**CS330 - Programming Languages**

PLP1

Focus of the course:

1. Provide the tools for understand and *critical evaluation* of programming languages constructs and implementations.

Two general approaches:

* 1. Horizontal:
     + Focuses on several languages.
     + Discuss each in depth.
  2. Vertical:
* Present general language concepts and constructs in some particular sequence.
* Individual languages are used ONLY for illustration.

Our approach is hybrid:

* Vertical and
* Somewhat detailed discussions of languages from very different (not necessarily most popular) paradigms: Scheme, Prolog.

1. Give flavor of different types of languages

* Somewhat extensive discussion

## Scheme: example of Functional languages.

## Prolog: example of Logic languages

3. Prepare for the study of advanced topics such as compiler design.

Name some languages you know:

Go, C, C++, perl (scripting), python, ruby, java,

Diagram

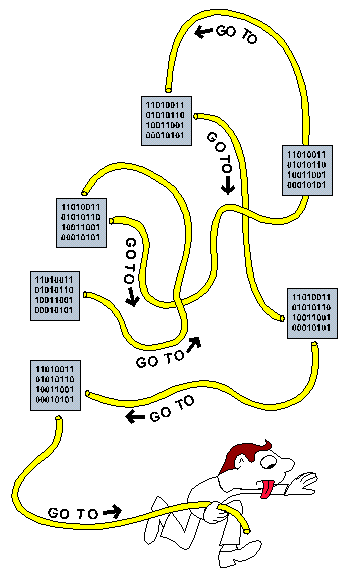
Description automatically generated

Chapter 1 - Introduction

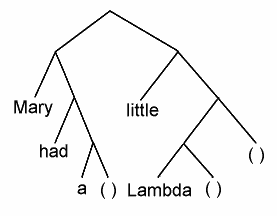
Why are there so many programming languages?

* Orientation toward special hardware, ex., FORTRAN I
  + Both If and Do statements (post-test loop) are implemented with 1 machine instruction on [IBM 704](https://www.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP704.html).
  + *Efficient object code*
* Evolution: we've learned better ways of doing things over time

Ex., goto-control 🡪 structured programming 🡪 OOP



* Orientation toward special purposes
  + Functional Languages such as Lisp/scheme/ML, designed for symbolic manipulation



In Lisp/scheme, this tree is represented by the following list:

( *(mary had a ()* (little (Lambda) )

* + Perl for text processing (more details [here](https://www.perl.org/))
    - Try it with online IDE such as [Jdoodle](https://www.jdoodle.com/execute-perl-online/)
  + C for system programming
  + Prolog for logical reasoning: focus on relationship

ex., opposite(east, west).

* Personal Preference - there are diverse ideas about what language is better or more pleasant to use.

What makes a language successful, or [popular](http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html)

* Easy to learn (BASIC, Pascal, Scheme, Java)
* Expressive power - what features and abstraction facilities are provided, although
  + All languages are [Turing Complete](https://www.youtube.com/watch?v=RPQD7-AOjMI) (detail [here](https://en.wikipedia.org/wiki/Turing_machine))
    - Turing machine is extremely simple, but can compute any algorithm, giving sufficient time and space
  + Features not available in a language can often be simulated/emulated, but at a cost.
  + The choice of programming language construct limits the control, data structure and abstractions used to some extent, example later.
* Easy to implement (limited hardware resources required): Basic, PASCAL
* Excellent compiler: Fortran I, II, III
* Backing of a powerful sponsor (ex., DoD): [Ada](http://www.nap.edu/openbook.php?record_id=5463&page=7), [COBOL](http://en.wikipedia.org/wiki/COBOL)
* Wide dissemination at minimal cost: BASIC, PASCAL, java
* Open source, ex., C (unix, linux, etc)

1.3 Why study programming languages

* Make it easier to learn new languages – learn how to learn:
  + Easier if you already know a similar language
  + Easier if you are exposed to the entire spectrum of languages
  + Easier if you have a core set of knowledge

