PROGRAMMING LANGUAGES

Module 1

INTRODUCTION TO LANGUAGES

Introduction

What makes a language successful?

- easy to learn (BASIC, Pascal, LOGO, Scheme)
- easy to express things, easy use once fluent, "powerful" (C, Common Lisp, APL, Algol-68, Perl)
- easy to implement (BASIC, Forth)
- possible to compile to very good (fast/small) code (Fortran)
- backing of a powerful sponsor (COBOL, PL/1, Ada, Visual Basic)
- wide dissemination at minimal cost (Pascal, Turing, Java)

Introduction

Why do we have programming languages? What is a language for?

- way of thinking -- way of expressing algorithms
- languages from the user's point of view
- abstraction of virtual machine -- way of specifying what you want
- the hardware to do without getting down into the bits
- languages from the implementor's point of view

WHY STUDY PROGRAMMING LANGUAGES?

Help you choose a language.

- C vs. Modula-3 vs. C++ for systems programming
- Fortran vs. APL vs. Ada for numerical computations
- Ada vs. Modula-2 for embedded systems
- Common Lisp vs. Scheme vs. ML for symbolic data manipulation
- Java vs. C/CORBA for networked PC programs

WHY STUDY PROGRAMMING LANGUAGES?

Ada is a modern programming language designed for large, long-lived applications – and embedded systems in particular – where reliability and efficiency are essential. It was originally developed in the early 1980s (this version is generally known as Ada 83).

Fortran (formerly FORTRAN, derived from "Formula Translation") is a general-purpose, imperative programming language that is especially suited to numeric computation and scientific computing. Originally developed by IBM in the 1950s for scientific and engineering applications.

WHY STUDY PROGRAMMING LANGUAGES?

Modula-3 is a programming language conceived as a successor to an upgraded version of Modula-2 known as Modula-2+. While it has been influential in research circles (influencing the designs of languages such as Java, C#, and Python) it has not been adopted widely in industry.

Pascal is an imperative and procedural programming language, which Niklaus Wirth designed in 1968–69 and published in 1970, as a small, efficient language intended to encourage good programming practices using structured programming and data structuring.

WHY STUDY PROGRAMMING LANGUAGES?

Scheme is a functional programming language and one of the two main dialects of the programming language Lisp.

Cobra is a general-purpose, object-oriented programming language. It supports both static and dynamic typing.

ALGOL 68 (short for ALGOrithmic Language 1968) is an imperative computer programming language that was conceived as a successor to the ALGOL 60 programming language, designed with the goal of a much wider scope of application and more rigorously defined syntax and semantics.

WHY STUDY PROGRAMMING LANGUAGES?

Common Lisp (CL) is a dialect of the Lisp programming language, published in ANSI standard document.

ML is a general-purpose functional programming language developed by Robin Milner and others in the early 1970s at the University of Edinburgh. ML stands for ***MetaLanguage***.

WHY STUDY PROGRAMMING LANGUAGES?

Make it easier to learn new languages some languages are similar; easy to walk down family tree

- concepts have even more similarity; if you think in terms of iteration, recursion, abstraction (for example), you will find it easier to assimilate the syntax and semantic details of a new language than if you try to pick it up in a vacuum. Think of an analogy to human languages: good grasp of grammar makes it easier to pick up new languages.

WHY STUDY PROGRAMMING LANGUAGES?

Help you make better use of whatever language you use

- understand unintelligible features:
  - In C, help you understand unions, arrays & pointers, separate compilation, var args, catch and throw
  - In Common Lisp, help you understand first-class functions/closures, streams, catch and throw, symbol internals

WHY STUDY PROGRAMMING LANGUAGES?

Help to make better use of whatever language you use

- understand implementation costs: choose between alternative ways of doing things, based on knowledge of what will be done underneath:
- use simple arithmetic equal
- avoid call by value with large data items in Pascal
- avoid the use of call by name in Algol 60
- choose between computation and table lookup (e.g. for cardinality operator in C or C++)

WHY STUDY PROGRAMMING LANGUAGES?

