NARSIMHA REDDY ENGINEERING COLLEGE
(Autonomous)
Approved by AICTE, NewDelhi & Affiliated to JNTUH, Hyderabad
Accredited by NBA&NAAC with A Grade

SCRIPTING LANGUAGES

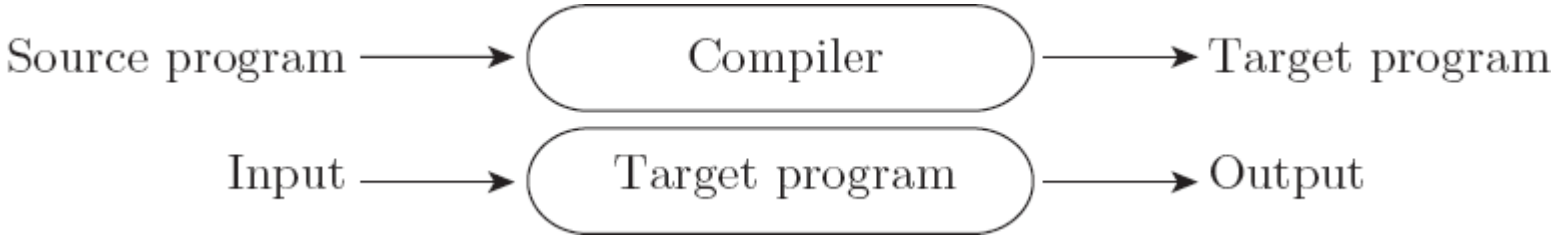
By
G Sunil Kumar
Assistant Professor
Department of CSE

Compilation vs. Interpretation

- Compilation vs. interpretation
  - not opposites
  - no absolute distinction

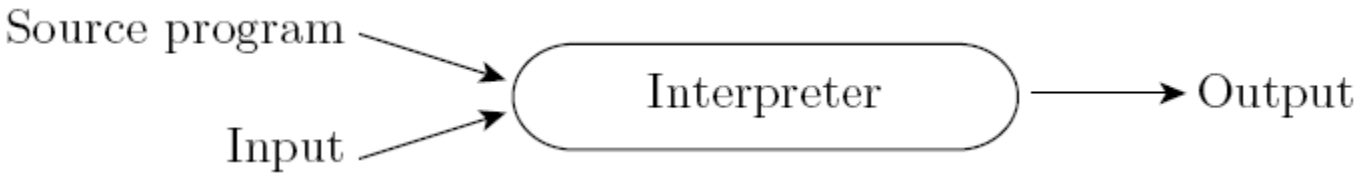
Compilation vs. Interpretation

- Pure compilation
  - compiler translates source program into equivalent target program, then goes away
  - often high-level language (source code) translated to machine language (object code)
  - OS later executes target program on machine
  - target program is locus of control



Compilation vs. Interpretation

- Pure interpretation
  - interpreter stays around for execution of program
  - interpreter is locus of control during execution
  - interpreter implements virtual machine

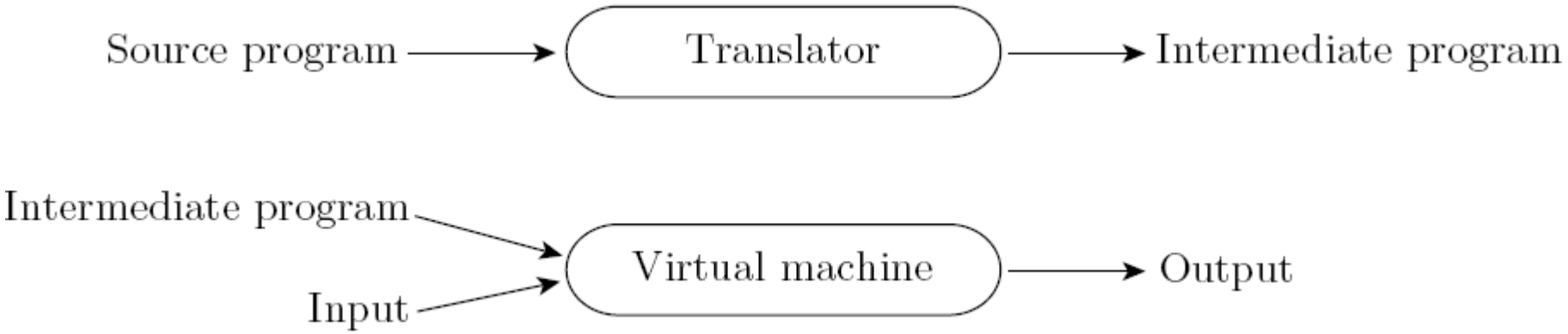


Compilation vs. Interpretation

- Interpretation
  - greater flexibility
  - better error messages (e.g., good source-level debugger)
  - dynamically create code and then execute it
- Compilation
  - better performance

Compilation vs. Interpretation

- Most language implementations mix compilation and interpretation
- Common case compilation or pre-processing – followed by interpretation



Compilation vs. Interpretation

- Compilation *not* required to produce machine code for hardware
- Compilation *translates* one language into another, fully analyzing input's meaning
- Compilation requires semantic *understanding* of input
- Preprocessing does not require semantic understanding, allows some errors through
- Compiler hides subsequent steps
- Preprocessor does not hide subsequent steps