### Improved background for choosing appropriate languages – it’s always a [tradeoff](https://www.garfieldtech.com/blog/language-tradeoffs)

* understand relatively obscure 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, symbol internals
* Simulate useful features in languages that don’t support them.
  + Ex., what data structure would you use to represent the following expression in C++: A\*(B–C)

BTW, the lisp representation of this expression: (\* A (- B C))

\*

A -

B C

Easily implemented in C, C++, java,

Ex.,

struct node

{

string symbol;

… // other book keeping

node \*left, \*right;

};

double eval (node\* r)

{

switch (r->symbol)

{

case ‘+’: return eval(r->left)+eval(r->right);

break;

case ‘-‘: return eval(r->left)-eval(r->right);

break;

case ‘\*’: return eval(r->left)\*eval(r->right);

break;

case ‘/’: return eval(r->left)/eval(r->right);

break;

default:

return lookup(r->symbol);

}

}

If we have to use a language that supports array but not pointers, how should we represent the tree?

A heap array

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| \* | A | - |  |  | B | C |  |  |  |

0 1 2 3 4 5 6

In heap storage, left child of a node at index k has an index of 2k+1, right child will be at 2k+2

3 parallel arrays

Symbol (char)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| \* | A | - |  |  | B | C |  |

0 1 2 3 4 5 6 7

Left (int)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | -1 | 5 |  |  | -1 | -1 |  |

Right (int)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | -1 | 6 |  |  | -1 | -1 |  |

Example of [Turing Machine](https://www.youtube.com/watch?v=-ZS_zFg4w5k):

A diagram of a machine

Description automatically generated

Defining ++ operation (or TM) with a table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| State | Current symbol | Write | Head Move | next state |
| A | Blank | Blank | Right | A |
| A | 0 | 0 |  | B // at leftmost |
| A | 1 | 1 |  | B // at leftmost |
| B | 0 | 0 | Right | B |
| B | 1 | 1 | Right | B |
| B | Blank | Blank | Left | C // at rightmost |
| C | 0 | 1 | Left | D // done |
| C | 1 | 0 | Left | C |
| C | Blank | 1 |  | D |
| D | Blank | Blank |  | halt |
| D | 0 | 0 | Left | D |
| D | 1 | 1 | Left | D |

Tape that extends infinitely on both ends

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | 0 | 0 | 1 | 1 |  |  |

^

^: location of **read/write** head

State:

From ++, we can build + and then \*

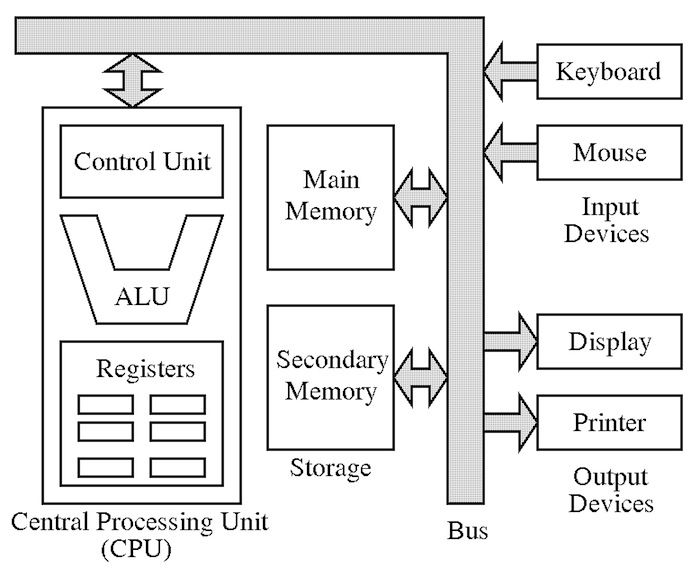
7+5 🡺 7++ + 5-- 🡪 …..

