Lecture 08: Language Translation

Concepts

- Source Program -- program written in a high-level programming language. (a.k.a. source code)
- Object Program -- the source program after it has been translated into machine language. (a.k.a. object code)

A variety of tools exist to support software development/construction:

- Translator : interpreter, compiler, assembler
- Linker
- Loader

Language translation tools

Language translators convert source **code** into language that the computer processor understands. Source code has various structures and **commands**, but computer processors only understand **machine language**.

Different types of translations must occur to turn source code into machine language, which is made up of bits of binary **data**. The three major types of language translators are compilers, assemblers, and interpreters.

A. Compilers

A compiler is a special program that takes written source code (high-level-language) and translates it into an executable machine-language program.
When a compiler executes, it analyzes all of the language statements in the source code and builds the machine language **object** code.

Once the translation is done, the machine-language program can be run any number of times. However, it can only be run on one type of computer since each type of computer has its own machine language. Therefore if the program is to run on another type of computer it has to be re-translated, using a different compiler, into the appropriate machine language.
It is a slow mode of translation, but fast in execution.
Examples of compilers: COBOL, C, C++, Pascal

History bit: The first compiler was written by Grace Hopper, in 1952, for the A-0 programming language.

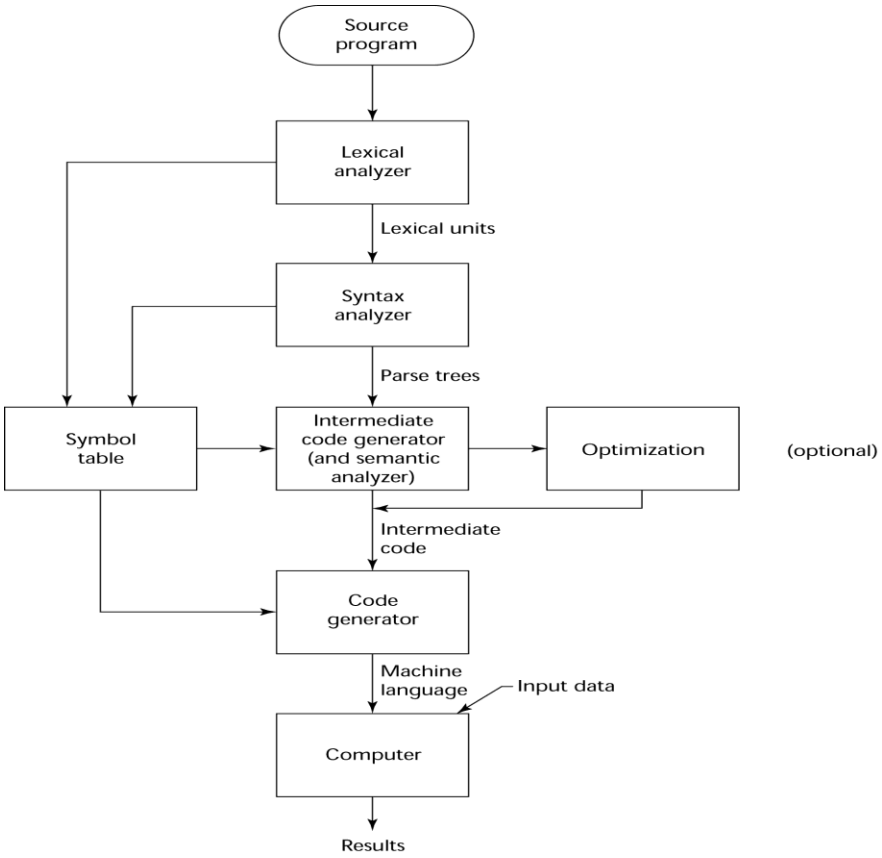
Compiler characteristics:

- spends a lot of time analyzing and processing the program
- the resulting executable is some form of machine- specific binary code
- the computer hardware executes the resulting code
- program execution is fast

The compilation process

The compilation process has several phases which in summary include:

- Lexical analysis – involves converting characters in the source program into lexical units/tokens
- Syntax analysis - transforming lexical units into parse trees which represent the syntactic structure of program
- Semantics analysis – generation of intermediate code
- Code generation - machine code is generated



1. Lexical Analysis

A.K.A Scanning, it involves the scanner breaking down the high-level language program into its smallest meaningful symbols (tokens, atoms). A symbol table is then started with a list of all tokens (symbols) found.

