CS351 Organization of Programming Languages

Introduction to Programming Languages

Jakramate Bootkrajang

Areerat Trongratsameethong

2 / 2566

Outline

- Why do we have programming language?
- Why study programming languages?
- Programming language paradigms
- Compiler vs interpreter
- Overview of compilation process

Why do we have programming language?

- 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? [1/2]

- Help you choose a language.
  - C++ vs. Rust for systems programming
  - Fortran vs. Julia for numerical computations
  - Python vs. JavaScript for web applications
  - Ada vs. C for embedded systems
  - Common Lisp vs. Scheme vs. ML for symbolic data manipulation
  - Java vs. Scala for application servers
- Make it easier to learn new languages
  - Familiarity with related languages
  - Understanding core concepts that reappear

Why study programming languages? [2/2]

- Help you make better use of whatever language you use
  - Specialized features
    - unions, first-class functions, …
  - Implementation costs
    - Garbage collection, tail recursion
  - Emulating missing features
    - Recursion (with loops and stacks)
    - First-class functions (with objects)…or vice versa!
- Learn the terminologies used in PL

First-class functions (with objects): the language supports passing functions as arguments to other functions, returning them as the values from other functions, and assigning them to variables or storing them in data structures.

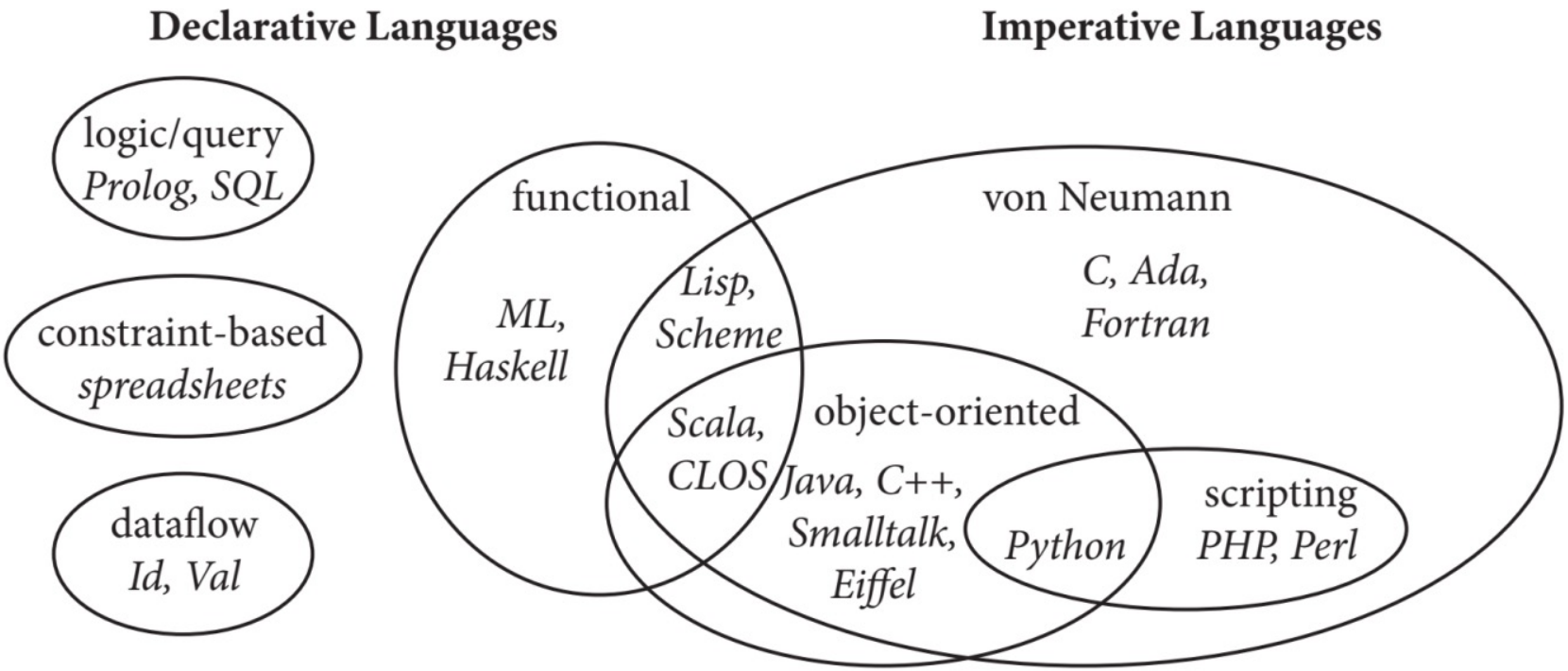
Languages you currently know

- What is your favorite language, and why do you like it?

Why are there so many?

- 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 works well (and what people like)

Languages landscape



Scott, M. L. (2016). *Programming Language Pragmatics*.

Imperative languages

- Focus is on how the computer should do it.
- Tightly coupled with machine architecture.
- Better performance can be expected.
- Python code snippet for item searching

```
1 def look_for_h(names):
2     for name in names:
3         result = []
4         if "h" in name:
5             result.append(name)
6     return result
```

Declarative languages

- Focus is on what the computer is to do
- Are in some sense “higher level”
- They are more in tune with the programmer’s point of view

```
SELECT * FROM names WHERE name LIKE "%h%";
```

Paradigms of PL: von Neumann languages

- The most familiar and successful.
- The basic means of computation is the modification of variables.
- Examples: C, Fortran, Ada 83, Python

```
