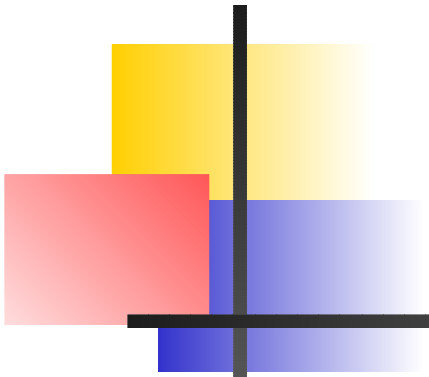
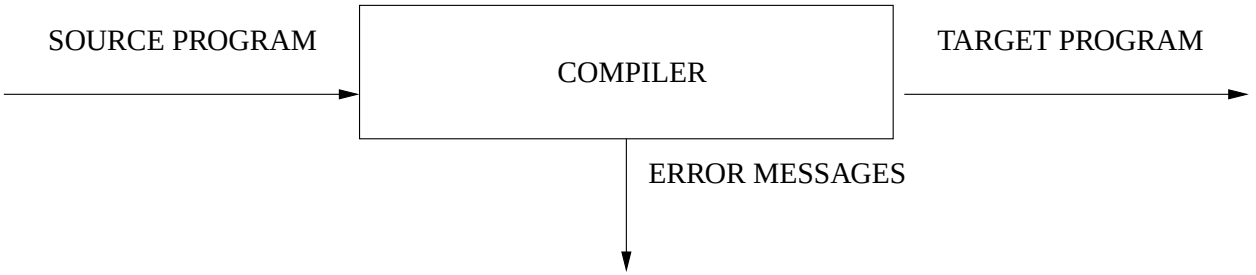


Compilers

SITE : http://www.info.univ-tours.fr/~mirian/

Introduction

Compiler: A program that reads a program written in one language (the *source* program) and translates it into an equivalent program in another language (the *target* program)
Important part of this translation process: the compiler reports to its user the presence of errors in the source program

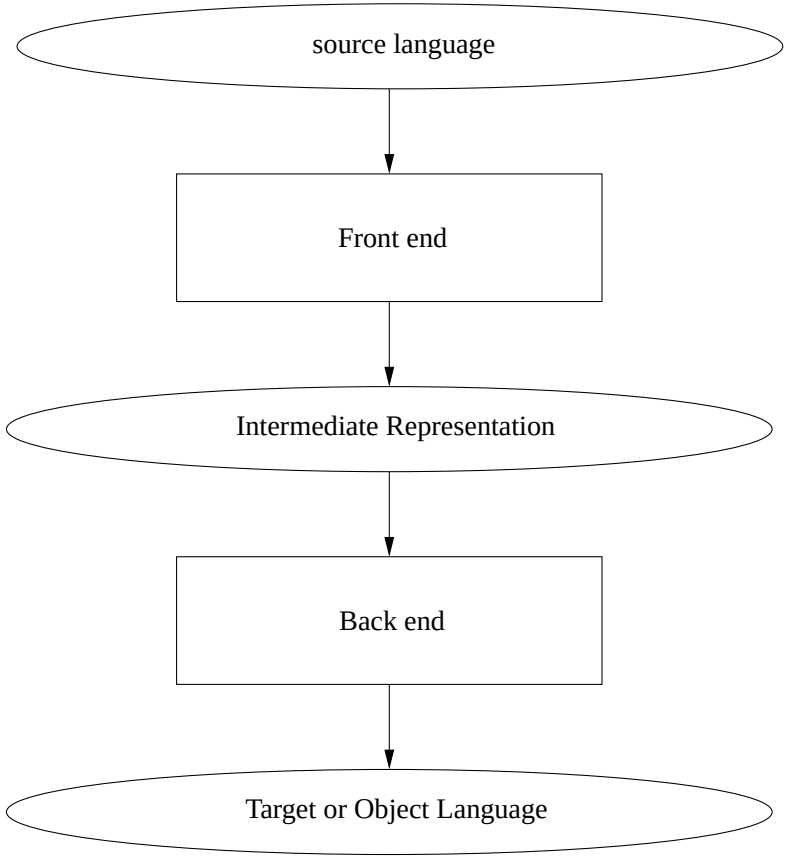


Interpreters: Instead of producing a target program as a translation, an interpreter performs the operations implied by the source program

So... we need to build too many compilers!?

