Program translation.

- On the very earliest computers programs were written and entered in binary form.
- Some computers required the program to be entered one binary word at a time, using swithes on the front panel of the computer, one switch for each bit in the word.
- Because of that

– the size and the complexity of the programs were severe limited

– the debugging was very difficult task

– in general, development of the programs was very difficult and error prone.

Idea: use the computer itself to ease the programmer's work by **translation** from a more human-readable form of the program into executable binary code.

Assembly and High-level languages.

• First translator programs, known as the assemblers, were available at the beginning of 1950's.

• Translation from assembly languages to the binary form was fairly straightforward (on command-by-command basis)

• Later, the **high-level procedural languages** have appeared.

– FORTRAN (1957), Algol (1958), COBOL (1960)

– today there are hundreds of *third-generation* high-level programming languages (C, Pascal, BASIC, etc.)

Based on function-oriented programming concept, these languages attempted to free the programmer from the details of the machine's instructions.

• Translator for these languages became known as **compilers**

• A different type of high-level language translator, known as an **interpreter**, appeared for Lisp around 1962

Compilers vs Interpreters.

- High-level procedural languages may be compiled, or they may be interpreted

- A compiler operates on the entire program, translate it and generating a permanent binary module representing the program. This module can then be executed.

Translation and execution are separate processes.

- An interpreter translate source code and executes it, one source code program line at a time.

Translation and execution are interlaced processes.

Compilers vs Interpreters.

- Execution of interpreted code is much more slowly than compiled code.

- However, when a program is to be executed only once or twice, or when a program is frequently modified, it is easier and faster not to have to compile in order to run the program.

- The translation process for interpreters is simplified, since the code does not need to take into account different possible conditions; only actual conditions do matter.

- Some techniques are more easily to implement with interpreter than with a compiler, e.g self-modifying code.

Pre-processing.

- Pre-processing is a kind of translation from a high-level code to some other high-level code.

- Sometime it is used for the translation of **object-oriented (fourth-generation)** programming languages:

– pre-processing translates a source code (say, in C++) into the code in some lower-level programming language (e.g. in C)

– then the compilation of the result is performed

Components of programming languages.

Before we go into some details of compilation process, let us discuss relevant components of programming languages

- The lexical component. The lexicon is a list of all legal words (elementary expressions) in the language, together with information about the word.

E.g. in Java “import”, “public”, “else” are legal words. They go together with an information about its meanings and roles.

Components of programming languages (cont.)

- The syntactical component. Syntax define the form and structures of legal expressions of the language.

E.g. in Java

```
...
double private x,y;
...
```

is not a correct fragment of code, although lexically it is OK.

At the same time

```
...
private double x,y;
...
```

is a correct fragment.

Components of programming languages (cont.)

• The semantic component deals with the meaning of the expressions. For the programming languages semantic component defines a course of action to be performed, while executing a particular (fragment of a) program.

E.g. in Java, the fragment

```
...

if (m < n)
System.out.println("The minimum is " + m);

...
```

means “Check if `m < n`. If it is true than print the message, otherwise skip this instruction”

Programming language descriptions.

- The precise description of a programming language is needed for the translator:

– to analyse of the (source) program, and

– to generate correct and unambiguous code

- Key part of the precise description of any programming language is the **syntax rules** that make up its grammar. These rules are used in the parsing process.

- The parsing process consists of determining which rule applies in a particular instance as one traces through the sentence.

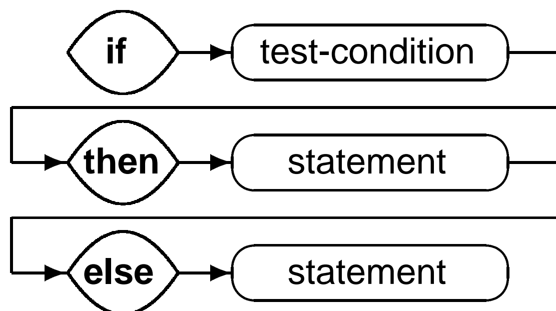
Example.

Pascal program fragment:

```
if testval > 5 then
   setval := 20
else
  for i := 1 to 20 do
  begin
   valuej := 2*i;
   writeln (i, valuej)
end


;
```

- First stage of parsing:

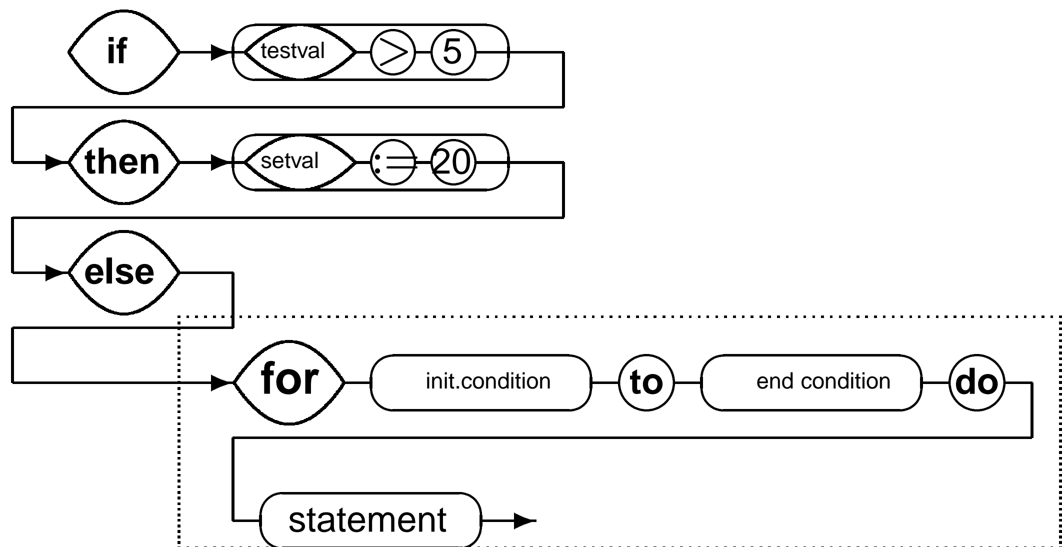


Example(cont.)

```
if testval > 5 then
   setval := 20
else
  for i := 1 to 20 do
  begin
   valuej := 2*i;
   writeln (i, valuej)
end

;
```

- Second stage of parsing:



- Third stage, etc

Informal narrative description.

One of the ways of describing the grammar of a high-level programming language is an **(informal) narrative description** of the language.

Example:

Pascal is written free form with no required layout. Tokens are separated by spaces, tabs, carriage returns, or comments.. . .

The basic Pascal structure is the block. A block consists of the following components:

– label declarations

– constant definitions

. . .

– procedure and functions declarations

–program statements

. . .

Informal narrative description.

- It is **easily understandable** by a human and it is useful as the reference source.
- But, it is **insufficiently precise**.

Grammars.

- Syntactic rules, as we have seen, define syntactic constructions in terms of other syntactic entities, some of which need to be defined themselves, some are already defined.
- The grammar is a set of such rules.
- Any grammar define correct sentences (programs) in terms of elementary parts – **symbols.** When one defines a grammar for a programming language **tokens** are considered as such elementary symbols.
- **Token** is the smallest group of character symbols that has a specific meaning within the language. Examples (e.g. in Java): ‘+’, ‘-’, ‘;’ ‘class’, ‘if’, etc

Terminal and Non-terminal symbols.

● In **general** the symbols in a grammar is divided into two parts: **terminal symbols** and **nonterminal symbols.**

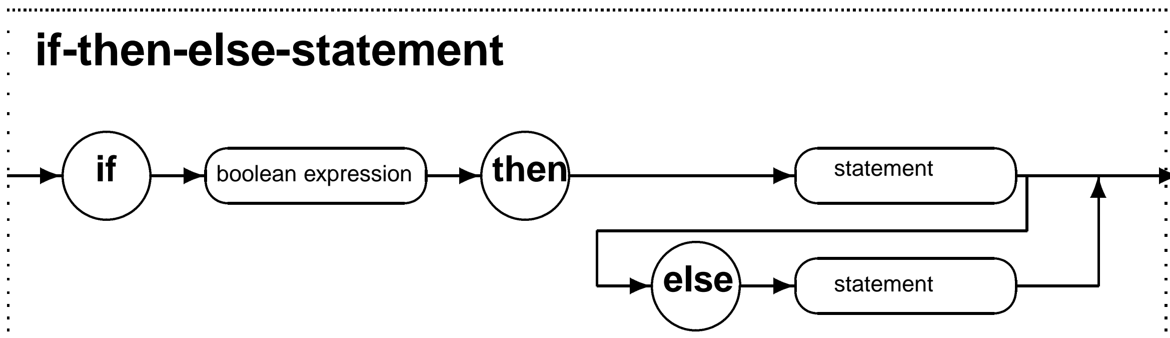
– terminal symbols represent elementary parts of the language, which don't need to be defined further (tokens).

– non-terminal symbols represent parts of the language which need further, more detailed definition (**if-then-else-statement** in the above example of the Pascal fragment)

Syntax diagrams.

One way of representing grammar is by **syntax diagrams**.

Example:



Here we have non-terminal symbols:

– **if-then-else-statement**

– **boolean expression**

– **statement**

And terminal symbols:

– **if**

– **then**

– **else**

Syntax diagrams (cont.)

- The grammar of the entire programming language can be presented by the set of syntax diagrams (productions).
- One diagram should be marked as **starting diagram (production)**

Backus-Naur form.

● Alternative and the most precise and suitable for automated processing, method for describing the grammar is known as **Backus-Naur form (BNF)**

● The rules of the language represented in BNF in a form of productions, i.e the rules of the form

“non-terminal —$>$ sequence of non-terminal and terminal symbols”

Example:

$<$ **if-then-else-statement** $>$ **—**$>$
if $<$ **boolean-expression** $>$ **then** $<$ **statement** $>$

$<$ **if-then-else-statement** $>$ **—**$>$
if $<$ **boolean-expression** $>$ **then** $<$**statement**$>$ **else** $<$ **statement** $>$

Bakus-Naur form (cont.)