 1 names = ["Jim", "John", "Jones"]
 2 Human = {}
 3
 4 i = 0
 5 for name in names:
 6     Human[i] = name
 7     i += 1
 8
 9 print(Human)
10 Human[0] = "Tom" # modify name
```

Paradigms of PL: Object-oriented languages

- OOP sees computations as interactions among semi-independent objects, each of which has both its own internal state and subroutines to manage that state.
- Examples: Ruby, Java, Smalltalk

```ruby
class Human
  @@species = 'H. sapiens' # common variable
  def initialize(name)
    @name = name
  end
  # basic getter method
  def get_name
    @name
  end
end
```

```ruby
names = ["John","Jim","Jones"]
harray = []

names.each do |name|
  harray.push(Human.new(name))
end

harray.each do |human|
  puts human.get_name()
end
```

Paradigms of PL: Functional languages

- Functional languages employ a computational model based on the recursive definition of functions.
  - Racket, Lisp, ML, Haskell, Scheme

```
(cons "Jim" (cons "John" (cons "Jones" empty)))
```

```
1 (define (sum-from-1-to-n-recursion n)
2   (cond
3     ((<= n 0) 0)    ; Base case: if n is <= 0, return 0
4     (else (+ n (sum-from-1-to-n-recursion (- n 1))))))) ;
```

Paradigms of PL: Logic languages

- Logic- or constraint-based languages take their inspiration from predicate logic.
- They model computation as an attempt to find values that satisfy certain specified relationships, using goal-directed search through a list of logical rules.
- Examples: Prolog

Prolog: example

Here is the new law we wrote plus a database of primitive propositions:

```
=====================================================

sisterOf(X,Y) :- female(Y), parents(Y,M,W), parents(X,M,W).

parents(bart, homer, marge).
parents(lisa, homer, marge).
parents(maggie, homer, marge).
female(marge).
female(lisa).
female(maggie).

=====================================================
```

Here is a query:

```
=====================================================

?- sisterOf(bart, Z).
Z = lisa ;
Z = maggie .
```

So many languages but one machine language

```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a = a - b;
        else b = b - a;
    }
    return a;
}
```

```
(define gcd
  (lambda (a b)
    (cond ((= a b) a)
          ((> a b) (gcd (- a b) b))
          (else (gcd (- b a) a)))))
```

```
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).
```

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

```
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
```

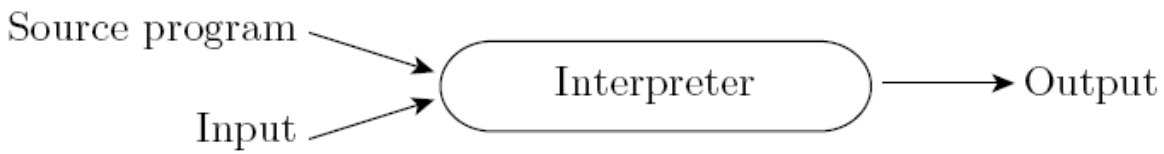
Scott, M. L. (2016). *Programming Language Pragmatics*.

