Chapter 1 - Introduction
Introduction

• Why are there so many programming languages?
  ▫ Evolution -- 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).
  - 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 implementer'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.

• Make it easier to learn new languages because some languages are similar.
  ▫ Concepts have even more similarity such as iteration, recursion, abstraction.
  ▫ Easier to assimilate the syntax and semantic details of a new language than if you try to pick it up in a vacuum.
Why Study Programming Languages?

- Easy to walk down family tree.
  - 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?

- Help you make better use of whatever language you use.
  - Understand obscure features:
    - In C, help you understand unions, arrays & pointers, separate compilation, \texttt{varargs, catch and throw}.
    - In Common Lisp, help you understand first-class functions/closures, streams, catch and throw, symbol internals.
  - Understand implementation costs: choose between alternative ways of doing things, based on knowledge of what will be done underneath:
    - Use simple arithmetic equal (use $x^2$ instead of $x^2$).
    - Use C pointers or Pascal "with" statement to factor address calculations.
    - 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?

- Figure 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).

- Figure 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.
Language Groups

• Group languages as
  ▫ 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.
Compilation vs. Interpretation

In this lecture, we will discuss how to translate source code into machine executable code.
Compilation vs. Interpretation

• Compilation vs. interpretation
  ▫ Not opposites.
  ▫ Not a clear-cut distinction.

• Pure Compilation
  ▫ The compiler translates the high-level source program into an equivalent target program (typically in machine language).
    • The “target program” is called the object code.
    • You translate once and run many times.
Compilation

- Compilation is the conceptual process of translating source code into a CPU-executable binary target code
- Compiler runs on the same platform $X$ as the target code
Cross Compilation

- Compiler runs on platform $X$, target code runs on platform $Y$
Compilation vs. Interpretation

- Pure Interpretation
  - Interpreter stays around for the execution of the program.
  - Interpreter is the locus of control during execution.
  - You translate for each run.
Interpretation

- Interpretation is the conceptual process of running high-level code by an interpreter
Compilation vs. Interpretation

• Interpretation:
  ▫ Greater flexibility.
  ▫ Better diagnostics (error messages), easier to debug.

• Compilation
  ▫ Better performance.
Compilation vs. Interpretation

- Compilers “try to be as smart as possible” to fix decisions that can be taken at compile time to avoid to generate code that makes this decision at run time
  - Type checking at compile time vs. runtime
  - Static allocation
  - Static linking
  - Code optimization

- Compilation leads to better performance in general
  - Allocation of variables without variable lookup at run time
  - Aggressive code optimization to exploit hardware features
Compilation vs. Interpretation

• Benefit of interpretation?
  ▫ Interpretation facilitates interactive debugging and testing
    • Interpretation leads to better diagnostics of a programming problem
    • Procedures can be invoked from command line by a user
    • Variable values can be inspected and modified by a user
  ▫ Some programming languages cannot be purely compiled into machine code alone
    • Some languages allow programs to rewrite/add code to the code base dynamically
    • Some languages allow programs to translate data to code for execution (interpretation)
Compilation vs. Interpretation

• The compiler versus interpreter implementation is often fuzzy
  ▫ One can view an interpreter as a virtual machine that executes high-level code
  ▫ Java is compiled to bytecode
  ▫ Java bytecode is interpreted by the Java virtual machine (JVM) or translated to machine code by a just-in-time compiler (JIT)
  ▫ A processor (CPU) can be viewed as an implementation in hardware of a virtual machine (e.g. bytecode can be executed in hardware)
Compilation vs. Interpretation

- Common case is compilation or simple pre-processing, followed by 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; pre-processing does not.
- A pre-processor will often let errors through. A compiler hides further steps; a pre-processor does not.
Compilation vs. Interpretation

- Many compiled languages have interpreted pieces, e.g., 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

- 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).
Preprocessing

- Most C and C++ compilers use a preprocessor to import header files and expand macros.

```c
#include <stdio.h>
#define N 99
...
for (i=0; i<N; i++)
```

```c
for (i=0; i<99; i++)
```
The CPP Preprocessor

- Early C++ compilers used the CPP preprocessor to generated C code for compilation

**Diagram:**
- C++ Source Code → C++ Preprocessor → C Source Code → C Compiler → Assembly or Object Code
The translator can be a compiler or an interpreter. It is considered to be a compiler if:
1. There is a thorough analysis of the program
2. The transformation is non-trivial.