Notation

–>	"is defined, or replaced by"
|	"OR", i.e A|B means select either A or B
<**name**>	nonterminal symbol
symbols	terminal symbols
[**symbols**]	optional symbols
{**symbols**}	the symbols are repeated 0 or more times

Examples (Pascal):

- <statement> —>
  [<label>:]<simple-statement> | <structured-statement>
- <identifier> –> <letter>{<letter> | <digit>}

Bakus-Naur (cont.)

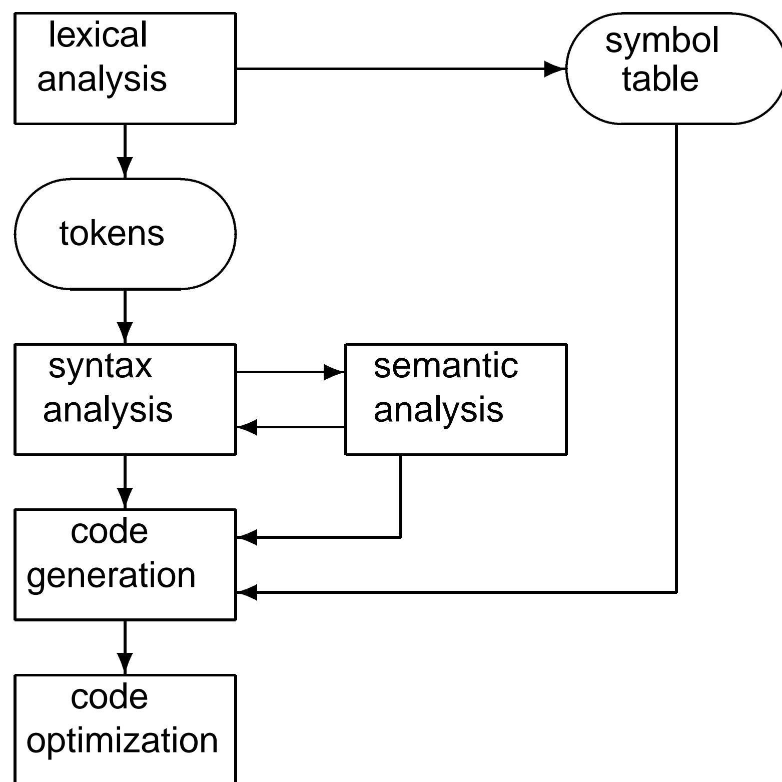
- The description of the grammar of a programming language can consist of tens or hundreds of productions (160 productions for Pascal)

- Given a grammar, a parsing for the program in the language defined by that grammar consists of determining which of the production applies in a particular place as one trace through the program. Or, how the program can be derived from **starting production** using the grammar.

- If the program, actually, can not be derived, then it is incorrect – there are syntactic mistakes.

The compilation process.

The compilation process is normally broken down into the four major steps.



Lexical analysis.

- **Lexical analysis (scanning):**

–text of the program is broken down into the smallest possible meaningful language components (tokens). These will consist e.g. number constants, identifiers (labels, variable names), etc.

–symbol table will be build of the various tokens used in the program, including the information on the type of every token (e.g. comparative token, variable token, etc.)

– Thus, the program text is reduced to the sequence of tokens.

Syntax analysis.

- Once lexical analysis is complete, the compiler moves to the syntax analysis stage to parse the result of lexical analysis into **individual productions.**

- One possible technique (we informally have discussed before) is **recursive descent**.

- Most difficult part of compilation process, especially because of detecting possible syntactical mistakes in programs.

- Semantic analysis is usually active at the same time. It is responsible in partucularly for the detecting data types inconsistencies.

Code generation.

- Memory locations and registers will be asisgned to the various data objects: variables, arrays, etc.
- The code will be generated for each of the productions identified during the syntax and semantic steps

Code optimization.

- The code is analysed to determine if there any ways to

–reduce the amount of code,

–to eliminate repeated operations,

–to reorganize parts of the program to execute faster and more efficiently

–etc

Example (Java) :

```
for (int i = 0; i < 5; i++)
{
 a = b + 1;
}
```

There is no need to execute `a = b + 1` five times !

The main steps carried out by a typical high-level language compiler.

1. Lexical analysis. Source text of program is converted into a sequence of lexemes (tokens), representing distinct elements of the program, e.g. number denotations, variable identifiers, key words. The information on the type of every lexem is stored in the symbol table.

2. Syntax analysis (parsing). Program is analysed to uncover the grammar from of the constructions used. These are examined so as to fit them into the syntactical rules of programs in the language. The outcome is to group and restructure the lexems in a way which identifies the kinds of programs constructs, such as statements, etc, being used.

3. Code generation and optimization. Finally, the program constructions identified in the syntax analysis are converted into machine-code sequences. Memory locations and registers are assigned to the various data objects: variables, arrays, etc. The code is analysed to determine if there any ways

– to reduce the amount of code,

– to eliminate repeated operations,

– to reorganize parts of the program

to execute faster and more efficiently.