Compiler vs interpreter

- Compilation vs. interpretation
  - not opposites
  - no absolute distinction
- Interpretation
  - greater flexibility
  - better error messages (e.g., good source-level debugger)
  - dynamically create code and then execute it
- Compilation
  - better performance

Pure interpretation

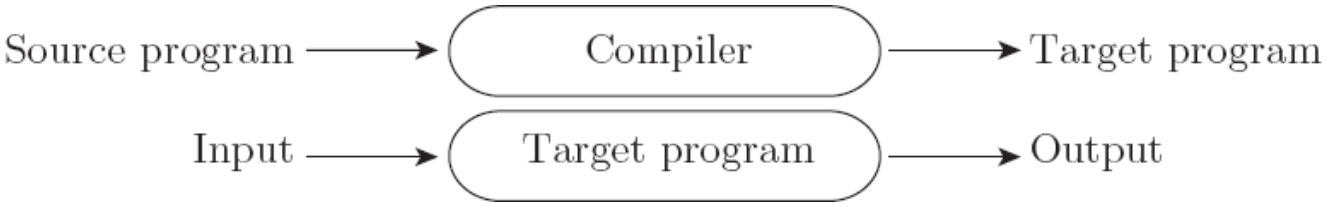
- interpreter stays around for execution of program
- interpreter is locus of control during execution



Scott, M. L. (2016). *Programming Language Pragmatics*.

Pure compilation

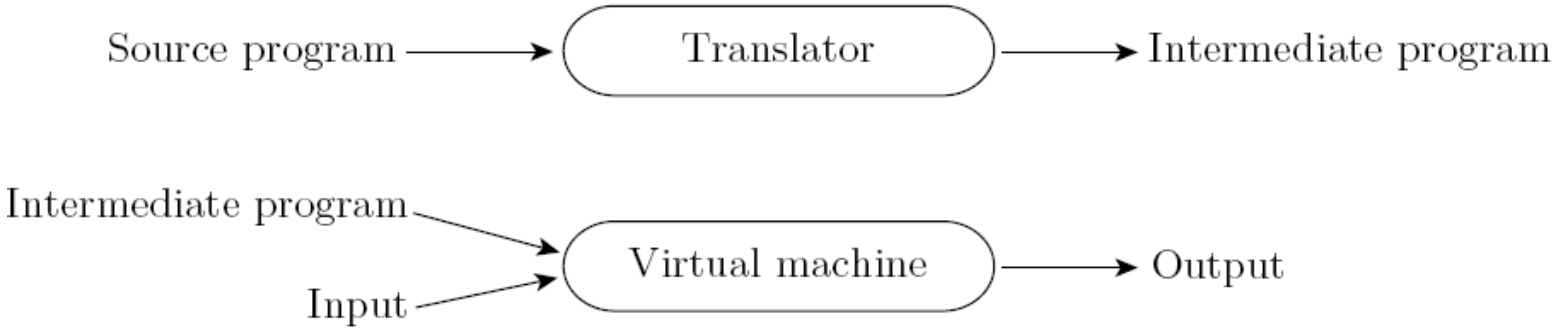
- compiler translates source program into equivalent target program, then goes away
- often high-level language (source code) translated to machine language (object code)
- OS later executes target program on machine
- target program is locus of control



Scott, M. L. (2016). *Programming Language Pragmatics*.