The Virtual Machine acts as an interpreter.
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:
Pure Compilation and Static Linking

- Adopted by the typical Fortran systems
- Library routines are separately linked (merged) with the object code of the program

```
extern printf();
```

```
_printf
_fget
_fscanf ...
```

```
Binary Executable
```

```
Source Program
```

```
Compiler
```

```
Incomplete Object Code
```

```
Static Library Object Code
```

```
Linker
```

```
Executable
```
Compilation, Assembly, and Static Linking

- Facilitates debugging of the compiler

Diagram:

- Source Program
- Compiler
- Assembly Program
- Assembler
- Linker
- Binary Executable
- Static Library
- Object Code

Code:
```c
extern printf();

_putchar
_fget
_fscanf
...
```
Compilation, Assembly, and Dynamic Linking

- Dynamic libraries (DLL, .so, .dylib) are linked at run-time by the OS (via stubs in the executable)

```c
extern printf();
```

- Shared Dynamic Libraries
  - _printf, _fget, _fscan, ...

Source Program → Compiler → Assembly Program → Assembler → Incomplete Executable → Output
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:
  - Bootstrapping

```
Pascal compiler, in Pascal, that generates machine language → Pascal compiler, in P-code, that generates P-code, running on the P-code interpreter → Pascal compiler, in P-code, that generates machine language

Pascal compiler, in Pascal, that generates machine language → Pascal compiler, in P-code, that generates machine language, running on the P-code interpreter → Pascal compiler, in machine language, that generates machine language
```
Bootstrapping

- Pascal came shipped with three things:
  - A Pascal to P-Code Compiler, written in Pascal.
  - A Pascal to P-Code Compiler, written in P-Code.
  - A P-code interpreter, in Pascal.

- The interpreter is hand translated into machine lang.
Bootstrapping

- To get a simple (but slow) compiler, one could use the “Pascal to P-Code compiler, in P-Code” and the “P-Code to machine language interpreter” to compile Pascal code.

This interpreter is needed to run the Pascal to P-Code compiler.
Bootstrapping (Faster)

- To create a faster compiler, we modify the “Pascal to P-code compiler, in *Pascal*” so that it is a “Pascal to machine language compiler, in *Pascal*.”
  - This is **MUCH HARDER** than creating a P-Code to Machine language interpreter.
Bootstrapping (Faster)
Bootstrapping (Faster)

- The first step is to **construct** the “Pascal to machine language compiler, in **P-Code**.”
- The second step is to run the “Pascal to machine language compiler, in **Pascal**” through the “**P-Code**” version to produce the “**Machine lang.**” version.
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.
JIT Compilers

Source Program

Compiler/Interpreter

Bytecode or IL Representation

JIT Compiler

Input

Machine Language

Output

Bytecode Interpreter

Input

Output
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.
  - Silicon compilers.
  - Query language processors.
# Programming Environment Tools

- **Tools**

<table>
<thead>
<tr>
<th>Type</th>
<th>Unix examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Editors</td>
<td>vi, emacs</td>
</tr>
<tr>
<td>Pretty printers</td>
<td>cb, indent</td>
</tr>
<tr>
<td>Pre-processors (esp. macros)</td>
<td>cpp, m4, watfor</td>
</tr>
<tr>
<td>Debuggers</td>
<td>adb, sdb, dbx, gdb</td>
</tr>
<tr>
<td>Style checkers</td>
<td>lint, purify</td>
</tr>
<tr>
<td>Module management</td>
<td>make</td>
</tr>
<tr>
<td>Version management</td>
<td>sccs, rcs</td>
</tr>
<tr>
<td>Assemblers</td>
<td>as</td>
</tr>
<tr>
<td>Link editors, loaders</td>
<td>Id, Id-so</td>
</tr>
<tr>
<td>Perusal tools</td>
<td>More, less, od, nm</td>
</tr>
<tr>
<td>Program cross-reference</td>
<td>ctags</td>
</tr>
</tbody>
</table>
An Overview of Compilation - Phases