Compilation vs. Interpretation

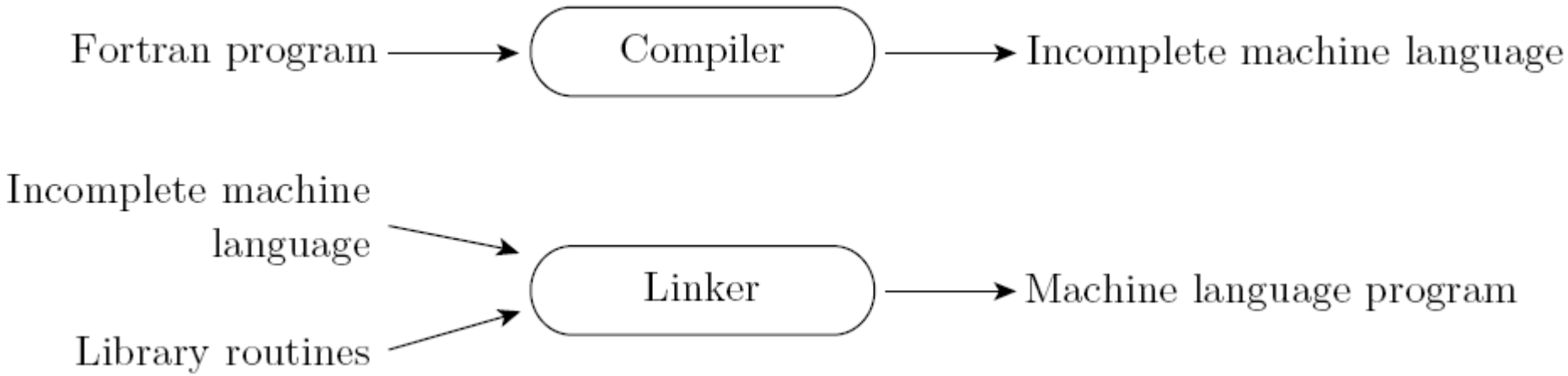
- Compiled languages have interpreted features
  - input/output formats
- Compiled languages may use “virtual instructions”
  - set operations
  - string operations
- Compiled languages might only produce virtual instructions, e.g., Java byte code

Compilation vs. Interpretation

- Implementation strategy: Preprocessor
  - removes comments and white space
  - groups characters into *tokens* (keywords, identifiers, numbers, symbols)
  - expands abbreviations and textual macros
  - identifies higher-level syntactic structures (loops, subroutines)
  - preserves structure of source in intermediate form

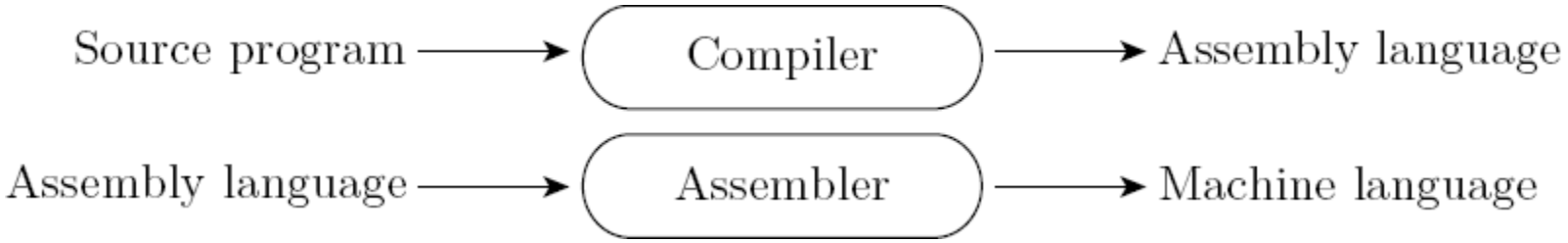
Compilation vs. Interpretation

- Implementation strategy: Library and linking
  - compiler uses *linker* program to merge appropriate subroutines from *library*



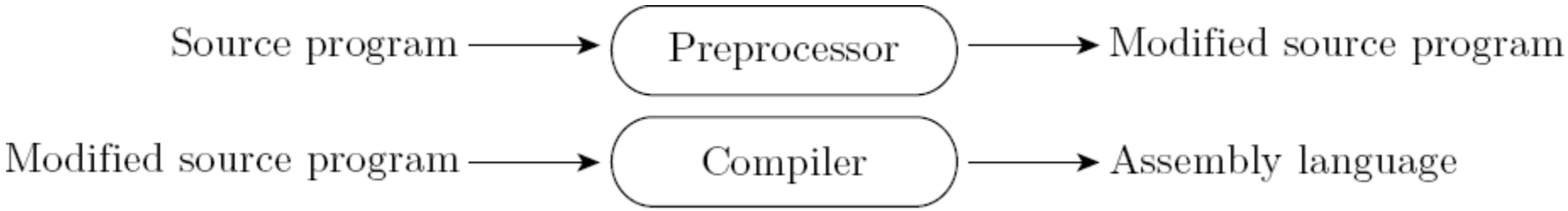
Compilation vs. Interpretation

- Implementation strategy: Post-compilation assembly
  - facilitates debugging (assembly easier to read)
  - isolates compiler from changes in format of machine code files (e.g., between OS releases)



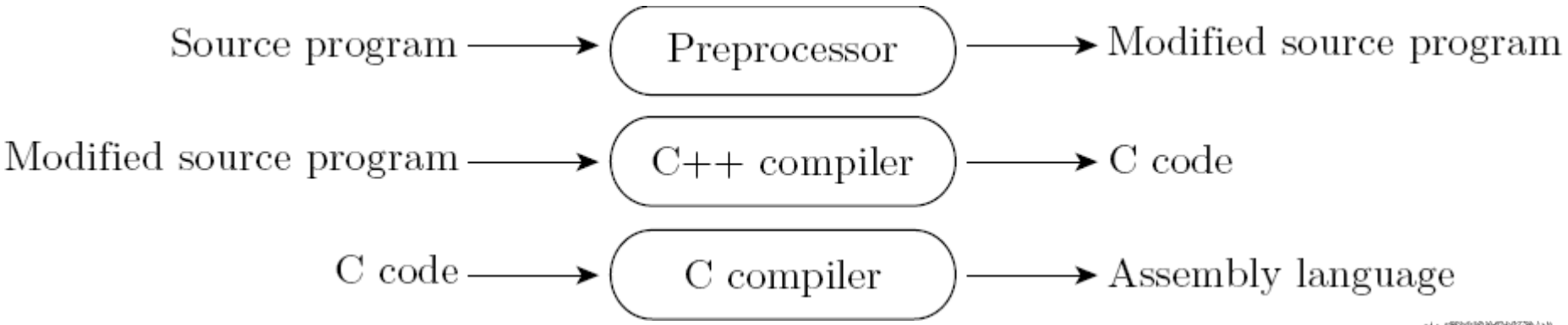
Compilation vs. Interpretation

- Implementation strategy: Conditional compilation
  - preprocessor deletes portions of code, several program versions share same source
  - e.g., C's preprocessor