Tokens are the smallest meaningful units of the language and these are faster to manipulate than characters e.g.:

- "Reserved words": do, if, float, while
- Special characters: (,{ , + ,-, =, !, /,
- Names & numbers myValue , 3.07e02

Token Type	Classification Number
symbol	1
number	2
=	3
+	4
−	5
;	6
==	7
if	8
else	9
(	10
)	11

Figure 1: Symbol Table Classification

2. Syntax Analysis

A.K.A Parsing, this phase entails the compiler determining whether the tokens recognized by the scanner are syntactically legal statements. This is performed by a parser.

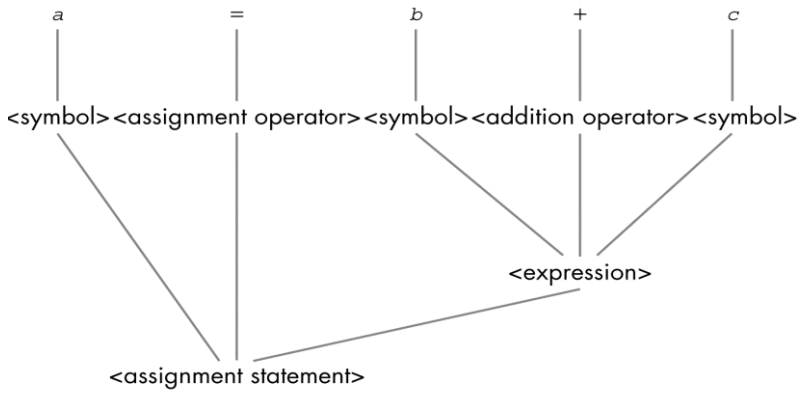
The *syntax* of the language is usually defined via a formal context-free (CF) grammar. This language grammar is defined by a set of rules that identify legal (meaningful) combinations of symbols. The parser applies these rules repeatedly to the program until leaves of parse tree are "atoms" i.e. each application of a rule results in a node in the parse tree. However, in cases when there is no pattern match, a syntax error arises.

In summary, the output of a parser is either a *parse tree* or *error message if the tree can't be constructed*

Example

High-level language statement: a = b + c

The resulting parse tree for the above HLL statement is as shown below:



3. Semantic Analysis

This is simply discovery of meaning in a program using the symbol table.

The compiler does this by making a first pass over the parse tree to determine whether all branches of the tree are semantically valid. If they are valid, the compiler can generate machine language instructions, if not, there is a semantic error and machine language instructions are not generated

Some of the tasks in this phase include:

- checking that
  - all identifiers are unique
  - identifiers are used according to their kind
  - Making sure identifiers are declared before use
  - Type checking for assignments and operators
  - Checking types and number of parameters to subroutines
  - Making sure functions contain return statements
  - Making sure there are no repeats among switch statement labels
- keeping track of types of identifiers
  - type defines structure and ways of correct use
  - type tells how to generate code for a particular use of an identifier

The symbol table is therefore an important structure assisting semantic analysis. It maps each identifier to all information known about it (type, structure, scope).

4. Intermediate Code generation

This phase involves going through the parse tree from bottom up, turning rules into code. E.g. a sum expression results in the code that computes the sum and saves the result.
The result is inefficient code in a machine-independent language

5. Machine independent optimization

This is a phase where various transformations that improve the code are performed:

- Find and reuse common sub-expressions
- Take calculations out of loops if possible
- Eliminate redundant operations

6. Target code generation

Here intermediate code is converted into machine instructions on intended target machine and the storage addresses for entries in symbol table are determined.

7. Machine-dependent optimization

Make improvements that require specific knowledge of machine architecture, e.g.

- Optimize use of available registers
- Reorder instructions to avoid waits

B. Interpreters

An interpreter translates a high-level language program instruction-by-instruction into machine executable code i.e. it analyzes and executes each line of source code in order. Instead of requiring a step before program execution, an interpreter processes the program as it is being executed. It is therefore slower in execution.