- There are thousands of source languages (Fortran, C, Pascal or specialized languages) and target languages (another programming language or a machine language): Too many compilers?
- Despite an apparent complexity, the basic tasks that any compiler must perform are essentially the same
- At the highest level, a compiler has a *front end* and a *back end*
- It is desirable for the **front end** to deal with aspects of the input language, but to keep it as independent of the machine as possible
- The **back end** should concentrate on dealing with the specifics of output language, and try to remain independent of the input.
- Problem of generating a suite of compilers for n different languages to m different machines.

Highest level of abstraction of a compiler

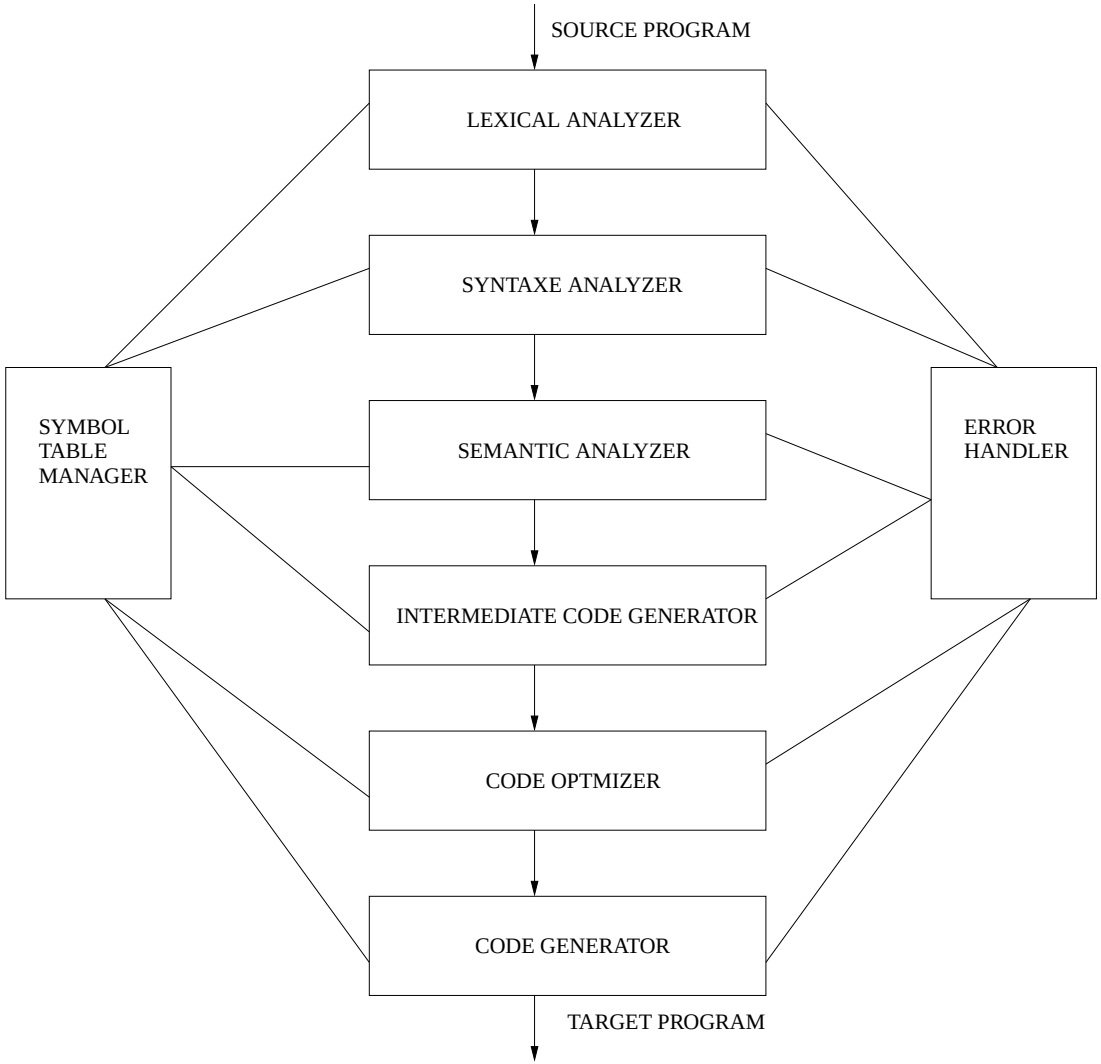


The phases of a compiler

- Conceptually, a compiler operates in *phases*, each of which transforms the source program from one representation to another
- In practice, some of the phases may be grouped together
- One can say that the phases are grouped into two parts:

1. **Analysis** - The typical compiler front end
   Breaks up the source program into constituent pieces and creates an intermediate representation of the source program
   (a) Lexical analysis
   (b) Syntax analysis
   (c) Semantic analysis
2. **Synthesis**
   Constructs the desired target program from the intermediate representation
   The back end corresponds to the phases of code generation and optimization

Phases of a Compiler



Lexical Analysis: Scanner

- Read a stream of characters and breaks it up into **tokens** (groups of characters having some relation among them)
- Each **token** represents **a sequence of characters treated as one entity**, *i.e.*, they are the smallest program units that are individually meaningful
- The blanks separating the characters of the tokens would normally be eliminated

Example: In Java and C

abc: identifiers

if and **return**: reserved words

42: integer literals

1.2E-3: floating point literals

$+$ **;** $<=$: Operators

$\{$ $($; : punctuation

"a string" : string literals

$/*$ **a comment**: comments

Example

In lexical analysis the characters in the assignment statement

position := initial + rate * 60

would be grouped into the following tokens:

Type of the token	**Value of the token**
identifier	**position**
the assignment symbol	:=
identifier	**initial**
the plus sign	+
identifier	**rate**
the multiplication sign	*
numbers	**60**

Note: The types in the first column are usually represented by codes

Lexical Analysis (cont.)

- The implementation of a lexical analyzer is based on **finite state automata**
  Example: The identifiers in Pascal are described by a **regular expression**