Compilation vs. Interpretation

- Implementation strategy: Source-to-source translation
  - generate intermediate program in another language (e.g., C++ to C, various to JavaScript)

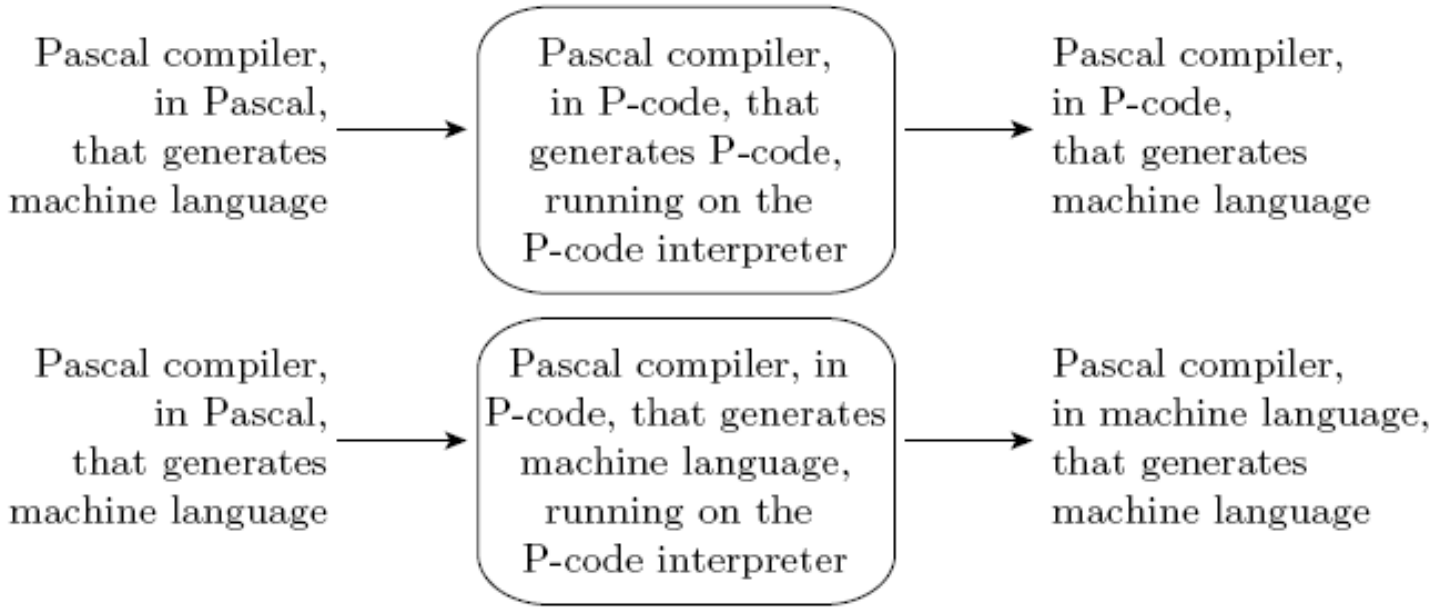


Compilation vs. Interpretation

- Implementation strategy: Compilation of interpreted languages
  - Compiler generates code guessing about runtime circumstances
  - If correct, code is fast
  - If not, dynamic check reverts to normal interpreter

Compilation vs. Interpretation

- Implementation strategy: Bootstrapping



Compilation vs. Interpretation

- Implementation strategy: Dynamic and Just-in-Time compilation
  - Deliberately delay compilation until last possible moment
    - compile source code on the fly – dynamically created source -- optimize program for particular input
    - use machine-independent intermediate code but compile to machine code when executed (e.g., Java just-in-time-compiler, .NET CIL)

Compilation vs. Interpretation

- Implementation strategy: Microcode
  - Assembly-level instruction set not implemented in hardware; runs on interpreter.
  - Interpreter written in low-level instructions (*microcode* or *firmware*), stored in read-only memory, executed by hardware

Compilation vs. Interpretation

- Compilers exist for some interpreted languages, but not pure
  - selective compilation of part + sophisticated preprocessing of rest
  - interpretation of part still necessary for reasons above
- Unconventional compilers
  - text formatters
  - silicon compilers
  - query language processors