Help you make better use of whatever language you use

- figure out how to do things in languages that don't support them clearly:
  - lack of suitable control structures in Fortran
  - use comments and programmer discipline for control structures
  - lack of recursion in Fortran

WHY STUDY PROGRAMMING LANGUAGES?

Help to make better use of whatever language you use

- figure out how to do things in languages that don't support them clearly:
    - lack of named constants and enumerations in Fortran
    - use variables that are initialized once, then never changed
    - lack of modules in C and Pascal use comments and programmer discipline

Definition

A **Compiler** is a program that converts high-level language to assembly language.
An **Assembler** is a program that converts the assembly language to machine-level language. An assembler translates assembly language programs into machine code. The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.
A **Preprocessor,** generally considered as a part of compiler, is a tool that produces input for compilers. It deals with macro-processing, augmentation, file inclusion, language extension, etc.

Definition

An interpreter, like a compiler, translates high-level language into low-level machine language.

The difference lies in the way they read the source code or input.

A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes.
In contrast, an interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence. If an error occurs, an interpreter stops execution and reports it; whereas a compiler reads the whole program even if it encounters several errors.

Definition

Linker is a computer program that links and merges various object files together in order to make an executable file.

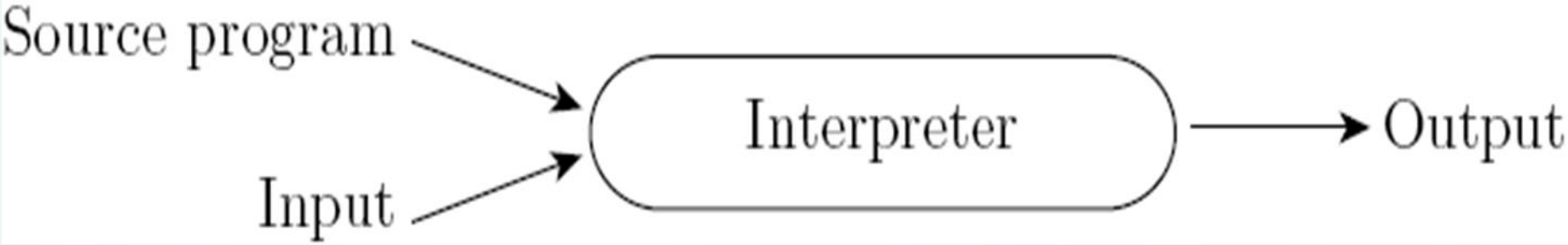
All these files might have been compiled by separate assemblers.

The major task of a linker is to search and locate referenced module/routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references.

COMPILATION vs. INTERPRETATION

Pure Interpretation

- Interpreter stays around for the execution of the program
- Interpreter is the point of control during execution



COMPILATION vs. INTERPRETATION

Interpretation:

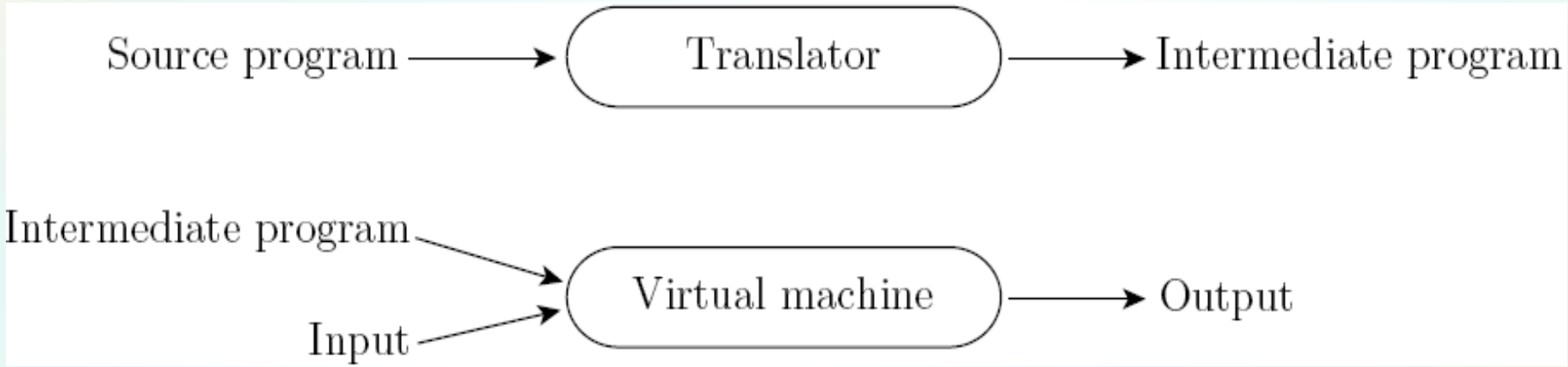
- Greater flexibility
- Better diagnostics (error messages)

