Chapter 1 :: Introduction

Programming Language Pragmatics

Michael L. Scott

Introduction

- early computers (1940s) cost millions of dollars and were programmed in machine language
  - machine’s time more valuable than programmer’s
  - machine language: bit sequences to perform low-level tasks; close to hardware
  - tedious

Introduction

- example: Euclid’s algorithm for GCD

```
55 89 e5 53  83 ec 04 83  e4 f0 e8 31  00 00 00 89  c3 e8 2a 00
00 00 39 c3  74 10 8d b6  00 00 00 00  39 c3 7e 13  29 c3 39 c3
75 f6 89 1c  24 e8 6e 00  00 00 8b 5d  fc c9 c3 29  d8 eb eb 90
```


Introduction

- less error-prone method needed
  - assembly language: binary operations expressed with mnemonic abbreviations

```
    pushl   %ebp                    jle     D
    movl    %esp, %ebp              subl    %eax, %ebx
    pushl   %ebx                B:  cmpl    %eax, %ebx
    subl    $4, %esp                jne     A
    andl    $-16, %esp          C:  movl    %ebx, (%esp)
    call    getint                  call    putint
    movl    %eax, %ebx              movl    -4(%ebp), %ebx
    call    getint                  leave
    cmpl    %eax, %ebx              ret
    je      C                   D:  subl    %ebx, %eax
A:  cmpl    %eax, %ebx              jmp     B
```


Introduction

- assembly language is specific to a certain machine, however
  - tedious to re-write code for each computer type
  - machine-independent language desired
  - Fortran (mid-1950s) used a compiler to bridge the gap between high-level language and machine-dependent code
  - many other languages followed

Introduction

- Why are there so many programming languages?
  - we've learned better ways of doing things over time
  - socio-economic factors: proprietary interests, commercial advantage
  - orientation toward special purposes
  - orientation toward special hardware
  - diverse ideas about what is pleasant to use

Introduction

- What makes a language successful?
  - easy to learn (BASIC, Pascal, LOGO, Scheme, Python)
  - easy to express things, easy to 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, C#)
  - 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?

- studying programming languages can help you choose the right language for an application
  - 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?

- makes 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 (at least Indo-European)

Why study programming languages?

- helps you make better use of whatever language you use
  - understanding obscure features
    - in C, helps you understand unions, arrays, pointers, separate compilation
    - in Common Lisp, helps you understand first-class functions/closures, streams, catch and throw, symbol internals

Why study programming languages?

- helps you make better use of whatever language you use (2)
  - understanding implementation costs: choosing between alternative ways of doing things, based on knowledge of what will be done underneath
    - using simple arithmetic equal (use x*x instead of x**2)
    - using C pointers or Pascal "with" statement to factor address calculations
    - avoiding call by value with large data items in Pascal
    - avoiding the use of call by name in Algol 60
    - choosing between computation and table lookup (e.g. for cardinality operator in C or C++)

Why study programming languages?

- helps you make better use of whatever language you use (3)
  - figuring out how to do things in languages that don't support them explicitly
    - lack of suitable control structures in Fortran
      - use comments and programmer discipline for control structures
    - lack of recursion in Fortran, CSP, etc.
      - write a recursive algorithm then use mechanical recursion elimination (even for things that aren't quite tail recursive)

Why study programming languages?

- helps you make better use of whatever language you use (4)
  - figuring out how to do things in languages that don't support them explicitly
    - 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
    - lack of iterators in just about everything
      - fake them with (member?) functions

Imperative languages

- can group languages according to statement type
  - imperative
    - von Neumann (Fortran, Pascal, Basic, C)
    - object-oriented (Smalltalk, Eiffel, C++?)
    - scripting languages (Perl, Python, JavaScript, PHP)
  - declarative
    - functional (Scheme, ML, pure Lisp, FP)
    - logic, constraint-based (Prolog, VisiCalc, RPG)

Imperative languages

- imperative languages, particularly the von Neumann languages, predominate
  - they will occupy the bulk of our attention
- we also plan to spend a lot of time on functional and logic languages

Programming Language History

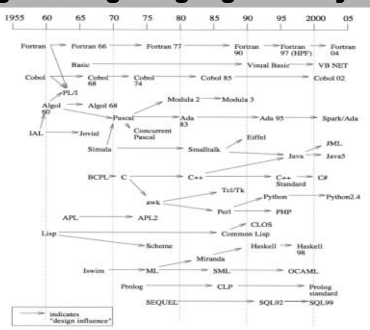


Figure 1.2: A Snapshot of Programming Language History

Source: Tucker & Noonan (2007)

Programming Language Properties

- programming languages have four properties:
  - syntax
  - naming
  - types
  - semantics

Source: Tucker & Noonan (2007)

Programming Language Properties

- syntax
  - precise description of all grammatically correct programs of that language
  - answers questions such as
    - what are the basic statements for the language?
    - how do I write …?
    - why is this a syntax error?