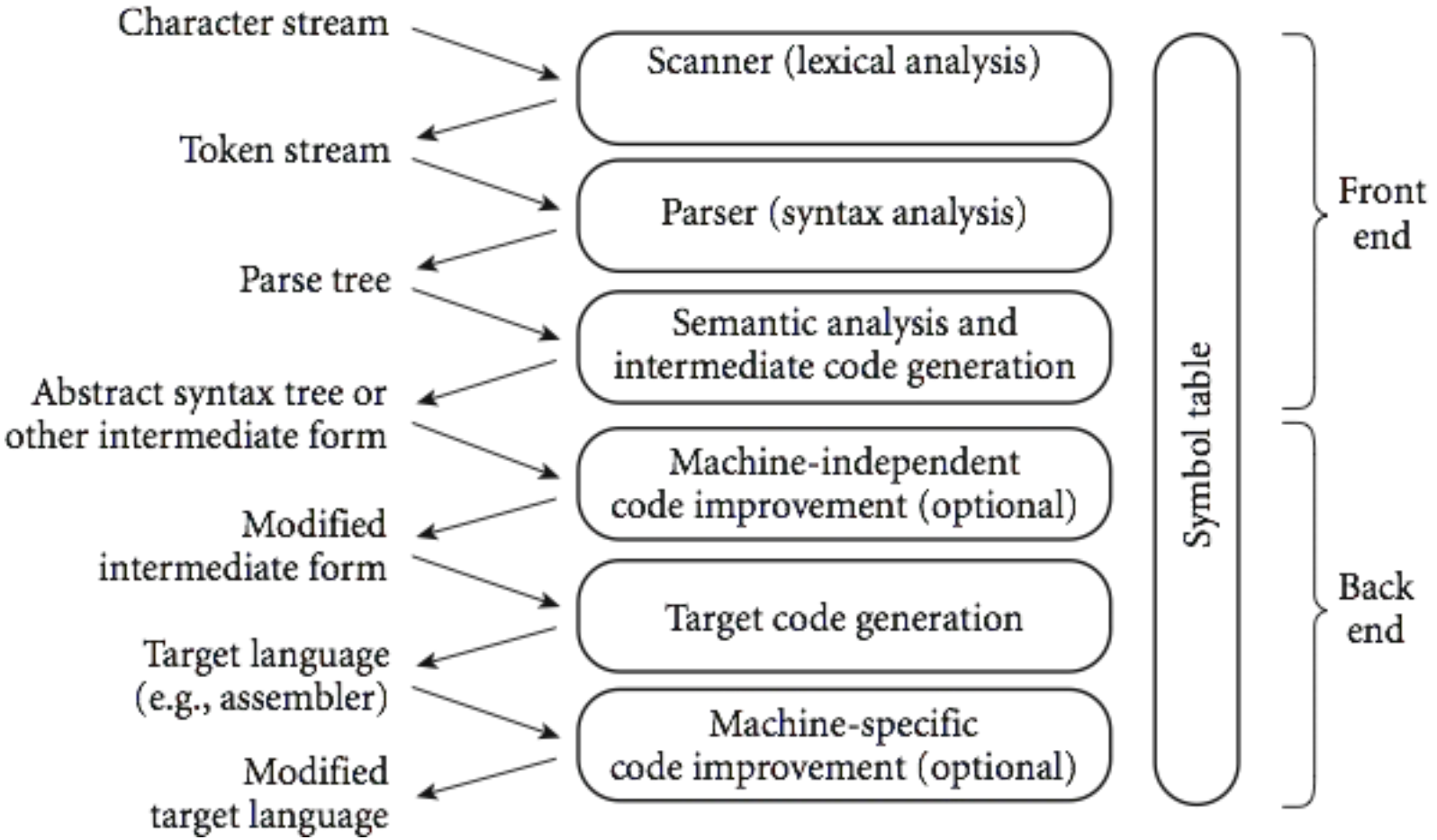
Programming Environment Tools

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

An Overview of Compilation

- Phases of Compilation



An Overview of Compilation

- ***Lexical Analysis (Scanning)***
  - recognize *regular language* using DFA
  - take input character stream
  - divide program into "tokens", smallest meaningful units to save time (char-by-char processing slow)
  - recognize identifiers, constants, keywords, operators
  - produce token stream
  - do simple tasks early to reduce complexity later

An Overview of Compilation

- ***Syntax Analysis (Parsing)***
  - recognize *context-free language (CFG)* using PDA
  - take token stream (but could take character stream with no scanner, might be quite messy)
  - discover context-free grammatical structure of program
  - output error messages
  - produce concrete syntax (parse) tree

An Overview of Compilation

- ***Intermediate form*** (IF)
  - produced if no errors in syntax or static "semantics"
  - machine code for idealized machine; e.g. stack machine or with unlimited number of registers
  - chosen to balance machine independence, ease of optimization, ease of translation to final form, compactness
  - might use several intermediate forms
  - use abstract syntax trees and symbol table in our interpreters

An Overview of Compilation

dent optimization

e program, optionally produce
" program – faster, smaller, etc.

lly optimize

mediate form program

subexpression elimination, copy
de elimination, loop
function calls, tail recursion

An Overview of Compilation

- ***Code generation***
  - produce assembly language or relocatable machine language from intermediate form and symbol table
  - assign memory locations, registers, etc.

- ***Machine-specific optimization***
  - take output of code generation
  - Optionally improve using specific details of machine, e.g., special instructions, addressing modes, co-processors

An Overview of Compilation

- ***Symbol table***
  - track information about identifiers throughout all phases
  - may be (partially) retained to support debugging, error recovery, reflection/metaprogramming

An Overview of Compilation

- Lexical and Syntax Analysis: GCD program (in C)

```
int main() {
int i = getint(), j = getint();
while (i != j) {
if (i > j) i = i - j;
else j = j - i;
}
putint(i);
}
```

An Overview of Compilation

- Lexical and Syntax Analysis: GCD program tokens
  - L*exical analysis* (scanning) and parsing recognize structure of program, group characters into *tokens*

```
int     main   (   )        {
int     i      =   getint   (   )   ,   j   =   getint   (   )   ;
while   (      i   !=       j   )   {
if      (      i   >        j   )   i   =   i   -       j   ;
else    j      =   j        -   i   ;
}
putint  (      i   )        ;
}
```

