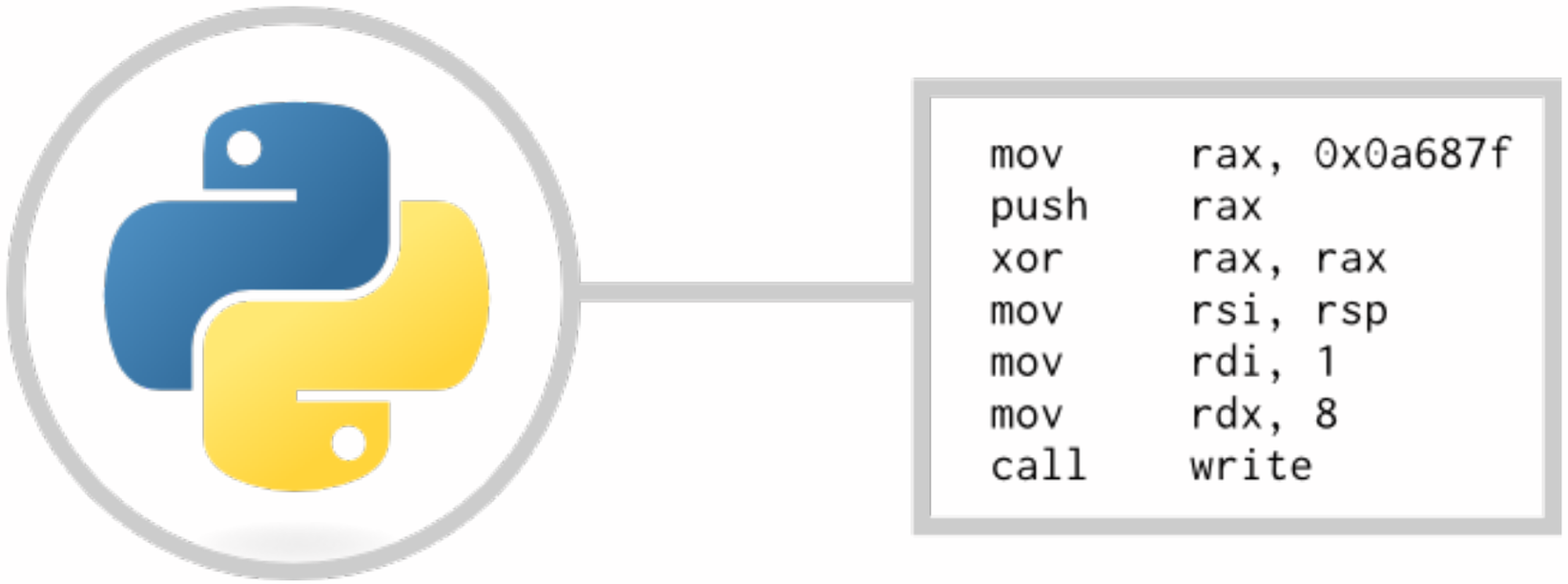
CS 480

Translators (Compilers)



Instructor: Liang Huang

(some slides courtesy of J. Siek, Indiana and Z. Su, UCDavis)

Comments from CS 321

"IMPORTANT: Homework solutions should have been released with homeworks... "

"9-11pm recitation [actually 8-9pm] is not acceptable ... sunday 4-6 office hour or recitation [before the final on Monday] is not acceptable ..."

"It is truly appalling to me that Dr. Huang was allowed to teach a course here at OSU... The fact that he is going to be teaching another course here is incredibly disappointing, as there are no other options for graduating seniors who need the course (Translators, CS480)... Dr. Huang should not be assigned to any more teaching positions here... The TAs for this course are another example of poor selection..."

"... One of my friends actually decided to add a year to his college, just to avoid taking the class [CS 480]."

Comments from CS 321

"Dr. Huang is very good at explaining the material, and is clear with his instructions..."

"Dr. Huang is clearly an expert in the material presented in class. From the time I have spent interacting with him and the TAs, I can see that they are all very excited about the material..."

"I felt like the course was taught decently well..."

"... I appreciated the interesting questions Dr. Huang asked on quizzes but felt frustrated that our time to attempt them was sometimes so short. I really appreciated the extra credit programming project. It really helped me comprehend the CKY algorithm."

Comments from previous schools

(UPenn, USC, CUNY)

"Thanks so much for your review session and practice problems! They are extremely helpful for the midterm. I just want to let you know that I appreciate it."

"I really enjoyed this class. It's demanding but turns out to be interesting."

"One of the best class that I have ever taken at the college level."

"His projects are hard if you do not understand the material. It is very hard to get an A, but he guarantees you that if you do get an A, you will get a job in a major company like Google. He is harsh at grading, but at the end he will take into consideration everyone and your final grade will be higher."

"Although he may come across as arrogant sometimes whenever he says things such as "this is trivial", he knows his subject matter well. He likes to give out tough homework assignments."

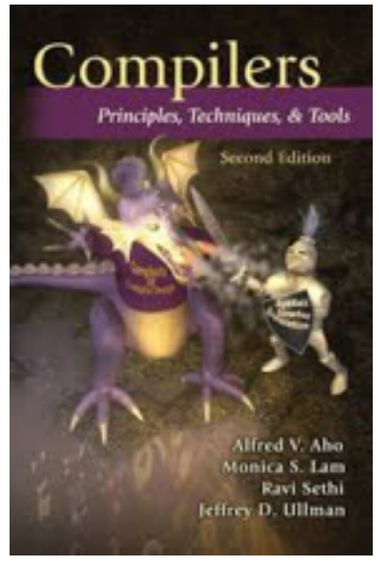
"One of the toughest course and is definitely worth your time... Not a class for slackers!"

"Professor Huang is great lecturer. His approach to push you to the limit. Even if you understand the half of what he is teaching and able to implement it, you will be prepared to the most difficult interviews in the top software developing companies. Should take this class only to see what are you really worth."

"One of the toughest courses in the department, but well worth your time. This course will help you understand, in great depth, many famous algorithms. On top of that, you will come out of the course a better programmer. This course is not for everyone, but what course is anyway?"

Admin

- Course Homepage (HWs, slides, schedule, etc.)
  - http://classes.engr.oregonstate.edu/eecs/winter2016/cs480/
- Optional Textbook: 2nd edi. Dragon
  - I'm against the high prices of textbooks!



- Canvas for Discussions, Grades, and Solutions
  - the first person to report each bug will be rewarded
  - technical questions should be asked on Canvas first
  - helping others will also be rewarded
- email us cs480-winter16-orst@googlegroups.com for other questions
  - in general, do *not* email us individually

Grades and Late Policy

- 5 programming HWs (40%)
  - I'll provide skeleton code for each HW
- 2 midterms (30%) -- around weeks 4 and 8
- 1 quiz (5%)
- Final project (20%) -- in groups of three
- class participation (5%)
- NO FINAL EXAM
- Late policy: you can submit only one HW late by 24 hours (with no penalty); other late HWs will not be graded.
  - HW1 is the easiest so save it for later HWs.

Grading Curves

Standard OSU (e.g., previous 480s)		**my courses *before* withdraw deadline**		**my courses *after* withdraw deadline**	
A	93 - 100	**A/A-**	25%	**A/A-**	30%
A-	90 - 92				
B+	87 - 89	**B+/B/B-**	30%	**B+/B/B-**	40%
B	83 - 86				
B-	80 - 82	**C+/C**	20%	**C+/C**	20%
C+	77 - 79				
C	73 - 76	**C-/D+**	10%	**C-/D+**	5%
C-	70 - 72				
D+	67 - 69	**F**	15%	**F**	5%
D	63 - 66				
D-	60 - 62				
F	< 60				

Programming Projects

School/Course	source	implementation	target
this course	Python subsets	Python	C
	Markdown supersets		LaTeX
Indiana, Colorado, Utah	Python subsets	Python	C
			Assembly
Stanford, Davis, NYU, ...	Java subsets/variants: Cool, Mini-Java, etc.	your choice	Assembly
Rutgers, Princeton, ...	OCaml subset	OCaml	Assembly
OSU CS 480 *previous years*	C subset/variant: IBTL	your choice	gforth

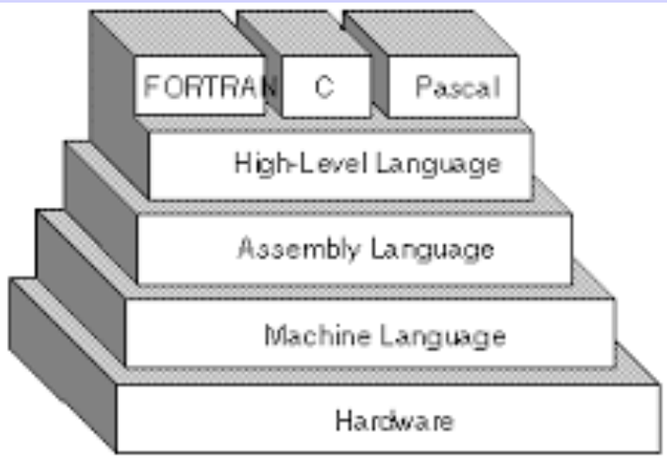
Why Study Compilers?

