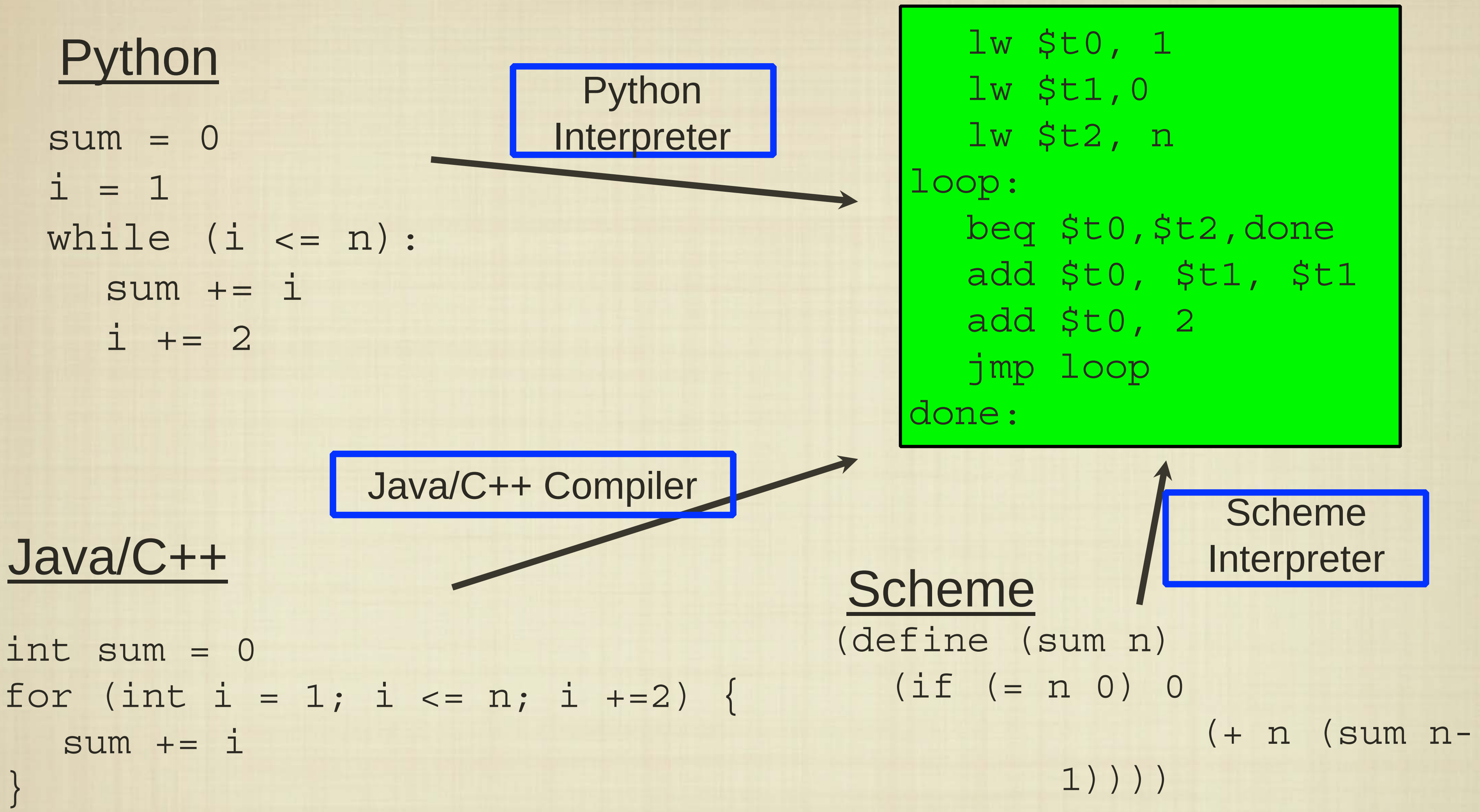
Parsing and Compilers

Spring 2014

Carola Wenk

Languages So Far



Python

```
sum = 0
i = 1
while (i <= n):
   sum += i
   i += 2
```

```
   lw $t0, 1
   lw $t1,0
   lw $t2, n
loop:
   beq $t0,$t2,done
   add $t0, $t1, $t1
   add $t0, 2
   jmp loop
done:
```