Source: Tucker & Noonan (2007)

Programming Language Properties

- naming
  - many entities in a program have names
    - variables, types
    - functions, parameters
    - classes, objects
  - named entities in a running program bound by
    - scope
    - visibility
    - type
    - lifetime

Source: Tucker & Noonan (2007)

Programming Language Properties

- types
  - collection of values and collection of operations on those values
  - simple types: numbers, characters, booleans, …
  - structured types: strings, lists, trees, hash tables
  - complex types: functions, classes, …
  - a language's type system helps to determine legal operations and to detect type errors

Source: Tucker & Noonan (2007)

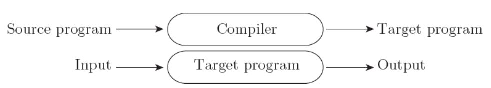
Programming Language Properties

- semantics
  - the meaning of a program
  - provides answers to questions
    - what does each statement mean?
    - what underlying model governs run-time behavior, such as function calls?
    - how are objects allocated to memory at run-time?
    - how do interpreters work in relation to semantics?

Source: Tucker & Noonan (2007)

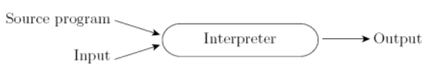
Compilation vs. Interpretation

- compilation vs. interpretation
  - not opposites
  - not a clear-cut distinction
- pure compilation
  - compiler translates a high-level source program into an equivalent target program (typically in machine language), and then goes away



Compilation vs. Interpretation

- pure interpretation
  - interpreter stays around for the execution of the program
  - interpreter is the locus of control during execution

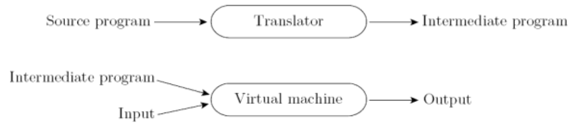


Compilation vs. Interpretation

- interpretation
  - greater flexibility
  - better diagnostics (error messages)
- compilation
  - better performance

Compilation vs. Interpretation

- common case is compilation or simple pre-processing, followed by interpretation
- some 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; pre-processing does not
- a pre-processor will often let errors through; a compiler hides further steps, while a pre-processor does not

Compilation vs. Interpretation

- many compiled languages have interpreted pieces, e.g., print formats in Fortran or C
- most use "virtual instructions"
  - set operations in Pascal
  - string manipulation in Basic
- some compilers produce nothing but virtual instructions, e.g., Pascal P-code, Java byte code, Microsoft COM+

Compilation vs. Interpretation

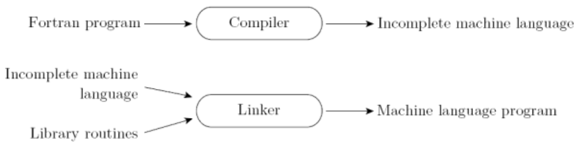
- many compilers are self-hosting
  - they are written in the language they compile
  - e.g., C compiler written in C
- how?
  - bootstrapping
  - write small interpreter
  - hand-translate small number of statements into assembly
  - extend through incremental runs of the compiler through itself

Compilation vs. Interpretation

- implementation strategies
  - preprocessor
    - removes comments and white space
    - groups characters into *tokens* (keywords, identifiers, numbers, symbols)
    - expands abbreviations in the style of a macro assembler
    - identifies higher-level syntactic structures (loops, subroutines)

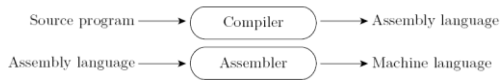
Compilation vs. Interpretation

- 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:



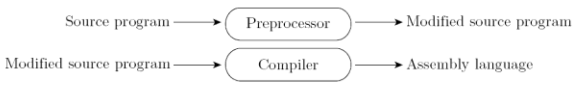
Compilation vs. Interpretation

- 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)