- it is the capstone course of the undergraduate CS curriculum
  - automata & formal language theory (CS 321)
  - programming language theory (CS 381)
  - data structures (CS 261) and algorithms (CS 325)

  - computer organization & assembly language (ECE 375)
  - operating systems (CS 344)
- it tells you how high-level languages *really* work on machines
- it can be used in other fields...
  - deterministic parsing => natural language parsing
  - syntax-directed translation => machine translation

Evolution of Programming Languages

> "I'm a terribly unscholarly person, and lazy. That was my motivating force in most of what I did, was how to avoid work."
>
> -- John Backus (1924, PA -- 2007, OR)



- programming in machine code or assembly was painful
- Fortran (1957) was the first compiler (Backus); *extremely ugly!*
- LISP (1958) was the first interpreter (John McCarthy)
- ALGOL (1958) (designed by Backus, Bauer, Perlis, ...) influenced most modern languages such as C, Java, Python
  - Backus-Nauer Form (BNF) as standard context-free grammar form
- later: bytecode interpretation (Python, Java, 1990s) and just-in-time compiling (trace monkey for javascript, pypy for python, 2000s)

What is a compiler?

```
$ cat a.c
#include<stdio.h>

int main() {
  int s = 0;
  for (int i = 1; i <= 5; i++)
    s += i;
}
```

```
$ gcc -std=c99 a.c
  # -std=c99 is for "int i" in the for-loop
  # not needed on Mac; better: clang or g++

$ objdump -d a.out # disassembler on Linux, or
$ otool -tV  a.out # on Mac (need XCode)
```

http://osxdaily.com/2014/02/12/install-command-line-tools-mac-os-x/

Linux

```
0000000000400474 <main>:
  400474:   55          push   %rbp
  400475:   48 ...      mov    %rsp, %rbp
  400478:   c7 ...      movl   $0x0, -0x8(%rbp)
  40047f:   c7 ...      movl   $0x1, -0x4(%rbp)
  400486:   eb ...      jmp    400492 <main+0x1e>
  400488:   8b ...      mov    -0x4(%rbp), %eax
  40048b:   01 ...      add    %eax, -0x8(%rbp)
  40048e:   83 ...      addl   $0x1, -0x4(%rbp)
  400492:   83 ...      cmpl   $0x5, -0x4(%rbp)
  400496:   7e ...      jle    400488 <main+0x14>
  400498:   b8 ...      mov    $0x0, %eax
  40049d:   c9          leaveq
  40049e:   c3          retq
  40049f:   90          nop
```

Linux x86_64

http://www.x86-64.org/documentation/assembly.html
http://www.cs.cmu.edu/~fp/courses/15213-s07/misc/asm64-handout.pdf

Mac

```
_main:
0100000f50   pushq    %rbp
0100000f51   movq     %rsp, %rbp
0100000f54   movl     $0x0, -0x4(%rbp)
0100000f5b   movl     $0x0, -0x8(%rbp)
0100000f62   movl     $0x1, -0xc(%rbp)
0100000f69   cmpl     $0x5, -0xc(%rbp)
0100000f70   jg       0x100000f91
0100000f76   movl     -0xc(%rbp), %eax
0100000f79   movl     -0x8(%rbp), %ecx
0100000f7c   addl     %eax, %ecx
0100000f7e   movl     %ecx, -0x8(%rbp)
0100000f81   movl     -0xc(%rbp), %eax
0100000f84   addl     $0x1, %eax
0100000f89   movl     %eax, -0xc(%rbp)
0100000f8c   jmp      0x100000f69
0100000f91   movl     -0x4(%rbp), %eax
0100000f94   popq     %rbp
0100000f95   retq
```

Mach-O x86_64

More Example of x86_64 Assembly

```
int fact_while(int x) {
    int result = 1;
    while (x > 0) {
       result *= x;
       x--; }
    return result;
}
```

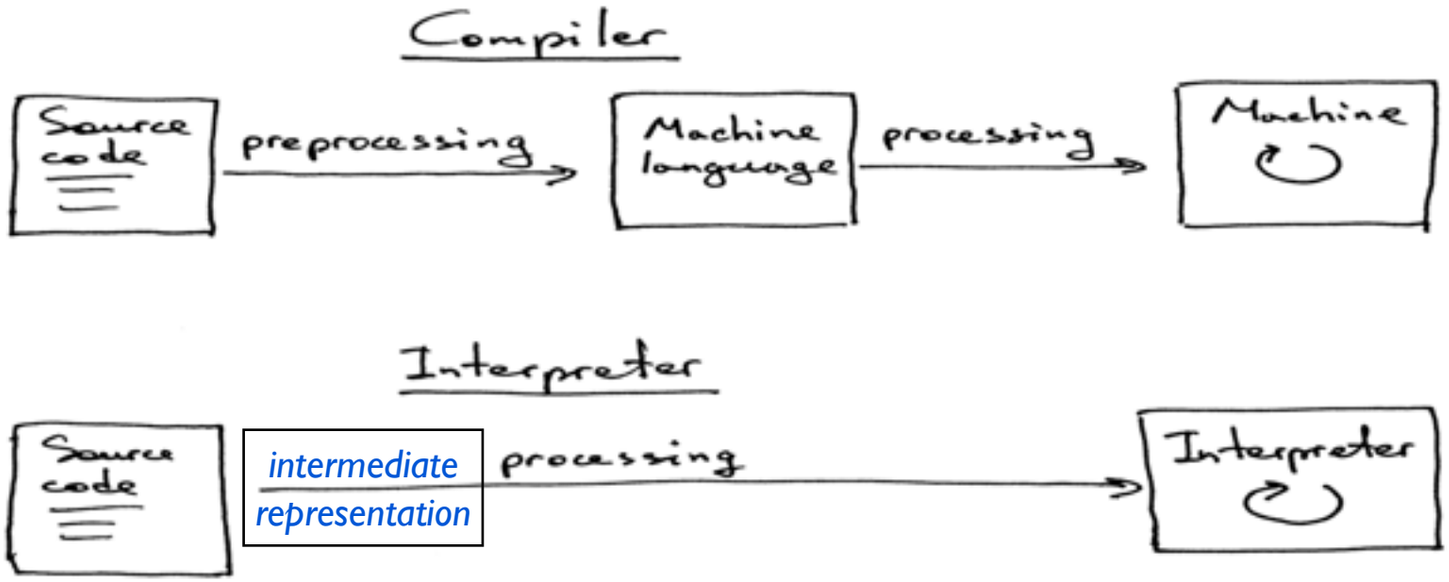
```
   x86-64 implementation of fact_while
   x in register %edi
1  fact_while:
2    movl    $1, %eax      result = 1
3    jmp     .L12          goto middle
4  .L13:                  loop:
5    imull   %edi, %eax      result *= x
6    decl    %edi            x--
7  .L12:                  middle:
8    testl   %edi, %edi      Test x
9    jg      .L13            if >0 goto loop
10   rep ; ret               else return
```

Instruction		Effect	Description
`leaq`	S, D	$D \leftarrow$ `&`S	Load effective address
`incq`	D	$D \leftarrow D$ `+ 1`	Increment
`decq`	D	$D \leftarrow D$ `- 1`	Decrement
`negq`	D	$D \leftarrow$ `-`D	Negate
`notq`	D	$D \leftarrow$ `~`D	Complement
`addq`	S, D	$D \leftarrow D$ `+` S	Add
`subq`	S, D	$D \leftarrow D$ `-` S	Subtract
`imulq`	S, D	$D \leftarrow D$ `*` S	Multiply

http://www.x86-64.org/documentation/assembly.html
http://www.cs.cmu.edu/~fp/courses/15213-s07/misc/asm64-handout.pdf

Compilation vs. Interpretation

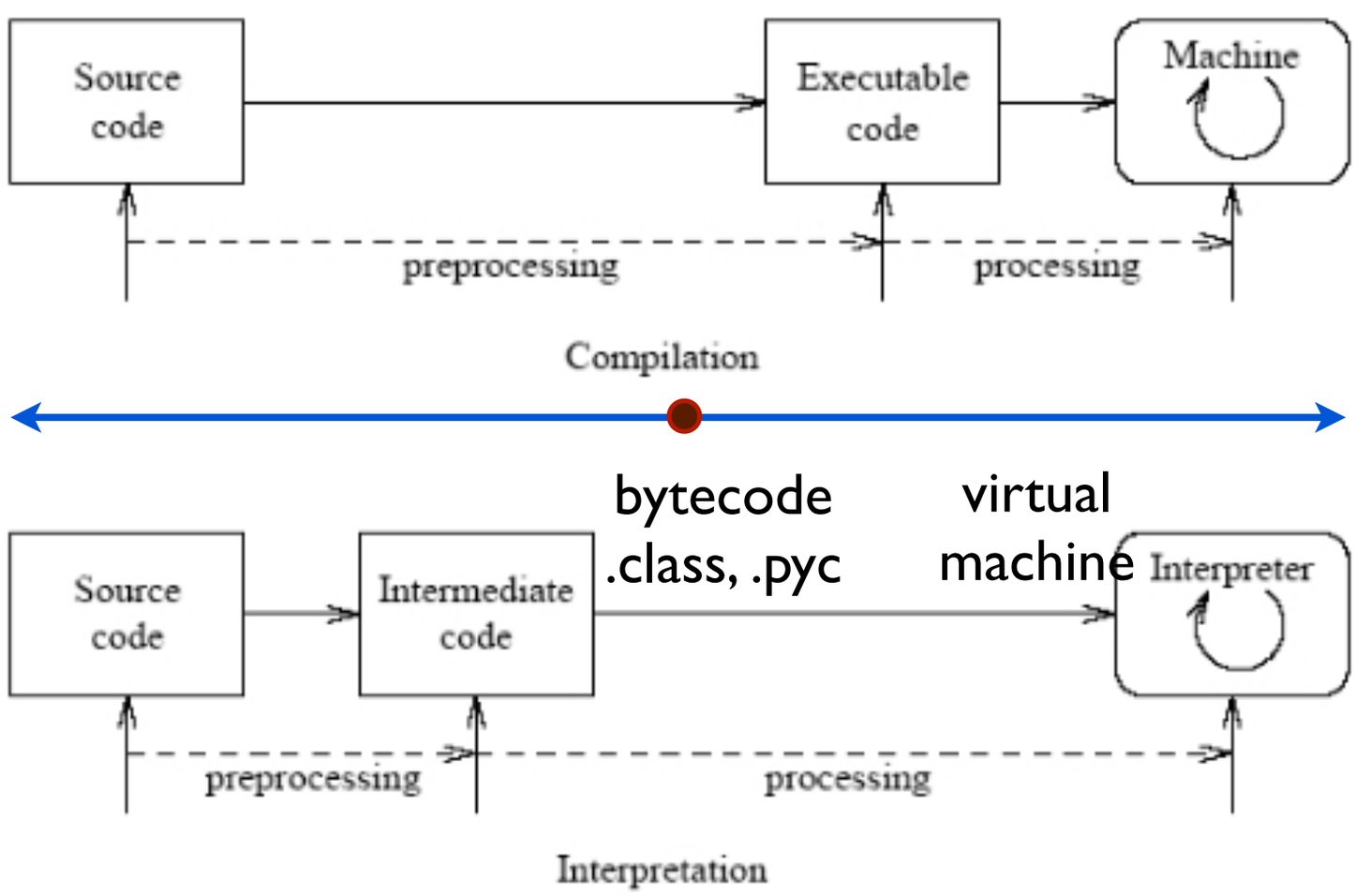
oversimplified view



C/C++, ML, ...