Java/C++

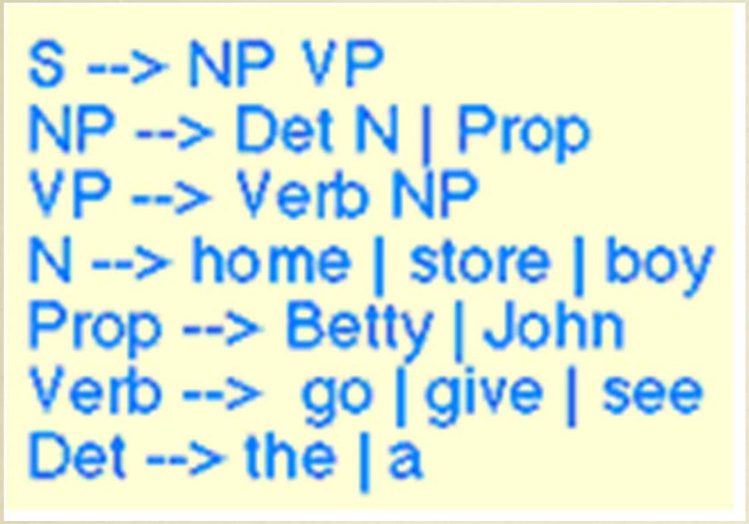
```
int sum = 0
for (int i = 1; i <= n; i +=2) {
   sum += i
}
```

Scheme

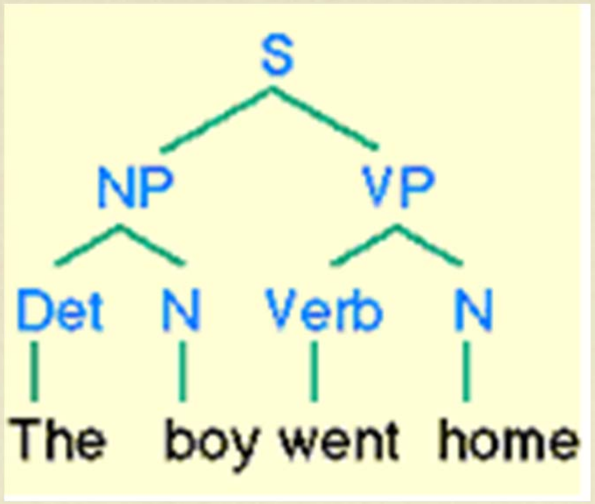
```
(define (sum n)
   (if (= n 0) 0
                (+ n (sum n-
         1))))
```

We’ve seen four languages, how do we actually turn a program into machine instructions?

Language Structure



```
S --> NP VP
NP --> Det N | Prop
VP --> Verb NP
N --> home | store | boy
Prop --> Betty | John
Verb -->  go | give | see
Det --> the | a
```



Every language has a <u>grammar</u>: the rules by which it is spoken and written.

When we hear or see a statement in English, we

1. break it into *tokens* and
2. *parse* the tokens into a structure that gives us the meaning.

Language Grammar

- Any programming language needs to have a “grammar”, so that we can logically transform a program into its corresponding machine instructions.

- What does such a grammar look like?

  Languages grammars are usually specified in Backus-Naur Normal Form (BNF).

- How do we check whether a program is grammatically correct?

- It’s a lot like English: we take a program and see if the grammar could have possibly generated it.