- Character Stream
- Token Stream
- Parse Tree
- Abstract syntax tree
- Modified intermediate form
- Machine language
- Modified target language
- Scanner (lexical analysis)
- Parser (syntax analysis)
- Semantic analysis & intermediate code gen.
- Machine-independent optimization (optional)
- Target code generation.
- Machine-specific optimization (optional)
A Typical Compilation Process

- Try g++ with -v, -E, -S flags
Compilation Phases and Passes

- Compilation of a program proceeds through a fixed series of phases
  - Each phase uses an (intermediate) form of the program produced by an earlier phase
  - Subsequent phases operate on lower-level code representations
- Each phase may consist of a number of passes over the program representation
  - Pascal, FORTRAN, C languages designed for one-pass compilation, which explains the need for function prototypes
  - Single-pass compilers need less memory to operate
  - Java and ADA are multi-pass
Compiler Front- and Back-end

Front end analysis

Source program (character stream)

Scanner (lexical analysis)

Tokens

Parser (syntax analysis)

Parse tree

Semantic Analysis and Intermediate Code Generation

Back end synthesis

Abstract syntax tree or other intermediate form

Machine-Independent Code Improvement

Modified intermediate form

Target Code Generation

Assembly or object code

Machine-Specific Code Improvement

Modified assembly or object code
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.
Scanner: Lexical Analysis

- Lexical analysis breaks up a program into tokens
  - Grouping characters into non-separable units (tokens)
  - Changing a stream to characters to a stream of tokens

```plaintext
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.
```

```plaintext
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 .
```
Scanner: Lexical Analysis

- Recognize structures without regard to meaning and group them into tokens.

```plaintext
program gcd(input, output);
```

- The purpose of the scanner is to simplify the parser by reducing the size of the input.
Lexical Analysis

- Lexical analyzer: reads input characters and produces a sequence of tokens as output (nexttoken()).
  - Trying to understand each element in a program.
  - **Token**: a group of characters having a collective meaning.

```cpp
const pi = 3.14159;
```

Token 1: (const, -)
Token 2: (identifier, ‘pi’)  
Token 3: (=, -)  
Token 4: (realnumber, 3.14159)  
Token 5: (;, -)
Interaction of Lexical Analyzer with Parser

Source program

Lexical analyzer

token

Nexttoken()

symbol table

parser
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).
Parser: Syntax Analysis

- Checks whether the token stream meets the grammatical specification of the language and generates the syntax tree.
  - A syntax error is produced by the compiler when the program does not meet the grammatical specification.
  - For grammatically correct program, this phase generates an internal representation that is easy to manipulate in later phases
    - Typically a syntax tree (also called a parse tree).
- A grammar of a programming language is typically described by a context free grammar, which also defines the structure of the parse tree.
Parser: Syntax Analysis

• The Syntax analysis catches all malformed statements.
• The parse tree is sometimes called a concrete syntax tree because it contains how all tokens are derived.
• Much of this information is extraneous for the “meaning” of the code (e.g., the only purpose of “;” is to end a statement).
Parser: Syntax Analysis

- Parsing organizes the tokens into a context-free grammar (i.e., syntax).
Context-Free Grammars

- A context-free grammar defines the syntax of a programming language
- The syntax defines the syntactic categories for language constructs
  - Statements, Expressions, Declarations
- Categories are subdivided into more detailed categories
  - A Statement is a
    - For-statement
    - If-statement
    - Assignment

\[
<\text{statement}> ::= <\text{for-statement}> | <\text{if-statement}> | <\text{assignment}>
\]
\[
<\text{for-statement}> ::= \text{for} ( <\text{expression}> ; <\text{expression}> ; <\text{expression}> ) <\text{statement}>
\]
\[
<\text{assignment}> ::= <\text{identifier}> : = <\text{expression}>
\]
Context-Free Grammars