Lisp, BASIC

realistic and modern view

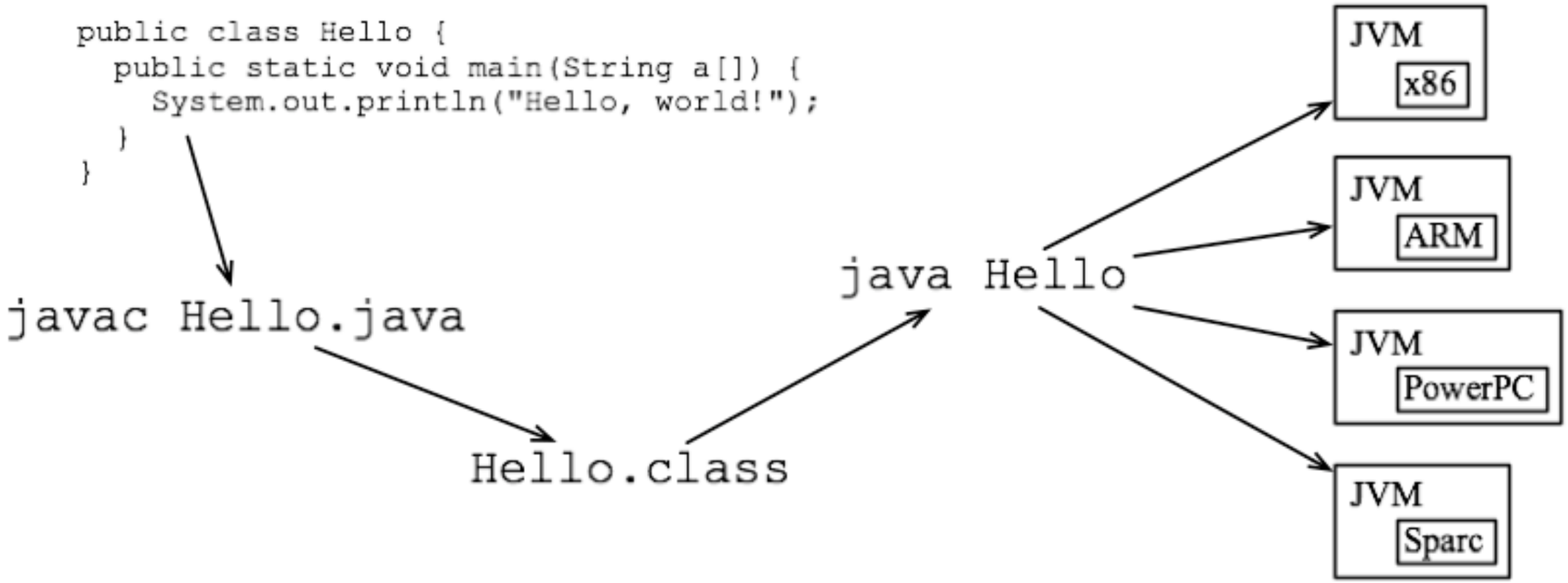


C/C++, ML, ...

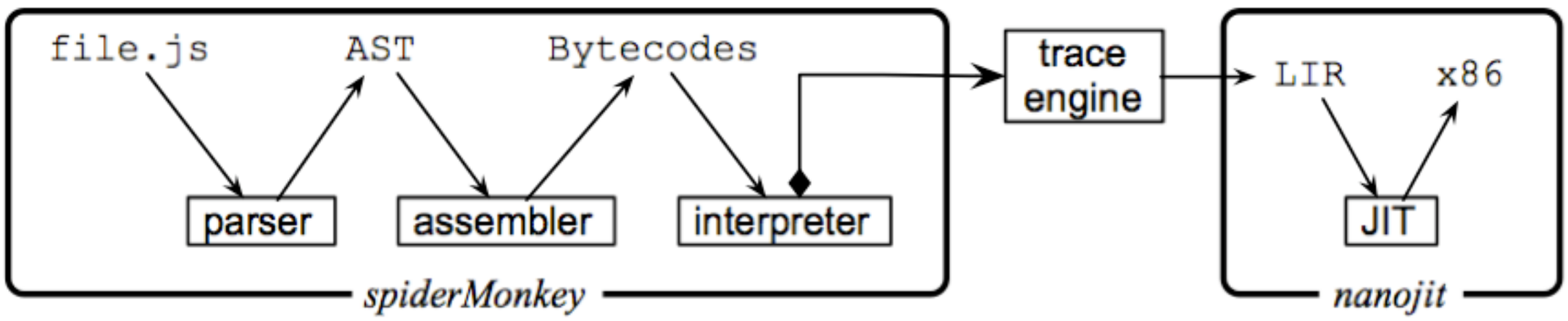
just-in-time compilation (e.g. pypy)

Java, Python, Ruby, Perl, Matlab

Bytecode (VM) vs. Just-in-time



trace monkey: JIT compiler for JavaScript in Firefox



Java Bytecode Example

```
outer:
for (int i = 2; i < 1000; i++) {
    for (int j = 2; j < i; j++) {
        if (i % j == 0)
            continue outer;
    }
    System.out.println (i);
}
```

```
$ javac Hello.java
$ javap -c Hello.class  # Java disassembler
```

```
0:   iconst_2
1:   istore_1
2:   iload_1
3:   sipush  1000
6:   if_icmpge       44
9:   iconst_2
10:  istore_2
11:  iload_2
12:  iload_1
13:  if_icmpge       31
16:  iload_1
17:  iload_2
18:  irem
19:  ifne    25
22:  goto    38
25:  iinc    2, 1
28:  goto    11
31:  getstatic       #84; // Field java/lang/System.out:Ljava/io/PrintStream;
34:  iload_1
35:  invokevirtual   #85; // Method java/io/PrintStream.println:(I)V
38:  iinc    1, 1
41:  goto    2
44:  return
```

```
Compiled from "Hello.java"
public class Hello {
  public Hello();
    Code:
       0: aload_0
       1: invokespecial #1    // Method java/lang/Object."<init>":(
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2    // Field java/lang/System.out:Ljava/i
       3: ldc           #3    // String Hello, world!
       5: invokevirtual #4    // Method java/io/PrintStream.println
       8: return
}
```

Python Bytecode Example

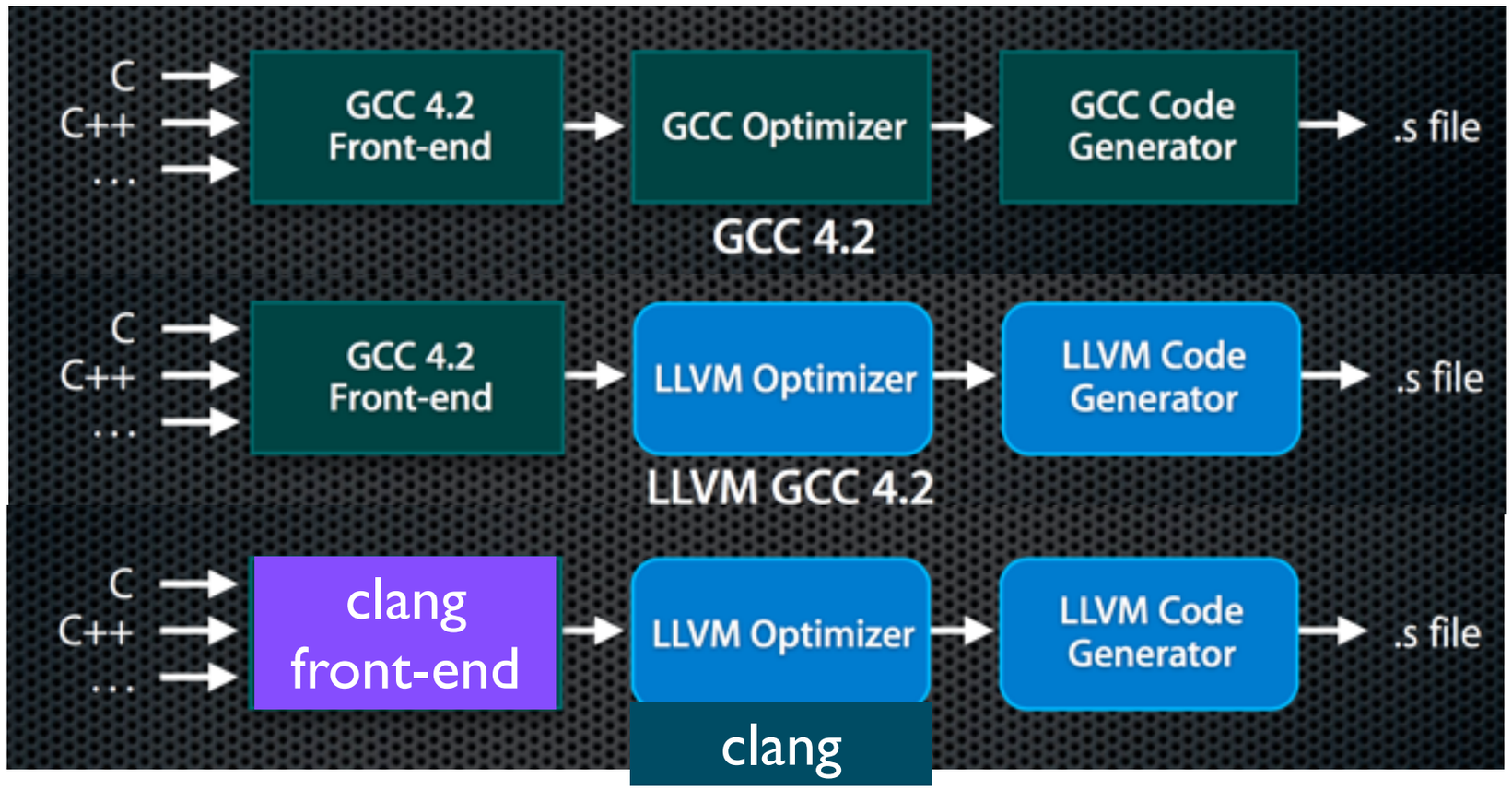
```
def myfunc(alist):
    return len(alist)
```

```
>>> import dis                  # Python disassembler
>>> dis.dis(myfunc)
  2           0 LOAD_GLOBAL              0 (len)
              3 LOAD_FAST                0 (alist)
              6 CALL_FUNCTION            1
              9 RETURN_VALUE
```

```
>>> bytecode = dis.Bytecode(myfunc)
>>> for instr in bytecode:
...     print(instr.opname)
...
LOAD_GLOBAL
LOAD_FAST
CALL_FUNCTION
RETURN_VALUE
```