7\*6 🡺 7+7+7+7+7+7

1.2 Spectrum of Programming Languages

Languages can be categorized as

* Imperative (programmers specify **how** to solve a problem)
* Procedure – based on [von Neumann](https://learnlearn.uk/alevelcs/von-neumann-architecture/) (Fortran, Pascal, Basic, C)
* Object-oriented (Smalltalk, Eiffel, java, C++)
* [scripting languages](https://thebittheories.com/what-is-a-scripting-language-c208857bd62c) (Perl, JavaScript, PHP)
* declarative (programmers specify **what** need to be done)
* functional (Scheme, pure Lisp, FP)
* logical, constraint-based (Prolog, VisiCalc, RPG, SQL)



Example – Calculating Greatest Common Divisor (GCD) (figure 1.2)

// use **algo** like syntax

Algorithm for gcd (a, b)

While (a<>b)

{

If (a > b)

a =a-b;

else // b>=a

b= b-a;

}

Return a;

C++:

int gcd(int a, int b)

{

while (a!=b)

{

if (a > b)

a =a-b;

else

b= b-a;

}

return a;

}

Scheme:

(define gcd

(lambda (a b) ;; parameter setup

(cond ((= a b) a) ;; if a == b return a // in prefix

((> a b) (gcd (- a b) b))

(else (gcd (- b a) a))))

)

Prolog:

gcd(A, B, G) :- A = B, G =A.

gcd(A, B, G) :- A > B, C is A-B, gcd(C, B, G).

gcd(A, B, G) :- B > A, C is B-A, gcd (C, A, G).

we can run this with a query such as :

?- gcd(6,4,Goat).

Goat=2;

?- gcd(9, 361, 3).

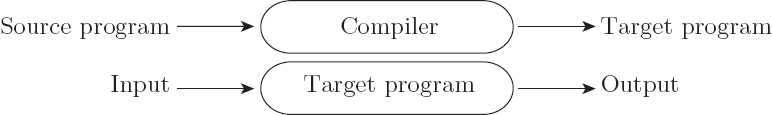
No.

-------------- Reading Assignment for MLK day: PLP 1.4 – 1.5 ----------------------

* 1. Compilation vs. Interpretation

Pure Compilation

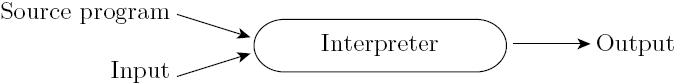
* The compiler translates the high-level source program into an equivalent target program (typically in machine language), and then goes away:



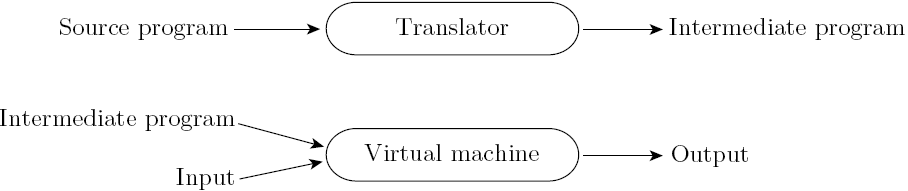
* Better performance

Pure Interpretation

* Interpreter stays around for the execution of the program
* Interpreter is the locus of control during execution
* Greater flexibility
* Better diagnostics (higher, language level error messages)
* Slower execution



* Most language implementations include a mixture of both compilation and interpretation. Ex, Pascal 🡪p-code 🡪 ML



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

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)
  + Some allows conditional compilation – part of code maybe deleted, ex.,

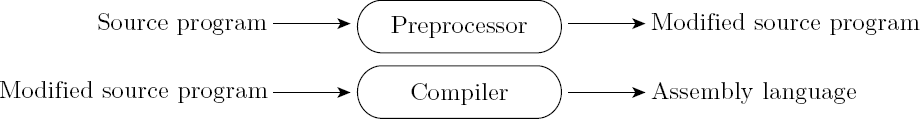
/\* C preprocessor allows us to turn on and off of prinf \*/