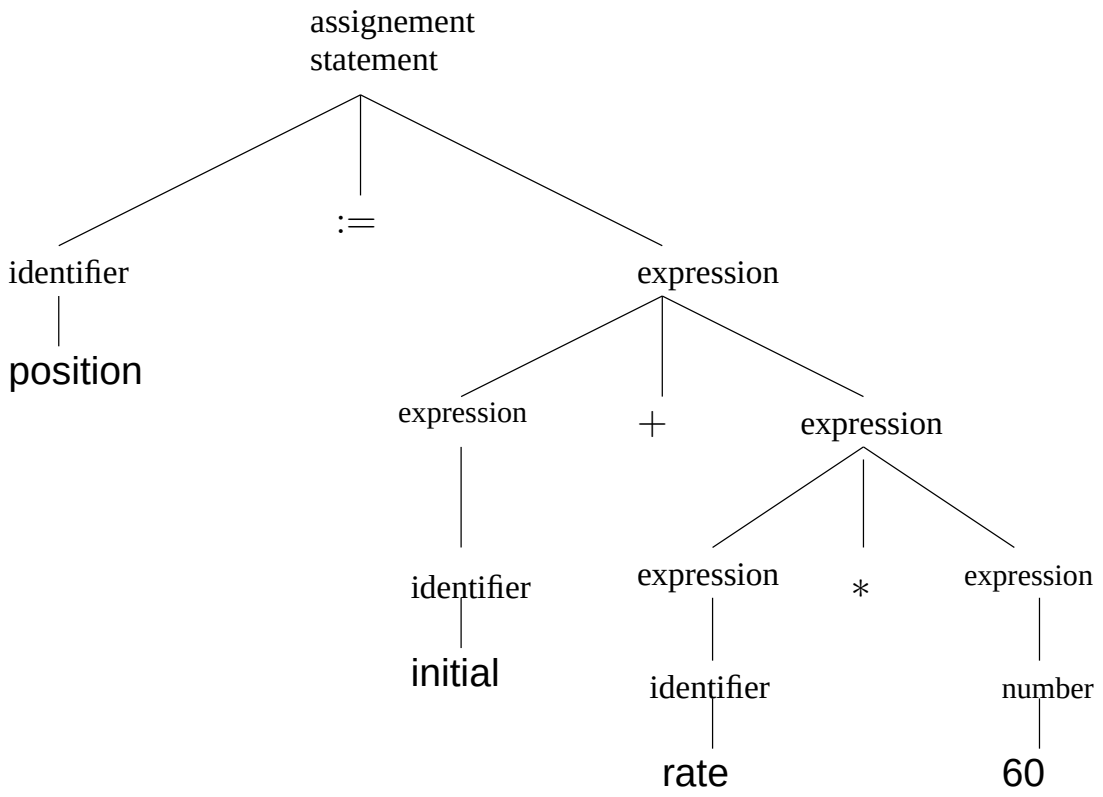
$$letter\ (letter \mid digit \mid underline)^*$$

- Lexical analysis can be complicated in some languages (ex, Fortran)