LLVM and Clang

- LLVM ("low-level virtual machine") is a standardized intermediate representation (and related tools)
- llvm-gcc uses gcc front end and llvm backend
- clang is a new front end
  - much faster compiling than gcc; more informative error msgs



use clang/gcc for AST and LL

```
$ cat b.c
int main() {
  int y = 1;
  int x = y + 3;
}
```

```
$ gcc -cc1 -emit-llvm b.c
$ less b.ll
define i32 @main() #0 {
  %y = alloca i32, align 4
  %x = alloca i32, align 4
  store i32 1, i32* %y, align 4
  %1 = load i32* %y, align 4
  %2 = add nsw i32 %1, 3
  store i32 %2, i32* %x, align 4
  ret i32 0
}
```

```
$ gcc -cc1 -ast-dump b.c
`-FunctionDecl 0x1028cc8e0 <b.c:3:1, line:6:1> main 'int ()'
  `-CompoundStmt 0x1028ccb28 <line:3:12, line:6:1>
    |-DeclStmt 0x1028cca08 <line:4:3, col:12>
    | `-VarDecl 0x1028cc990 <col:3, col:11> y 'int'
    |   `-IntegerLiteral 0x1028cc9e8 <col:11> 'int' 1
    `-DeclStmt 0x1028ccb10 <line:5:3, col:16>
      `-VarDecl 0x1028cca30 <col:3, col:15> x 'int'
        `-BinaryOperator 0x1028ccae8 <col:11, col:15> 'int' '+'
          |-ImplicitCastExpr 0x1028ccad0 <col:11> 'int' <LValueToRValue>
          | `-DeclRefExpr 0x1028cca88 <col:11> 'int' lvalue Var 0x1028cc990 'y' 'int'
          `-IntegerLiteral 0x1028ccab0 <col:15> 'int' 3
```

use clang/gcc for Assembly

```
$ cat b.c
int main() {
  int y = 1;
  int x = y + 3;
}
```

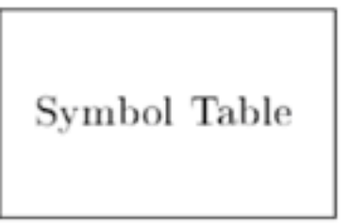
```
$ gcc -cc1 -emit-llvm b.c
$ less b.ll
define i32 @main() #0 {
  %y = alloca i32, align 4
  %x = alloca i32, align 4
  store i32 1, i32* %y, align 4
  %1 = load i32* %y, align 4
  %2 = add nsw i32 %1, 3
  store i32 %2, i32* %x, align 4
  ret i32 0
}
```

```
$ gcc -S -O3 b.c
$ cat b.s
    xorl    %eax, %eax
    popq    %rbp
    retq
```

```
$ gcc -cc1 -emit-llvm b.c
$ llc b.ll
$ cat b.s
_main:                  ## @main
## BB#0:
    movl    $1, -4(%rsp)
    movl    $4, -8(%rsp)
    xorl    %eax, %eax
    ret
```

```
$ gcc -S b.c
$ cat b.s
    movl    $0, %eax
    movl    $1, -4(%rbp)
    movl    -4(%rbp), %ecx
    addl    $3, %ecx
    movl    %ecx, -8(%rbp)
    popq    %rbp
    retq
```

Compiler Pipeline



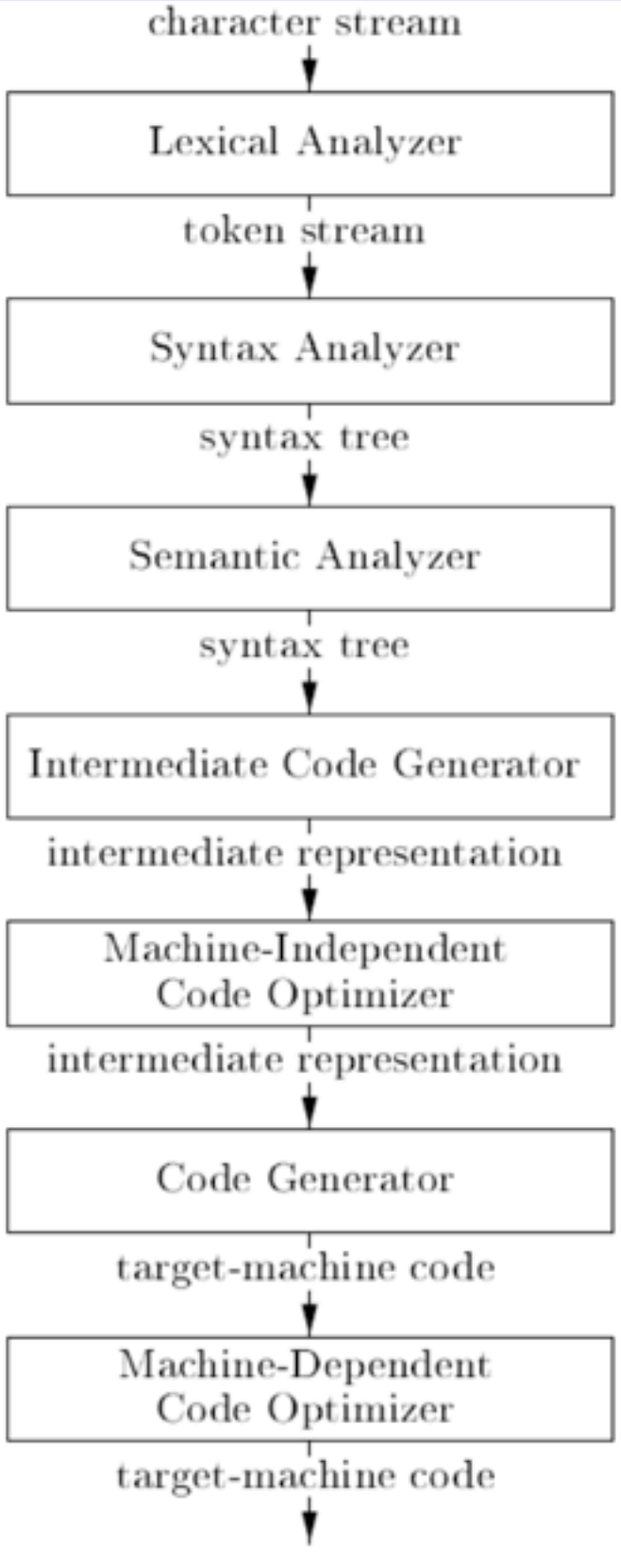
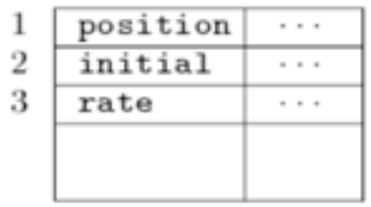