**#ifdef DEBUG**

**printf("debug:x = %d, y = %f\n", x, y);**

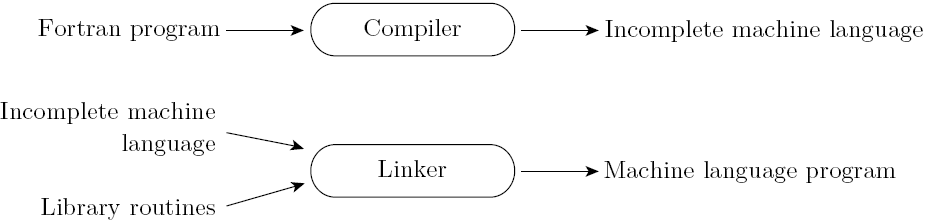
**...**

**#endif**



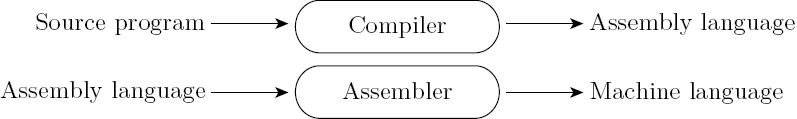
Compilation vs. pre-processing:

* 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.
* Many compiled languages have interpreted pieces, e.g., formatting in Fortran or C
* Most compilers use “virtual instructions” in addition to machine code
* set operations in Pascal
* string manipulation in Basic
* Some compilers only produce virtual instructions, e.g., Pascal P-code, Java byte code, assembly (?), C
* Library 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:

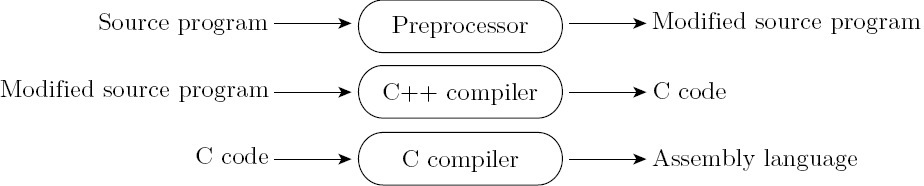


Post-compilation Assembly

* Source program is compiled into 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)



* 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:



Let’s say I designed a new language called zinc. What language should we use to write a compiler for my new language? Zinc

[**Bootstrapping**](https://www.huffpost.com/entry/pull-yourself-up-by-your-bootstraps-nonsense_n_5b1ed024e4b0bbb7a0e037d4) – compiler of a language written in that language

How do we compile/interpret/execute the compiler written in the same language?

1. Write another translator
2. Use another compiler written in another language
3. Grow your compiler in size,

Ex., Start by hand translating your first (very small) compiler

Example: PASCAL bootstrapping

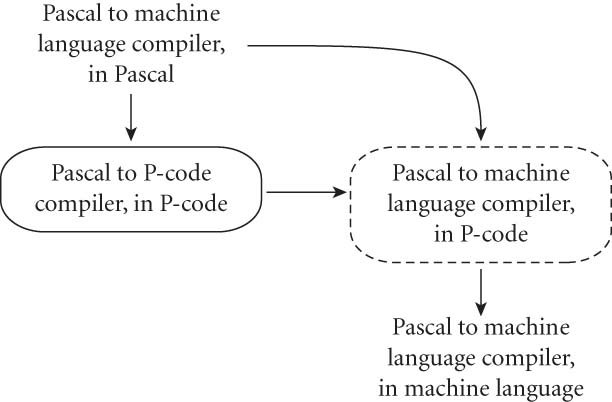
Early Pascal compilers were built with the following tool set

* A Pascal to p-code compiler (written) in Pascal,
* A Pascal to p-code compiler in p-code
* A p-code interpreter written in Pascal

*Step 1 - hand translate the p-code interpreter from Pascal to local ML*

Result

* *A p-code interpreter written in ML*

**

[Pascal](https://www.youtube.com/watch?v=1dAJFNOWoQY) – a Quick Intro

Timing of compilation/interpretation

* Early 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.
* Late (or delayed) - Just-in-Time (JIT) Compilation