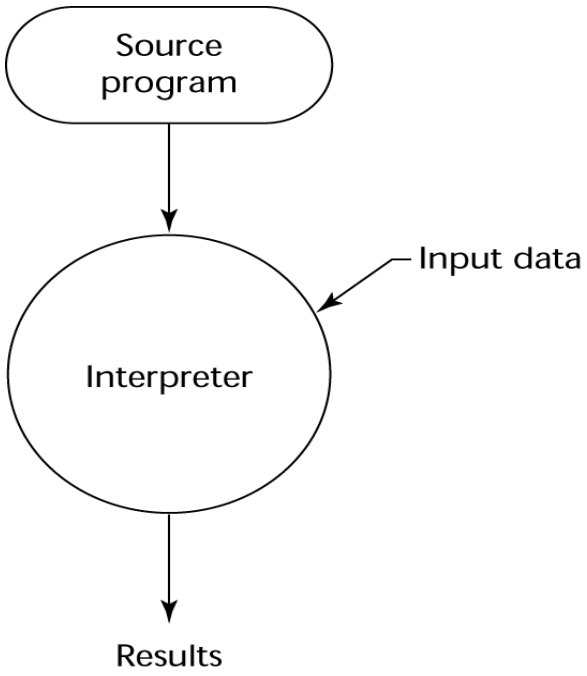
One use of interpreters is to execute HLL programs and another is that they can let you use a machine-language program meant for one type of computer on a completely different type of computer, thus performing a last moment translation service.

Interpreter characteristics:

- relatively little time is spent analyzing and processing the program
- the resulting code is some sort of intermediate code
- the resulting code is interpreted by another program
- program execution is relatively slow

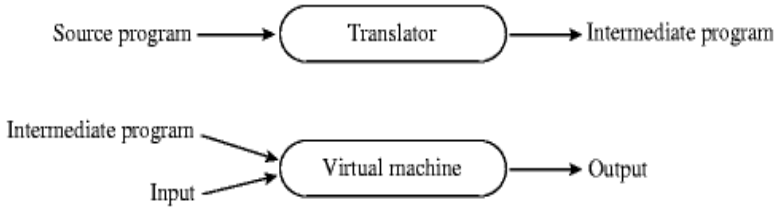
Examples of interpreters: BASIC, APL, Perl, Python, Smalltalk

The Interpretation Process



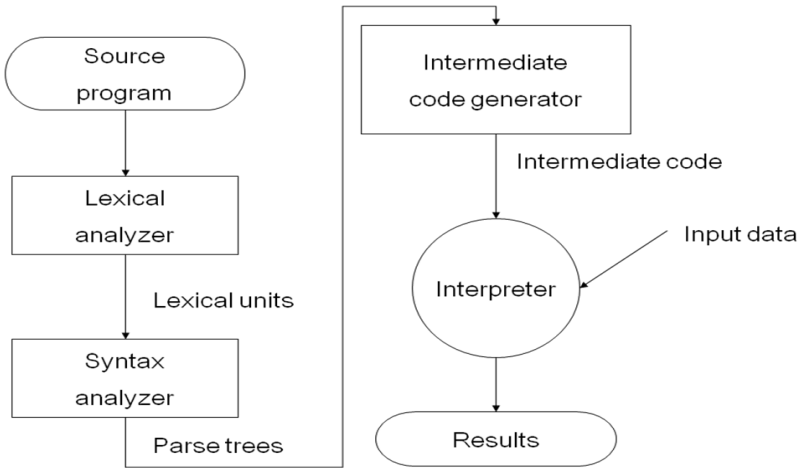
C. Hybrid Implementation Systems

These systems compromise between compilers and pure interpreters. A high-level language program is translated to an intermediate language that allows easy interpretation. It is faster than pure interpretation.