Figure 1.6: Phases of a compiler



SYMBOL TABLE

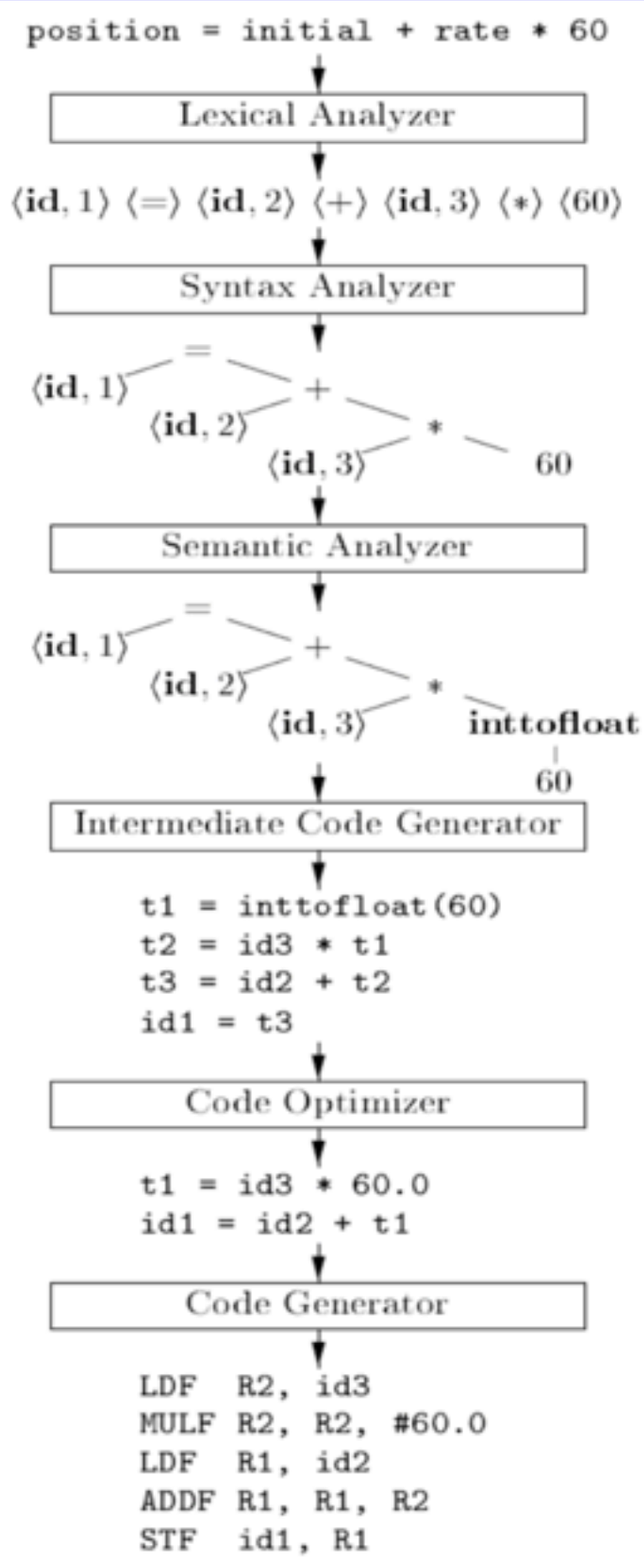
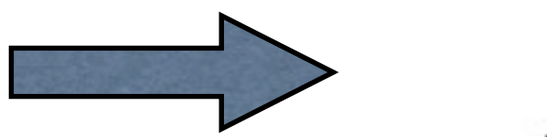


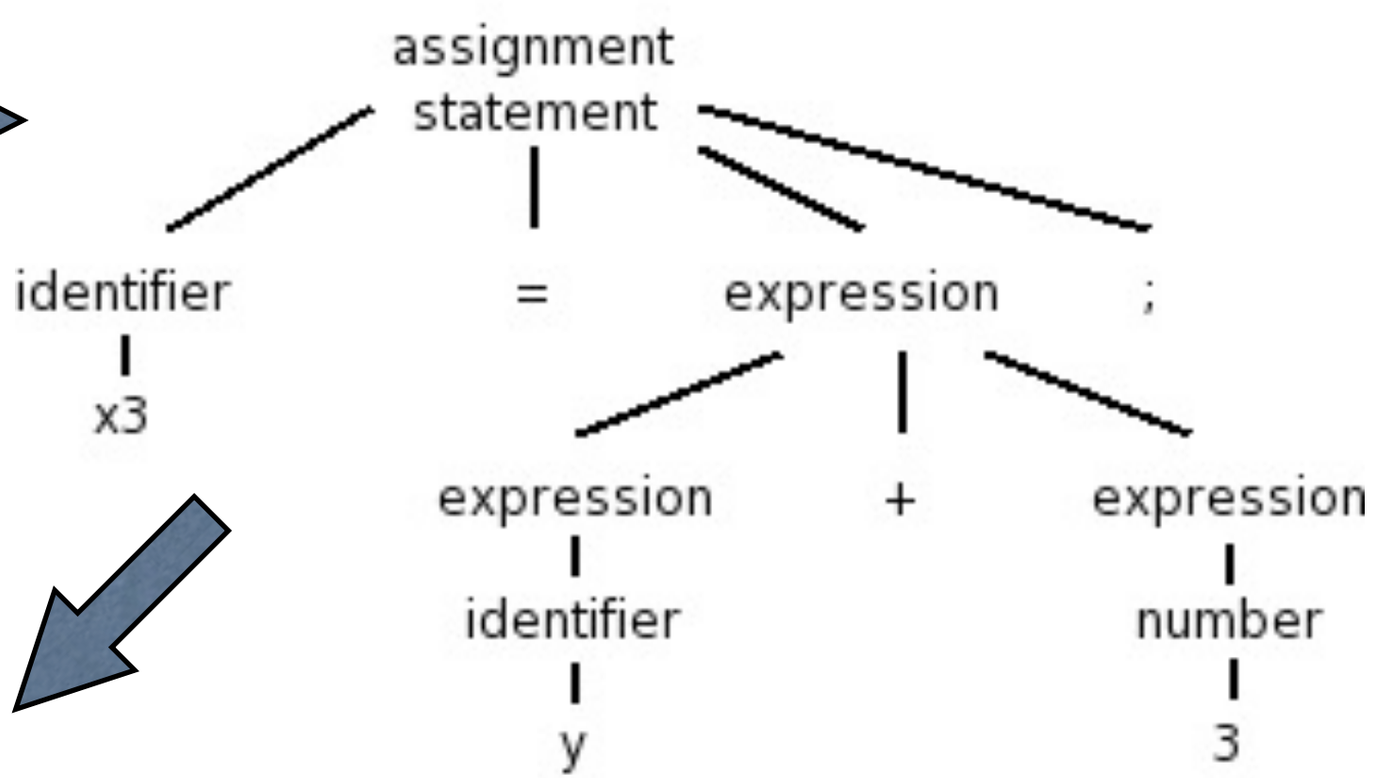
Figure 1.7: Translation of an assignment statement

Syntax-Directed Translation

1. parse high-level language program into a syntax tree
2. generate intermediate or machine code recursively

```
x3 = y + 3;
```





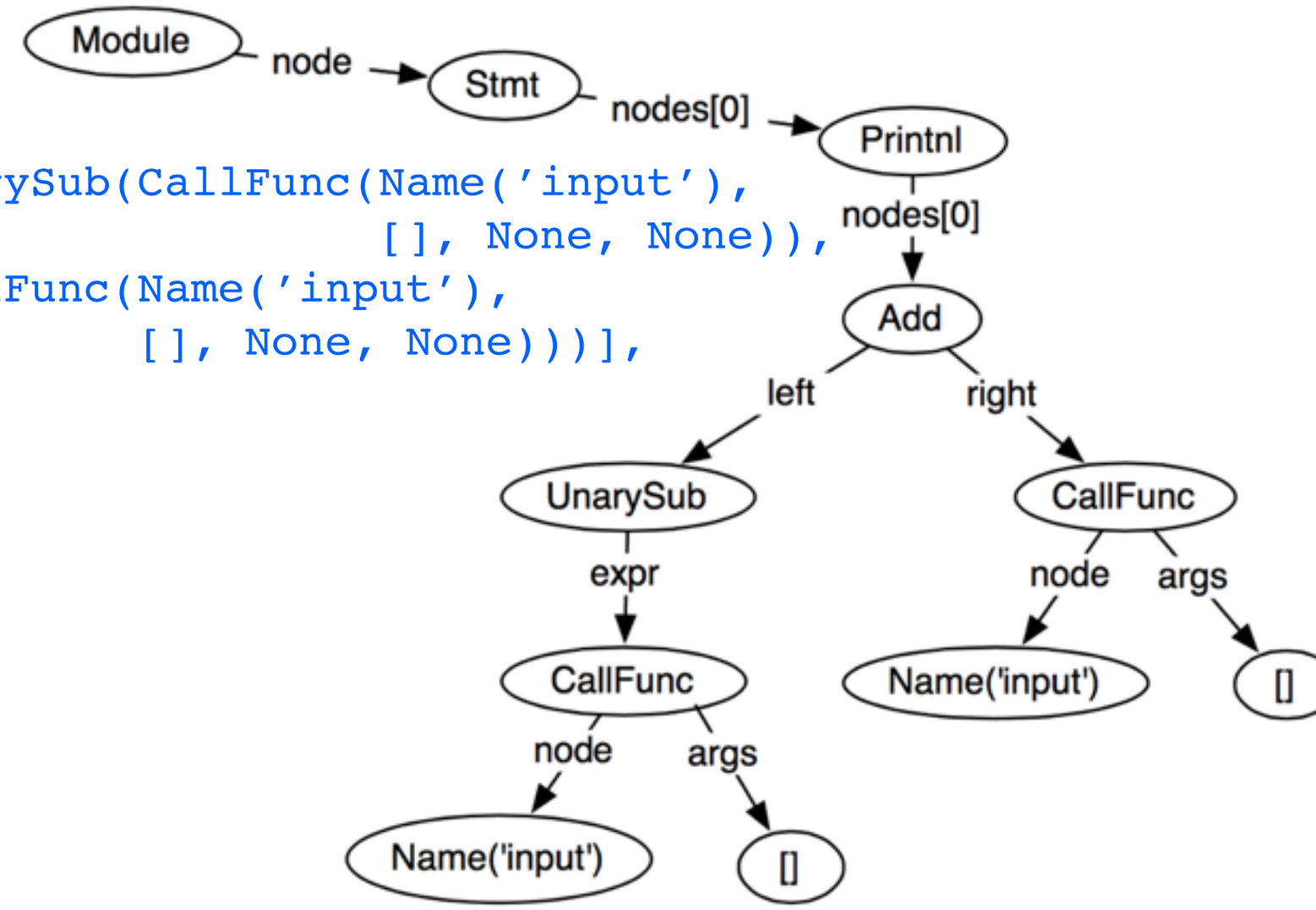

```
LD      R1,  id2
ADDF    R1,  R1, #3.0  // add float
RTOI    R2,  R1        // real to int
ST      id1, R2
```

