

Write a Compiler (in Python)

David Beazley
<http://www.dabeaz.com>

November 2018

Introduction

Materials

- Download and extract the following zip file
<http://www.dabeaz.com/python/compilers.zip>
- Contains exercises and project descriptions
- Software requirements:
 - Python 3.7 (Anaconda recommended)
 - llvmlite and clang
 - SLY

Deep Thought

Programming

```
int fact(int n) {  
    int r = 1;  
    while (n > 0) {  
        r *= n;  
        n--;  
    }  
    return r;  
}
```

"Metal"



How does it all work????

Metal

Machine Code

```
5548 89e5 897d fcc7 45f8 0100 0000 837d  
fc00 0f8e 1800 0000 8b45 fc0f af45 f889  
45f8 8b45 fc83 c0ff 8945 fce9 deff ffff  
8b45 f85d c300 0000 0000 0000 0000 0000  
3500 0000 0000 0001 0000 0000 0000 0000  
0000 0000 0000 0000 1400 0000 0000 0000  
017a 5200 0178 1001 100c 0708 9001 0000  
2400 0000 1c00 0000 88ff ffff ffff ffff  
3500 0000 0000 0000 0041 0e10 8602 430d  
0600 0000 0000 0000 0000 0000 0100 0006  
0100 0000 0f01 0000 0000 0000 0000 0000
```

"Metal"



CPU is low-level (a glorified calculator)

Assembly Code

fact:

```
pushq  %rbp  
movq  %rsp, %rbp  
movl  %edi, -4(%rbp)  
movl  $1, -8(%rbp)
```

L1:

```
cmpl  $0, -4(%rbp)  
jle   L2  
movl  -4(%rbp), %eax  
imull -8(%rbp), %eax  
movl  %eax, -8(%rbp)  
mov   -4(%rbp), %eax  
addl  $-1, %eax  
movl  %eax, -4(%rbp)  
jmp   L1
```

L2:

```
movl  -8(%rbp), %eax  
popq  %rbp  
retq
```

Machine Code



```
5548 89e5 897d fcc7 45f8 0100 0  
fc00 0f8e 1800 0000 8b45 fc0f a  
45f8 8b45 fc83 c0ff 8945 fce9 d  
8b45 f85d c300 0000 0000 0000 0  
3500 0000 0000 0001 0000 0000 0  
0000 0000 0000 0000 1400 0000 0  
017a 5200 0178 1001 100c 0708 9  
2400 0000 1c00 0000 88ff ffff f  
3500 0000 0000 0000 0041 0e10 8  
0600 0000 0000 0000 0000 0000 0  
0100 0000 0f01 0000 0000 0000 0
```

"Human" readable
machine code

High Level Programming

Source Code

```
int fact(int n) {  
    int r = 1;  
    while (n > 0) {  
        r *= n;  
        n--;  
    }  
    return r;  
}
```

"Human understandable"
programming



```
fact:  
    pushq  %rbp  
    movq   %rsp, %rbp  
    movl   %edi, -4(%rbp)  
    movl   $1, -8(%rbp)  
  
L1:  
    cmpl   $0, -4(%rbp)  
    jle    L2  
    movl   -4(%rbp), %eax  
    imull  -8(%rbp), %eax  
    movl   %eax, -8(%rbp)  
    mov    -4(%rbp), %eax  
    addl   $-1, %eax  
    movl   %eax, -4(%rbp)  
    jmp    L1  
  
L2:  
    movl   -8(%rbp), %eax  
    popq   %rbp  
    retq
```

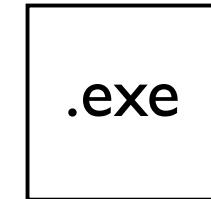
Compilers

Source Code

```
int fact(int n) {  
    int r = 1;  
    while (n > 0) {  
        r *= n;  
        n--;  
    }  
    return r;  
}
```

Executable

compiler



run

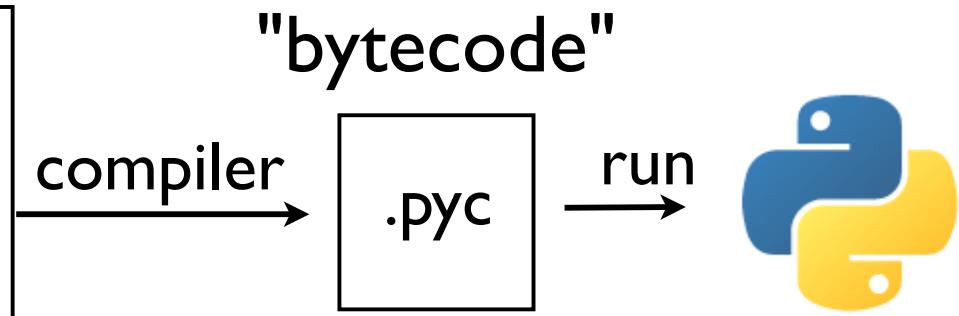


Compilers: Focused on program translation

Virtual Machines

Source Code

```
def fact(n):  
    r = 1  
    while n > 0:  
        r *= n  
        n--  
    return r;
```



Many languages run virtual machines that work like high level CPUs (Python, Java, etc.)

Transpilers

Source Code

```
int fact(int n) {  
    int r = 1;  
    while (n > 0) {  
        r *= n;  
        n--;  
    }  
    return r;  
}
```

translate →

Source Code

```
def fact(n):  
    r = 1  
    while n > 0:  
        r *= n  
        n = n - 1  
    return r
```

- Translation to a different language
- Example: Compilation to Javascript, C, etc.

Background

- Compilers are one of the most studied topics in computer science
- Huge amount of mathematical theory
- Interesting algorithms
- Programming language design/semantics
- The nature of computation itself

Compiler Building

- It's one of the most complicated programming projects you will ever undertake
- Many layers of abstraction and tooling
- Involves just about every topic in computer science (algorithms, hardware, etc.)
- Difficult software design (lots of parts)

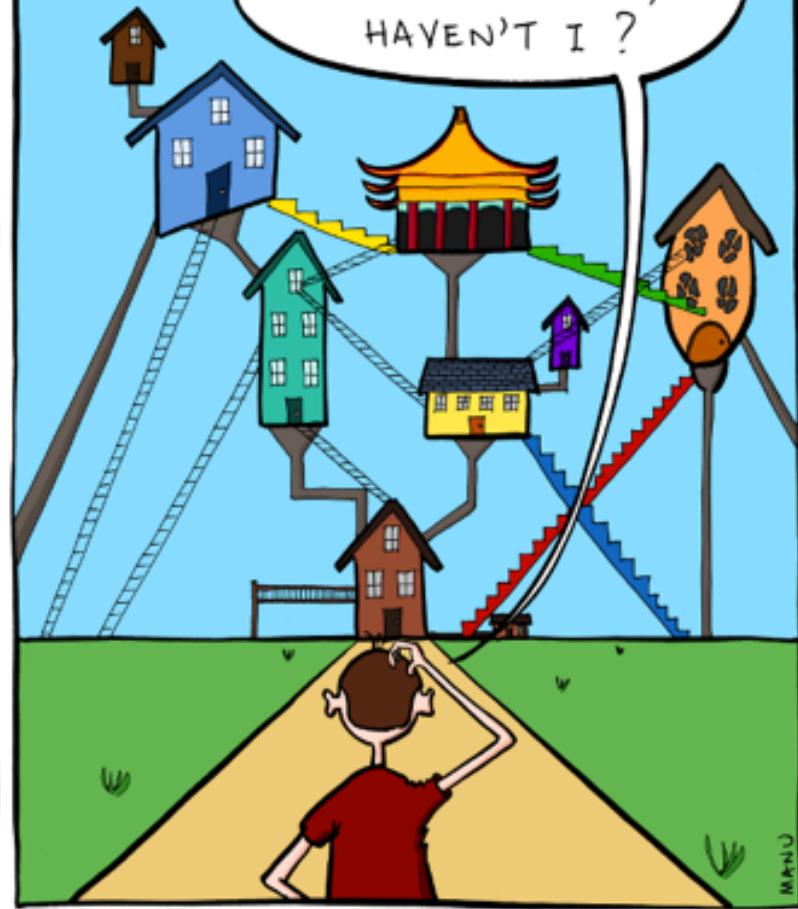
THE LIFE OF A SOFTWARE ENGINEER.

CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.



MUCH LATER...

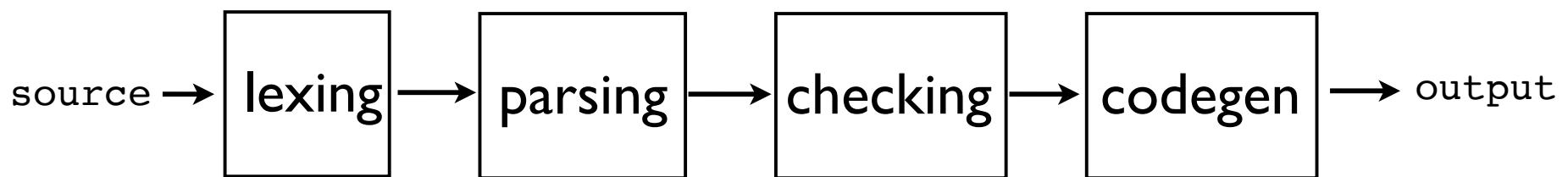
OH MY. I'VE DONE IT AGAIN,
HAVEN'T I ?



<http://www.bonkersworld.net>

Behind the Scenes

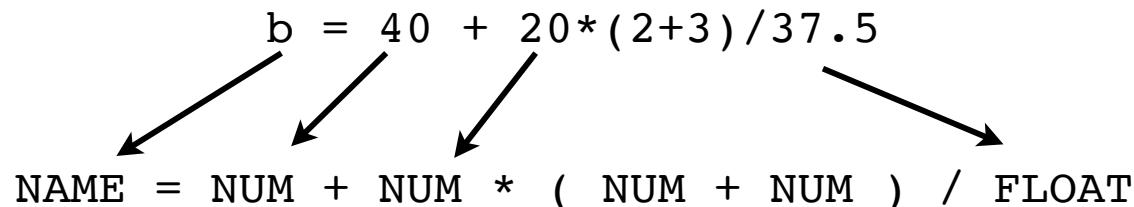
- Compilers consist of many "stages"



- Each, responsible for a different aspect.
- Mental model: processing pipeline (or workflow)

Lexing

- Splits input text into tokens



- Detects illegal symbols

The diagram shows a fragment of code: `b = 40 * $5`. A red arrow points to the dollar sign `$`, which is labeled "Illegal Character" in red text.

- Analogy: Take text of a sentence and break it down into valid words from the dictionary

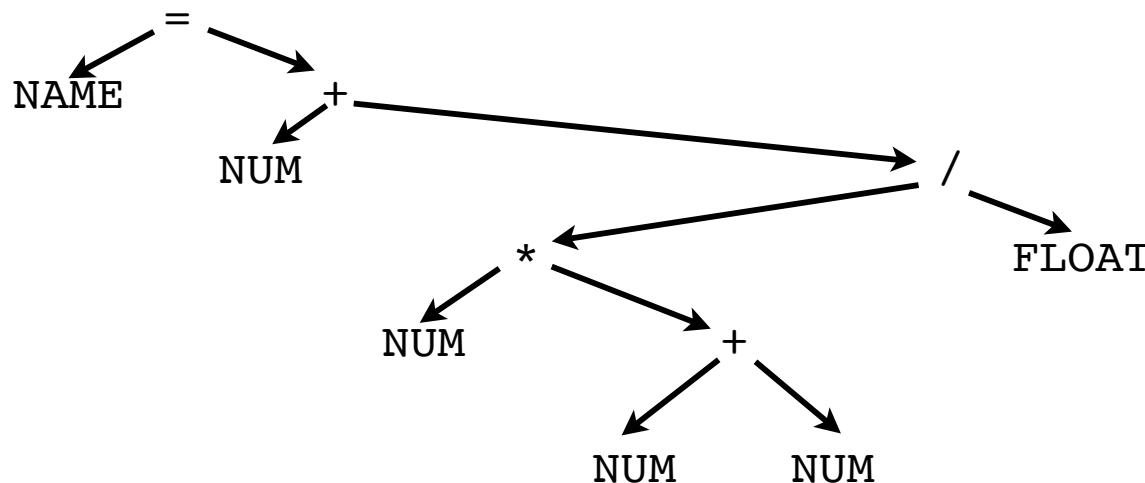
Parsing

- Checks that input is structurally correct

b = 40 + 20*(2+3)/37.5

- Builds a tree representing the structure

NAME = NUM + NUM * (NUM + NUM) / FLOAT

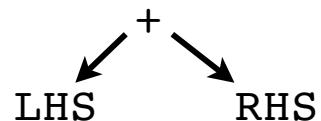


Type Checking

- Enforces the rules (aka, the "legal department")

b = 40 + 20*(2+3)/37.5	(OK)
c = 3 + "hello"	(TYPE ERROR)
d[4.5] = 4	(BAD INDEX)

- Example: + operator



1. LHS and RHS must be the same type
2. The type must implement +

Code Generation

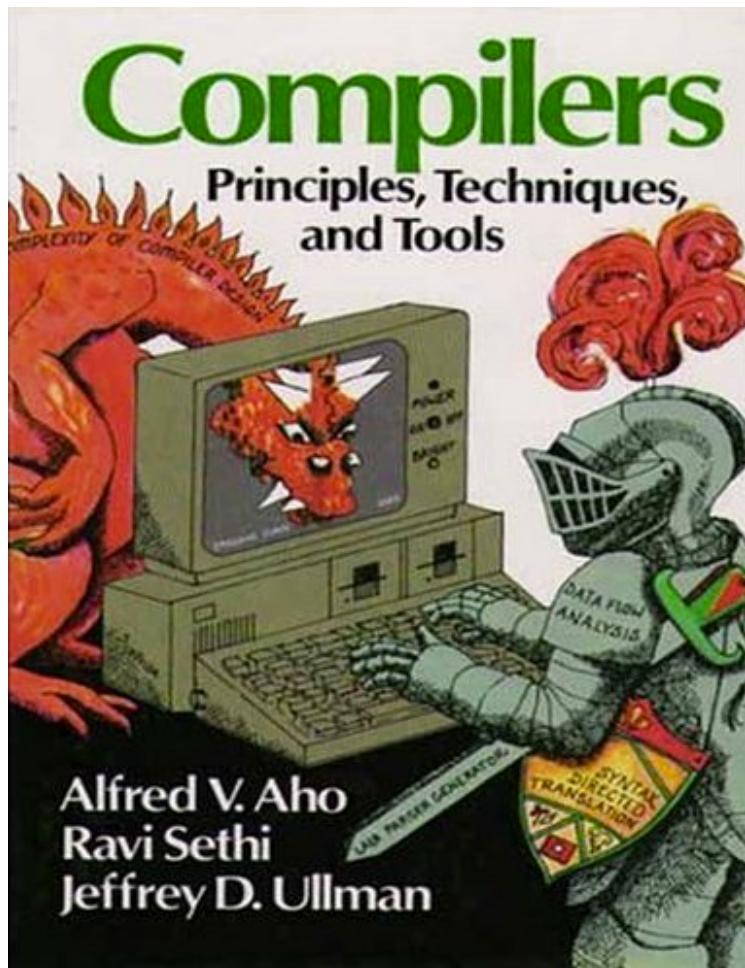
- Generation of "output code":

```
b = 40 + 20*(2+3)/37.5
      ↓
LOAD R1, 40
LOAD R2, 20
LOAD R3, 2
LOAD R4, 3
ADD R3, R4, R3 ; R3 = (2+3)
MUL R2, R3, R2 ; R2 = 20*(2+3)
LOAD R3, 37.5
DIV R2, R3, R2 ; R2 = 20*(2+3)/37.5
ADD R1, R2, R1 ; R1 = 40+20*(2+3)/37.5
STORE R1, "b"
```

Why Write a Compiler?