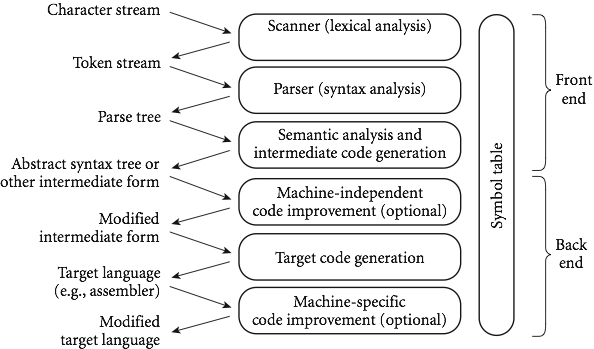
In some cases a programming system may deliberately delay compilation until the last possible moment. Ex., Java byte code and JVM

* Java classes contain platform-neutral bytecodes that can be interpreted by a JVM on many different computer architectures 🡪 slower, but more portable
* All bytecode can also be compiled to machine code at run time, just before execution 🡪 slow at start up, then speed up.

Compiler vs. interpreter (continued)

* Compilers exist for some interpreted languages, but they aren't pure:
  + Selective compilation of compliable 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., ex SQL processor in DBMS
  1. An Overview of Compilation Process

Phases of compilation:



Many Compiler components can be created automatically from a (formal) description of the source and/or target language.

The phases (or passes):

1. ***Scanning***:

* divides the program into tokens, which are the smallest meaningful units; this saves time, since character-by-character processing is slow.
* we can fine tune the scanner better if its job is simple; it also reduces complexity (lots of it) for later stages.
* We can use a parser to take characters directly as input, but it isn't efficient.
* Example:

/\* GCD Program (in C) \*/

**int main() {**

**int i = getint(), j = getint(); /\* some hypothetic I/O procedure \*/**

**while (i != j) {**

**if (i > j) i = i - j;**

**else j = j - i;**

**}**

**putit(i);**

**}**

Ex., 7, 5 🡪 2, 5 🡪 2, 3 🡪 2,1 🡪 1,1 🡺 GCD=1

19, 38 🡪 19, 19 🡺 GCD=19

Tokens – figure 1.21

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| int | main | ( | ) | { | int |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

1. ***Parsing*** – understanding the syntax

* Parsing discovers the "context free" structure of the program.
* Parsing organizes tokens into a *parse tree* that represents higher-level constructs in terms of their constituents
* Informally, it finds the structure which you can describe with syntax diagrams (the "circles and arrows" in a Pascal manual - [example](http://primepuzzle.com/tp2/syntax-diagrams.html))
* Recursive rules known as *Context-Free Grammar* (CFG) define the ways in which these constituents combine. Example, while loop in C can be defined as:

*iteration-statement → while ( expression ) statement*

The statement is often a list enclosed in braces:

*statement → compound-statement*

*compound-statement → { block-item-list\_opt }*

where

*block-item-list\_opt → block-item-list*

or

*block-item-list\_opt → ϵ*

and

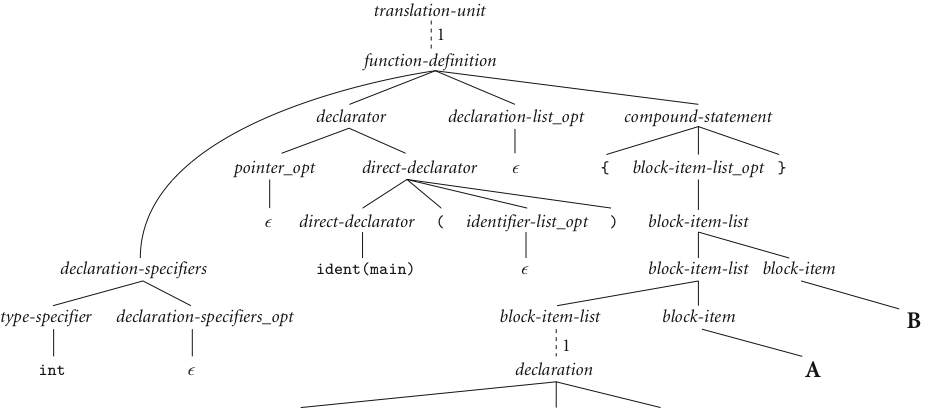
*block-item-list → block-item*

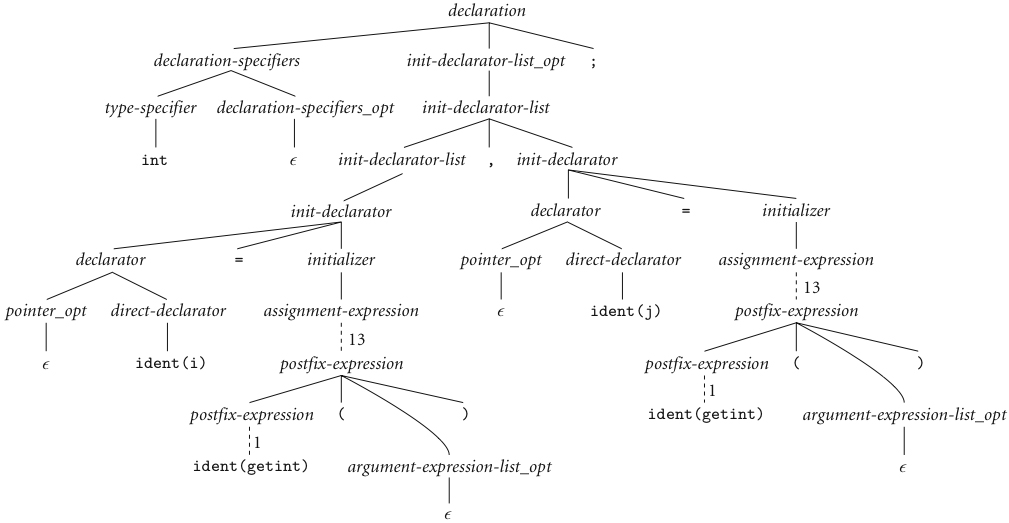
*block-item-list → block-item-list block-item*

*block-item → declaration*

*block-item → statement*

GCD Program Parse Tree

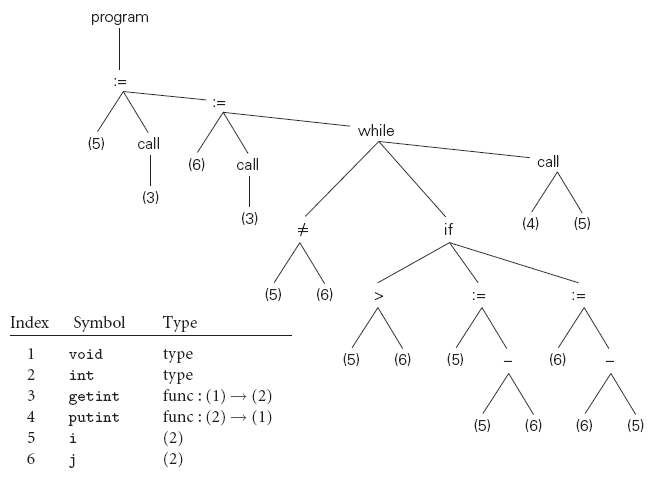






1. ***Semantic analysis*** - the discovery of *meaning* of the program

* The compiler performs STATIC semantic analysis to determine meaning that can be figured out at compile time.
  1. Recognize multiple occurrences of the same identifier, such as i in our example, often by creating a symbol table
  2. Track data type of identifiers and expressions
  3. Parameter matching
  4. Type checking
* This often results in the generation of a syntax tree. Ex.,



* Some things can't be figured out until run time. They are part of the program's DYNAMIC semantics. Example:
  1. Variables must have values before they appear on the RHS of an expression.
  2. Pointers must refer to valid object when dereferenced (i.e., going to the pointee)
  3. Array subscript must be in bound.

1. ***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

1. ***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, yet important

1. ***Code generation phase*** - produces assembly language or re-locatable machine language

* 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 symbols (ex., 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