Compilation

- Better performance

COMPILATION vs. INTERPRETATION

Most language implementations include a mixture of both compilation and interpretation



COMPILATION vs. INTERPRETATION

Note that compilation does NOT have to produce machine language for some sort of hardware

Compilation is ***translation*** from one language into another, with full analysis of the meaning of the input

Compilation entails semantic *understanding* of what is being processed;

A pre-processor will often let errors through. A compiler hides further steps; a pre-processor does not

COMPILATION vs. INTERPRETATION

Implementation strategies:

- Preprocessor
  - Looks for compiler directives (e.g., #include)
  - Removes comments and white space
  - Groups characters into *tokens* (keywords, identifiers, numbers, symbols)
  - Identifies higher-level syntactic structures (loops, subroutines)

COMPILATION vs. INTERPRETATION

Implementation strategies:

- Library of Routines and Linking
  - Compiler uses a *linker* program to merge the appropriate *library* of subroutines (e.g., math functions such as sin, cos, log, etc.) into the final program:

COMPILATION vs. INTERPRETATION

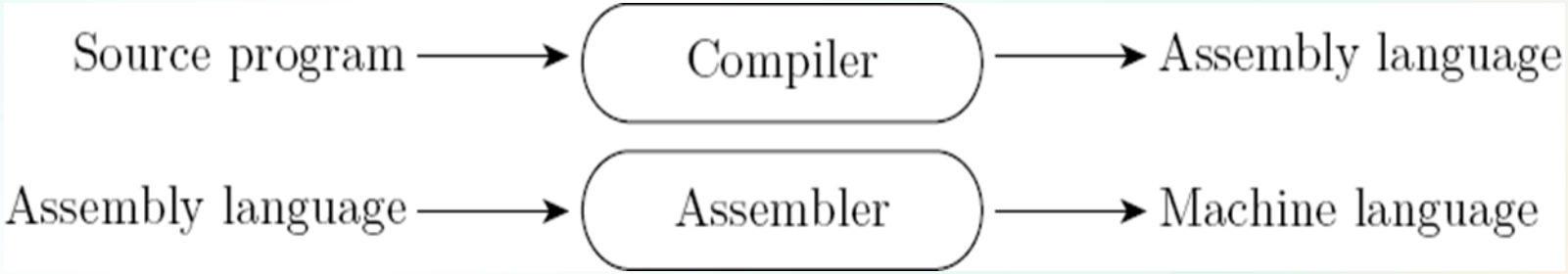
Compilation refers to the processing of source code files (.c, .cc, or .cpp) and the creation of an 'object' file.

This step doesn't create anything the user can actually run. Instead, the compiler merely produces the machine language instructions that correspond to the source code file that was compiled.
For instance, if you compile (but don't link) three separate files, you will have three object files created as output, each with the name <filename>.o or <filename>.obj (the extension will depend on your compiler).
Each of these files contains a translation of your source code file into a machine language file -- but you can't run them yet! You need to turn them into executables your operating system can use. That's where the linker comes in.