Examples of:

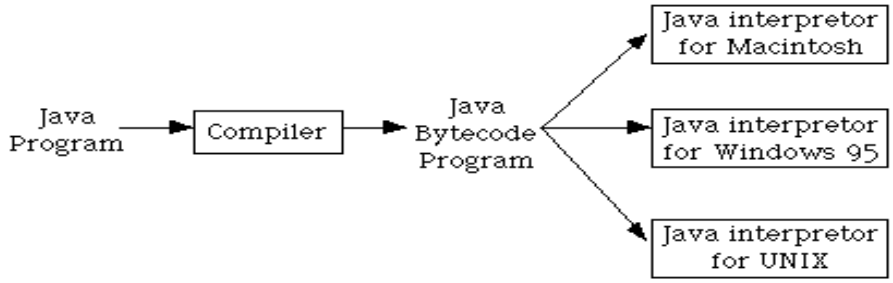
- Perl programs are partially compiled to detect errors before interpretation.
- All initial implementations of Java were hybrid; the intermediate form, byte code, provides portability to any machine that has a byte code interpreter and a run-time system (together, these are called Java Virtual Machine).



Case study: The Java Virtual Machine

The designers of Java chose to use a combination of compilation and interpretation. Programs written in Java are compiled into machine language, but it is a machine language for a computer that doesn't really exist. This so-called "virtual" computer is known as the 'Java Virtual Machine'. The machine language for the Java virtual machine is called 'Java Bytecode'. One of the main selling points of Java is that it can actually be used on any computer. All that the computer needs is an interpreter for Java bytecode. Such an interpreter simulates the Java virtual machine.

This is one of the essential features of Java: the same compiled program can be run on many different types of computer.

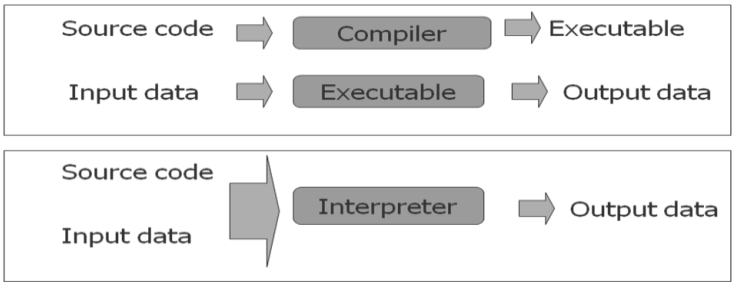


Why, you might wonder, use the intermediate Java bytecode at all? Why not just distribute the original Java program and let each person compile it into the machine language of whatever computer they want to run it on? There are many reasons, but the main one is a security concern. Many Java programs are meant to be downloaded over a network. This leads to obvious security concerns: you don't want to download and run a program that will damage your computer or your files. The bytecode interpreter acts as a buffer between you and the program you download.

It should be noted that there is no necessary connection between Java and Java bytecode. A program written in Java could certainly be compiled into the machine language of a real computer. And programs written in other languages could be compiled into Java bytecode. However, it is the combination of Java and Java bytecode that is platform-independent, secure, and network-compatible while allowing you to program in a modern high-level object-oriented language.

Compiler vs Interpreter

- Compilers translate a source (human-writable) program to an executable (machine-readable) program while Interpreters convert a source program and execute it at the same time.
- Interpretation is ‘simple’ and Compilation is ‘complicated’:
  - compilation involves *understanding* of the whole source program.
- Not all computing languages are compiled. Some are interpreted. In interpreted languages, the source code is always present and when it is run it is interpreted line-by-line and then it is executed. Most scripting languages are of this sort – eg. JavaScript and VBScript



- In a production environment where throughput is more critical, a compiled language is preferred.

D. Assembler

An assembler translates **assembly language** into machine language. Assembly language is one step removed from machine language. It uses computer-specific commands and structure similar to machine language, but instead of numbers, it uses names.

An assembler is similar to a compiler, but it is specific to translating programs written in assembly language into machine language. To do this, the assembler takes basic computer instructions from assembly language and converts them into a pattern of bits for the computer processor to use to perform its operations.

Other Programming Tools

Editors

Editors are used for the creation and modification of programs, and possibly the associated documentation. They can be general-purpose text or document editors, or they can be specialized for a target language - Monolingual (e.g., Java editor) or multilingual

Linkers

Combine object-code fragments into a larger program can be monolingual or multilingual. In a broader sense, they are tools for linking specification modules, able to perform checking and binding across various specification modules.

Loader

This tool loads machine language program into memory.