Python Abstract Syntax Tree

```
print - input() + input()
```

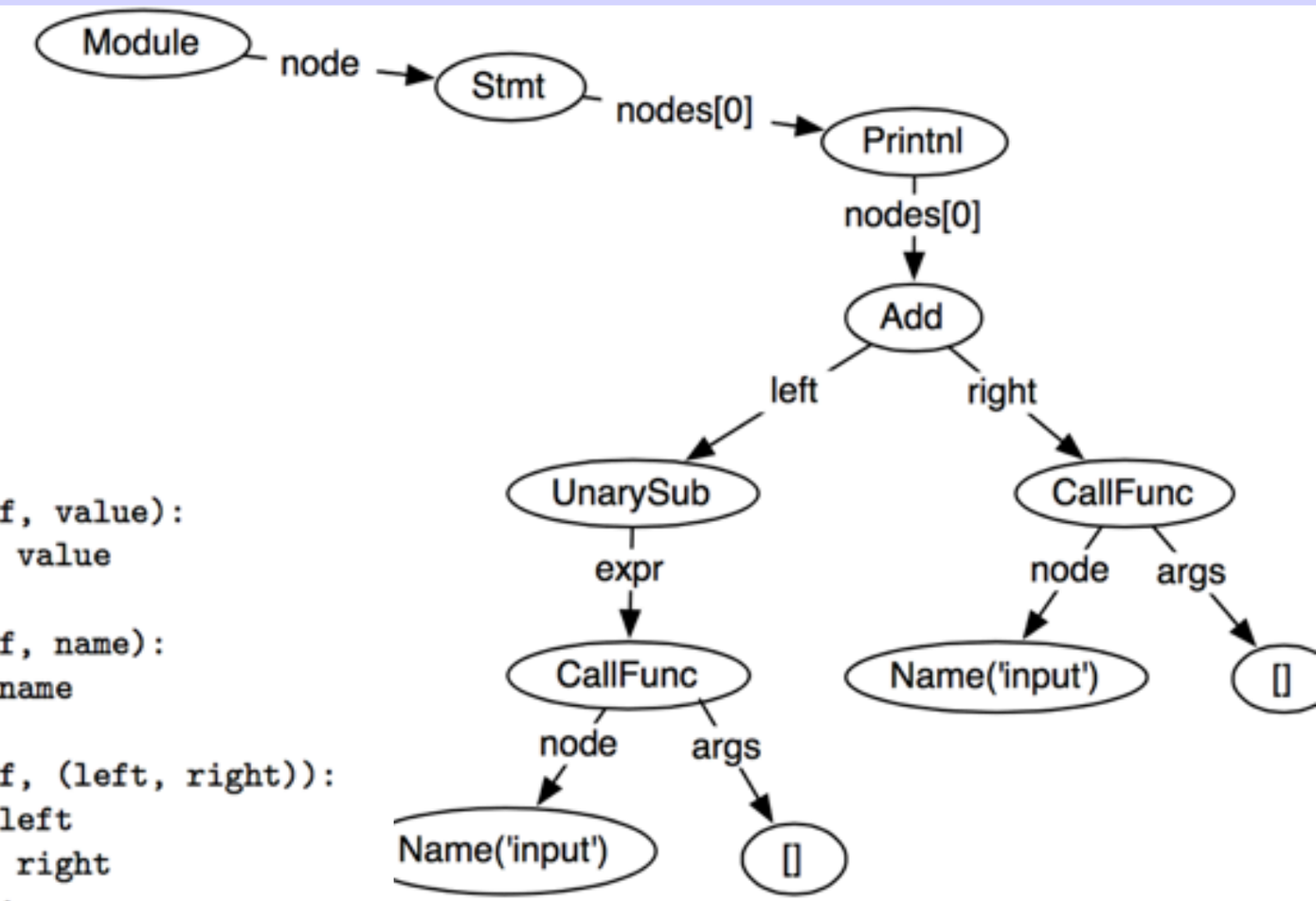
```
>>> import compiler
>>> ast = compiler.parse("print - input() + input()")
>>> ast
Module(None, Stmt([Printnl([Add((UnarySub(CallFunc(Name('input'), [], None,
None)), CallFunc(Name('input'), [], None, None)))], None)]))
```

```
Module(None,
       Stmt([Printnl([Add((UnarySub(CallFunc(Name('input'),
                                             [], None, None)),
                          CallFunc(Name('input'),
                                   [], None, None)))],
                     None)]))
```



Python classes for AST

```
print - input() + input()
```




```
class Module(Node):
    def __init__(self, doc, node):
        self.doc = doc
        self.node = node
class Stmt(Node):
    def __init__(self, nodes):
        self.nodes = nodes
class Printnl(Node):
    def __init__(self, nodes, dest):
        self.nodes = nodes
        self.dest = dest
class Assign(Node):
    def __init__(self, nodes, expr):
        self.nodes = nodes
        self.expr = expr
class AssName(Node):
    def __init__(self, name, flags):
        self.name = name
        self.flags = flags
class Discard(Node):
    def __init__(self, expr):
        self.expr = expr
```

```
class Const(Node):
    def __init__(self, value):
        self.value = value
class Name(Node):
    def __init__(self, name):
        self.name = name
class Add(Node):
    def __init__(self, (left, right)):
        self.left = left
        self.right = right
class UnarySub(Node):
    def __init__(self, expr):
        self.expr = expr
# CallFunc is for calling the 'input' function
class CallFunc(Node):
    def __init__(self, node, args):
        self.node = node
        self.args = args
```

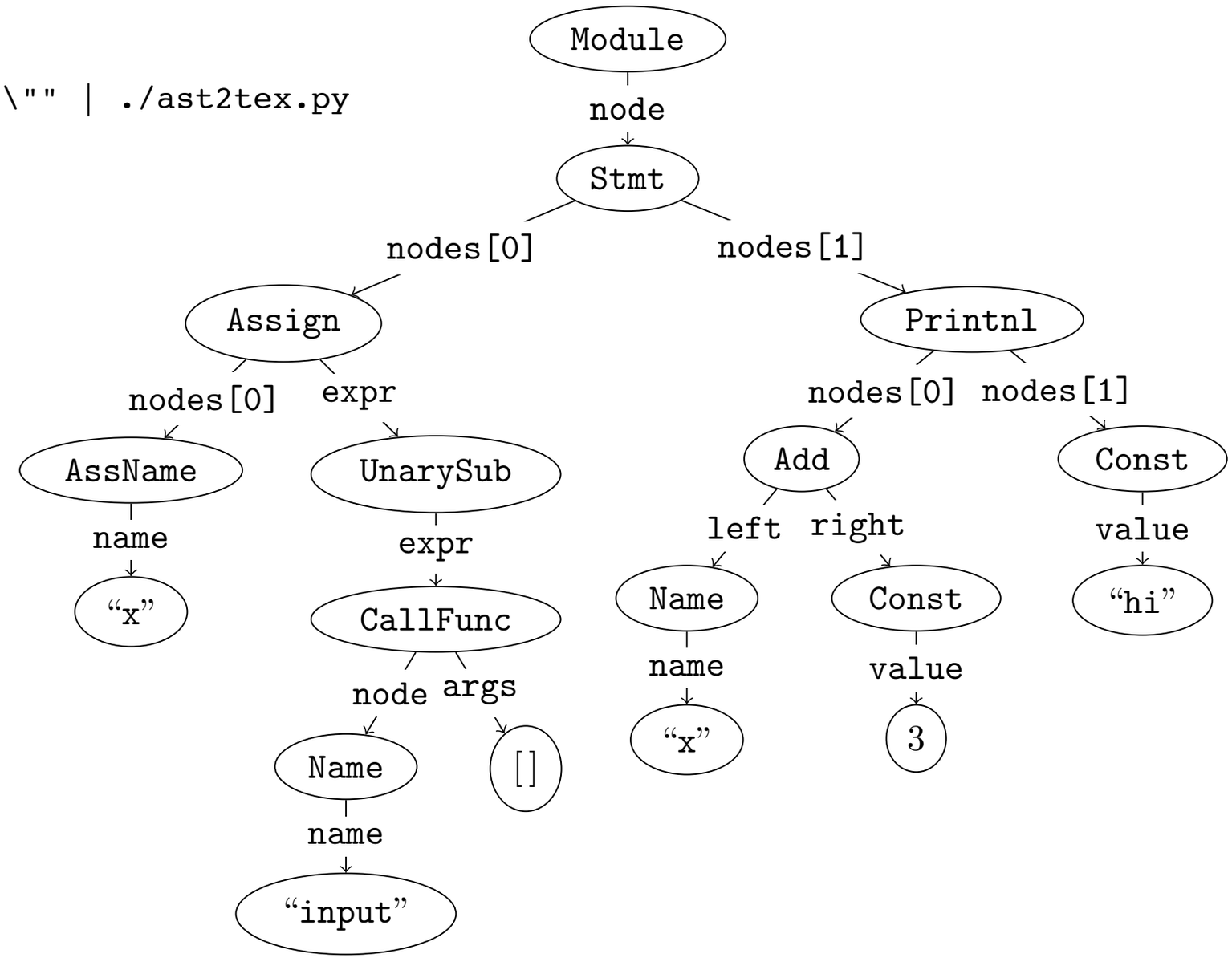
FIGURE 2. The Python classes for representing P_0 ASTs.