- Doing so demystifies a huge amount of detail about how computers and languages work
- It makes you a more informed developer
- Confidence : If you can write a compiler, chances are you can code just about anything (few tasks are ever *that* insane)

Books



- The "Dragon Book"
- Very mathematical
- Typically taught to graduate CS students
- Intense

Teaching Compilers

- Mathematical approach
 - Lots of formal proofs, algorithms, possibly some implementation in a functional language (LISP, ML, Haskell, etc.)
- Systems approach
 - Some math/algorithms, software design, computer architecture, implementation of a compiler in C, C++, Java.

Our Approach

- An implementation approach, but using Python as an implementation language
- We are going to build an actual compiler for a simple programming language
- Just because it's simple doesn't mean it will be easy--will be a project that could be expanded into a much larger language

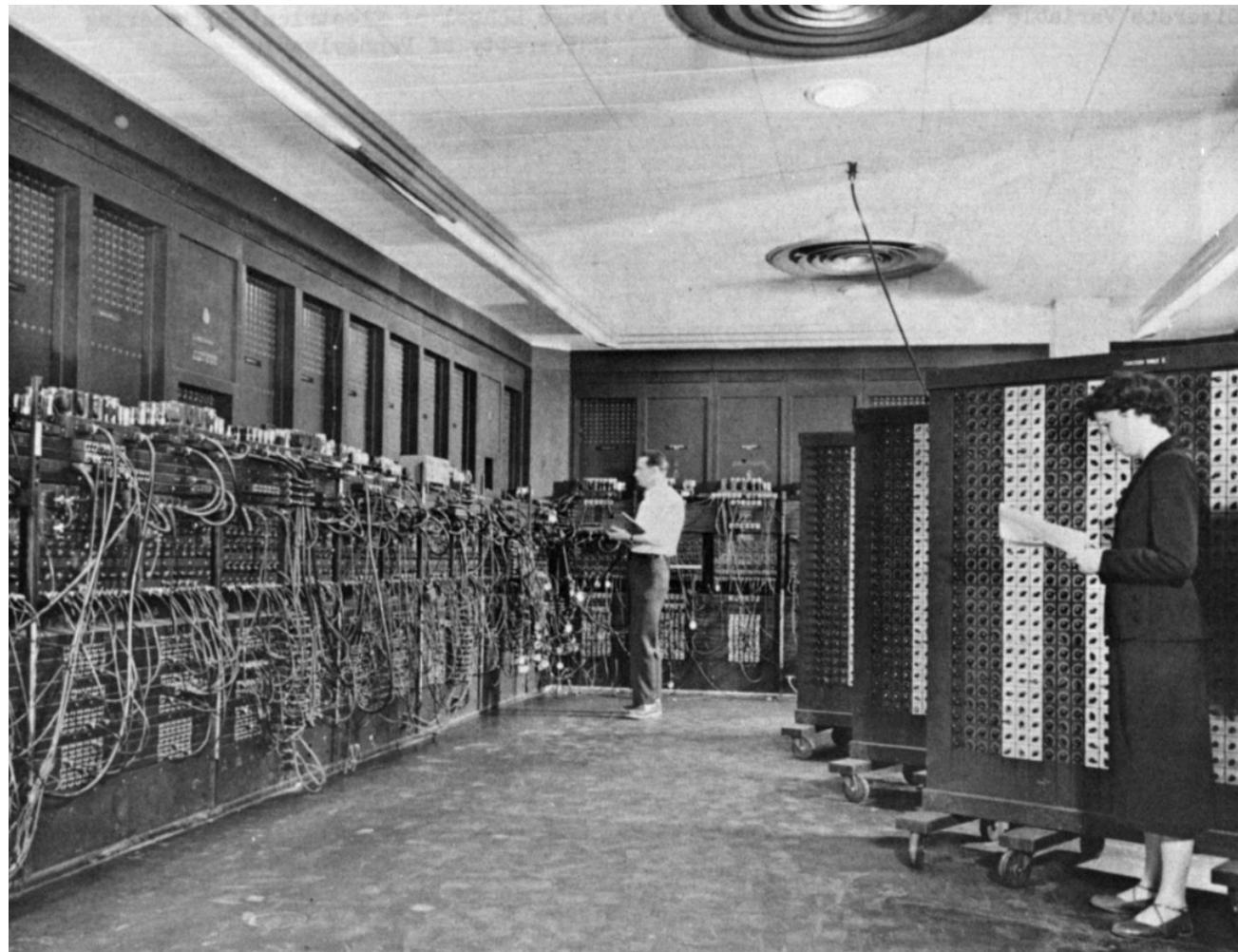
Disclaimer

- The project is comparable in scope and complexity to a compilers programming project given to CS students at a university
- I took such a course as a grad student.
- I taught such a course as a CS professor

Part 0

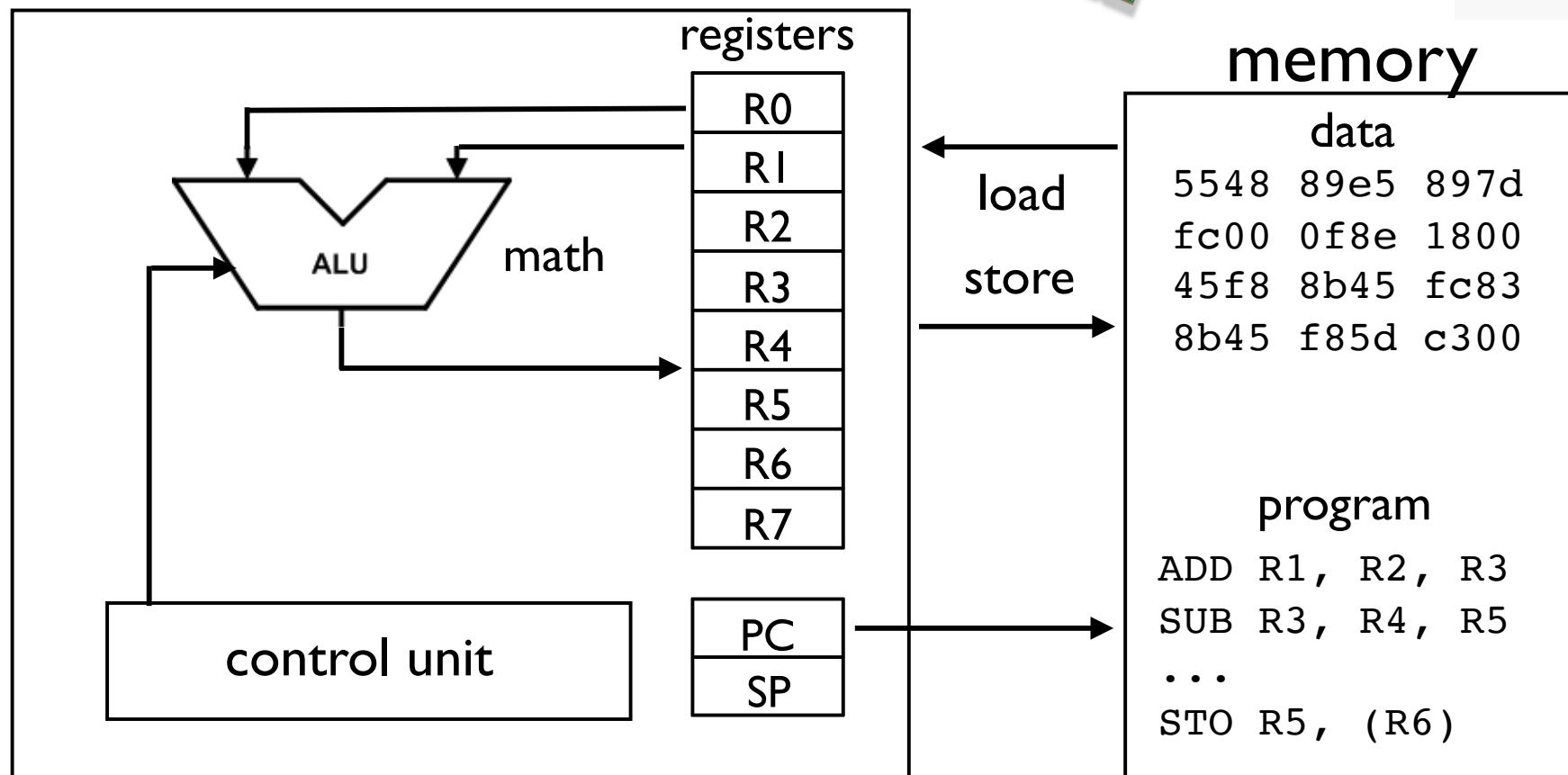
Preliminaries

What is a Computer?



What is a Computer?

CPU (Central Processing Unit)



What is a Computer?

- Computers are not especially "smart"
- Glorified calculator (with program memory)
- Example:

2 + 3 * 4 - 5

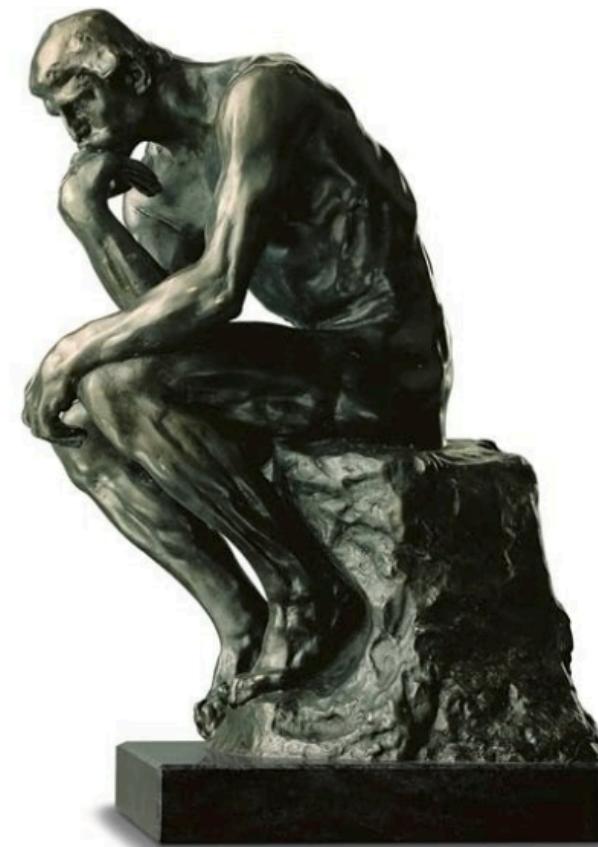
MOV 2, R1	; R1 = 2
MOV 3, R2	; R2 = 3
MOV 4, R3	; R3 = 4
MUL R2, R3, R4	; R4 = R2 * R3 = 3 * 4 = 12
ADD R1, R4, R5	; R5 = R1 + R4 = 2 + 12 = 14
MOV 5, R6	; R6 = 5
SUB R5, R6, R7	; R7 = R5 - R6 = 14 - 5 = 9

What is Computation?

"Computer Science is no more about computers than astronomy is about telescopes."

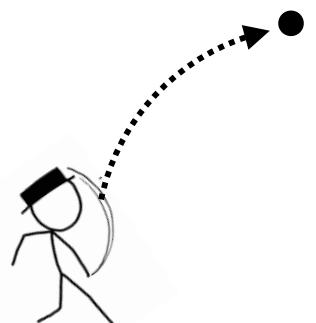
What does it actually mean to "compute" something?

What can be computed?



Math vs. Computation

- Math: Expressing relationships/properties



$$\text{height} = -16*t^{**2} + v*t$$

- Computation: A process/procedure

Compute: $5!$ $\rightarrow 5 * 4 * 3 * 2 * 1$

```
result = 1
n = 5
while n > 0:
    result = result * n
    n = n - 1
```

What is Computation?

- But, what is the actual essence of "computation?"
- Give me a minimal definition of it...