Debuggers

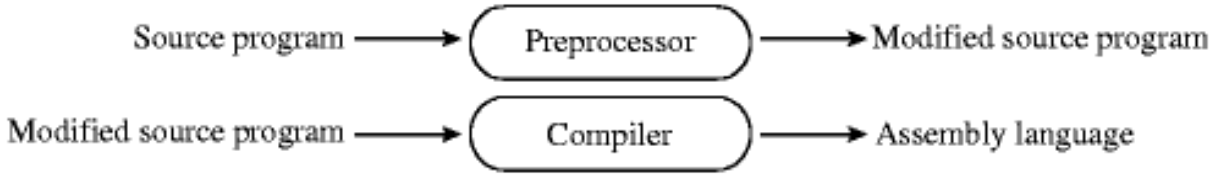
These are special kinds of interpreters where

- execution state inspectable
- execution mode definable
- there is animation to support program understanding

Preprocessor

Preprocessor macros (instructions) are commonly used to specify that code from another file is to be included. A preprocessor processes a program immediately before the program is compiled to expand embedded preprocessor macros.

For example: The C preprocessor expands the macros; #include, #define, e.tc.



Machine-dependent vs machine-independent aspects of translation

The *object code* is *machine-dependent* meaning that a *compiled* program can only be executed on a machine for which it has been compiled, whereas an *interpreted* program is not *machine-dependent* because the *machine-dependent* part is in the interpreter itself.

Hardware compilation

Hardware compilation involves taking user's specifications to automatically generate IC's.

Some compiler output targets hardware at a very low level e.g. Field Programmable Gate Arrays. Such compilers are said to be *hardware compilers* or synthesis tools because the source code they compile effectively controls the final configuration of the hardware and how it operates; the output of the compilation are not instructions that are executed in sequence - only an interconnection of transistors or lookup tables.

Hardware Description Language (HDL)

This is a computer language that facilitates the documentation, design, and manufacturing of digital systems, particularly very large-scale integrated circuits, and combines program verification techniques with expert system design methodologies.

For HDLs, 'compiler' refers to synthesis, a process of transforming the HDL code listing into a physically realizable gate netlist. The netlist output can take any of many forms e.g. a "simulation" netlist with gate-delay information.

On the other hand, a software compiler converts the source-code listing into a microprocessor-specific object-code, for execution on the target microprocessor. As HDLs and programming languages borrow concepts and features from each other, the boundary between them is becoming less distinct.

The Silicon compiler

This is software that translates the electronic design of a chip into the layout of the logic gates, including the actual masking from one transistor to another. The source of the compilation is either a high-level description or the netlist.

Silicon compilation takes place in three major steps:

- Convert a hardware-description language such as Verilog or VHDL or FpgaC into logic (typically in the form of a "netlist").
- Place equivalent logic gates on the IC. Silicon compilers typically use standard-cell libraries so that they do not have to worry about the actual integrated-circuit layout and can focus on the placement.
- Routing the standard cells together to form the desired logic.

The role of a formal semantics of a language

A formal semantics for a programming language is a mathematically precise description of the intended *meaning* of each construct in the language.

This is in contrast to formal *syntax* for a language, which tells us which sequences of symbols are correctly formed programs, a formal *semantics* tells us what programs will actually do when we run them. It could be a mathematical description of semantics in terms of algebra, attribute grammar e.t.c.

Formal semantics are thus grounded in mathematics and as a result, they can be more precise and less ambiguous than semantics given in natural language.

Programming languages whose semantics are described formally can reap many benefits. For example:

- Formal semantics offer a complete, rigorous definition of a language. This can serve both as a reference for programmers wishing to understand subtle points of the language, and as a touchstone for implementers of the language (e.g. compiler writers), by **providing a standard against which the correctness of an implementation may be judged.**
- Formal semantics also provide a foundation for mathematical proofs about programs. In order to be able to formulate claims about programs as precise mathematical statements, one has to have a precise mathematical definition of the language in question – formal semantics.
- The ideas of semantics provide guidance for language designers. The attempt to provide a formal semantics can highlight unnecessary complications and suggest simpler, cleaner definitions.
- Formal semantics can establish unambiguous and uniform standards for implementations of a language.

Further Reading

Sommerville, I., *Software Engineering*, 8th ed., Pearson, 2006.
Pressman, Roger S., *Software Engineering-A Practitioner's Approach*, 5th ed., McGraw-Hill, 2001