• Example (while loop in C)

\[
\text{iteration-statement} \rightarrow \text{while ( expression ) statement}
\]

statement, in turn, is often a list enclosed in braces:
\[
\text{statement} \rightarrow \text{compound-statement}
\]
\[
\text{compound-statement} \rightarrow \{ \text{block-item-list opt} \}
\]
where
\[
\text{block-item-list opt} \rightarrow \text{block-item-list}
\] or
\[
\text{block-item-list opt} \rightarrow \epsilon
\] and
\[
\text{block-item-list} \rightarrow \text{block-item}
\]
\[
\text{block-item-list} \rightarrow \text{block-item-list block-item}
\]
\[
\text{block-item} \rightarrow \text{declaration}
\]
\[
\text{block-item} \rightarrow \text{statement}
\]
Example: Micro Pascal

- `<Program>::= program <id> (<id> <More_ids>) ; <Block>`
  
  `<Block>::= <Variables> begin <Stmt> <More_Stmts> end`

- `<More_ids>::= , <id> <More_ids>`
  | ε

- `<Variables>::= var <id> <More_ids> : <Type> ; <More_Variables>`
  | ε

- `<More_Variables>::= <id> <More_ids> : <Type> ; <More_Variables>`
  | ε

- `<Stmt>::= <id> := <Exp>`
  | if <Exp> then <Stmt> else <Stmt>
  | while <Exp> do <Stmt>
  | begin <Stmt> <More_Stmts> end

- `<Exp>::= <num>`
  | <id`
  | <Exp> + <Exp`
  | <Exp> - <Exp>`
Parsing Examples

- \( \text{Pos} = \text{init} + \text{rate} \times 60 \Rightarrow id1 = id2 + \text{id3} \times \text{const} \Rightarrow \) syntax error (\( \text{exp ::= exp + exp} \) cannot be reduced).

- \( \text{Pos} = \text{init} + \text{rate} \times 60 \Rightarrow id1 = id2 + \text{id3} \times \text{const} \Rightarrow \)

\[
\begin{align*}
\text{Pos} & \Rightarrow id1 \quad \text{:=} \quad + \\
& \quad \text{id2} \quad \text{*} \quad \text{id3} \quad 60
\end{align*}
\]
A Pascal Example

program gcd(input, output);
var i, j: integer;
begin
  read(i,j); // get i & j from read
  while i<>j do
    if i>j then i := i-j
    else j := j-1;
  writeln(i)
end.
Parsing Examples

- Syntax Tree
  - GCD Program Parse Tree

```
<table>
<thead>
<tr>
<th>Index</th>
<th>Symbol</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>void</td>
<td>type</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>type</td>
</tr>
<tr>
<td>3</td>
<td>getint</td>
<td>func:(1) → (2)</td>
</tr>
<tr>
<td>4</td>
<td>putint</td>
<td>func:(2) → (1)</td>
</tr>
<tr>
<td>5</td>
<td>i</td>
<td>(2)</td>
</tr>
<tr>
<td>6</td>
<td>j</td>
<td>(2)</td>
</tr>
</tbody>
</table>
```

```
program
  ::= (5) call (3)
     | (6) call (3) while ≠ (5) if (4) (5)
     | > (5) (6) := (5) (5) := (5) (6) (5) (6)
```
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.
Semantic Analysis

- Semantic analysis discovers the **meaning of a program** by creating an **abstract syntax tree** that removes “extraneous” tokens.
- To do this, the analyzer builds & maintains a **symbol table** to map identifiers to information known about it. (i.e., scope, internal structure, etc...)
- By using the symbol table, the semantic analyzer can **catch problems not caught by the parser**.
  - Identifiers are declared before used
  - Subroutine calls provide correct number and type of arguments.
Semantic Analysis

```
program
  id(GCD)
  ( id(INPUT) more_ids )
  ;
  block
```

```
program
(5) read
  (3) (6) read
  (3) (7) Rest of code

<table>
<thead>
<tr>
<th>Index</th>
<th>Symbol</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>INTEGER</td>
<td>type</td>
</tr>
<tr>
<td>2</td>
<td>TEXTFILE</td>
<td>type</td>
</tr>
<tr>
<td>3</td>
<td>INPUT</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>OUTPUT</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>GCD</td>
<td>program</td>
</tr>
<tr>
<td>6</td>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>J</td>
<td>1</td>
</tr>
</tbody>
</table>
```
Semantic Analysis