COMPILATION vs. INTERPRETATION

Implementation strategies:

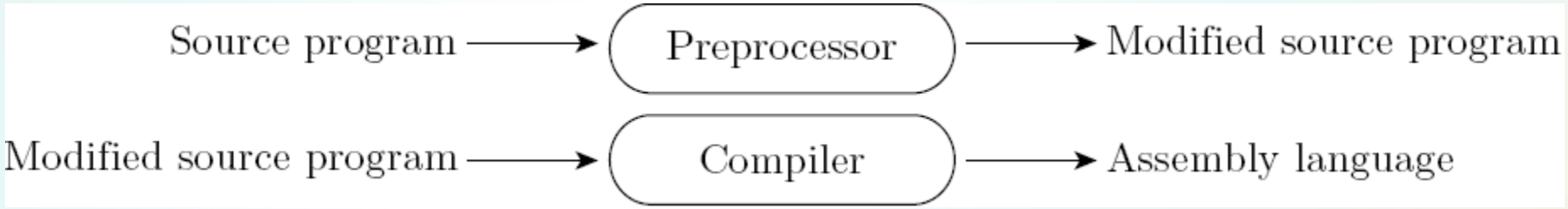
- Post-compilation Assembly
  - Facilitates debugging (assembly language easier for people to read)
  - Isolates the compiler from changes in the format of machine language files (only assembler must be changed, is shared by many compilers)



COMPILATION vs. INTERPRETATION

Implementation strategies:

- The C Preprocessor (conditional compilation)
  - Preprocessor deletes portions of code, which allows several versions of a program to be built from the same source



COMPILATION vs. INTERPRETATION

Implementation strategies:

- **DYNAMIC AND JUST-IN-TIME COMPILATION**
  - In some cases a programming system may deliberately delay compilation until the last possible moment.
    - The Java language definition defines a machine-independent intermediate form known as *byte code*. Byte code is the standard format for distribution of Java programs.
    - The main C# compiler produces .NET Common Intermediate Language (CIL), which is then translated into machine code immediately prior to execution.

COMPILATION vs. INTERPRETATION

Implementation strategies:

- Assembly-level instruction set is not implemented in hardware; it runs on an interpreter.
- Interpreter is written in low-level instructions (*microcode* or *firmware*), which are stored in read-only memory and executed by the hardware.