Multiple Statements

```
x = - input()
print x+3, "hi"
```

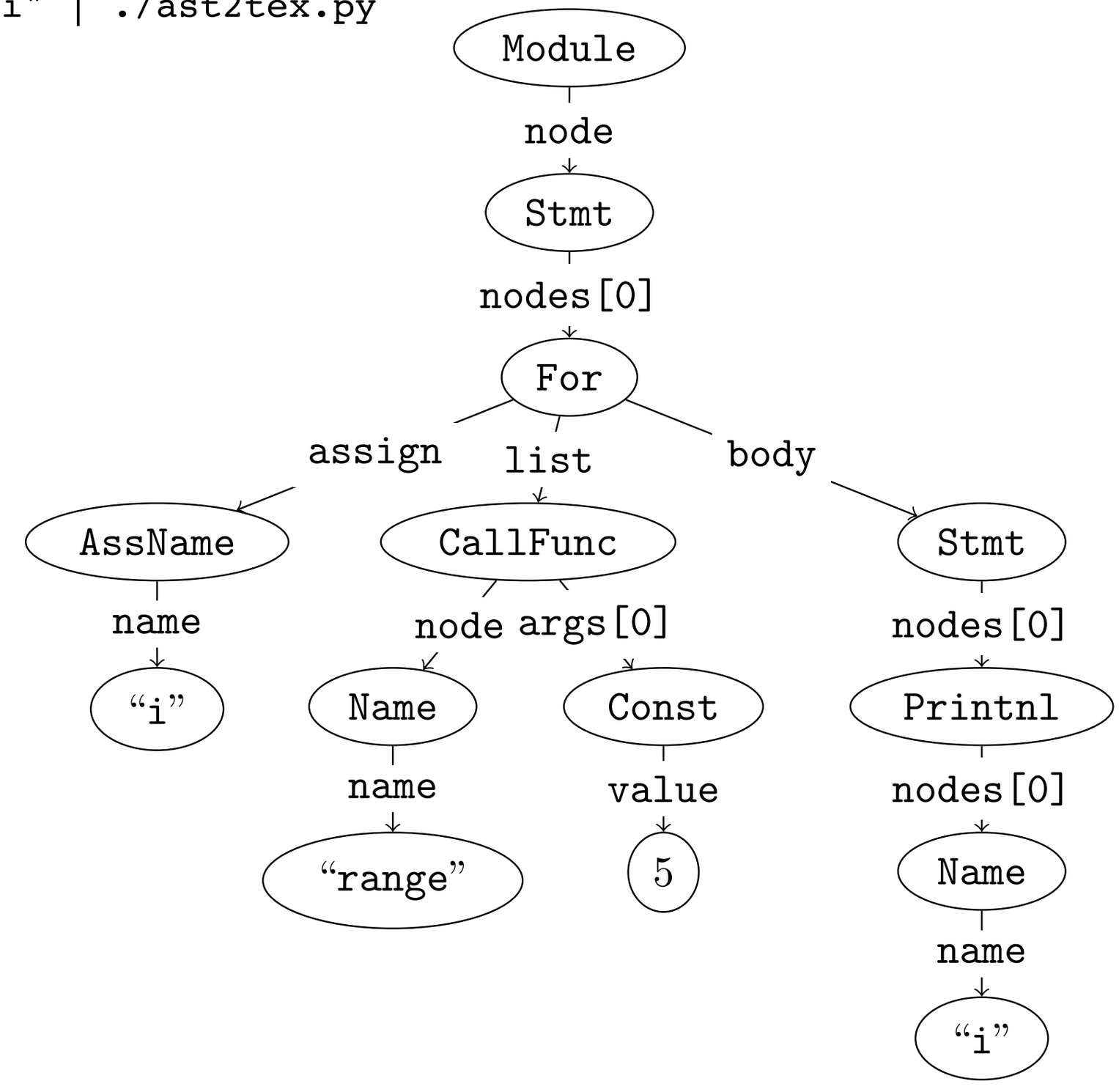
```
$ echo -e "x = -input()\nprint x+3, \"hi\"" | ./ast2tex.py
```

```
Module
| .node: Stmt
| | .nodes[0]: Assign
| | | .nodes[0]: AssName
| | | | .name: str x
| | | .expr: UnarySub
| | | | .expr: CallFunc
| | | | | .node: Name
| | | | | | .name: str "input"
| | | | | .args: list []
| | .nodes[1]: Printnl
| | | .nodes[0]: Add
| | | | .left: Name
| | | | | .name: str "x"
| | | | .right: Const
| | | | | .value: int 3
| | | .nodes[1]: Const
| | | | .value: str "hi"
```



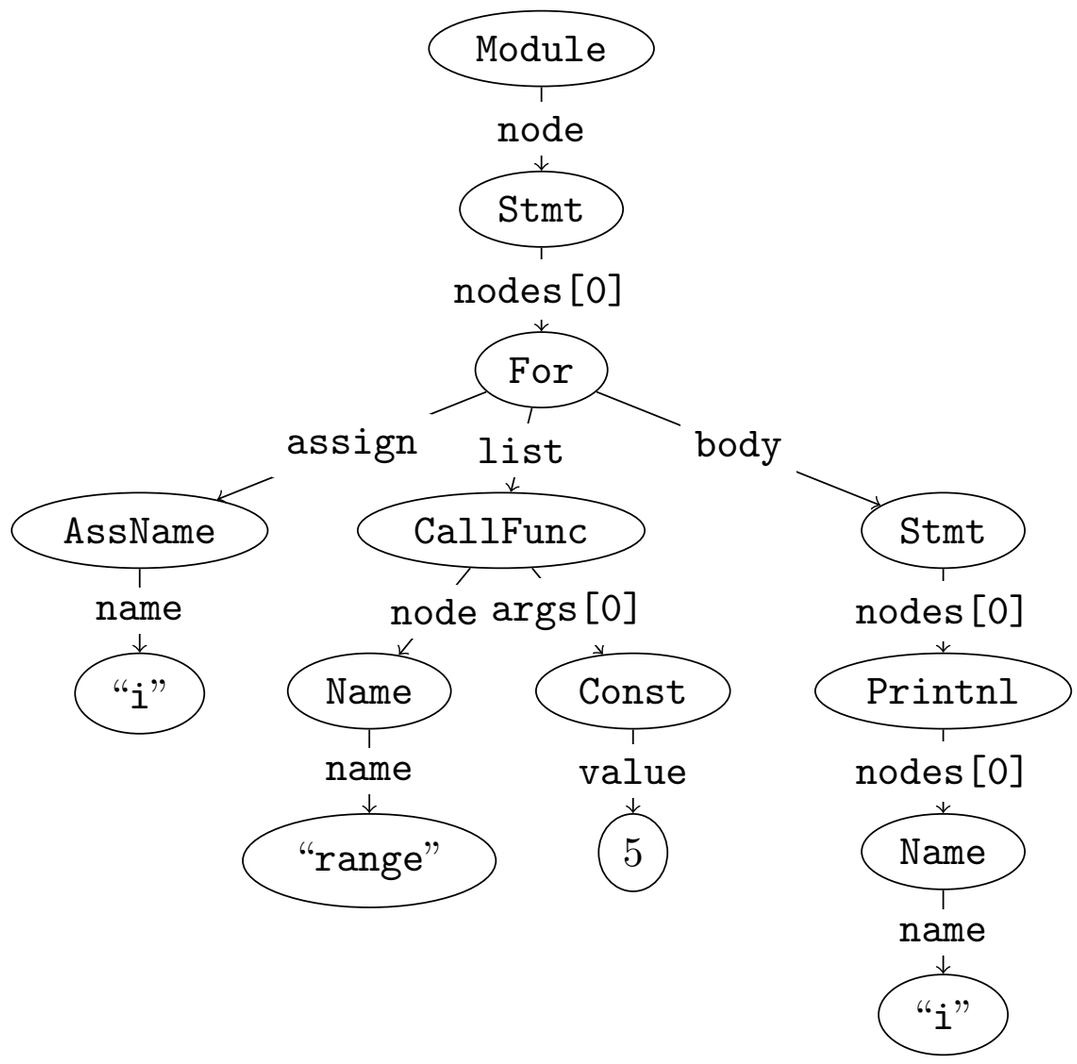
For Loops

```
$ echo "for i in range(5): print i" | ./ast2tex.py

Module
| .node: Stmt
| | .nodes[0]: For
| | | .assign: AssName
| | | | .name: str "i"
| | | .list: CallFunc
| | | | .node: Name
| | | | | .name: str "range"
| | | | .args[0]: Const
| | | | | .value: int 5
| | | .body: Stmt
| | | | .nodes[0]: Printnl
| | | | | .nodes[0]: Name
| | | | | | .name: str "i"
```



HW1 P_1 grammar

```
program : module
module : stmt+
stmt : (simple_stmt | for_stmt) NEWLINE
simple_stmt : "print" expr ("," expr)*
            | name "=" expr

for_stmt : "for" name "in" "range" "(" expr ")" ":" simple_stmt

expr : name
| decint
| "-" expr
| expr "+" expr
| "(" expr ")"
```



Actual Python Grammar

- INDENT makes Python NOT context-free
  - in C/C++/Java, if-then-else conflict is NOT context-free

```
file_input: (NEWLINE | stmt)* ENDMARKER

...

stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
small_stmt: (expr_stmt | print_stmt  | del_stmt | pass_stmt | flow_stmt |
             import_stmt | global_stmt | exec_stmt | assert_stmt)
expr_stmt: testlist (augassign (yield_expr|testlist) |
                     ('=' (yield_expr|testlist))*)
augassign: ('+=' | '-=' | '*=' | '/=' | '%=' | '&=' | '|=' | '^=' |
            '<<=' | '>>=' | '**=' | '//=')

print_stmt: 'print' ( [ test (',' test)* [','] ] |
                      '>>' test [ (',' test)+ [','] ] )

...

compound_stmt: if_stmt | while_stmt | for_stmt | try_stmt | with_stmt | funcdef | classdef | decorated
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
while_stmt: 'while' test ':' suite ['else' ':' suite]
for_stmt: 'for' exprlist 'in' testlist ':' suite ['else' ':' suite]
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT
...
```

https://docs.python.org/2/reference/grammar.html

SDT Example: Python to LISP

- example of syntax-directed translation from infix to prefix

```
def generate_c(n):
    if isinstance(n, Module):
        return generate_c(n.node) + "\n"
    elif isinstance(n, Stmt):
        return generate_c(n.nodes[0])
    elif isinstance(n, Printnl):
        return generate_c(n.nodes[0])
    elif isinstance(n, Const):
        return '%d' % n.value
    elif isinstance(n, UnarySub):
        return '(- %s)' % generate_c(n.expr)
    elif isinstance(n, Add):
        return '(+ %s %s)' % (generate_c(n.left), generate_c(n.right))
    elif isinstance(n, Mul):
        return '(* %s %s)' % (generate_c(n.left), generate_c(n.right))
    else:
        raise sys.exit('Error in generate_c: unrecognized AST node: %s' % n)
```