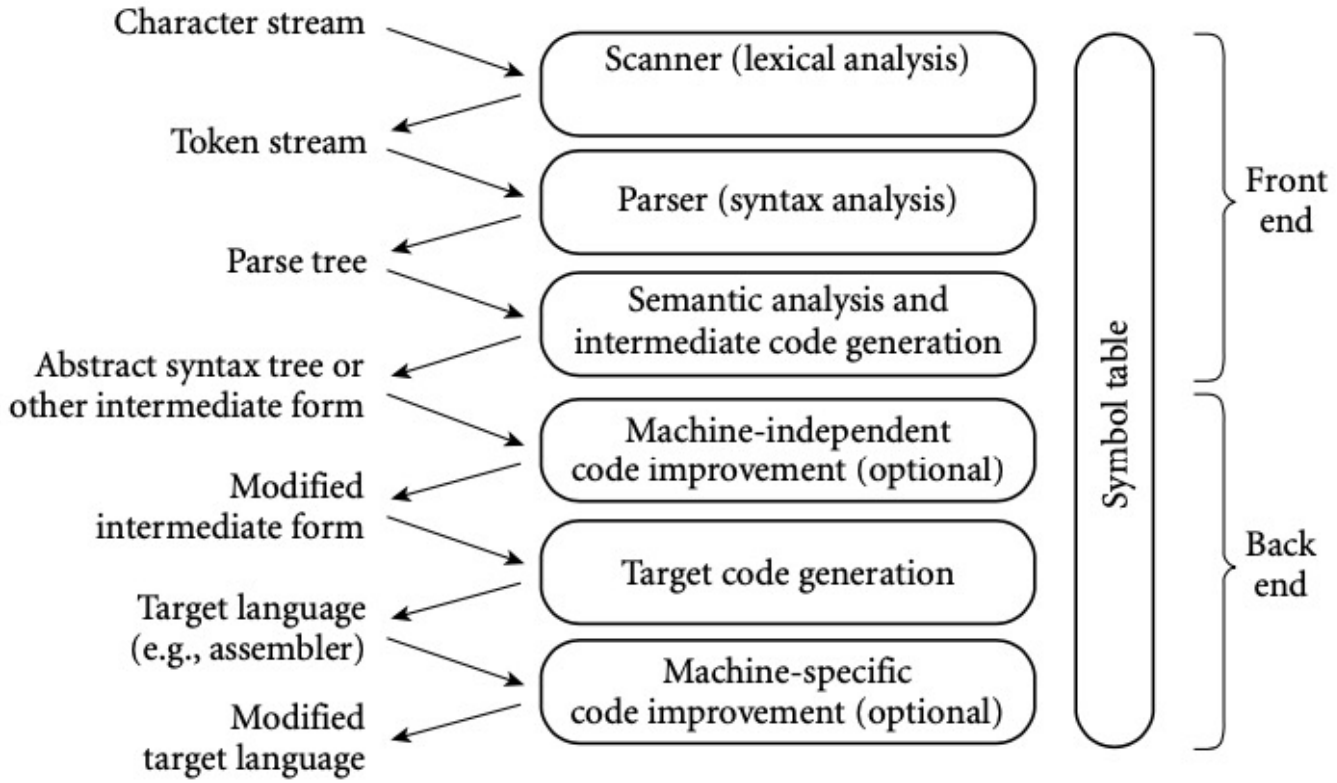
Mix compilation / interpretation

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



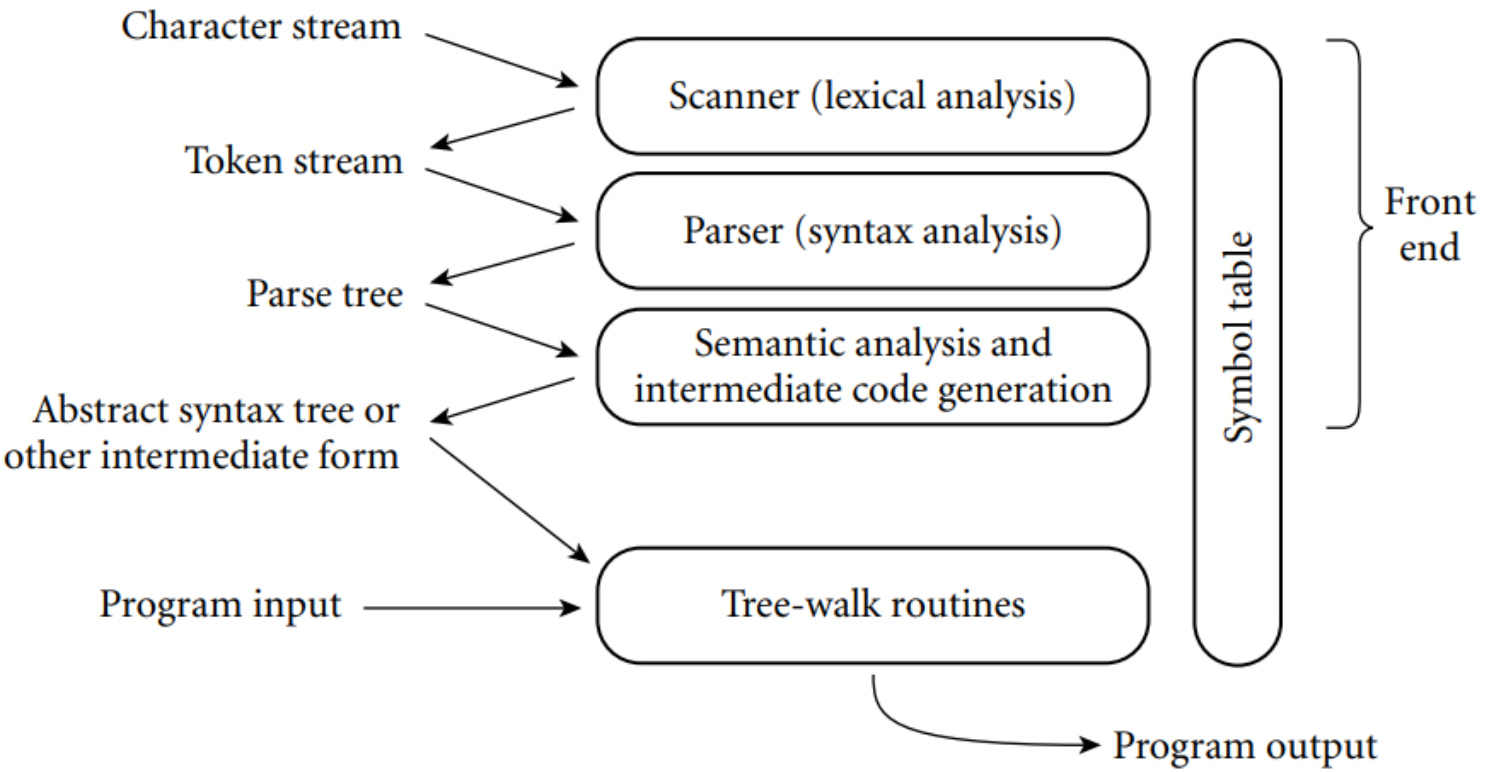
Scott, M. L. (2016). *Programming Language Pragmatics*.

Compilation Process Overview



Scott, M. L. (2016). *Programming Language Pragmatics*.

Interpretation Process Overview



Scott, M. L. (2016). *Programming Language Pragmatics*.