An Overview of Compilation

An Overview of Compilation

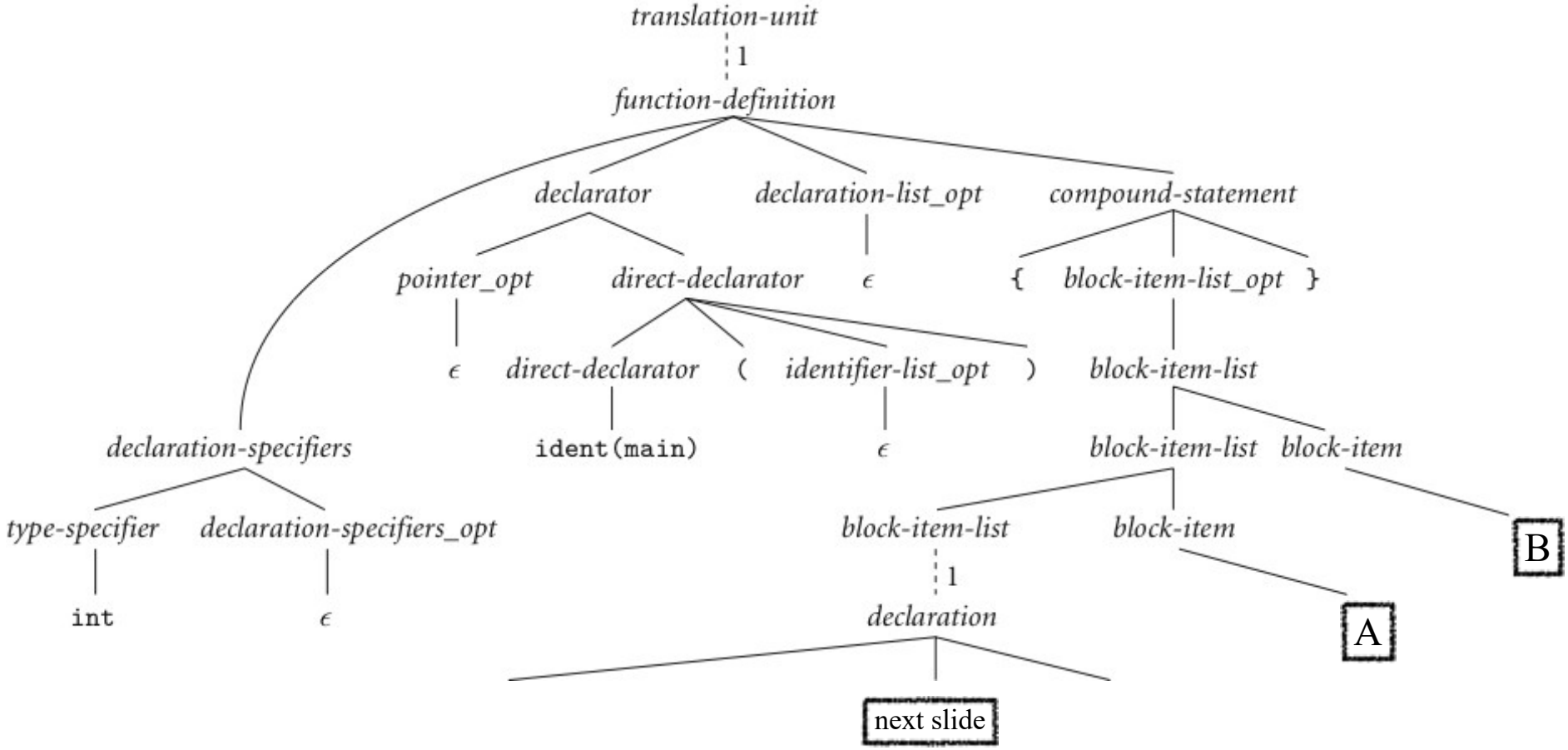
- Context-Free Grammar and Parsing: Example (`while` loop in C)

