:

greeting -> **hi** | **good morning** | **good afternoon** | **good evening**

Another grammar:

greeting -> **hi** | **good** *time_specification*
time_specification -> **morning** | **afternoon** | **evening**

Can also be written as:

greeting -> **Hi**
greeting -> **good** *time_specification*
time_specification -> **morning**
time_specification -> **afternoon**
time_specification -> **evening**

Grammar for the greeting language

Terminals:

- **hello**
- **good**
- **morning**
- **afternoon**
- **evening**

Non-terminals:

- *greeting*
- *time_specification*

Productions:

- *greeting* -> **hi** | **good** *time_specification*
- *time_specification* -> **morning** | **afternoon** | **evening**

Start symbol:

- *greeting*

Lesson 2

"Terminals", "Non-Terminals" and "Productions"

A *terminal* is something that you write in the language itself. → in this: five words **hello**, **good morning**, **afternoon** and **evening**

In grammar, we also use the names *greetings* and *time_specification*. They do not appear in the greeting language, but only in the grammar. They are called *non-terminals*, and we write them in *italics*.

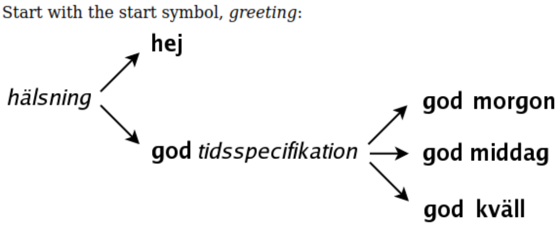
A *production* is a rule in the grammar, which states that a certain non-terminal can be replaced with something else.

The start symbol

In this case greeting is the start symbol.

How do you find out which language a grammar describes?

Start with the start symbol, *greeting*:

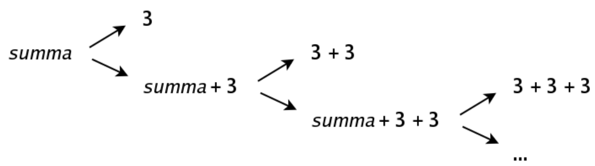


We have formed a tree, and the language, ie, which sequences of terminals are allowed, we find by looking at the leaves in the tree.

A grammar for an infinitely large language

- *sum* -> **3** | *sum* **+ 3**

If the *sum* is the start symbol, we can expand it as follows:



A grammar for sums

- **4 + 5**
- **4 + 5 + 6**
- **7**
- **2 + 3 + 4 + 5 + 6**

A first try:

- *sum* -> **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | *sum* **+ 0** | *sum* **+ 1** | *sum* **+ 2** | *sum* **+ 3** | *sum* **+ 4** | *sum* **+ 5** | *sum* **+ 6** | *sum* **+ 7** | *sum* **+ 8** | *sum* **+ 9**

The grammar works, but it is very long. We can make it shorter by introducing the non-terminal *number*:

- *number* -> **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
- *sum* -> *digit* | *sum* **+** *digit*

Make number into a terminal (responsibility of the scanner to recognize „1", „2", … and pass to parser) :

Then the grammar becomes even shorter:

- *sum* -> **digit** | *sum* **+ digit**

Lexical analysis and syntactic analysis

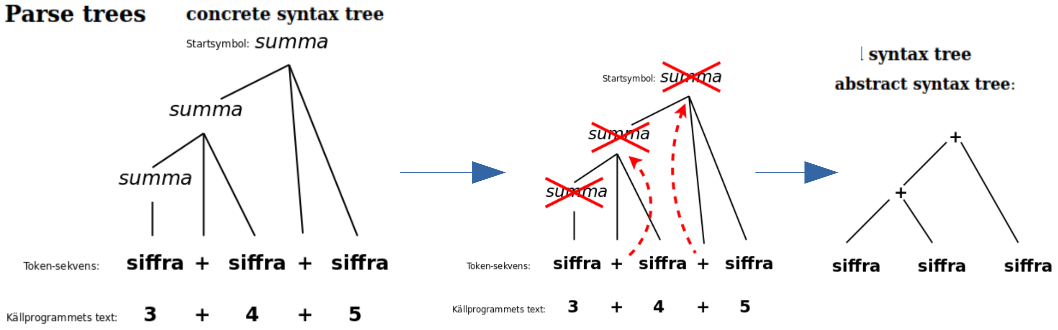
The scanner reads the source program text, and instead creates a sequence of terminals, or "tokens", which are passed on to the parser:

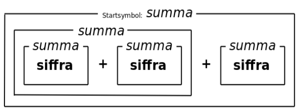
identifier (x), **equals sign**, **identifier** (lower_coordinate), **plus sign**, **integer** (14), **semicolon**

The parser reads this sequence of tokens, compares it to the grammar, and determines if there is an allowed sequence of tokens according to the grammar, and which of the productions in grammar are needed to generate this sequence.

Parse trees

concrete syntax tree





A grammar for real expressions

- **1 + 2**
- **2 * 3**
- **1 + 2 * 3**

An initial attempt to write the grammar:

- *E* -> **number**
- *E* -> *E* + *E*
- *E* -> *E* * *E*

More compact:

- *E* -> **number** | *E* + *E* | *E* * *u*

Ambiguous, so next attempt:

- *F* -> **number**
- *T* -> *F* * *F*
- *E* -> *T* + *T*

It's better, but it *only* fits expressions on the form **numbers * numbers + numbers * numbers**.

- *F* -> **number**
- *T* -> *F* | *T* * *F*
- *E* -> *T* | *E* + *T*

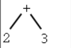
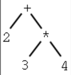
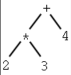
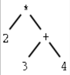
Now the grammar is correct, and it says the following:

- A *factor (F)* consists of a single number.
- A *term (T)* can either consist of a single *factor*, or you have a *term* (which may in turn be a single *factor* or several multiplied together), multiplying with a new *factor* at the end, so that it becomes a new *term*.
- An *expression (E)* can either consist of a lone *term*, or you have an *expression* (which may in turn be a lone *term* or several added together) where you add a new *term* at the end so that it becomes a new *expression*.

Yacc and Bison

```
%token number

%start expression

%%

factor : number ;
term : factor | term '*' factor ;
expression : term | expression '+' term ;

%%
```

Syntax and grammars

Infix notation	Tree	Value	Postfix notation	Prefix notation	Function notation	LISP
2 + 3		5	2 3 +	+ 2 3	plus(2, 3)	(plus 2 3)
2 + 3 * 4		14	2 3 4 * +	+ 2 * 3 4	plus(2, times(3, 4))	(plus 2 (times 3 4))
2 * 3 + 4		10	2 3 * 4 +	+ * 2 3 4	plus(times(2, 3), 4)	(plus (times 2 3) 4)
2 * (3 + 4)		14	2 3 4 + *	* 2 + 3 4	times(2, plus(3, 4))	(times 2 (plus 3 4))

Postfix: Stack machine. Easy to write an interpreter.

- Push numbers onto the top of the stack.
- **+**: Pop the two top numbers, add, and push the sum.

Syntax definition

```
if (a == b)
  printf("Same!\n");
else
  printf("Not same!\n");
```

if (*some expression*) *some statement* else *some other statement*

context-free grammar

statement -> **if** (*expression*) *statement* **else** *statement*
statement -> **if** (*expression*) *statement*
statement -> { *statement-list* } (forgot what?)

1. A set of **terminals** (Sw: **terminaler**) = terminal symbols = tokens
2. A set of **non-terminals** (Sw: **icke-terminaler**) = non-terminal symbols (compound grammatical constructs)
3. A set of **productions** (Sw: **produktioner**) = rules: non-terminal -> tokens/non-terminals. A production is **for** the non-terminal to the left.
4. What is the **start symbol** (Sw: **startsymbolen**)