Deep Idea

- Computation is repeated substitution

$$\begin{array}{r} 2 + 3 * 4 - 5 \\ \downarrow \text{substitute} \\ 2 + 12 - 5 \\ \downarrow \text{substitute} \\ 14 - 5 \\ \downarrow \text{substitute} \\ 9 \end{array}$$

- Maybe overly simplified, but computation is a process of substituting one thing for another
- Stops when no more substitutions possible

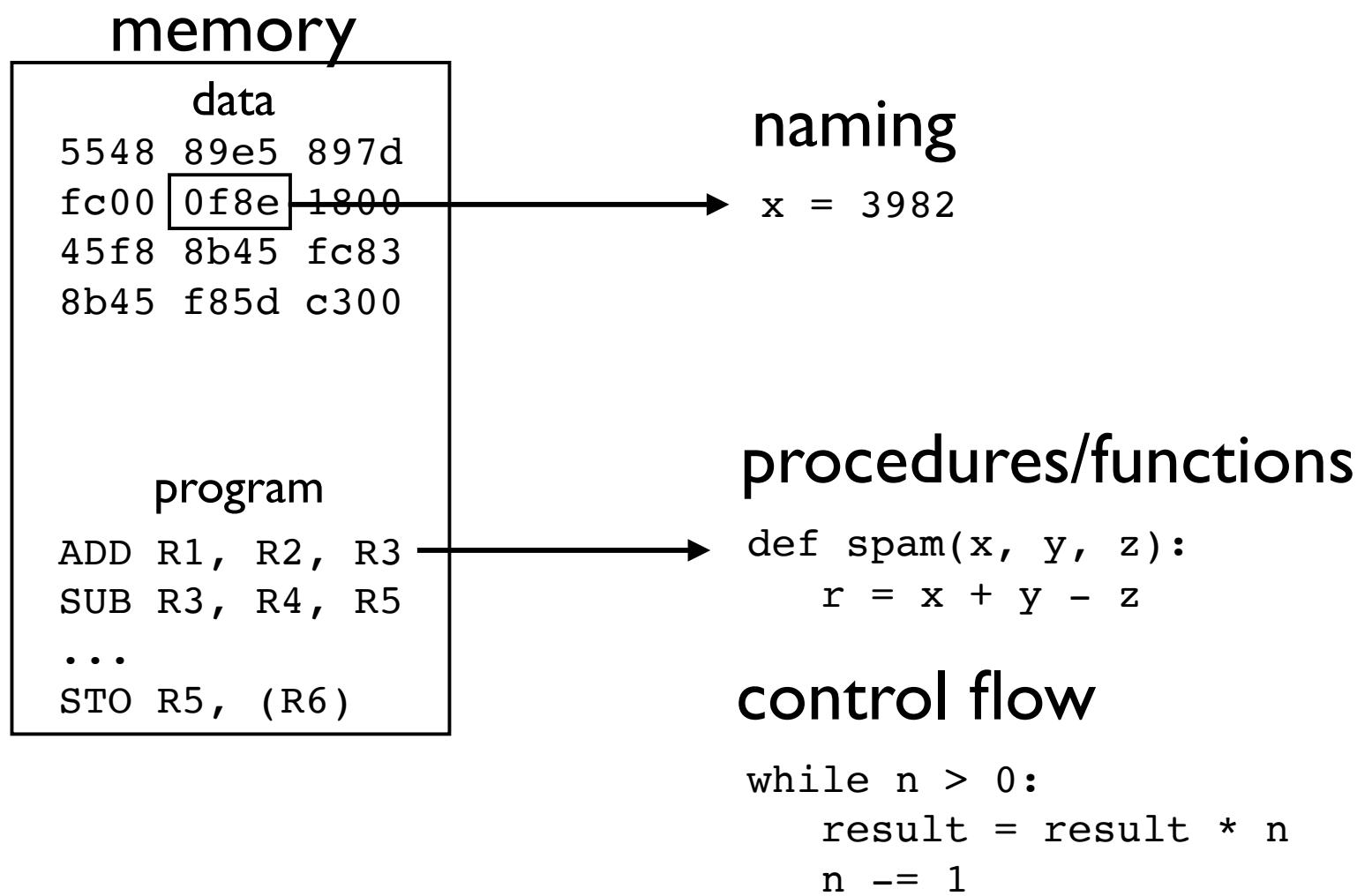
Substitution

- Substitution is a major part of compiler writing
- "High level" things are replaced by "lower level" things
- Part of moving from an abstract programming language down to the actual hardware

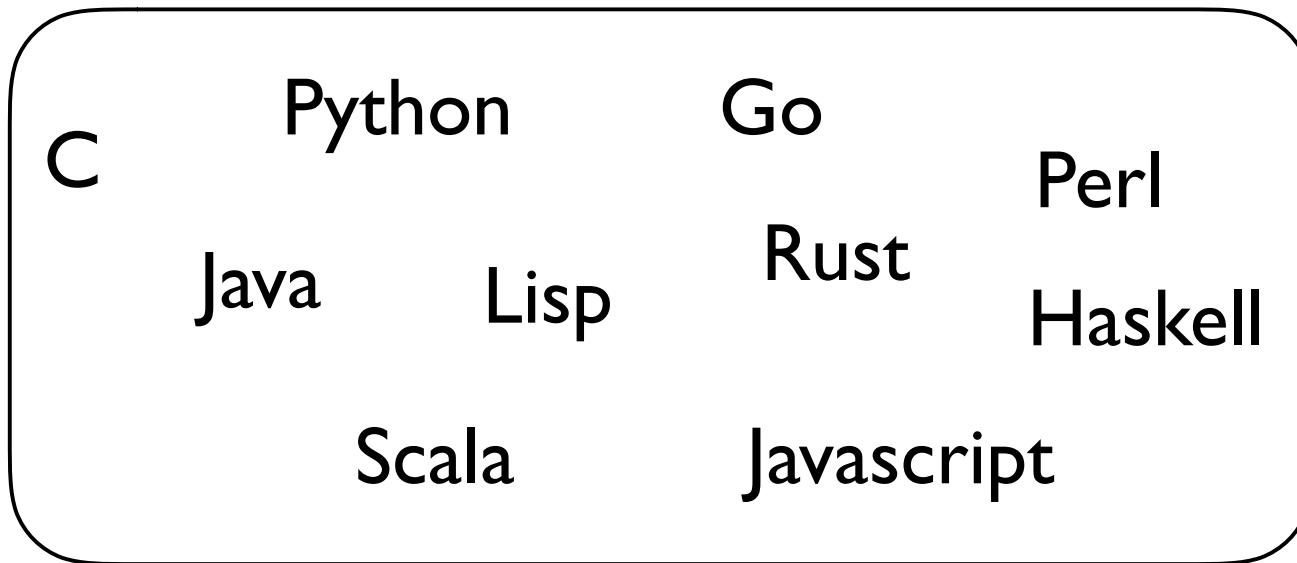
What is Programming?

- Describes a computational process. Yes.
- How? By banging keys on a keyboard??
- Key feature: Abstraction

Programming is Abstraction



Programming is Abstraction

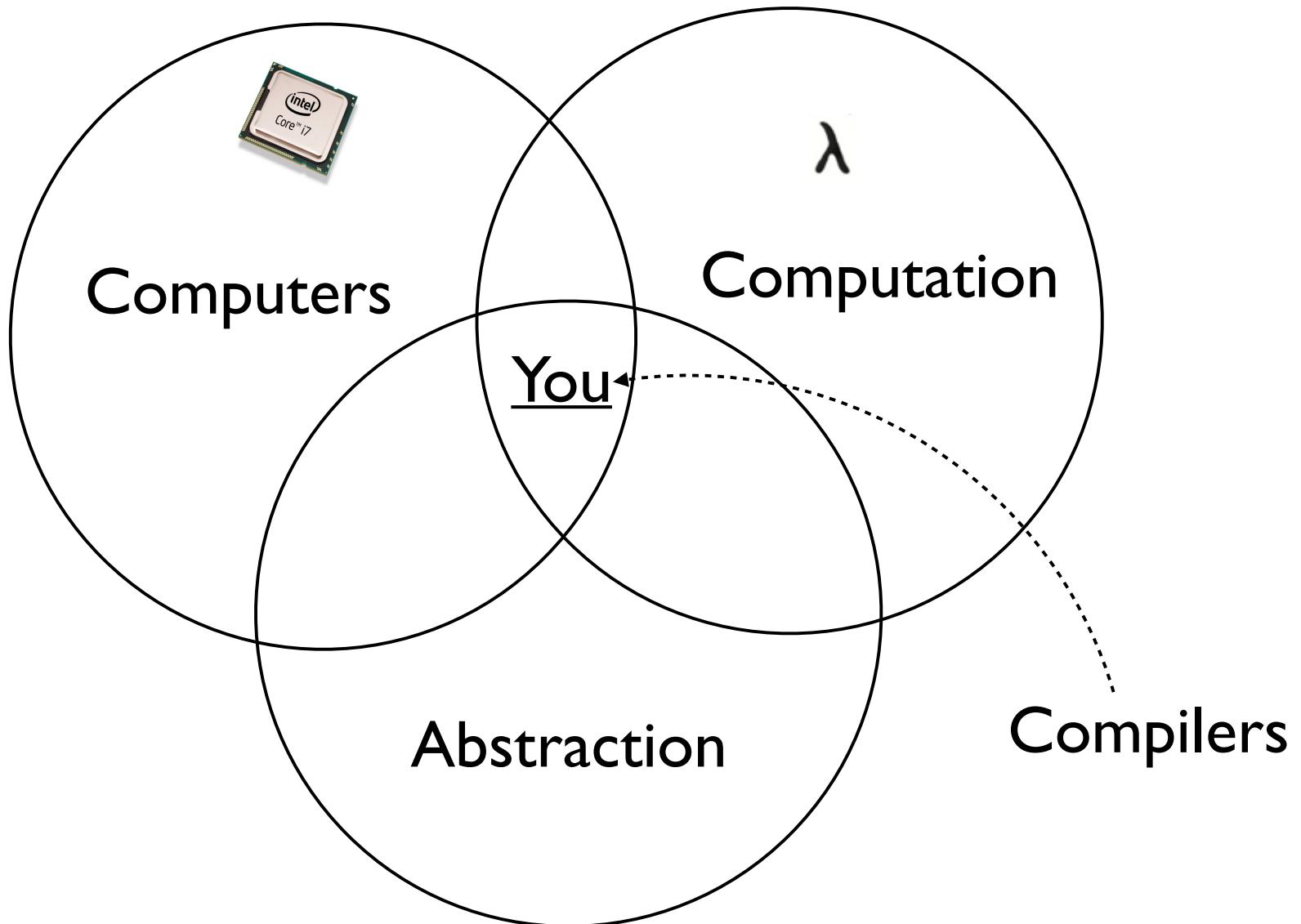


Abstraction



"Metal"

Big Picture



Exercise and Project 0

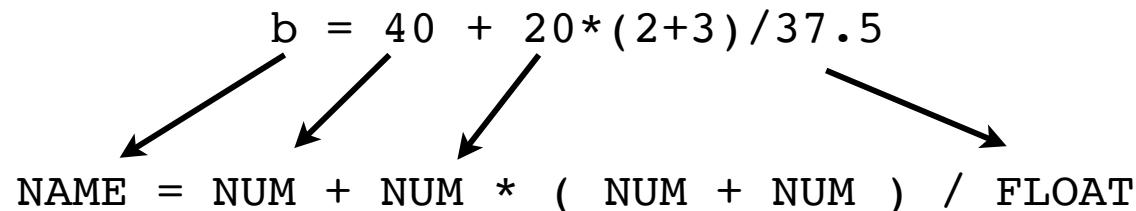
See: [compilers/doc/html/index.html](#)

Part I

Lexing

Lexing in a Nutshell

- Convert input text into a token stream



- Tokens have both a type and value

`b` → ('NAME', 'b')

`=` → ('ASSIGN', '=')

`40` → ('NUM', '40')

- Question: How to do it?

Tokenization w/ regex

- Each token is defined by a named re pattern

```
NAME      = r'(?P<NAME>[A-Za-z_][A-Za-z0-9_]* )'  
NUM       = r'(?P<NUM>\d+ )'  
ASSIGN    = r'(?P<ASSIGN>= )'  
SPACE    = r'(?P<SPACE>\s+ )'
```

- You make a master regex by joining

```
pat = re.compile(' | '.join([NAME, NUM, ASSIGN, SPACE]))
```

- You match text

```
>>> m = pat.match('1234')  
>>> m.group()
```

value → '1234'

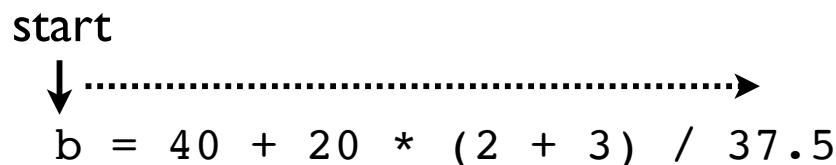
```
>>> m.lastgroup
```

type → 'NUM'

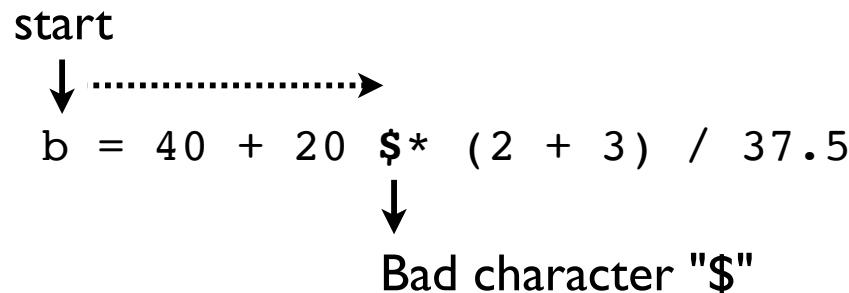
```
>>>
```

Linear Regex Scanning

- Must perform a linear text scan



- ALL characters must be matched
- Otherwise error:



Linear Scan Example

- Scan the text via iteration (generator)

```
def tokenizer(pat, text):  
    index = 0  
    while index < len(text):  
        m = pat.match(text, index)  
        if m:  
            yield m  
            index = m.end()  
        else:  
            raise SyntaxError('Bad char %r' % text[index])  
  
>>> for m in tokenizer(pat, 'foo 42'):  
...     tok = (m.lastgroup, m.group())  
...     print(tok)  
...  
('NAME', 'foo')  
('SPACE', ' ')  
('NUM', '42')  
>>>
```

Tricky Problems

- Longer matches must be checked first

```
LT      = r'(?P<LT><) '      # Matches <
LE      = r'(?P<LE><=) '     # Matches <=
ASSIGN = r'(?P<ASSIGN>=) ' # Matches =
```

- Bad:

```
pat = re.compile(' | '.join([LT,LE,ASSIGN]))  
'<=' → [ ('LT', '<'), ('ASSIGN', '=') ]
```

- Good:

```
pat = re.compile(' | '.join([LE,LT,ASSIGN]))  
'<=' → [ ('LE', '<=' ) ]
```

notice the order

Tricky Problems

- Don't be too greedy...

```
STRING = re.compile(r'\".*\"')
```



```
>>> text = '"Hello" + "World" '
>>> m = STRING.match(text)
>>> m.group()
'"Hello" + "World" '
>>>
```

- You often want shortest matches (added ?)

```
STRING = re.compile(r'\".*?\"')
```



```
>>> text = '"Hello" + "World" '
>>> m = STRING.match(text)
>>> m.group()
'"Hello"'
>>>
```

Tricky Problems

- Avoid writing patterns that are substrings

```
PRINT = r'(?P<PRINT>print)'  
NAME  = r'(?P<NAME>[a-zA-Z]+)'
```

```
pat = re.compile("|".join([PRINT, NAME]))
```

- Example:

"printable" → [('PRINT', 'print'), ('NAME', 'able')]

- Better to incorporate matching of keywords as a separate step elsewhere (not in the regex)

Exercise I

Lexing Tools

- Tokenizing is a "solved" problem
- Most people use tools for this
- Example: PLY, PyParsing, Antlr, etc.
- We will use SLY

SLY Example

```
from sly import Lexer

class MyLexer(Lexer):
    tokens = { NAME, NUMBER, PLUS, MINUS, TIMES,
               DIVIDE, EQUALS }

    # Ignored characters (between tokens)
    ignore = ' \t'

    # Token specifications
    PLUS    = r'\+'
    MINUS   = r'-'
    TIMES   = r'\*'
    DIVIDE  = r'/'
    EQUALS  = r'='
    NAME    = r'[a-zA-Z_][a-zA-Z0-9_]*'
    NUMBER  = r'\d+'
```

SLY Example

```
>>> text = "b = 40 + 20 * 2"
>>> lexer = MyLexer()
>>> for tok in lexer.tokenize(text):
...     print(tok.type, tok.value)
...
NAME b
EQUALS =
NUMBER 40
PLUS +
NUMBER 20
TIMES *
NUMBER 2
>>>
```

Token instance

.type = Token name
.value = Token value
.lineno = Line number
.index = Index in input string



Ignored Patterns

```
class MyLexer(Lexer):
    tokens = { NAME, NUMBER, PLUS, MINUS, TIMES,
               DIVIDE, EQUALS }

    # Ignored characters (between tokens)
    ignore = '\t'

    # Token specifications
    PLUS    = r'\+'
    MINUS   = r'-'
    ...
    # Ignored text patterns
    ignore_comment = r'\#.*'      # Comments
```

- Give a regex rule with `ignore_*` prefix



Optional Actions

```
class MyLexer(Lexer):
    tokens = { NAME, NUMBER, PLUS, MINUS, TIMES,
               DIVIDE, EQUALS }

    ...
    # Token specifications
    PLUS    = r'\+'
    MINUS   = r'-'
    NUMBER  = r'\d+'

    def NUMBER(self, t):
        t.value = int(t.value)      # Convert value
        return t
```

- Method triggers when token is matched

Position Tracking

```
class MyLexer(Lexer):
    tokens = { NAME, NUMBER, PLUS, MINUS, TIMES,
               DIVIDE, EQUALS }
    ...
    # Token specifications
    PLUS    = r'\+'
    MINUS   = r'-'
    ...
    @_('\n+')
    def ignore_newlines(self, t):
        self.lineno += t.value.count('\n')
    ...
```

- Write a method to match newlines and increment the lexer line number

Error Handling

```
class MyLexer(Lexer):
    tokens = { NAME, NUMBER, PLUS, MINUS, TIMES,
               DIVIDE, EQUALS }
    ...
    # Token specifications
    PLUS   = r'\+'
    MINUS  = r'-'
    ...
    def error(self, t):
        print('Bad character %r' % t.value[0])
        self.index += 1      # Skip one character
    ...
```

- Define an `error()` method
- Skip over the bad character

Project I

Under the Hood

- What does `re.compile()` actually do?

```
NAME      = r'(?P<NAME>[A-Za-z_][A-Za-z0-9_]* )'  
NUM       = r'(?P<NUM>\d+ )'  
ASSIGN    = r'(?P<ASSIGN>= )'  
WHITESPACE = r'(?P<WHITESPACE>\s+ )'
```

```
pat = " | ".join([NAME, NUM, ASSIGN, WHITESPACE])  
cpat = re.compile(pat)
```

- Have you ever looked?

re Compilation

- Compilation can be viewed using `sre_parse`

```
>>> import sre_parse
>>> sre_parse.parse(r'[a-zA-Z]+')
[(MAXREPEAT, (1, MAXREPEAT, [(IN, [(RANGE, (97, 122)), (RANGE, (65, 90))])]))]
>>>
```

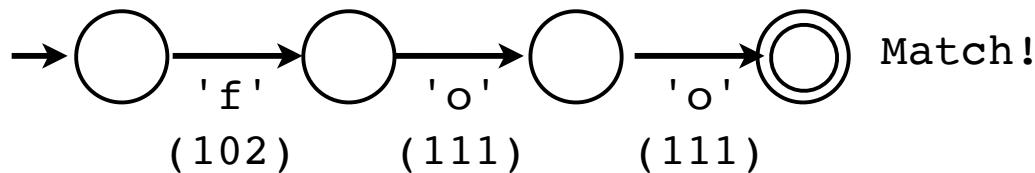
- Returns a somewhat verbose list of nested lists/tuples
- Let's see some examples:

re Compilation

- Simple strings

```
>>> import sre_parse
>>> sre_parse.parse(r'foo')
[(LITERAL, 102), (LITERAL, 111), (LITERAL, 111)]
>>>
```

- Defines a simple path (through an NFA)

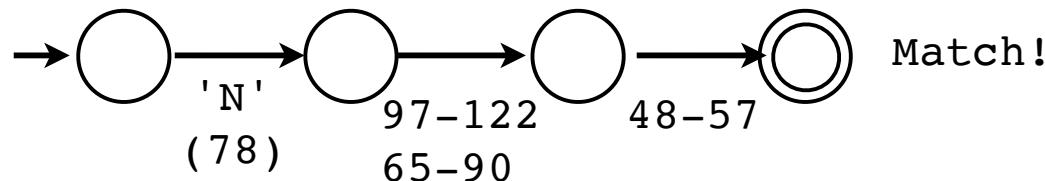


re Compilation

- Character ranges

```
>>> import sre_parse
>>> sre_parse.parse(r'N[a-zA-Z][0-9]')
[(LITERAL, 78),
 (IN, [(RANGE, (97, 122)), (RANGE, (65, 90))]),
 (IN, [(RANGE, (48, 57))])]
>>>
```

- Path accepts ranges of character codes



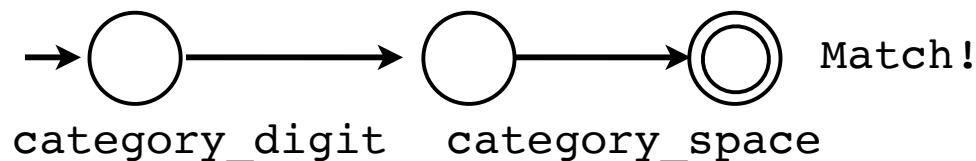
re Compilation

- Character classes

```
>>> import sre_parse
>>> sre_parse.parse(r'\d\s')
[ (IN, [ (CATEGORY, CATEGORY_DIGIT) ]),  

 (IN, [ (CATEGORY, CATEGORY_DIGIT) ]) ]
>>>
```

- Categories are predefined sets of chars

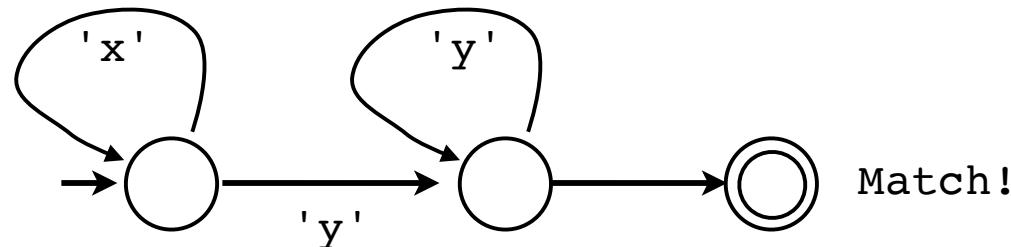


re Compilation

- Repetition

```
>>> import sre_parse
>>> sre_parse.parse(r'x*y+')
[ (MAXREPEAT, (0, MAXREPEAT, [(LITERAL, 120)])),,
 (MAXREPEAT, (1, MAXREPEAT, [(LITERAL, 121)]))]
>>>
```

- Introduces cycles (and counts)

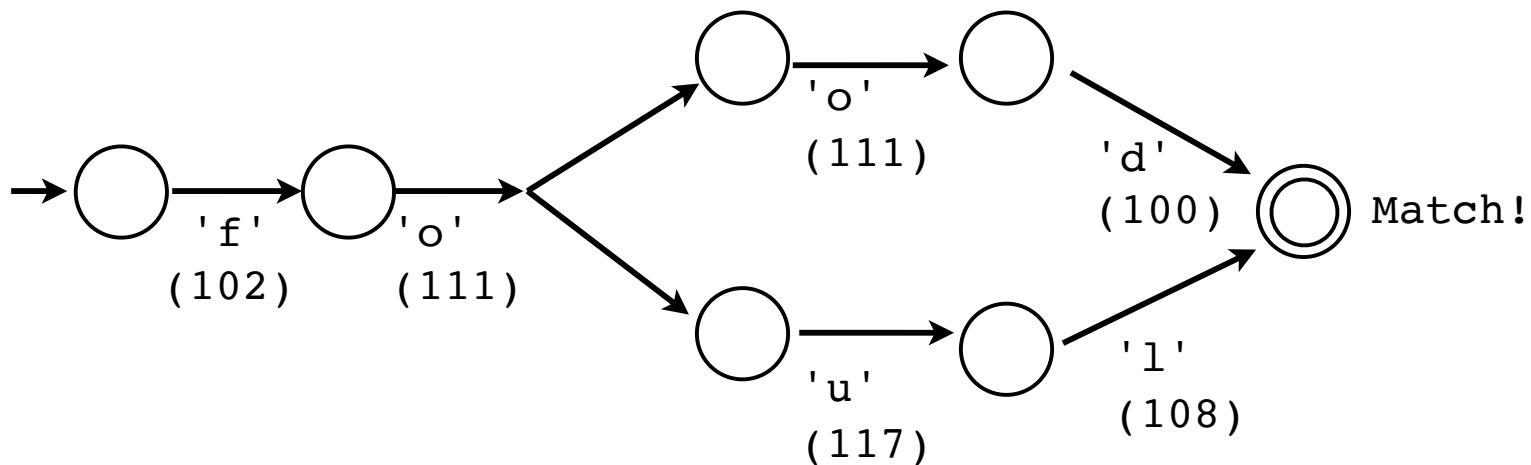


re Compilation

- "|" operation

```
>>> import sre_parse
>>> sre_parse.parse(r'food|foul')
[ (LITERAL, 102), (LITERAL, 111),
  (BRANCH, (None, [ [(LITERAL, 111), (LITERAL, 100)],
                     [(LITERAL, 117), (LITERAL, 108)] ])) ]
>>>
```

- Introduces branches

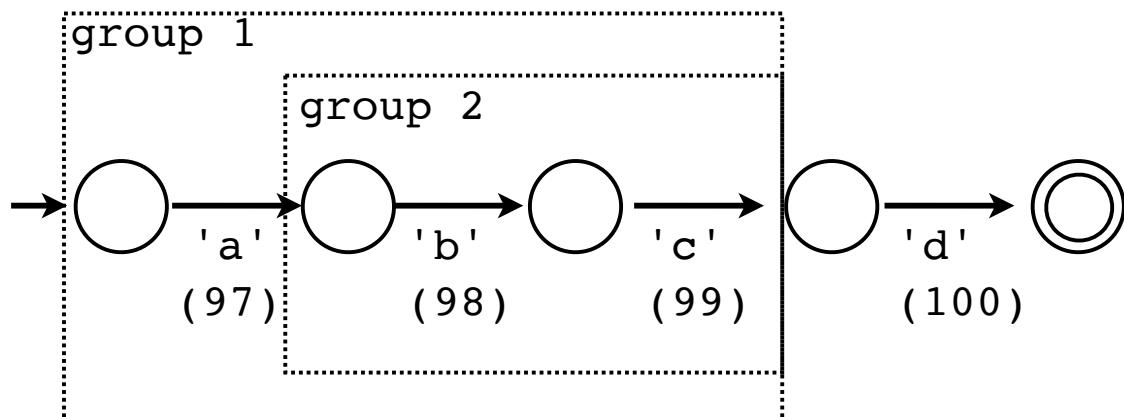


re Compilation

- "(pat)" grouping

```
>>> import sre_parse
>>> sre_parse.parse(r'(a(bc))d')
[ (SUBPATTERN, (1, [(LITERAL, 97),
    (SUBPATTERN, (2, [(LITERAL, 98), (LITERAL, 99)])
  ))]),
  (LITERAL, 100)]
>>>
```