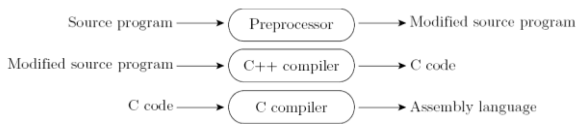
Compilation vs. Interpretation

- 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



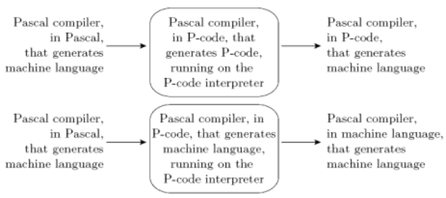
Compilation vs. Interpretation

- implementation strategies
  - source-to-source translation (C++)
    - C++ implementations based on the early AT&T compiler generated an intermediate program in C, instead of an assembly language



Compilation vs. Interpretation

- implementation strategies
  - bootstrapping



Compilation vs. Interpretation

- implementation strategies
  - compilation of interpreted languages
    - the compiler generates code that makes assumptions about decisions that won't be finalized until runtime
    - if these assumptions are valid, the code runs very fast; if not, a dynamic check will revert to the interpreter

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
      - Lisp or Prolog invoke the compiler on the fly, to translate newly created source into machine language, or to optimize the code for a particular input set
      - 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
  - microcode
    - 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.

Compilation vs. Interpretation

- compilers exist for some interpreted languages, but they aren't pure
  - selective compilation of compilable pieces and extra-sophisticated pre-processing of remaining source
  - interpretation of parts of code, at least, is still necessary for reasons above
- unconventional compilers
  - text formatters (LaTex)
  - silicon compilers
  - query language processors

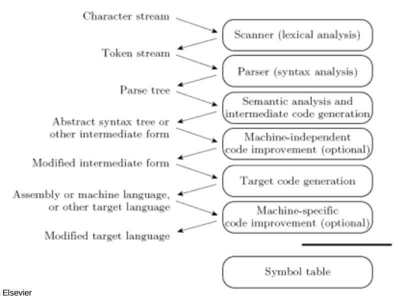
Programming Environment Tools

- 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	Id, Id-so
Perusal tools	More, less, od, nm
Program cross-reference	ctags

An Overview of Compilation

- Phases of Compilation



An Overview of Compilation

- scanner
  - 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

- parser
  - 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

- semantic analysis
  - the discovery of *meaning* in the program
  - the compiler actually does what is called STATIC semantic analysis, which is 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, which are part of the program's DYNAMIC semantics

An Overview of Compilation

- machine-independent code
  - intermediate form (IF) created after semantic analysis (*if* the program passes all checks)
  - IF's 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 term is a misnomer; we just *improve* code
  - the optimization phase is optional
- code generation phase
  - produces assembly language or (sometimes) 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
  - 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 (CFG) and parsing
    - parsing organizes tokens into a *parse tree* that represents higher-level constructs in terms of their constituents
    - potentially recursive rules known as a *context-free grammar* define the ways in which these constituents combine

An Overview of Compilation

- context-free grammar and parsing
  - example (Pascal program)

program ⟶ `PROGRAM id ( id` *more_ids* `) ;` *block* `.`

where

block ⟶ *labels constants types variables subroutines* `BEGIN` *stmt more_stmts* `END`

and

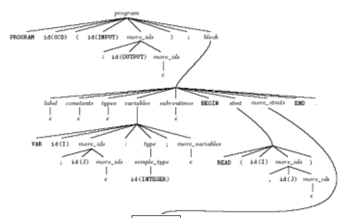
more_ids ⟶ `, id` *more_ids*

or

more_ids ⟶ ϵ

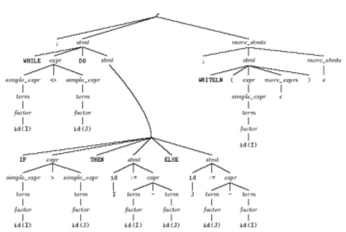
An Overview of Compilation

- context-free grammar and parsing
  - GCD program parse tree



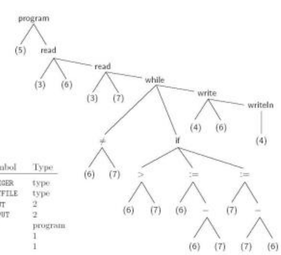
An Overview of Compilation

- context-free grammar and parsing
  - GCD program parse tree (continued)



An Overview of Compilation

- syntax tree
  - GCD program parse tree



Index	Symbol	Type
1	INTEGER	type
2	TEXTFILE	type
3	INPUT	2
4	OUTPUT	2
5	GCD	program
6	I	1
7	J	1

Figure 1.4: **Syntax tree and symbol table for the GCD program.**