iteration-statement → *while (expression) statement*

statement, in turn, 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*

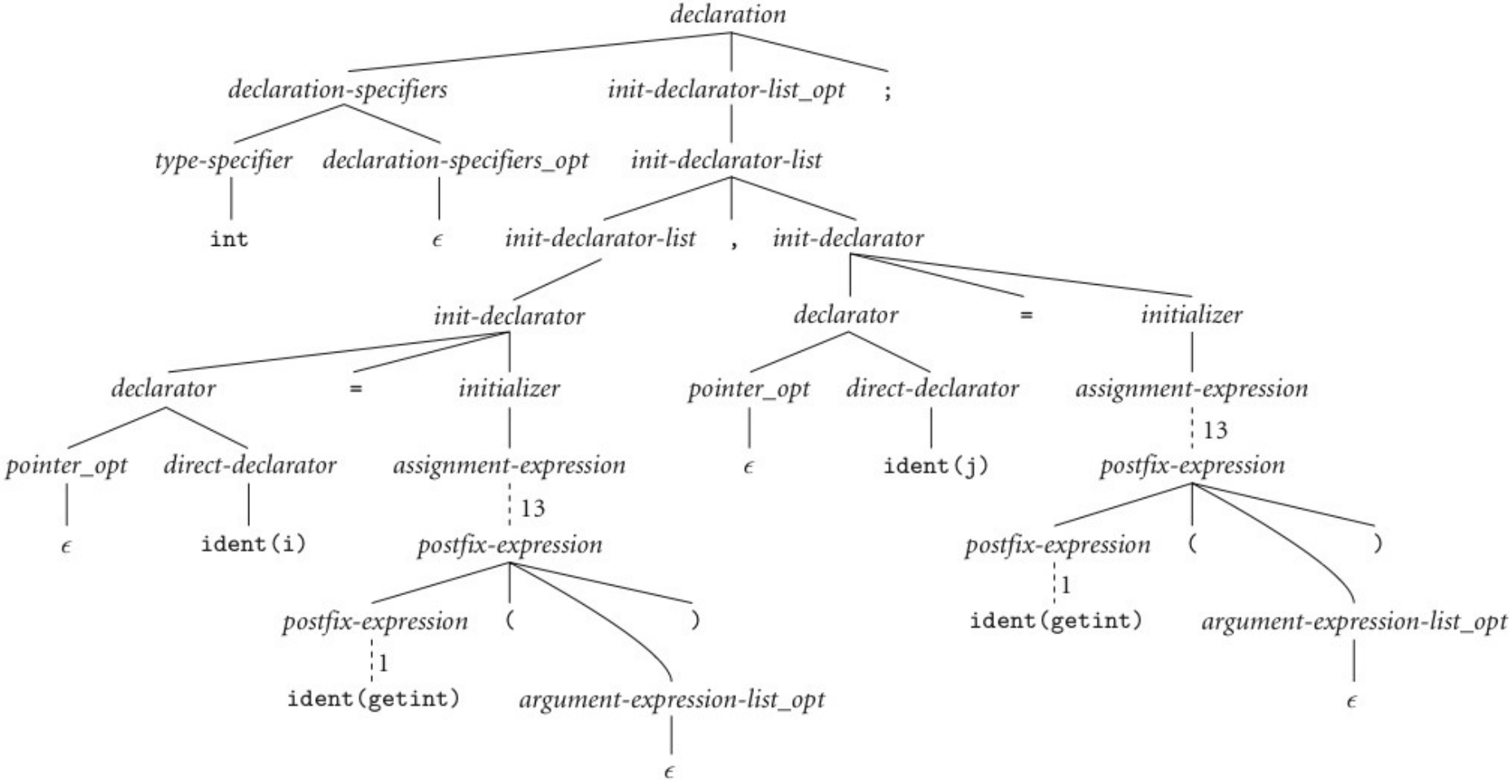
An Overview of Compilation

- Context-Free Grammar and Parsing: GCD Program Parse Tree



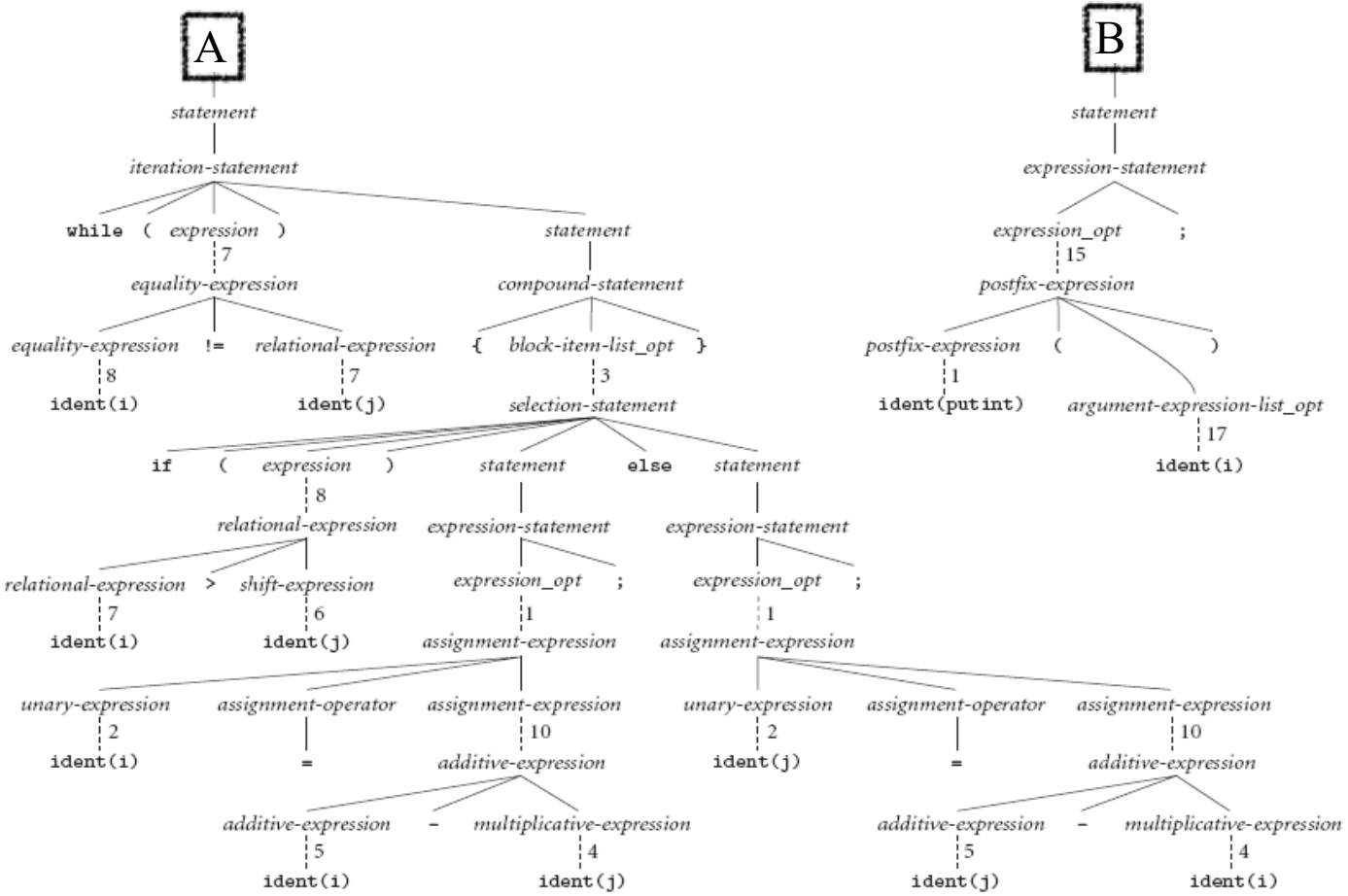
An Overview of Compilation

- Context-Free Grammar and Parsing (continued)