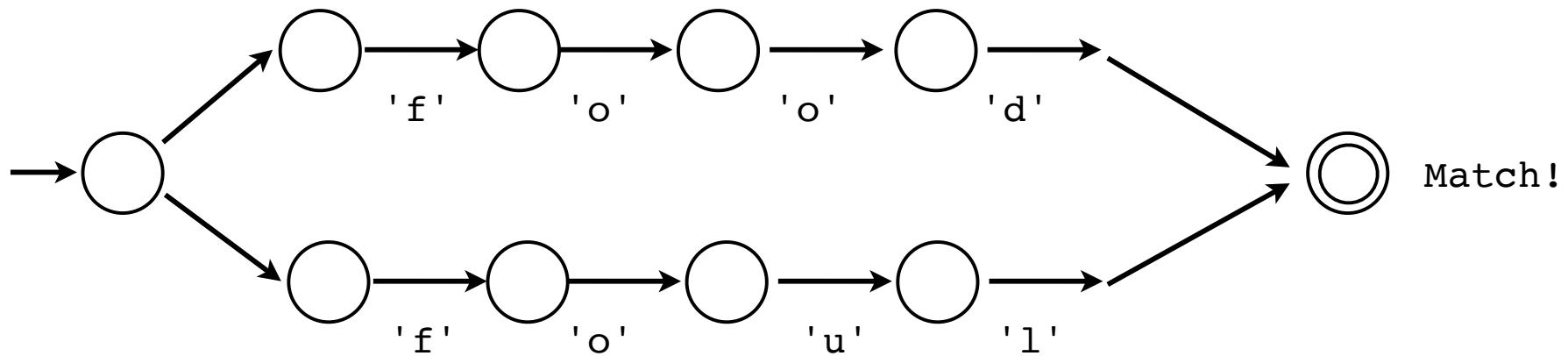
- Puts capture groups around path segments



re Matching Behavior

- Matching involves walking all possible paths

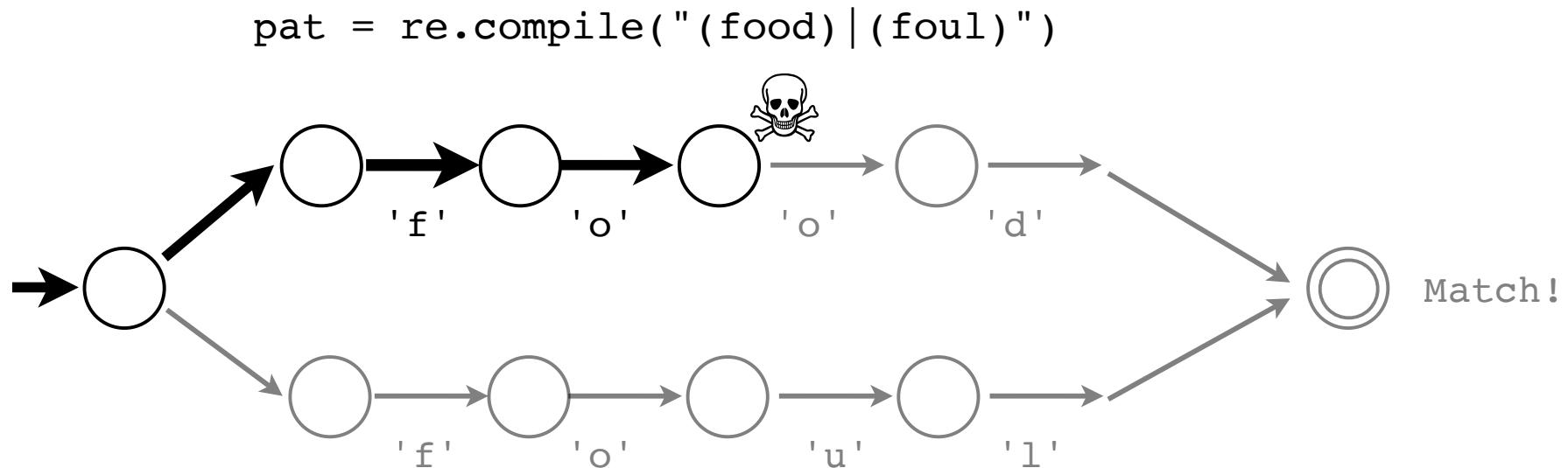
```
pat = re.compile("(food)|(foul)")
```



- They are simply tried in order (if branches)

re Matching Behavior

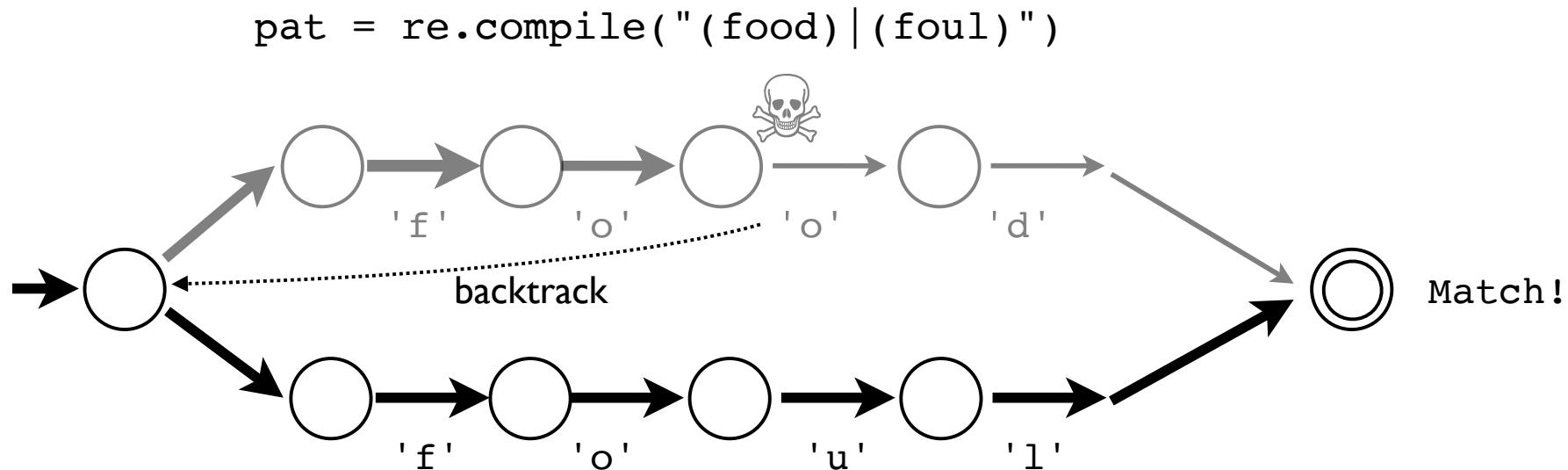
- Matching might reach a dead end



- Example: pat.match("foul")

re Matching Behavior

- Matching will backtrack to try other paths



- Example: pat.match("foul")

Backtracking Caution

- Backtracking in re patterns requires caution
- Inherently prevents regex matching directly on I/O streams (e.g., networks)
- Introduces pathological corner cases
- Exponential match times (sometimes)

See : Russ Cox,
"Regular Expression Matching Can Be Simple and Fast
(but is slow in Java, Perl, PHP, Python, Ruby, ...)"

Part 2

Parsing

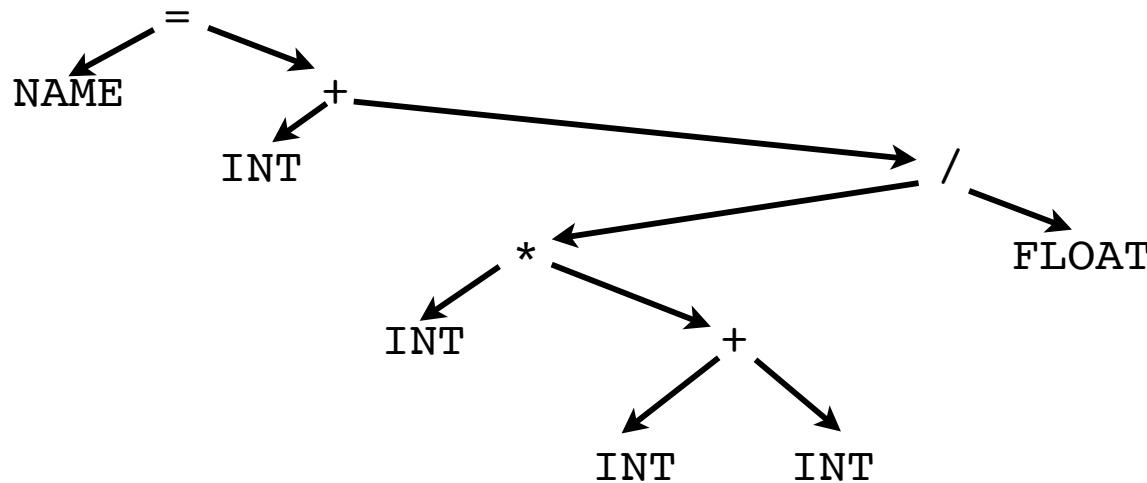
Parsing in a Nutshell

- Accept syntactically correct input

b = 40 + 20*(2+3)/37.5

- Usually builds a tree representing the structure

NAME = INT + INT * (INT + INT) / FLOAT



Disclaimer

- Parsing theory is a huge topic
- Highly mathematical
- Covered in great detail the first 3-5 weeks of a compilers course
- I'm going to cover the highlights

Problem: Specification

- How do you even describe a language?
- Example: Describe "assignment"

```
a = 0  
b = 2 + 3  
c = 2 + 3 * 4  
d = (2 + 3) * 4  
e = 0.5 * d
```

- By "describe" we mean a precise specification
- Already have tokens { NAME, NUM, FLOAT, ... }

Problem: Specification

- Example: Describe "assignment"

name = expression

- Okay, that's a start... but what's "expression"?
- It could be a number

name = 3

name = 3.4

- Or a combination of numbers and operators

name = 3 + 4

- Still a bit vague... how to precisely define?

Mathematical Approach

- Perhaps you describe things in terms of sets

```
S0 = { 0, 1, 2, ... } # Base: Start
```

```
Si = { s + t | s, t ∈ Si-1} # Induction
      ∪ { s - t | s, t ∈ Si-1}
      ∪ { s * t | s, t ∈ Si-1}
      ∪ { s / t | s, t ∈ Si-1}
      ∪ { ( t ) | t ∈ Si-1}
```

expression = $\bigcup S_i$, for all $i \geq 0$

- This is a precise mathematical description of what constitutes valid syntax
- It's the set of all syntactically valid things

Grammar Specification

- More practical to specify the language in terms of a formal grammar

```
assignment ::= NAME '=' expr ';'
```

```
expr      ::= expr '+' expr
           | expr '-' expr
           | expr '*' expr
           | expr '/' expr
           | '(' expr ')'
           | INT
           | FLOAT
           | NAME
```

- Notation is in BNF (Backus Normal Form)

Grammar Specification

- A BNF specifies substitutions

```
assignment ::= NAME '=' expr ';'
```

```
expr      ::= expr '+' expr
          | expr '-' expr
          | expr '*' expr
          | expr '/' expr
          | '(' expr ')'
          | INT
          | FLOAT
          | NAME
```

- Name on left can be replaced the sequence of symbols on the right (and vice versa).

Grammar Specification

- A BNF specifies substitutions

assignment ::= NAME '=' **expr** ';'

expr ::= expr '+' expr
| expr '-' expr
| expr '*' expr
| expr '/' expr
| '(' expr ')'
| INT
| FLOAT
| NAME

Can replace by any of
these sequences

- Name on left can be replaced the sequence of symbols on the right (and vice versa).

Grammar Specification

- A BNF specifies substitutions

```
assignment ::= NAME '=' expr ';'
```

```
expr      ::= expr '+' expr  
          | expr '-' expr  
          | expr '*' expr  
          | expr '/' expr  
          | '(' expr ')'  
          | INT  
          | FLOAT  
          | NAME
```

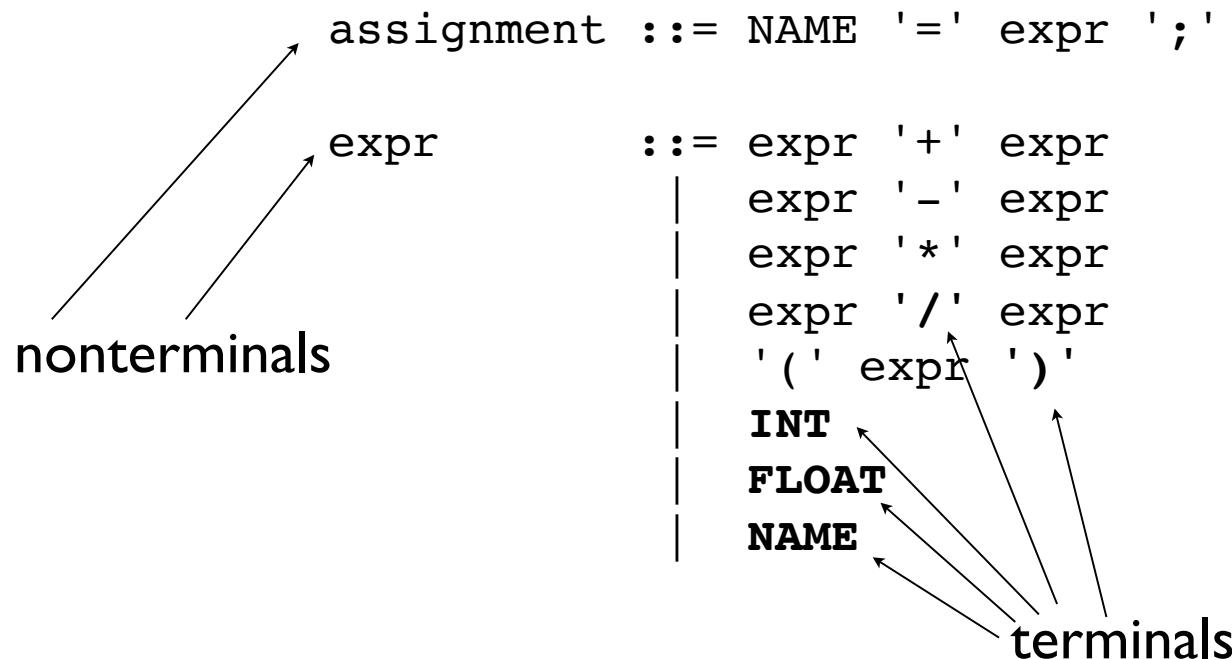
Examples

```
spam = 42;  
spam = 4+2;  
spam = (4+2)*3
```

- Name on left can be replaced the sequence of symbols on the right (and vice versa).