- Not all semantic rules can be checked at compile time.
  - Those that can are called **static** semantics of the language.
  - Those that cannot are called **dynamic** semantics of the language.
    - Arithmetic operations **do not overflow**.
    - Array subscripts expressions **lie within the bounds of the array**.
Semantic Analysis

- Semantic analysis is applied by a compiler to discover the meaning of a program by analyzing its parse tree or abstract syntax tree.
- A program without grammatical errors may not always be correct program.
  - pos = init + rate * 60
  - What if pos is a class while init and rate are integers?
  - This kind of errors cannot be found by the parser
  - Semantic analysis finds this type of error and ensure that the program has a meaning.
Semantic Analysis

- Static semantic checks (done by the compiler) are performed at compile time
  - Type checking
  - Every variable is declared before used
  - Identifiers are used in appropriate contexts
  - Check subroutine call arguments
  - Check labels
Semantic Analysis

- Dynamic semantic checks are performed at run time, and the compiler produces code that performs these checks
  - Array subscript values are within bounds
  - Arithmetic errors, e.g. division by zero
  - Pointers are not dereferenced unless pointing to valid object
  - A variable is used but hasn't been initialized
  - When a check fails at run time, an exception is raised
Semantic Analysis and Strong Typing

- A language is strongly typed "if (type) errors are always detected"
  - Errors are either detected at compile time or at run time
  - Examples of such errors are listed on previous slide
  - Languages that are strongly typed are Ada, Java, ML, Haskell
  - Languages that are not strongly typed are Fortran, Pascal, C/C++, Lisp
- Strong typing makes language safe and easier to use, but potentially slower because of dynamic semantic checks
- In some languages, most (type) errors are detected late at run time which is detrimental to reliability e.g. early Basic, Lisp, Prolog, some script languages
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.
Code Generation and Intermediate Code Forms

- A typical intermediate form of code produced by the semantic analyzer is an abstract syntax tree (AST)
- The AST is annotated with useful information such as pointers to the symbol table entry of identifiers

Example AST for the gcd program in Pascal
Code Generation and Intermediate Code Forms

- Other intermediate code forms
  - Intermediate code is something that is both close to the final machine code and easy to manipulate (for optimization). One example is the three-address code:
    \[
    \text{dst} = \text{op1} \quad \text{op} \quad \text{op2}
    \]
  - The three-address code for the assignment statement:
    
    \begin{align*}
    \text{temp1} & = 60 \\
    \text{temp2} & = \text{id3} + \text{temp1} \\
    \text{temp3} & = \text{id2} + \text{temp2} \\
    \text{id1} & = \text{temp3}
    \end{align*}

- Machine-independent Intermediate code improvement
  
  \begin{align*}
  \text{temp1} & = \text{id3} \times 60.0 \\
  \text{id1} & = \text{id2} + \text{temp1}
  \end{align*}
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 (sometime) relocatable machine language.
Optimization

• The process so far will produce correct code, but it may not be fast.
• Optimization will adjust the code to improve performance.
  ▫ A possible machine-independent optimization would be to keep the variables $i$ and $j$ in registers throughout the main loop.
  ▫ A possible machine-specific optimization would be to assign the variables $i$ and $j$ to specific registers.
Target Code Generation and Optimization

- From the machine-independent form assembly or object code is generated by the compiler.
  
  ```
  MOVF id3, R2  
  MULF #60.0, R2  
  MOVF id2, R1  
  ADDF R2, R1  
  MOVF R1, id1 
  ```

- This machine-specific code is optimized to exploit specific hardware features.
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.
Target Code Generation

- Code generation takes the abstract syntax tree and the symbol table to produce machine readable code.
- Simple code follows directly from the abstract syntax tree and symbol table.
Summary

- Compiler front-end: lexical analysis, syntax analysis, semantic analysis
  - Tasks: understanding the source code, making sure the source code is written correctly
- Compiler back-end: Intermediate code generation/improvement, and Machine code generation/improvement
  - Tasks: translating the program to a semantically the same program (in a different language).