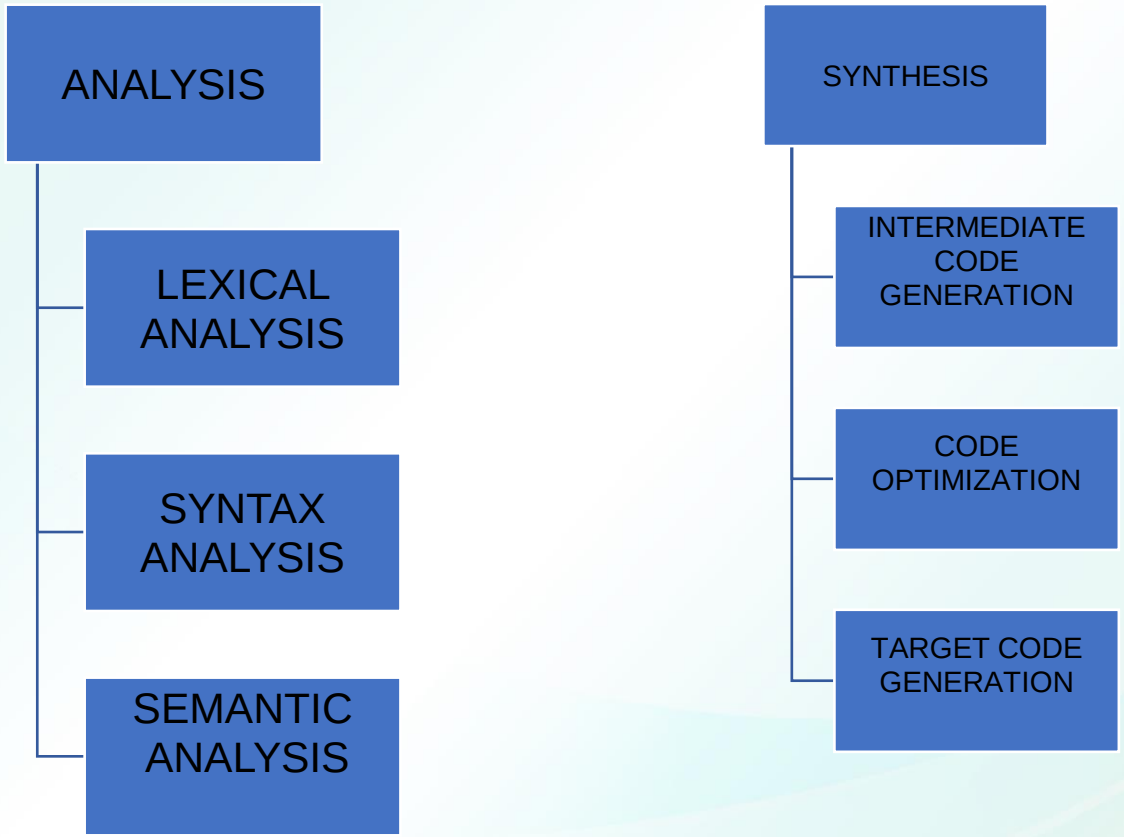
PROGRAMMING ENVIRONMENT TOOLS

TOOLS

Type	Unix examples
Editors	`vi`, `emacs`
Pretty printers	`cb`, `indent`
Pre-processors (esp. macros)	`cpp`, `m4`, `watfor`
Debuggers	`adb`, `sdb`, `dbx`, `gdb`
Style checkers	`lint`, `purify`
Module management	`make`
Version management	`sccs`, `rcs`
Assemblers	`as`
Link editors, loaders	`ld`, `ld-so`
Perusal tools	`More`, `less`, `od`, `nm`
Program cross-reference	`ctags`