Python Java C C++ Scheme

Backus-Naur Form

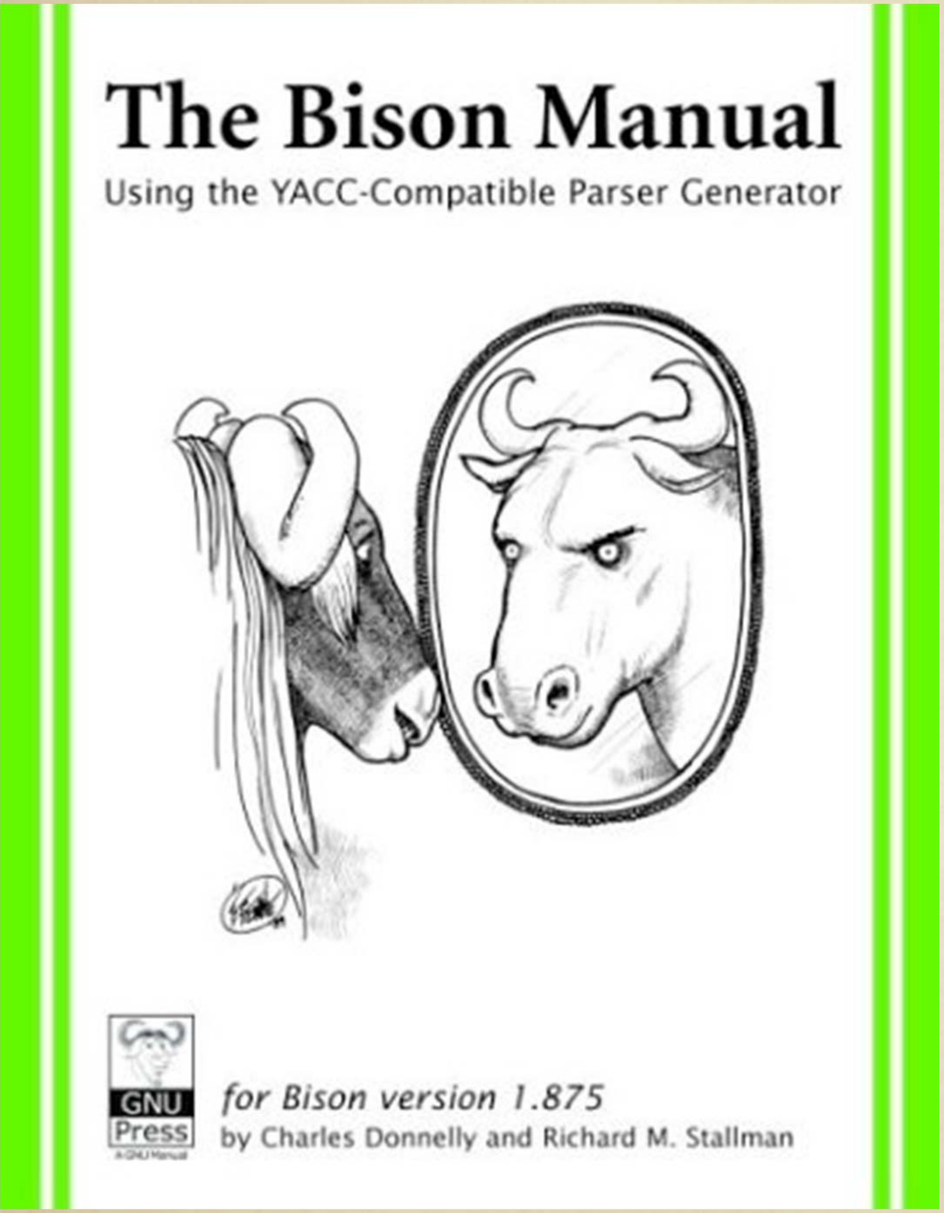
```
<postal-address> ::= <name-part> <street-address> <zip-part>

<name-part> ::= <personal-part> <last-name> <opt-suffix-part> <EOL>
                | <personal-part> <name-part>

<personal-part> ::= <first-name> | <initial> "."

<street-address> ::= <house-num> <street-name> <opt-apt-num> <EOL>

<zip-part> ::= <town-name> "," <state-code> <ZIP-code> <EOL>

<opt-suffix-part> ::= "Sr." | "Jr." | <roman-numeral> | ""

<opt-apt-num> ::= <apt-num> | ""
```

Backus-Naur Form is a set of rewrite rules that allows the compact specification of language rules.

To check if a particular sequence of characters matches a grammar, we need to establish whether that sequence could have been generated by the rules of the grammar.

[Wikipedia]

Parser Generators


The Bison Manual
Using the YACC-Compatible Parser Generator
GNU Press
for Bison version 1.875
by Charles Donnelly and Richard M. Stallman

So for each grammar, we need a parsing algorithm that can check whether any program is grammatically correct.

We won't get into this, but there are efficient algorithms for parsing.

Parsing algorithms actually don't care about the language, so most commonly "parser generators" take a grammar and output a parser (say in C).

It also turns out that we can use the parse to tell us how to generate machine instructions.