Scanner

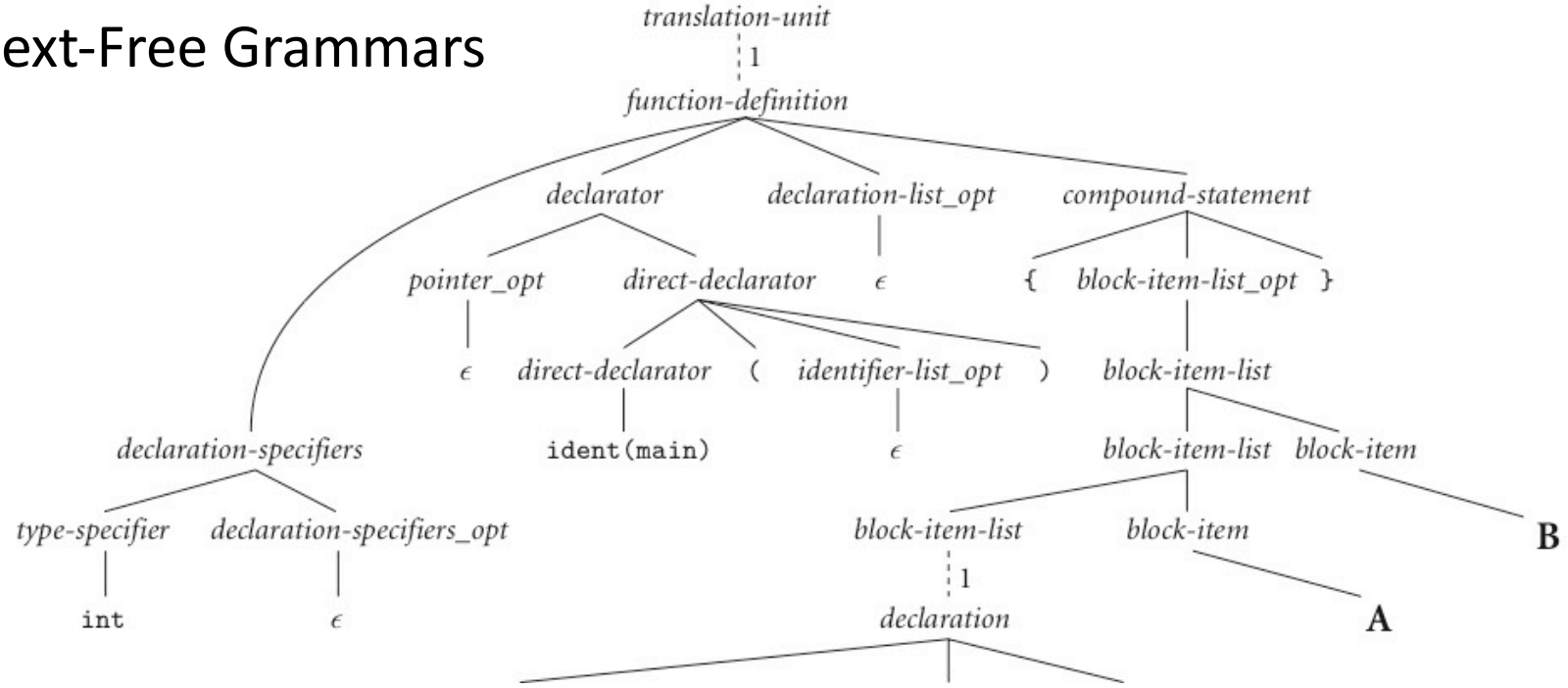
- Program tokenization
- Uses **regular expression**

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

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

Parsing

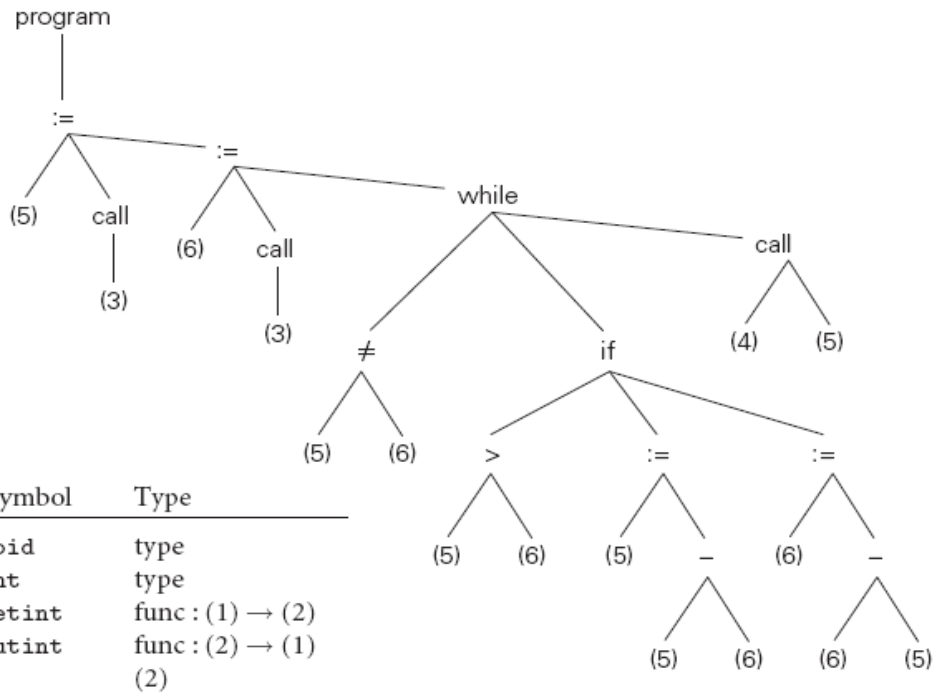
- Build tree of tokens (Parse Tree)
- Uses Context-Free Grammars