Terminals/Nonterminals

- Tokens are called "terminals"
- Rule names are called "nonterminals"



Terminology

- "terminal" - A symbol that can't be expanded into anything else (tokens).
- "nonterminal" - A symbol that can be expanded into other symbols (grammar rules)

Parsing Explained

- Problem: match text against a grammar

```
a = 2 * 3 + 4;
```

- Example: Does it match the assignment rule?

```
assignment ::= NAME '=' expr ';'
```

- How would you go about doing it?

Parsing Strategies

- Top Down: Start with the top-most grammar rule. Repeatedly expand non-terminal symbols until nothing but terminals matching the input text remain
- Bottom Up: Start with the raw input terminals. Repeatedly reduce symbols using grammar rules until the top-most grammar symbol is only remaining symbol.

Top-Down Parsing

- Start with the top rule and keep applying substitutions until you match all tokens

assignment ::= NAME '=' expr ';'



can you rewrite as?

NAME '=' INT '*' INT '+' INT ';'

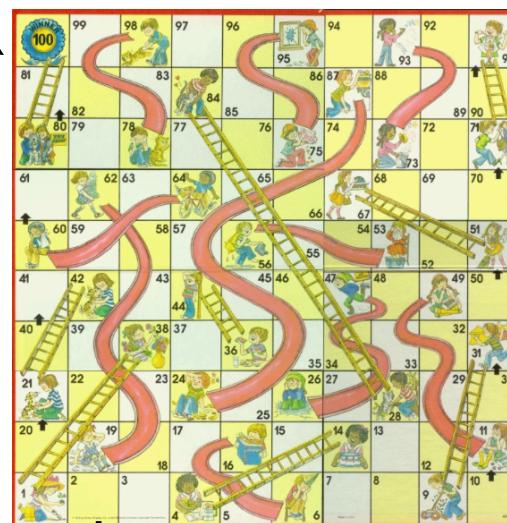
- Essentially, you keep replacing nonterminals with expansions until you get nothing but tokens

Top-Down Parsing

- Example:

assignment ::= NAME '=' expr ';'

start



grammar

end

input tokens NAME '=' INT '*' INT '+' INT ';'

input text a = 2 * 3 + 4;

Top-Down Parsing

- Example:

```
assignment ::= NAME '=' expr ';'  
           ::= NAME '=' expr '*' expr ';'
```



```
expr ::= expr '*' expr
```

input tokens NAME '=' INT '*' INT '+' INT ';'

input text a = 2 * 3 + 4;

Top-Down Parsing

- Example:

```
assignment ::= NAME '=' expr ';'  
           ::= NAME '=' expr '*' expr ';'
```

```
expr ::= expr '*' expr
```

Decision to expand on this rule
based on looking at the input
(more later)

input tokens NAME '=' INT '*' INT '+' INT ';'

input text a = 2 * 3 + 4;

Top-Down Parsing

- Example:

```
assignment ::= NAME '=' expr ';'  
          ::= NAME '=' expr '*' expr ';'   
          ::= NAME '=' INT '*' expr ';'   
          
```

expr ::= INT

input tokens NAME '=' INT '*' INT '+' INT ';'

input text a = 2 * 3 + 4;

Top-Down Parsing

- Example:

```
assignment ::= NAME '=' expr ';'  
          ::= NAME '=' expr '*' expr ';'   
          ::= NAME '=' INT '*' expr ';'   
          ::= NAME '=' INT '*' expr '+' expr ';'   
          
```



```
expr ::= expr '+' expr
```

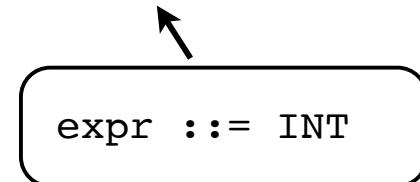
input tokens NAME '=' INT '*' INT '+' INT ';'

input text a = 2 * 3 + 4;

Top-Down Parsing

- Example:

```
assignment ::= NAME '=' expr ';'  
          ::= NAME '=' expr '*' expr ';'   
          ::= NAME '=' INT '*' expr ';'   
          ::= NAME '=' INT '*' expr '+' expr ';'   
          ::= NAME '=' INT '*' INT '+' expr ';'   
          
```



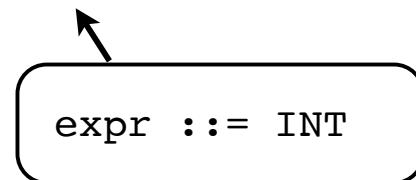
input tokens NAME '=' INT '*' INT '+' INT ';'

input text a = 2 * 3 + 4;

Top-Down Parsing

- Example:

```
assignment ::= NAME '=' expr ';'  
          ::= NAME '=' expr '*' expr ';'   
          ::= NAME '=' INT '*' expr ';'   
          ::= NAME '=' INT '*' expr '+' expr ';'   
          ::= NAME '=' INT '*' INT '+' expr ';'   
          ::= NAME '=' INT '*' INT '+' INT ';'
```



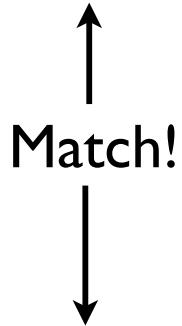
input tokens NAME '=' INT '*' INT '+' INT ';'

input text a = 2 * 3 + 4;

Top-Down Parsing

- Example:

```
assignment ::= NAME '=' expr ';'  
          ::= NAME '=' expr '*' expr ';'   
          ::= NAME '=' INT '*' expr ';'   
          ::= NAME '=' INT '*' expr '+' expr ';'   
          ::= NAME '=' INT '*' INT '+' expr ';'   
          ::= NAME '=' INT '*' INT '+' INT ';' ;
```



Observe: No further substitutions are possible
(fully expanded to terminals)

input tokens NAME '=' INT '*' INT '+' INT ';' ;

input text a = 2 * 3 + 4;

Bottom Up Parsing

- Start with the token sequence and apply rule reductions until you reach the top rule

assignment ::= NAME '=' expr ';'



can you reduce to this?

NAME '=' INT '*' INT '+' 'INT' ';'

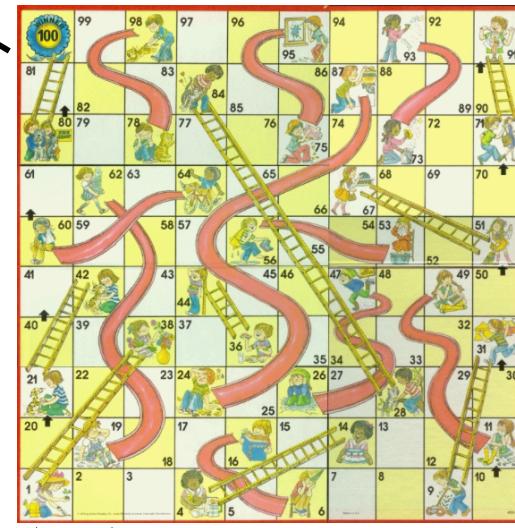
- Essentially, you replace token sequences with nonterminals until reduced to a single rule

Bottom-Up Parsing

- Example:

```
top rule      : assignment := NAME '=' expr ';'
```

end



grammar

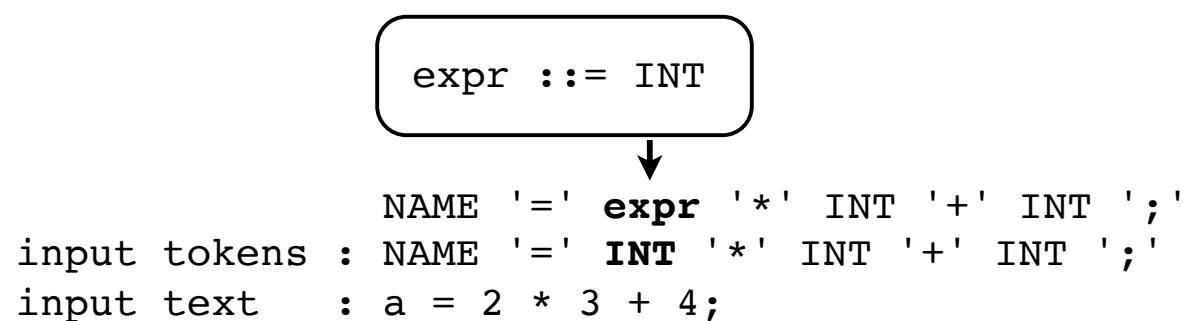
start

```
input tokens : NAME '=' INT '*' INT '+' INT ';'  
input text   : a = 2 * 3 + 4;
```

Bottom-Up Parsing

- Example:

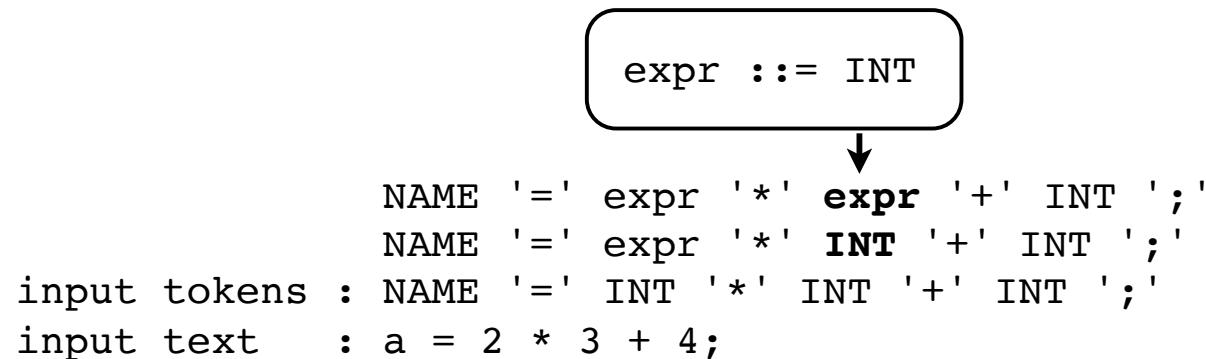
```
top rule      : assignment := NAME '=' expr ';'
```



Bottom-Up Parsing

- Example:

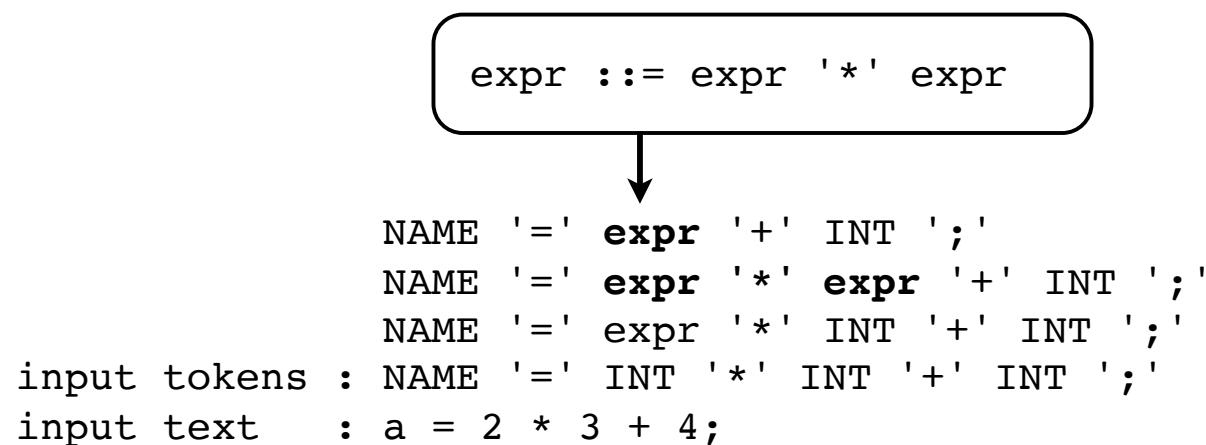
```
top rule      : assignment := NAME '=' expr ';'
```



Bottom-Up Parsing

- Example:

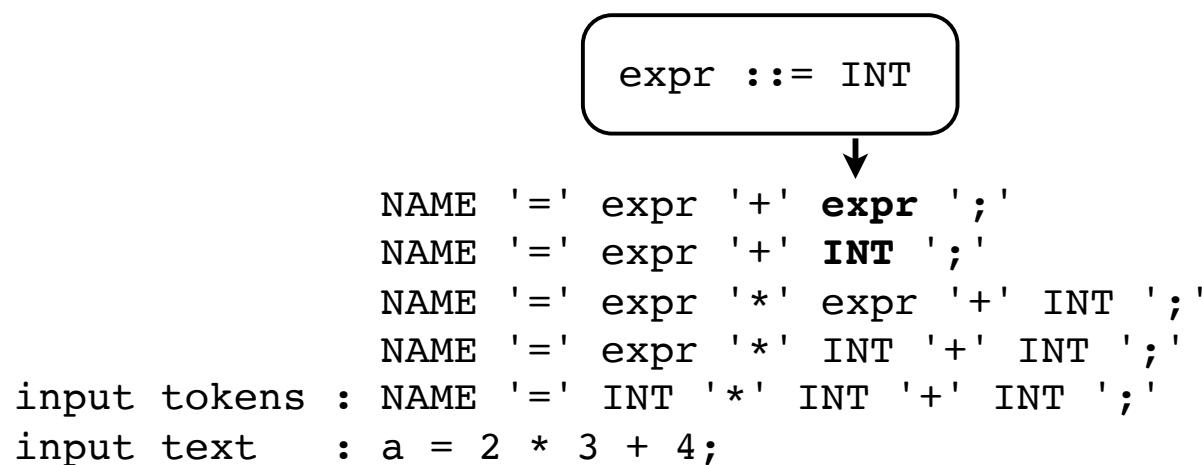
```
top rule      : assignment := NAME '=' expr ';'
```