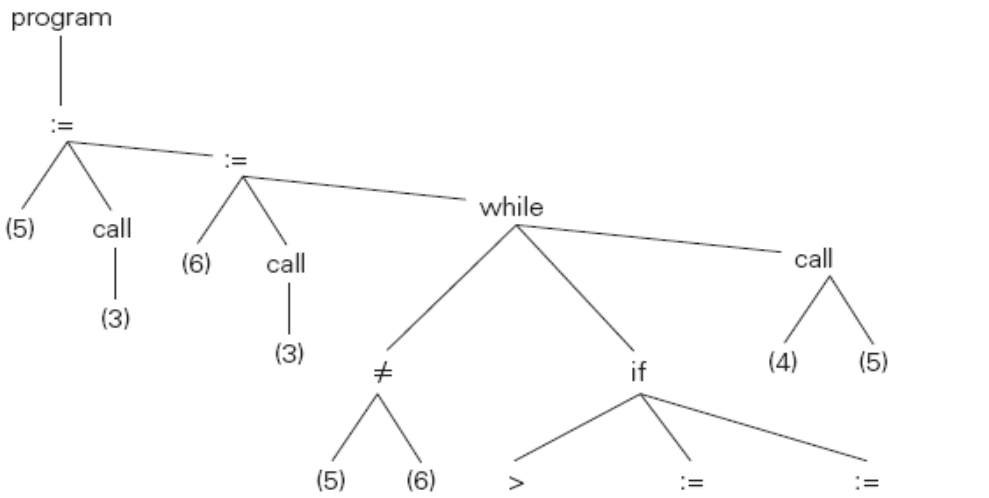
An Overview of Compilation

- Context-Free Grammar and Parsing (continued)



An Overview of Compilation

- Syntax Tree: GCD Program Parse Tree



Index	Symbol	Type
1	`void`	type
2	`int`	type
3	`getint`	func : (1) → (2)
4	`putint`	func : (2) → (1)
5	`i`	(2)
6	`j`	(2)

Ruby (on Rails)

G Sunil Kumar

M.Tech

Assistant Professor

Dept. of CSE

Narsimha Reddy Engineering College

Hyderabad

About the Section

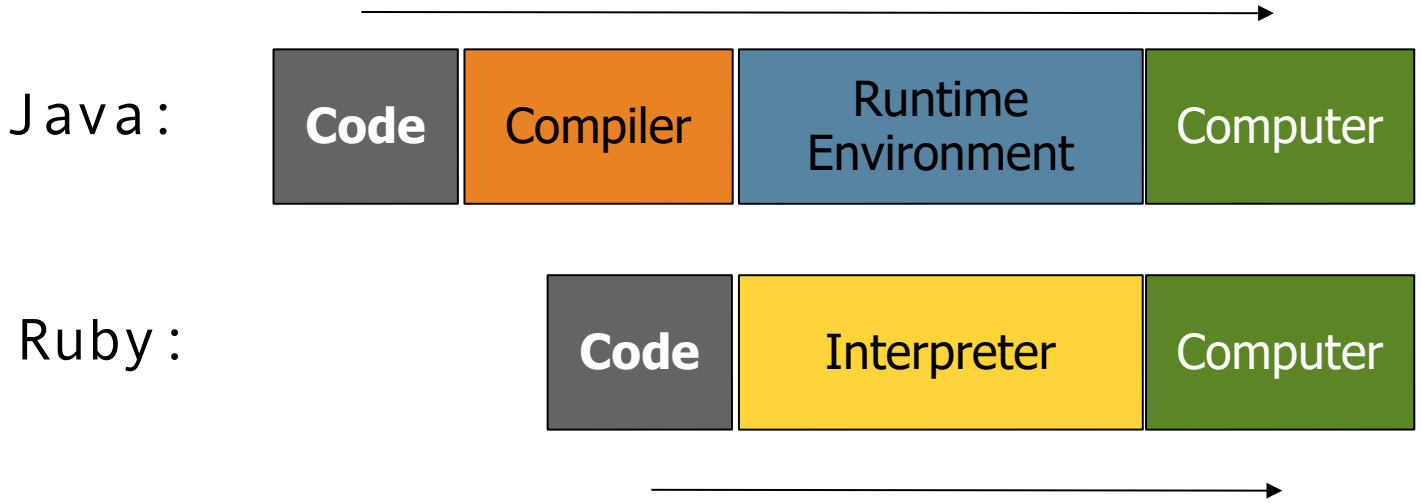
- Introduce the Ruby programming language
- Use Ruby to template web pages
- Learn about Ruby on Rails and its benefits

What is Ruby?

- Programming Language
- Object-oriented
- Interpreted

Interpreted Languages

- Not compiled like Java
- Code is written and then directly executed by an **interpreter**
- Type commands into interpreter and see immediate results



What is Ruby on Rails (RoR)

- Development framework for web applications written in Ruby
- Used by some of your favorite sites!

Advantages of a framework

- Standard features/functionality are built-in
- Predictable application organization
  - Easier to maintain
  - Easier to get things going

Installation

- Windows
  - Navigate to: http://www.ruby-lang.org/en/downloads/
  - Scroll down to "Ruby on Windows"
  - Download the "One-click Installer"
  - Follow the install instructions
    - Include RubyGems if possible (this will be necessary for Rails installation later)
- Mac/Linux
  - Probably already on your computer
  - OS X 10.4 ships with broken Ruby! Go here…
    - http://hivelogic.com/articles/view/ruby-rails-mongrel-mysql-osx

hello_world.rb

```
puts "hello world!"
```

puts vs. print

- "puts" adds a new line after it is done
  - analogous System.out.println()

- "print" does not add a new line
  - analogous to System.out.print()

Running Ruby Programs

- Use the Ruby interpreter

```
ruby hello_world.rb
```

  - "ruby" tells the computer to use the Ruby interpreter
- Interactive Ruby (irb) console

```
irb
```

  - Get immediate feedback
  - Test Ruby features

Comments

```
# this is a single line comment


=begin
   this is a multiline comment
   nothing in here will be part of the code
=end
```

Variables

- Declaration – No need to declare a "type"
- Assignment – same as in Java
- Example:

```
x = "hello world"        # String
y = 3                    # Fixnum
z = 4.5                  # Float
r = 1..10                # Range
```

Objects

- Everything is an object.
  - Common Types (Classes): Numbers, Strings, Ranges
  - nil, Ruby's equivalent of null is also an object
- Uses "dot-notation" like Java objects
- You can find the class of any variable

```
x = "hello"
x.class    → String
```

- You can find the methods of any variable or class

```
x = "hello"
x.methods
String.methods
```

Objects (cont.)