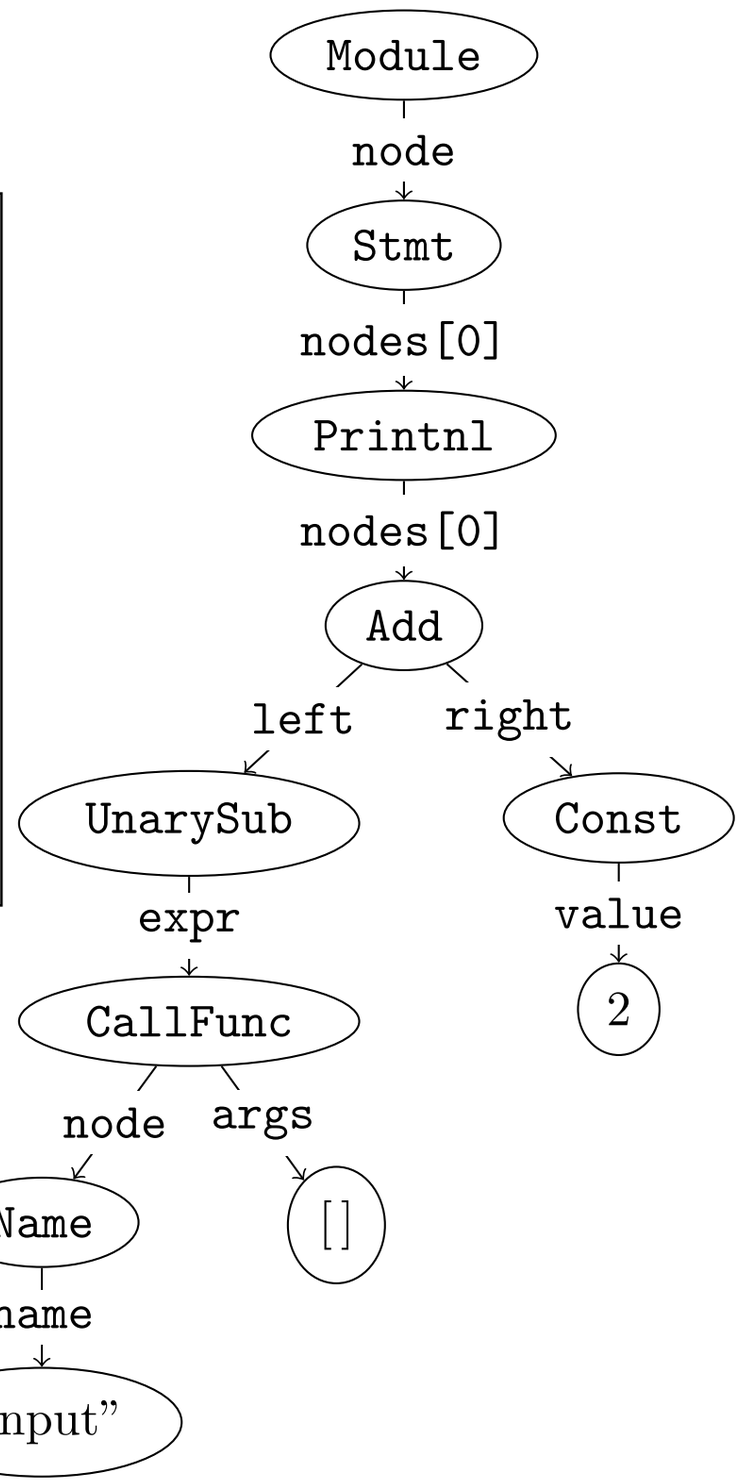
```
program : module
module : stmt
stmt : simple_stmt NEWLINE
simple_stmt : "print" expr

expr : decint
     | "-" expr
     | expr "+" expr
     | expr "*" expr
     | "(" expr ")"
```

```
$ echo "print (1 + -2) * 3" | python py2lisp.py
(* (+ 1 (- 2)) 3)
$ echo "(* (+ 1 (- 2)) 3)" | sbcl
-3
```

AST => Intermediate Code => Assembly

```
print - input() + 2
```

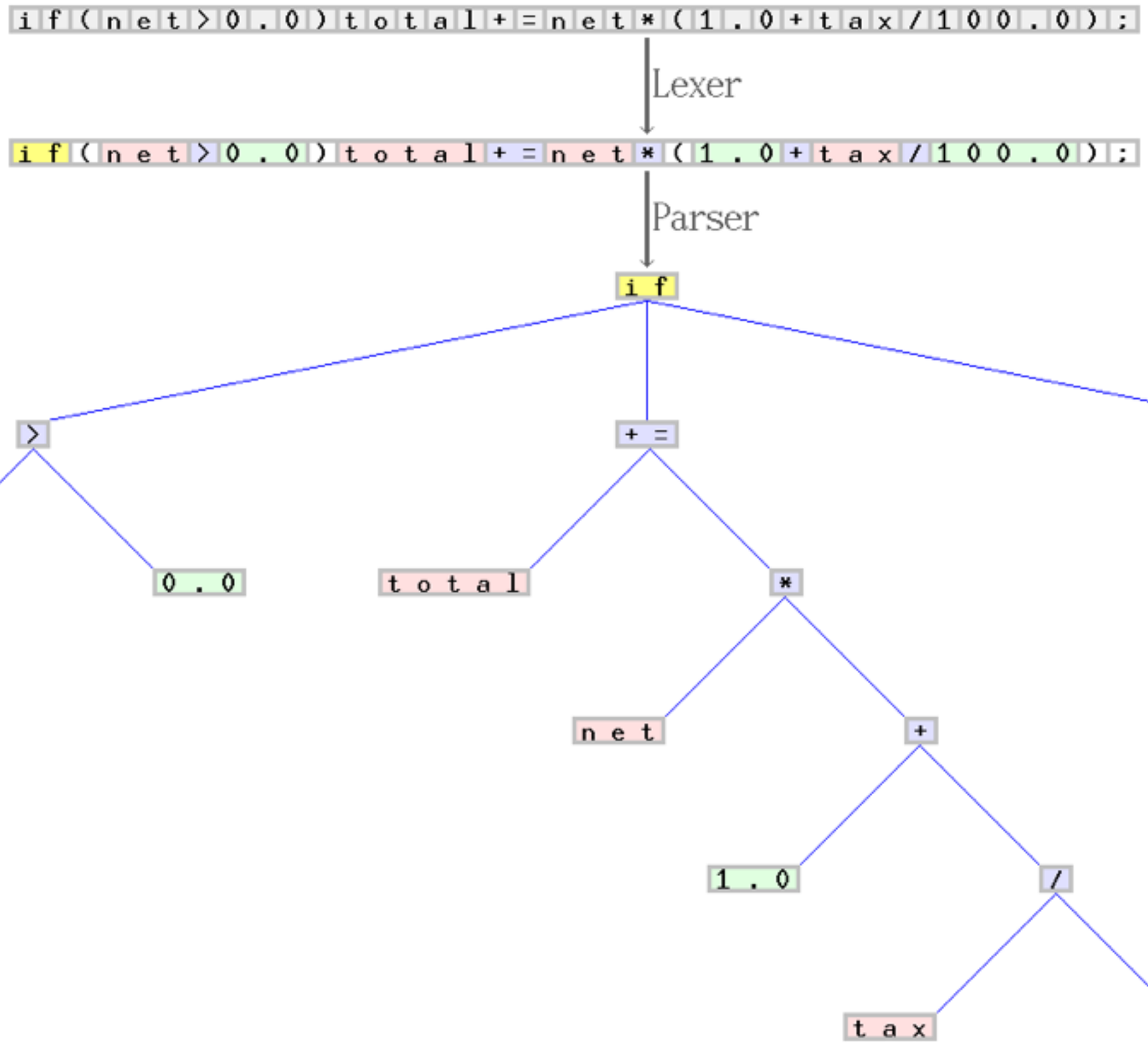
```
Module
| .node: Stmt
| | .nodes[0]: Printnl
| | | .nodes[0]: Add
| | | | .left: UnarySub
| | | | | .expr: CallFunc
| | | | | | .node: Name
| | | | | | | .name: str input
| | | | | | .args: list []
| | | | .right: Const
| | | | | .value: int 2
```




```
pushl %ebp
movl %esp, %ebp
subl $12,%esp
call input
movl %eax, -4(%ebp)
movl -4(%ebp), %eax
movl %eax, -8(%ebp)
negl -8(%ebp)
movl -8(%ebp), %eax,
movl %eax, -12(%ebp)
addl $2, -12(%ebp)
pushl -12(%ebp)
call print_int_nl
addl $4, %esp
movl $0, %eax
leave
ret
```

```
tmp0 = input()
tmp1 = - tmp0
tmp2 = tmp1 + 2
print tmp2
```

Lexer and Parser Example



hw2 (lex)

hw2 (yacc)
hw3 (LL)
hw4 (LR)

Example CFG in BNF: Postal Address

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

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

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

<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> | ""
```

- A postal address consists of a name-part, followed by a street-address part, followed by a zip-code part.
- A name-part consists of either: a personal-part followed by a last name followed by an optional suffix (Jr., Sr., or dynastic number) and end-of-line, or a personal part followed by a name part (this rule illustrates the use of recursion in BNFs, covering the case of people who use multiple first and middle names and/or initials).
- A personal-part consists of either a first name or an initial followed by a dot.
- A street address consists of a house number, followed by a street name, followed by an optional apartment specifier, followed by an end-of-line.
- A zip-part consists of a town-name, followed by a comma, followed by a state code, followed by a ZIP-code followed by an end-of-line.
- A opt-suffix-part consists of a suffix, such as "Sr.", "Jr." or a roman-numeral, or an empty string (i.e. nothing).
- A opt-apt-num consists of an apartment number or an empty string (i.e. nothing).

Note that many things (such as the format of a first-name, apartment specifier, ZIP-code, and Roman numeral) are left unspecified here. If necessary, they may be described using additional BNF rules.