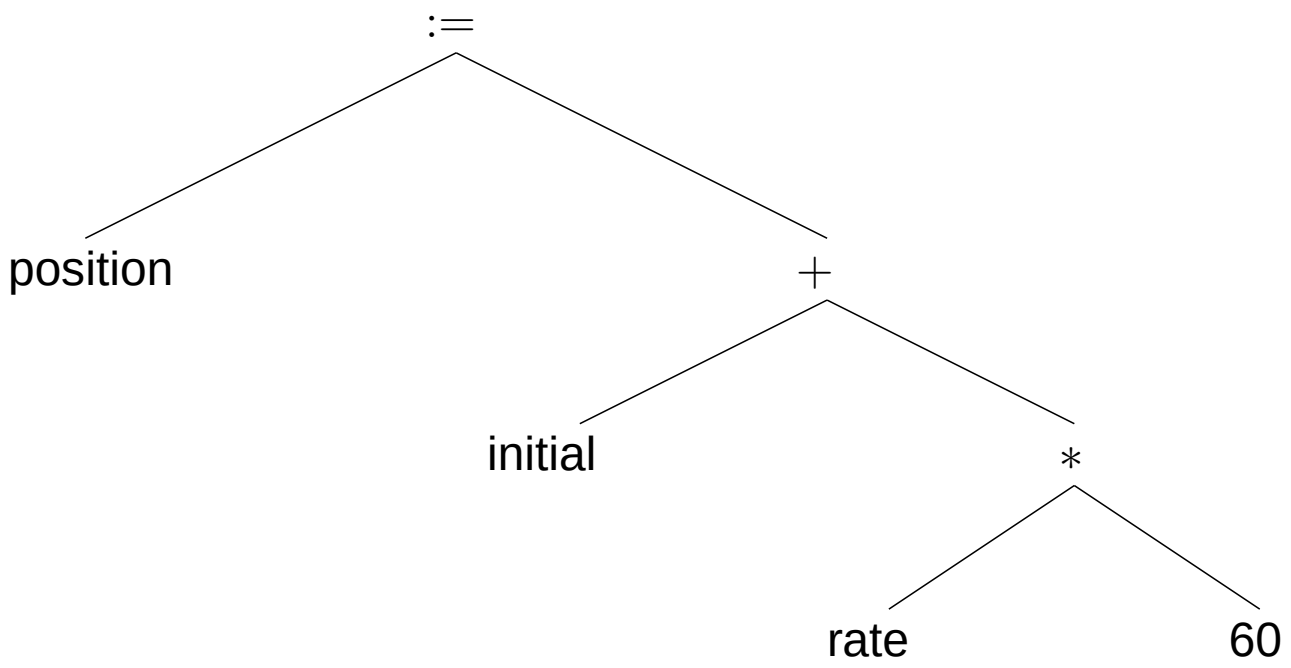
Syntax Analysis: Parser

- The parser **receives the tokens from the lexical analyzer and checks if they arrive in the correct order** (the one defined in the language)
- It involves grouping the tokens into grammatical phrases that are used by the compiler to synthesize output
- In general, **the syntax of a programming language is described by a *context free grammar***
- The job of the parser is to recover the hierarchical structure of the program from the stream of tokens received from the lexical analyzer
- The output of the parser depends on the implementation of the compiler: usually a tree
- **Parse tree**: describes the syntactic structure of the input
- **Syntax tree**: a compressed (and more common) representation of the parse tree in which the operators appear as the interior node and the operands of an operator are the children of the node

Parse tree


assignement
statement
identifier
:=
expression
position
expression
+
expression
identifier
expression
*
expression
initial
identifier
number
rate
60

Syntax tree



Semantic Analysis

- Checks the source program for semantic errors and gathers type information for the subsequent code generation phrase
- Uses the syntax analysis phase to identify the operators and operands of the expressions and statements