Generating Machine Instructions

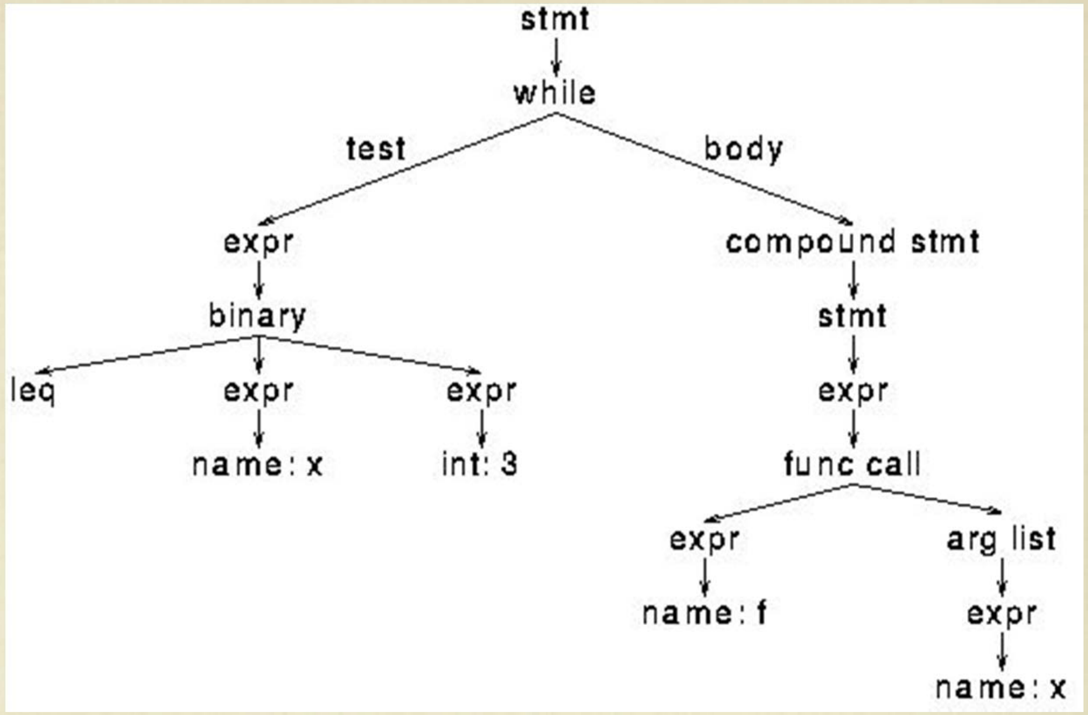
Python

```
while (x <= 3):
   f(x)
   x += 1
```

Machine Instructions

```
loop:
  code for “x <= 3”
jump_if_false done:
  code for “f(x)”
  code for “x += 1”
jump loop
done:
```