Scott, M. L. (2016). *Programming Language Pragmatics*.

Semantic Analysis

- Syntax Tree = Parse tree + semantic analysis



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)

Scott, M. L. (2016). *Programming Language Pragmatics*.

Target code generation

```
    pushl   %ebp                # \
    movl    %esp, %ebp          #  ) reserve space for local variables
    subl    $16, %esp           # /
    call    getint              # read
    movl    %eax, -8(%ebp)      # store i
    call    getint              # read
    movl    %eax, -12(%ebp)     # store j
A:  movl    -8(%ebp), %edi      # load i
    movl    -12(%ebp), %ebx     # load j
    cmpl    %ebx, %edi          # compare
    je      D                   # jump if i == j
    movl    -8(%ebp), %edi      # load i
    movl    -12(%ebp), %ebx     # load j
    cmpl    %ebx, %edi          # compare
    jle     B                   # jump if i < j
    movl    -8(%ebp), %edi      # load i
    movl    -12(%ebp), %ebx     # load j
    subl    %ebx, %edi          # i = i - j
    movl    %edi, -8(%ebp)      # store i
    jmp     C
B:  movl    -12(%ebp), %edi     # load j
    movl    -8(%ebp), %ebx      # load i
    subl    %ebx, %edi          # j = j - i
    movl    %edi, -12(%ebp)     # store j
C:  jmp     A
D:  movl    -8(%ebp), %ebx      # load i
    push    %ebx                # push i (pass to putint)
    call    putint              # write
    addl    $4, %esp            # pop i
    leave                       # deallocate space for local variables
    mov     $0, %eax            # exit status for program
    ret                         # return to operating system
```

Figure 1.6 Naive x86 assembly language for the GCD program.

Scott, M. L. (2016). *Programming Language Pragmatics*.

Code Optimization

- Optimized assembly for the previous assembly code

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

Scott, M. L. (2016). *Programming Language Pragmatics*.

Check Your Understanding

1. ภาษาเครื่อง และภาษา Assembly แตกต่างกันอย่างไร?
2. ทำไมจึงมีโปรแกรมมากมายหลายภาษา?
3. อะไรทำให้ภาษาโปรแกรมประสบความสำเร็จ
4. ให้ยกตัวอย่าง ภาษาโปรแกรม จำนวน 3 ภาษาของแต่ละกลุ่ม ดังต่อไปนี้
   - von Neumann, functional, object-oriented, logic
5. ภาษาที่อยู่ในรูปแบบ Declarative ต่างจาก Imperative อย่างไร?
6. ภาษาใดถือเป็นภาษาโปรแกรมระดับสูงภาษาแรก?
7. Compiler ต่างจาก interpreter อย่างไร?
8. Compiler และ interpreter มีข้อดี-ข้อเสีย อะไรบ้าง?
9. Java เป็น Compiler หรือ interpreter หรือทั้งคู่? แล้วเรารู้ได้อย่างไร?
10. Compiler ต่างจาก preprocessor อย่างไร?
11. ขั้นตอนการทำงานของ Compiler มีกี่ขั้นตอน แต่ละขั้นตอนทำหน้าที่อะไร

What's next?

- Syntax and Lexical Analysis
- Regular expression
- Context-free grammars