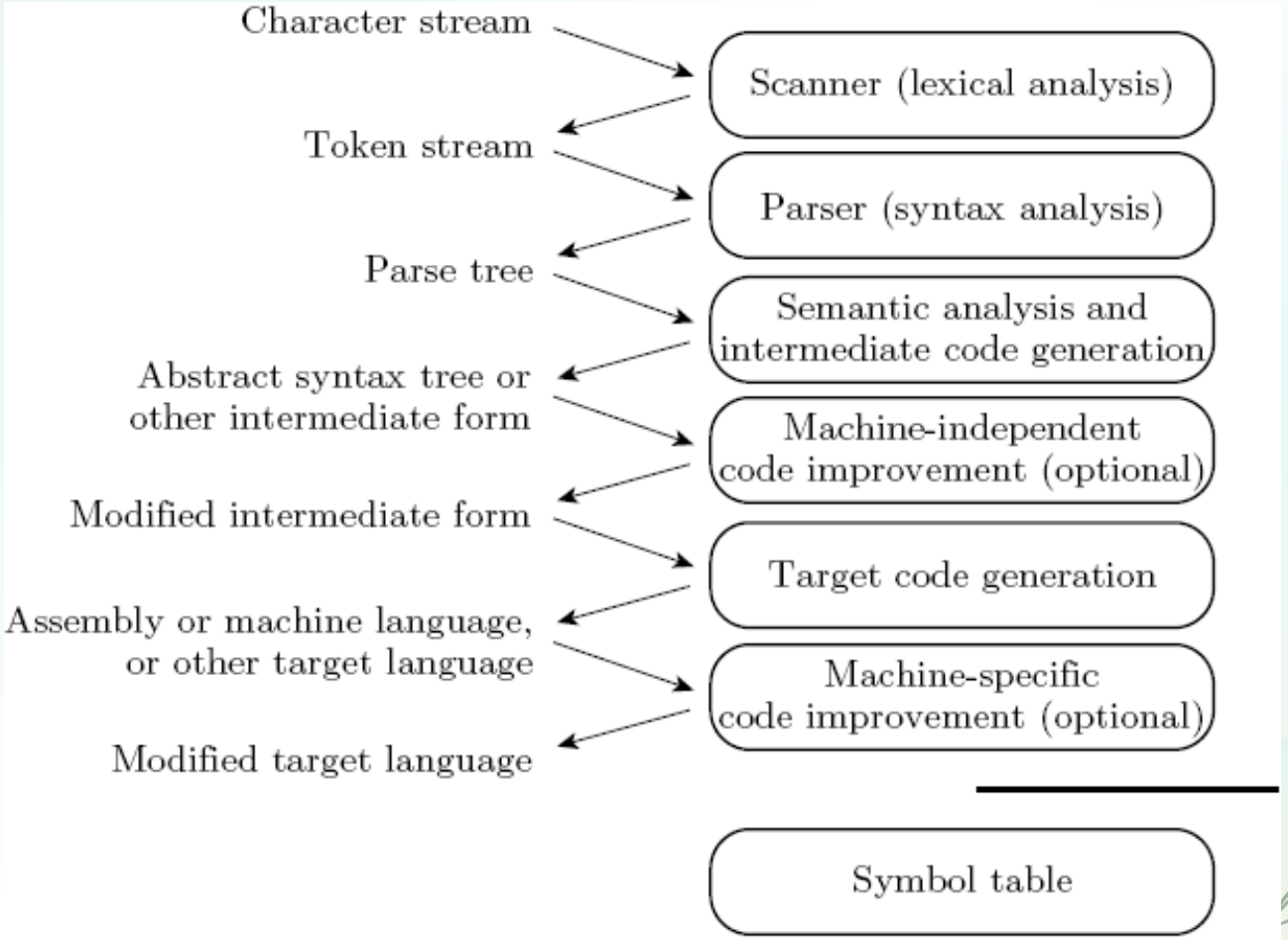
PHASE OF A COMPILER

Analysis of Language1
Synthesis of Language 2



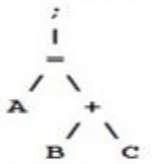
AN OVERVIEW OF COMPILATION

Overview:



AN OVERVIEW OF COMPILATION

*

Phase	Output	Sample
Programmer (source code producer)	Source string	`A=B+C;`
Scanner (performs *lexical analysis*)	Token string	`'A', '=', 'B', '+', 'C', ';'` And *symbol table* with names
Parser (performs *syntax analysis* based on the grammar of the programming language)	Parse tree or abstract syntax tree	
Semantic analyzer (type checking, etc)	Annotated parse tree or abstract syntax tree	

Intermediate code generator	Three-address code, quads, or RTL	`int2fp B t1`
		`+ t1 C t2`
		`:= t2 A`

Optimizer	Three-address code, quads, or RTL	`int2fp B t1` `+ t1 #2.3 A`

Code generator	Assembly code	`MOVF #2.3,r1`
		`ADDF2 r1,r2`
		`MOVF r2,A`

Peephole optimizer	Assembly code	`ADDF2 #2.3,r2` `MOVF r2,A`

AN OVERVIEW OF COMPILATION

***Scanning*:**

- divides the program into "tokens", which are the smallest meaningful units; this saves time, since character-by-character processing is slow
- we can tune the scanner better if its job is simple; it also saves complexity (lots of it) for later stages
- you can design a parser to take characters instead of tokens as input, but it isn't pretty
- scanning is recognition of a *regular language*, e.g., via DFA

AN OVERVIEW OF COMPILATION

***Parsing* is recognition of a *context-free language*, e.g., via PDA**

- Parsing discovers the "context free" structure of the program
- Informally, it finds the structure you can describe with syntax diagrams (the "circles and arrows" in a Pascal manual)

AN OVERVIEW OF COMPILATION

Lexical Analyzer or Linear Analyzer breaks the sentence into tokens. For Example following assignment statement :-

position = initial + rate * 60

Would be grouped into the following tokens:

1. The identifier **position**.
2. The assignment symbol **=**.
3. The identifier **initial**.
4. The plus sign.
5. The identifier **rate**.
6. The multiplication sign.
7. The number 60

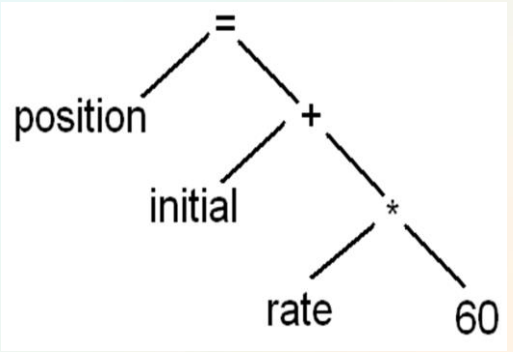
AN OVERVIEW OF COMPILATION

SYMBOL TABLE:

POSITION	Id1 & attributes
INITIAL	Id2 & attributes
RATE	Id3 & attributes

An expression of the form :

Position =Initial +60*Rate



gets converted to → id1 = id2 +60*id3

So the Lexical Analyzer symbols to an array of easy to use symbolic constants (TOKENS). Also, it removes spaces and other unnecessary things like comments etc.

AN OVERVIEW OF COMPILATION

***Semantic analysis* is the discovery of *meaning* in the program**

- The compiler actually does what is called STATIC semantic analysis. That's the meaning that can be figured out at compile time
- Some things (e.g., array subscript out of bounds) can't be figured out until run time. Things like that are part of the program's DYNAMIC semantics

AN OVERVIEW OF COMPILATION

Intermediate form **(IF) done after semantic analysis (*if* the program passes all checks)**

- IFs are often chosen for machine independence, ease of optimization, or compactness (these are somewhat contradictory)
- They often resemble machine code for some imaginary idealized machine; e.g. a stack machine, or a machine with arbitrarily many registers
- Many compilers actually move the code through more than one IF

AN OVERVIEW OF COMPILATION

Optimization **takes an intermediate-code program and produces another one that does the same thing faster, or in less space**

- The optimization phase is optional

Code generation phase **produces assembly language or (sometime) relocatable machine language**

AN OVERVIEW OF COMPILATION

Certain *machine-specific optimizations* (use of special instructions or addressing modes, etc.) may be performed during or after *target code generation*
***Symbol table*: all phases rely on a symbol table that keeps track of all the identifiers in the program and what the compiler knows about them**

- This symbol table may be retained (in some form) for use by a debugger, even after compilation has completed

AN OVERVIEW OF COMPILATION

Lexical and Syntax Analysis

- GCD Program (Pascal)

```
program gcd(input, output);
var i, j : integer;
begin
    read(i, j);
    while i <> j do
        if i > j then i := i - j
        else j := j - i;
    writeln(i)
end.
```

AN OVERVIEW OF COMPILATION

Lexical and Syntax Analysis

- GCD Program Tokens
  - Scanning (*lexical analysis*) and parsing recognize the structure of the program, groups characters into *tokens*, the smallest meaningful units of the program

```
program   gcd     (       input   ,       output  )       ;
var       i       ,       j       :       integer ;       begin
read      (       i       ,       j       )       ;       while
i         <>      j       do      if      i       >       j
then      i       :=      i       -       j       else    j
:=        j       -       i       ;       writeln (       i
)         end     .
```

AN OVERVIEW OF COMPILATION

Lexical and Syntax Analysis

- Context-Free Grammar and Parsing
  - Parsing organizes tokens into a *parse tree* that represents higher-level constructs in terms of their constituents
  - Potentially recursive rules known as *context-free grammar* define the ways in which these constituents combine

References

McGrath, Mike (2017). *C++ Programming.* In Easy Steps Limited.
Perkins, Benjamin (2016). *Beginning Visual C# 2015 programming.*
WroxSteve, Tale (2016). *C++.*
Chopra, R. (2015). *Principles of Programming Languages*. New Delhi: I.K. International Publishing.
Kumar, Sachin (2015).*Principles of programming Languages*. S. K. Kataria& Sons.

https://www.computerhope.com/jargon/p/programming-language.htm
https://www.tutorialspoint.com/compiler_design/compiler_design_syntax_analysis.htm
https://www.cs.iusb.edu/~dvrajito/teach/c311/c311_3_scope.html
https://www.tutorialspoint.com/compiler_design/compiler_design_semantic_analysis.htm
https://cs.lmu.edu/~ray/notes/controlflow/
https://teachcomputerscience.com/programming-data-types/
https://algs4.cs.princeton.edu/12oop/