Parse Tree

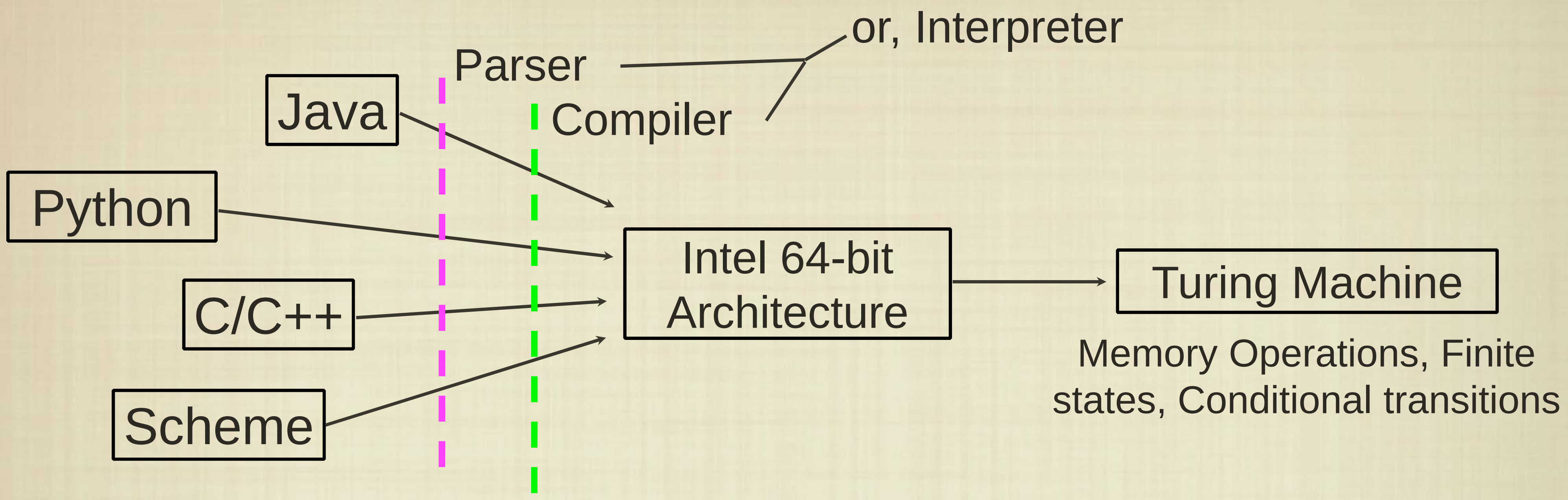


[Minka, Microsoft Research]

While checking the grammar, we can produce a parse tree, just as in English.

The general approach to translation is traverse the parse tree, using instruction templates for each node in the parse tree.

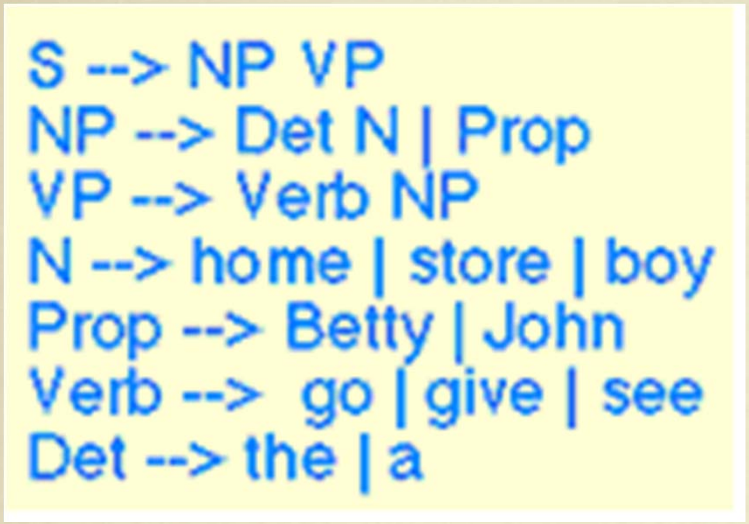
Different Languages


or, Interpreter
Parser
Compiler
Java
Python
C/C++
Scheme
Intel 64-bit Architecture
Turing Machine
Memory Operations, Finite states, Conditional transitions

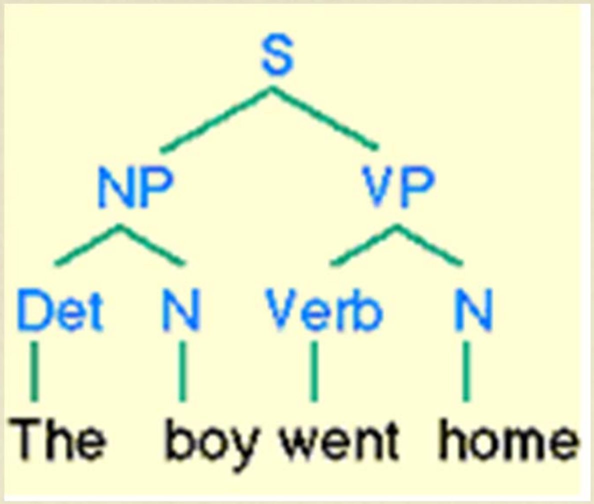
Any program written in a high-level language can be converted into machine instructions that are executed in a von Neumann architecture.

Every von Neumann machine implements a Turing machine.

Language Structure



```
S --> NP VP
NP --> Det N | Prop
VP --> Verb NP
N --> home | store | boy
Prop --> Betty | John
Verb -->  go | give | see
Det --> the | a
```



Every language has a <u>grammar</u>: the rules by which it is spoken and written.

When we hear or see a statement in English, we

Lex 1. break it into *tokens* and

Yacc *2. parse* the tokens into a structure that gives us the meaning.