Bottom-Up Parsing

- Example:

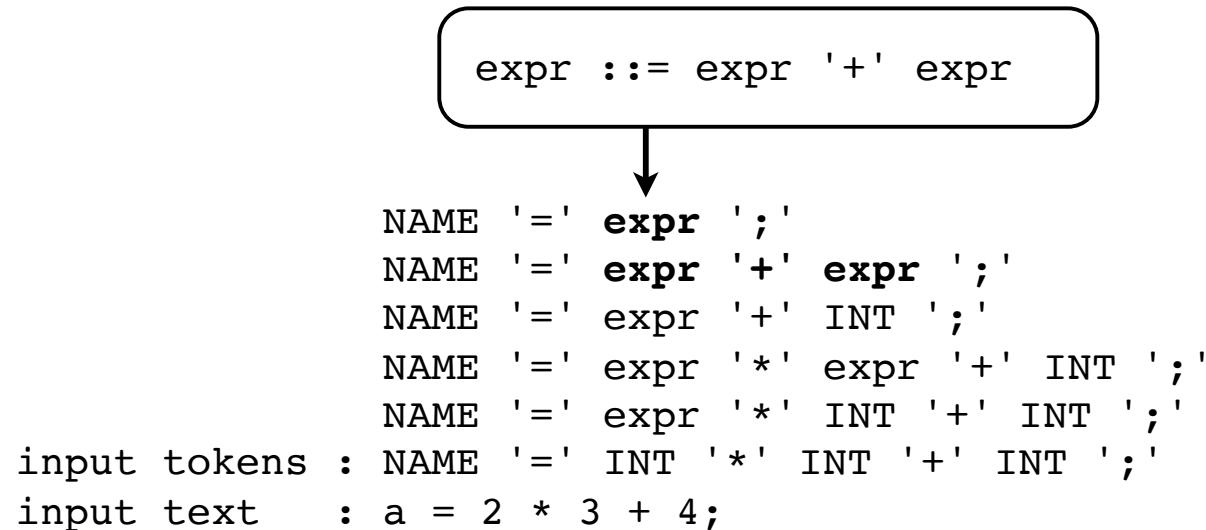
```
top rule      : assignment := NAME '=' expr ';'
```



Bottom-Up Parsing

- Example:

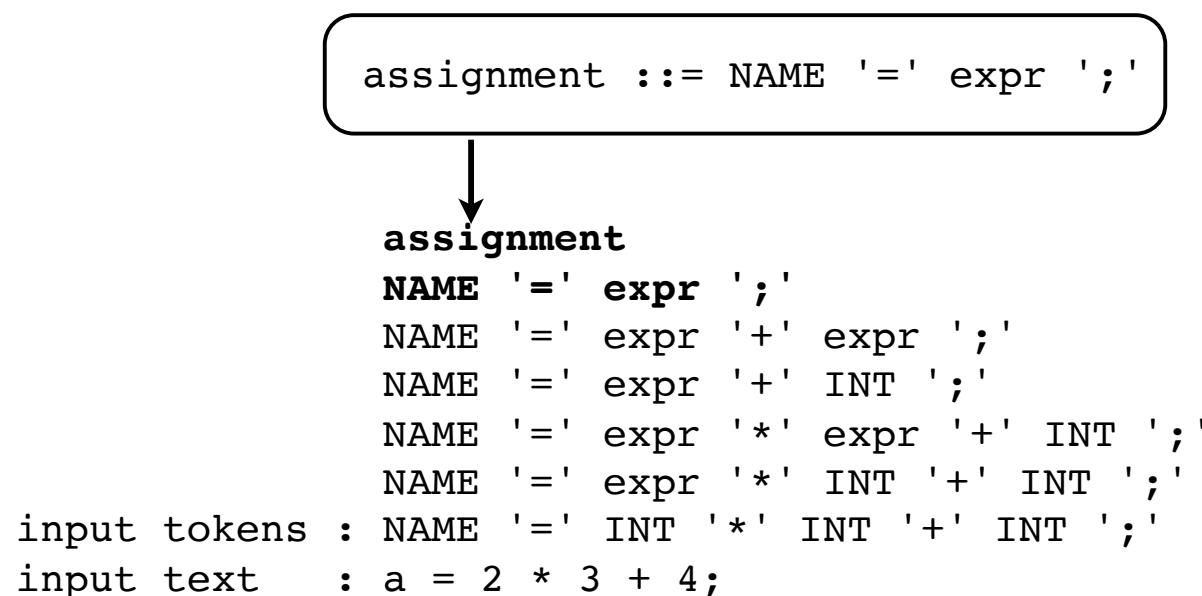
```
top rule      : assignment := NAME '=' expr ';'
```



Bottom-Up Parsing

- Example:

```
top rule      : assignment ::= NAME '=' expr ';'
```



Bottom-Up Parsing

- Example:

```
top rule      : assignment := NAME '=' expr ';'  
                ↑  
                Match!  
                ↓  
assignment  
NAME '=' expr ';'  
NAME '=' expr '+' expr ';'  
NAME '=' expr '+' INT ';'  
NAME '=' expr '*' expr '+' INT ';'  
NAME '=' expr '*' INT '+' INT ';'  
input tokens : NAME '=' INT '*' INT '+' INT ';'  
input text   : a = 2 * 3 + 4;
```

Observe: Input tokens are repeatedly reduced by grammar rules until you reach a single matching rule.

Problem: How to do it?

- Those "descriptions" were describing a general strategy, not an algorithm



- Frankly, a lot of hand-waving
- The actual algorithms are complex

Parsing Algorithms

- Most common parsing algorithms are based on a linear (left-to-right) scan of tokens
- Rule expansions/reductions are based solely on state of current progress and the next input token (lookahead)
- Common algorithms:
 - LL(1) (Top down)
 - LALR(1) (Bottom-up)

Terminology

- LL(n) - Top Down
 - Left-to-right scanning
 - Left-most rule expansion
 - n tokens of lookahead
- LR(n) - Bottom Up
 - Left-to-right scanning
 - Right-most rule reduction
 - n tokens of lookahead

DEMO

- Write a recursive descent parser

```
assignment ::= NAME '=' expr ';'
```

```
expr      ::= term '+' expr
            | term '-' expr
            | term
;
```

```
term      ::= factor '*' term
            | factor '/' term
            | factor
;
```

```
factor    : '(' expr ')'
            | NUM
            | NAME
;
```

Parsing Tools

- Parsing is almost always solved with tools
- Code generators
- Parser generators
- Why? Writing a parser by hand is painful!

DEMO

- Write a parser using SLY

```
assignment ::= NAME '=' expr ';'  
  
expr      ::= term '+' expr  
           | term '-' expr  
           | term  
;  
  
term      ::= factor '*' term  
           | factor '/' term  
           | factor  
;  
  
factor    : '(' expr ')'  
           | NUM  
           | NAME  
;
```

Syntax Directed Translation

- Tools work by attaching "actions" to the grammar rules (think callbacks/events)

Grammar

```
assignment ::= NAME '=' expr ';'  
  
expr0      ::= expr1'+' expr2  
                 | expr1'-' expr2  
                 | expr1'*' expr2  
                 | expr1'/' expr2  
                 | '(' expr1 ')'   
                 | INT  
                 | FLOAT  
                 | NAME
```

Actions

```
vars[NAME] = expr.val  
  
expr0.val = expr1.val + expr2.val  
expr0.val = expr1.val - expr2.val  
expr0.val = expr1.val * expr2.val  
expr0.val = expr1.val / expr2.val  
expr0.val = expr1.val  
expr.val = INT.val  
expr.val = FLOAT.val  
expr.val = vars[NAME]
```

- There is a propagation effect

Example

```
assignment ::= NAME '=' expr ';'
```



```
input text    : a = 2 * 3 + 4;
```

Example

```
assignment ::= NAME '=' expr ';'
```

NAME '=' INT '*' INT '+' INT ';'
('a') (2) (3) (4)

Tokens

input text : a = 2 * 3 + 4;

Example

```
assignment ::= NAME '=' expr ';'
```

NAME '=' **expr** '*' INT '+' INT ';' expr.val = INT.val
('a') (2) (3) (4)
 ↑
NAME '=' **INT** '*' INT '+' INT ';' Tokens
('a') (2) (3) (4)

input text : a = 2 * 3 + 4;

Example

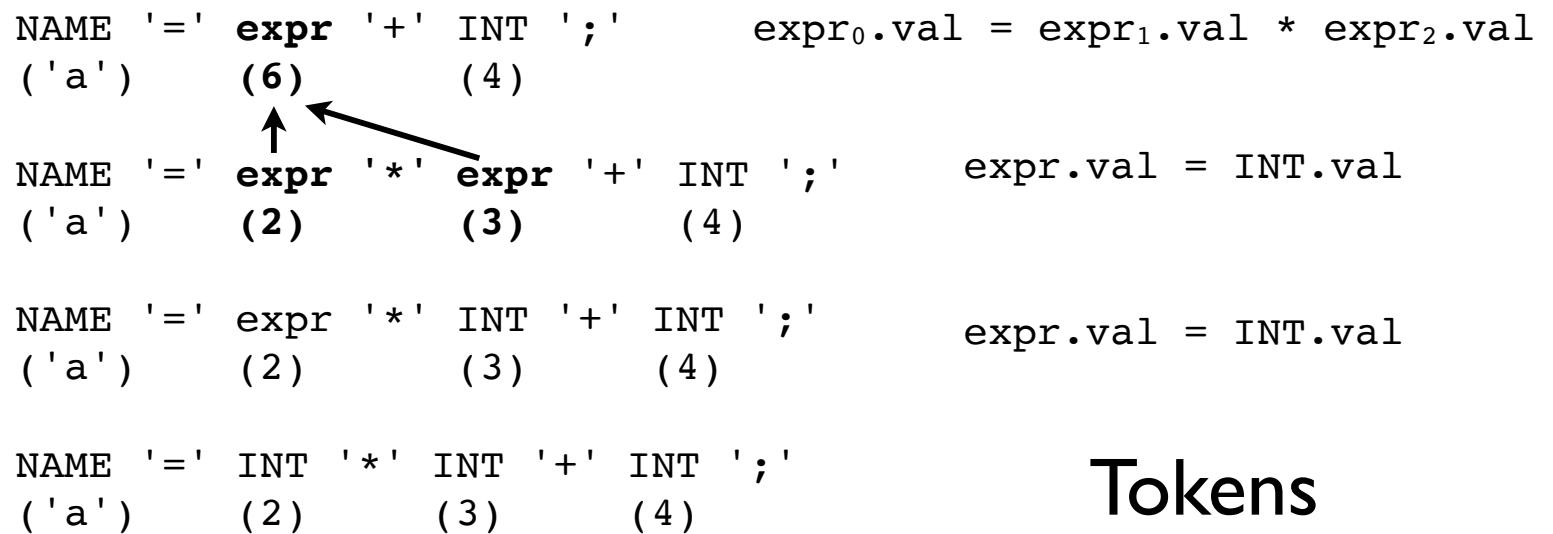
```
assignment ::= NAME '=' expr ';'
```

NAME '=' expr '*' expr '+' INT ';' ('a') (2) (3) (4)	expr.val = INT.val
NAME '=' expr '*' INT '+' INT ';' ('a') (2) (3) (4)	expr.val = INT.val
NAME '=' INT '*' INT '+' INT ';' ('a') (2) (3) (4)	Tokens

```
input text : a = 2 * 3 + 4;
```

Example

```
assignment ::= NAME '=' expr ';'
```



input text : a = 2 * 3 + 4;

Example

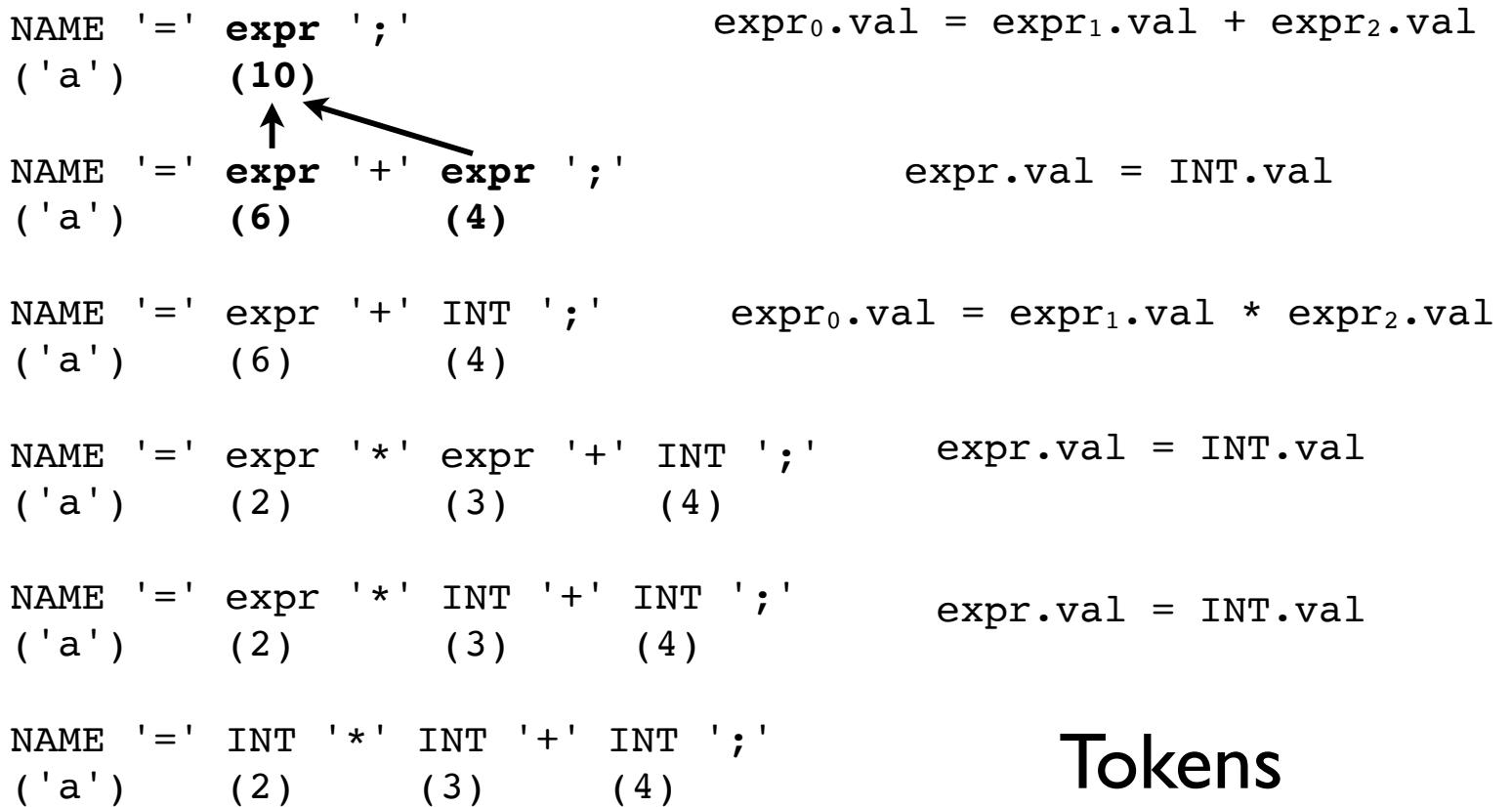
```
assignment ::= NAME '=' expr ';'
```