- In a context free grammar, it is not possible to represent a rule such as:
  All identifier should be declared before being used
  The verification of this rule concerns the semantic analysis.
- The basic idea is the use of a**symbol table** to store important information
- Important component of semantic analysis is **type checking**:
  The compiler checks that each operator has operands that are permitted by the source language specification
  Example: arithmetic operator applied to an integer and a real
- First phase that deals with the meanings of programming language constructs

Intermediate Code Generator

- Some compilers generate an explicit intermediate representation of the source program
- We can think of this intermediate representation as a program for an abstract machine
- This is the solution for avoiding the construction of $M \times N$ compilers (where M is the number of source language and N is the number of object language)
- Two important properties of the intermediate representation:
  - easy to produce
  - easy to translate into the target program
- Example: three-address code

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

Code Optimization

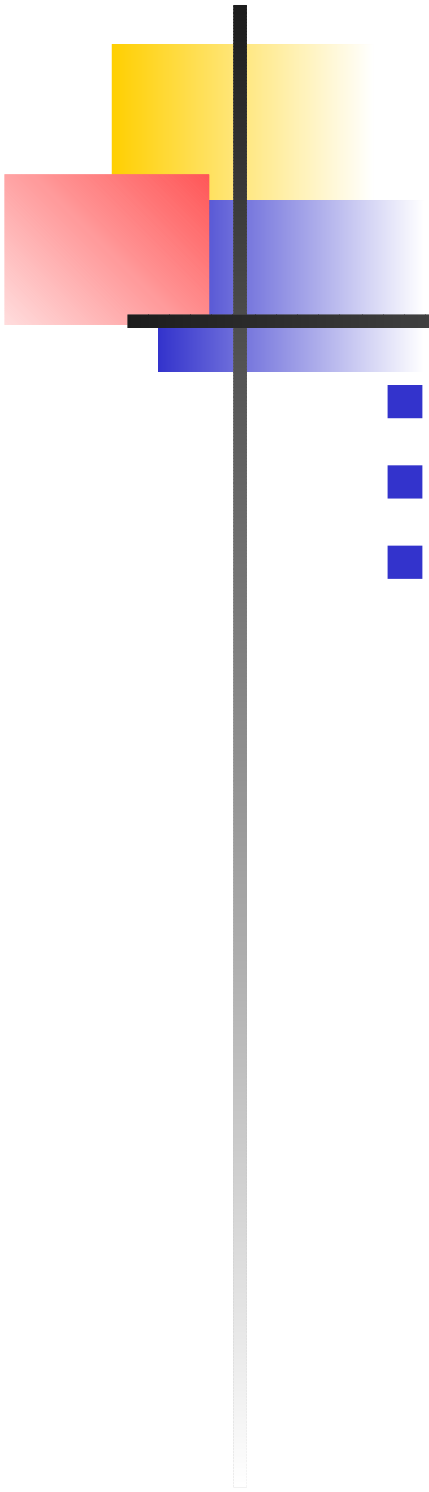
- Attempts to improve the intermediate code, so that faster-running machine code will result
- Some optimizations are trivial. Example: The intermediate code