- There are many methods that all Objects have
- Include the "?" in the method names, it is a Ruby naming convention for boolean methods
  - nil?
  - eql?/equal?
  - ==, !=, ===
  - instance_of?
  - is_a?
  - to_s

Numbers

- Numbers are objects
- Different Classes of Numbers
  - FixNum, Float

```
3.eql?2      → false
-42.abs      → 42
3.4.round    → 3
3.6.rount→ 4
3.2.ceil     → 4
3.8.floor → 3
3.zero?      → false
```

String Methods

"hello world".length → 11

"hello world".nil? → false

"".nil? → false

"ryan" > "kelly" → true

"hello_world!".instance_of?String → true

"hello" * 3 → "hellohellohello"

"hello" + " world" → "hello world"

"hello world".index("w") → 6

Operators and Logic

- Same as Java
  - Multiplication, division, addition, subtraction, etc.
- Also same as Java AND Python (WHA?!)
  - "and" and "or" as well as "&&" and "||"
- Strange things happen with Strings
  - String concatenation (+)
  - String multiplication (*)
- Case and Point: There are many ways to solve a problem in Ruby

if/elsif/else/end

- Must use "elsif" instead of "else if"
- Notice use of "end". It replaces closing curly braces in Java
- Example:

```
if (age < 35)
   puts "young whipper-snapper"
elsif  (age < 105)
   puts "80 is the new 30!"
else
   puts "wow… gratz..."
end
```

Inline "if" statements

- Original if-statement

```
age=100
if age < 105
   puts "don't worry, you are still young"
end
```

- Inline if-statement

```
age=95
puts "don't worry, you are still young" if age < 105
```

for-loops

- for-loops can use ranges
- Example 1:

```
for i in 1..10
    puts i
end
```

- Can also use blocks (covered next week)

```
3.times do
  puts "Ryan! "
end
```

for-loops and ranges

- You may need a more advanced range for your for-loop
- Bounds of a range can be expressions
- Example:

```
for i in 1..(2*5)
    puts i
end
```

while-loops

- Can also use blocks (next week)
- Cannot use "i++"
- Example:

```
i = 0
while i < 5
    puts i
    i = i + 1
end
```

unless

- "unless" is the logical opposite of "if"

- Example:

```
unless (age >= 105)        # if (age < 105)
  puts "young."
else
  puts "old."
end
```

until

- Similarly, "until" is the logical opposite of "while"
- Can specify a condition to have the loop stop (instead of continuing)
- Example

```
i = 0
until (i >= 5)        # while (i < 5), parenthesis not required
    puts I
    i = i + 1
end
```

Methods

- Structure

  def ***method_name***(***parameter1***, ***parameter2***, **…**)

  statements

  end

- Simple Example:

```
def print_ryan
  puts "Ryan"
end
```

Parameters

- No class/type required, just name them!
- Example:

```
def cumulative_sum(num1, num2)
  sum = 0
  for i in num1..num2
  sum = sum + i
  end
  return sum
end

# call the method and print the result
puts(cumulative_sum(1,5))
```

Return

- Ruby methods return the value of the last statement in the method, so…

```
def add(num1, num2)
  sum = num1 + num2
  return sum
end
```

can become

```
def add(num1, num2)
  num1 + num2
end
```

User Input

- "gets" method obtains input from a user
- Example

```
name = gets
puts "hello " + name + "!"
```

- Use chomp to get rid of the extra line

```
puts "hello" + name.chomp + "!"
```

- chomp removes trailing new lines

Changing types

- You may want to treat a String a number or a number as a String
  - to_i – converts to an integer (FixNum)
  - to_f – converts a String to a Float
  - to_s – converts a number to a String
- Examples

"3.5".to_i	→	3
"3.5".to_f	→	3.5
3.to_s	→	"3"

Constants

- In Ruby, constants begin with an Uppercase
- They should be assigned a value at most once
- This is why local variables begin with a lowercase
- Example:

```
Width = 5
def square
  puts  ("*" * Width + "\n") * Width
end
```

References

- Web Sites
  - http://www.ruby-lang.org/en/
  - http://rubyonrails.org/
- Books
  - Programming Ruby: The Pragmatic Programmers' Guide (http://www.rubycentral.com/book/)
  - Agile Web Development with Rails
  - Rails Recipes
  - Advanced Rails Recipes

Package Management with RUBYGEMS

RubyGems is a standardized packaging and installation framework for libraries and applications, making it easy to locate, install, upgrade, and uninstall Ruby packages.

- It provides users and developers with four main facilities

1. A standardized package format,
2. A central repository for hosting packages in this format,
3. Installation and management of multiple, simultaneously installed versions of the same library
4. End-user tools for querying, installing, uninstalling, and otherwise manipulating these packages.

- In the RubyGems world, developers bundle their applications and libraries into single files called gems.
- These files conform to a standardized format, and the RubyGems system provides a command-line tool, appropriately named gem, for manipulating these gem files.

Installing Ruby Gems

- To use RubyGems, we need to download and install the RubyGems system from the project's home page at http://rubygems.rubyforge.org.
- After downloading and unpacking the distribution, we can install it using the included installation script