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

can be performed by using two instructions:

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

- There is a great variation in the amount of code optimization different compilers perform. In those that do the most, called *optimizing compilers*, a significant fraction of the time of the compiler is spent on this phase
- However, there are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much



Code Generation

- Final phase of the compiler
- Generation of target code (in general, machine code or assembly code)
- Intermediate instructions are each translated into a sequence of machine instructions

Code Generation: example

Translation of the intermediate code

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

into some machine code:

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

where:

- the first and the second operands of each instruction specify a source and a destination, respectively
- The F in each instruction tells us that the instruction deals with floating-points numbers
- The $\#$ indicates that 60.0 is to be treated as a constant
- Registers 1 and 2 are used

Symbol Table Management

- A compiler needs to **record information concerning the objects found in the source program** (such as variables, labels, statements, type declarations, etc)
- A **Symbol Table** is a data structure containing a record for each identifier, with fields for the attributes of the identifiers
  Note: Attributes provide information about identifiers: storage allocation, type, scope, names, etc
- Symbol Table **allows the compiler to find a record for each identifier quickly and to store or retrieve data from that record quickly**
- When an identifier is detected by the lexical analyzer, it is entered in the symbol table. However, the attributes of an identifier cannot normally be determined during lexical analysis.

Error Detection and Reporting

- Each phase can encounter errors. However, after detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.
- One of the most important task of a compiler
- A difficult task due to the following 2 reasons:
  1. An error can hide another one
  2. An error can provoke a lot of other errors (all of them solved by correcting the first one)
- Two criteria can be followed:
  1. Stop when the first error is found (usually not very helpful, but can be used by an interactive compiler)
  2. Find all the errors

Phases and passes

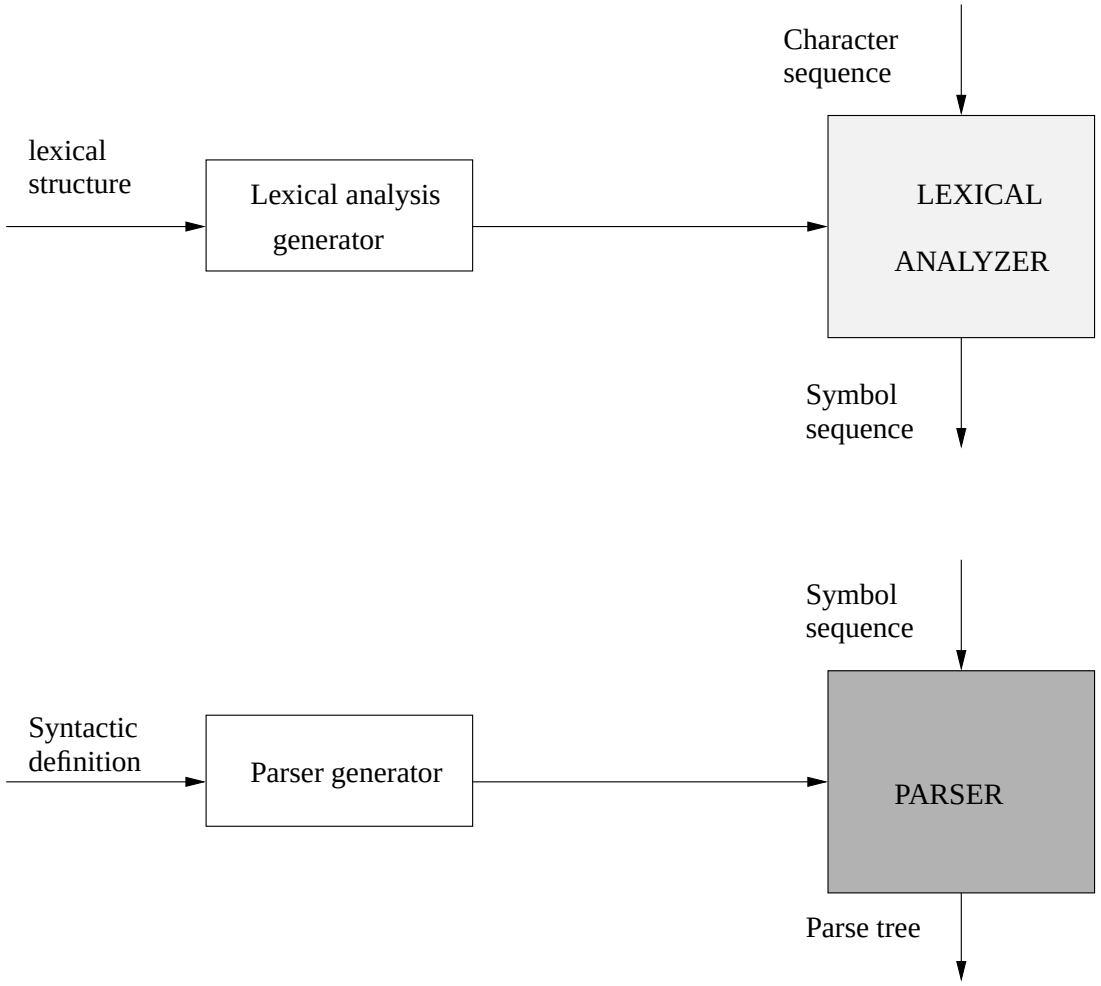
- In logical terms a compiler is thought of as consisting of **stages** and **phases**
- Physically it is made up of **passes**
- The compiler has one pass for each time the source code, or a representation of it, is read
- **Many compilers have just a single pass so that the complete compilation process is performed while the code is read once**
- The various phases described will therefore be executed in parallel
- Earlier compilers had a large number of passes, typically due to the limited memory space available
- Modern compilers are single pass since memory space is not usually a problem

Use of tools

The 2 main types of tools used in compiler production are:

1. a lexical analyzer generator
   - Takes as input the lexical structure of a language, which defines how its tokens are made up from characters
   - Produces as output a lexical analyzer (a program in C for example) for the language
   - Unix lexical analyzer Lex
2. a symbol analyzer generator
   - Takes as input the syntactical definition of a language
   - Produces as output a syntax analyzer (a program in C for example) for the language
   - The most widely know is the Unix-based YACC (Yet Another Compiler-Compiler), used in conjunction with Lex.
     Bison: public domain version

Lexical analyzer generator and parser generator



Applications of compiler techniques

- Compiler technology is useful for a more general class of applications
- Many programs share the basic properties of compilers: they read textual input, organize it into a hierarchical structure and then process the structure
- An understanding how programming language compilers are designed and organized can make it easier to implement these compiler like applications as well
- More importantly, tools designed for compiler writing such as lexical analyzer generators and parser generators can make it vastly easier to implement such applications
- Thus, compiler techniques - An important knowledge for computer science engineers
- Examples:
  - Document processing: Latex, HTML, XML
  - User interfaces: interactive applications, file systems, databases
  - Natural language treatment