NAME '=' expr '+' expr ';'	expr.val = INT.val	
('a') (6) (4)		
↑		
NAME '=' expr '+' INT ';'	expr ₀ .val = expr ₁ .val * expr ₂ .val	
('a') (6) (4)		
NAME '=' expr '*' expr '+' INT ';'	expr.val = INT.val	
('a') (2) (3) (4)		
NAME '=' expr '*' INT '+' INT ';'	expr.val = INT.val	
('a') (2) (3) (4)		
NAME '=' INT '*' INT '+' INT ';'	Tokens	
('a') (2) (3) (4)		

```
input text : a = 2 * 3 + 4;
```

Example

```
assignment ::= NAME '=' expr ';'
```



input text : a = 2 * 3 + 4;

Example

```
assignment ::= NAME '=' expr ';'
```

assignment

↑
NAME '=' expr ';'
('a') (10)

vars[NAME] = expr.val

expr₀.val = expr₁.val + expr₂.val

NAME '=' expr '+' expr ';'
('a') (6) (4)

expr.val = INT.val

NAME '=' expr '+' INT ';' expr₀.val = expr₁.val * expr₂.val
('a') (6) (4)

NAME '=' expr '*' expr '+' INT ';' expr.val = INT.val
('a') (2) (3) (4)

NAME '=' expr '*' INT '+' INT ';' expr.val = INT.val
('a') (2) (3) (4)

NAME '=' INT '*' INT '+' INT ';' Tokens
('a') (2) (3) (4)

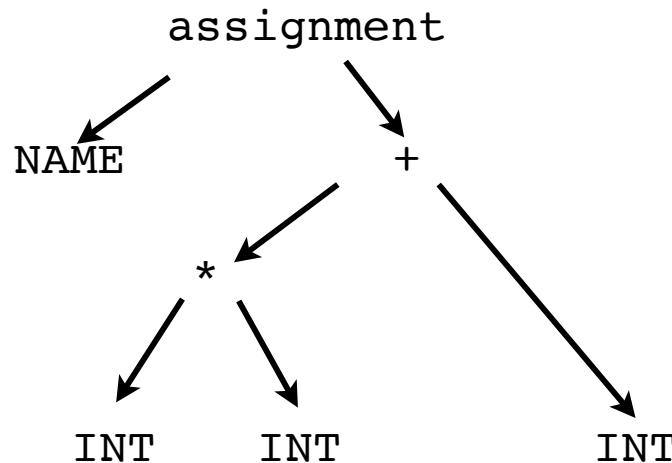
input text : a = 2 * 3 + 4;

DEMO

- Building a calculator with SLY

Abstract Syntax Trees

- Result of parsing is often a tree structure



- Captures the logical structure of the input
- Enables further analysis, checking, etc.

Tree Construction

- Actions can build data structures

Grammar

```
assignment ::= NAME '=' expr ';'  
  
expr0      ::= expr1'+' expr2  
                 | expr1'-' expr2  
                 | expr1'*' expr2  
                 | expr1'/' expr2  
                 | '(' expr1 ')' '  
                 | INT  
                 | FLOAT  
                 | NAME
```

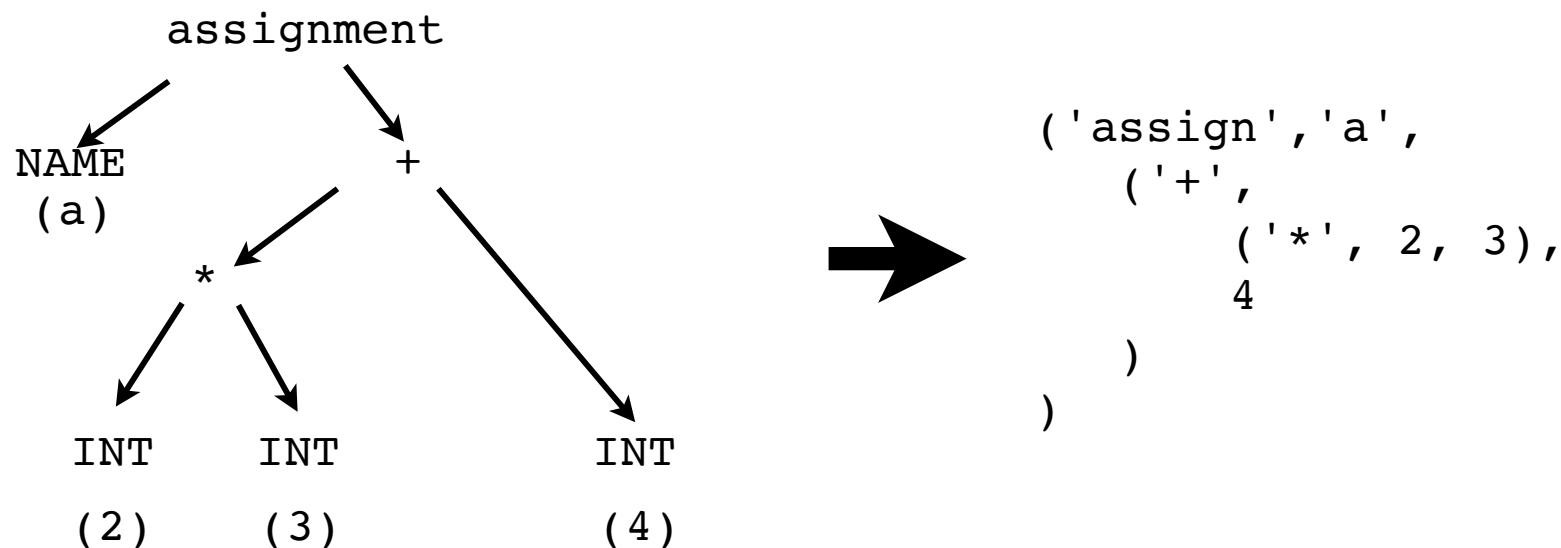
Actions

```
assignment = ('store', NAME, expr)  
  
expr0 = ('+', expr1, expr2)  
expr0 = ('-', expr1, expr2)  
expr0 = ('*', expr1, expr2)  
expr0 = ('/', expr1, expr2)  
expr0 = expr1  
expr = INT.val  
expr = FLOAT.val  
expr = ('load', NAME)
```

- Example: nested tuples

AST Representation

- Nodes represented by tuples, classes, etc.



DEMO

- Tree construction with SLY

AST Representation

- Nodes represented by tuples, classes, etc.

```
class Assignment:
```

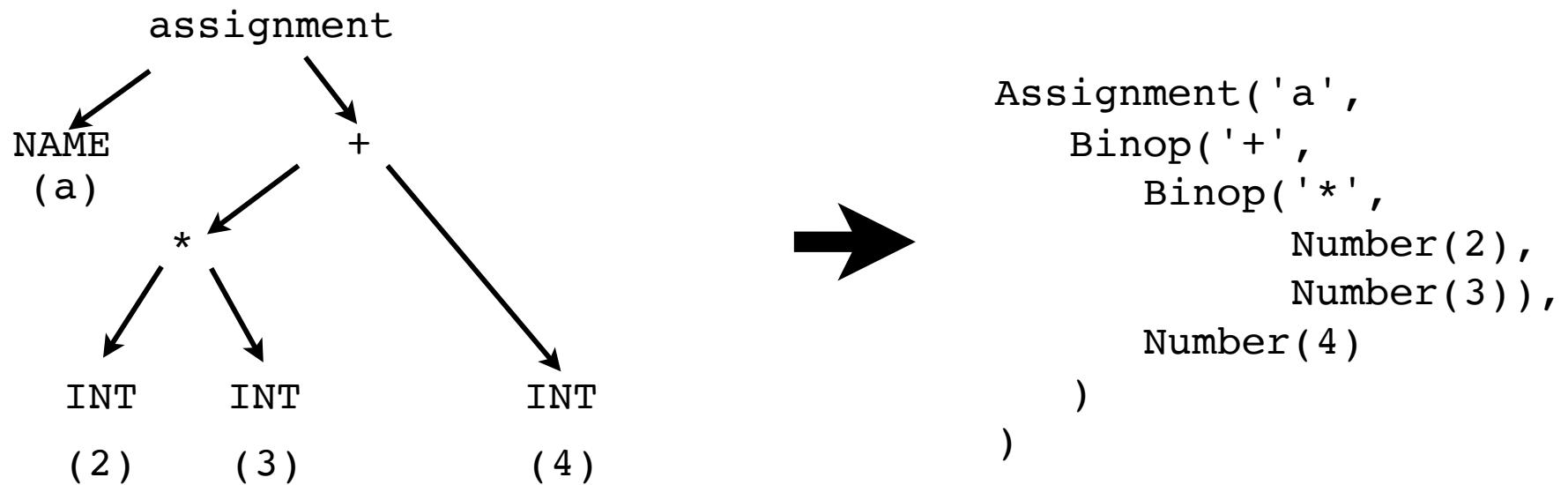
```
...
```

```
class Binop:
```

```
...
```

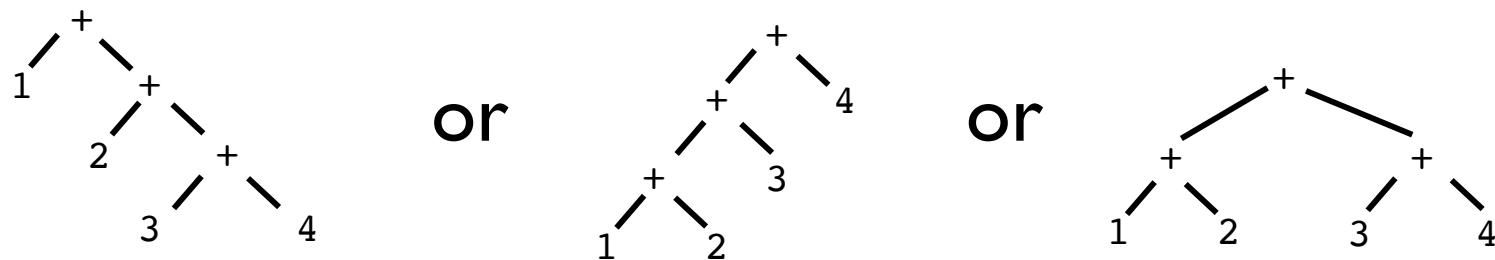
```
class Number:
```

```
...
```



Problem : Ambiguity

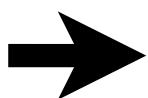
- Grammars must be unambiguous
- Consider: $1 + 2 + 3 + 4$
- There are many possible parses



- Ambiguity is bad!

Problem : Ambiguity

- To fix: must rewrite the grammar
- Example : (not only approach)

<pre>expr ::= expr '+' expr expr '-' expr expr '*' expr expr '/' expr '(' expr ')' ' INT FLOAT NAME</pre>		<pre>expr ::= expr '+' term expr '-' term expr '*' term expr '/' term term</pre> <pre>term ::= '(' expr ')' ' INT FLOAT NAME</pre>
---	--	---

- This is forcing left associativity

Problem : Ambiguity

- **Associativity example:**

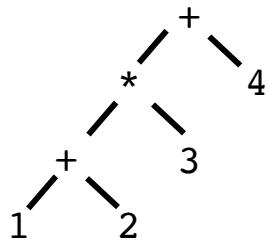
expr ::= expr '+' term	
expr '-' term	1 + 2 + 3 + 4
expr '*' term	
expr '/' term	expr + term
term	(expr + term) + term
	((expr + term) + term) + term
	((term + term) + term) + term
term ::= '(' expr ')'	
INT	
FLOAT	((1 + 2) + 3) + 4
NAME	



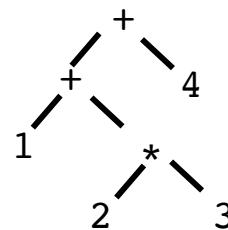
Notice how everything
is grouping to the left

Problem : Precedence

- Operators have different precedence
- Example : $1 + 2 * 3 + 4$



wrong



correct

- Multiplication is stronger than addition

Problem : Ambiguity

- To fix: more grammar rewriting

```
expr ::= expr '+' term  
      | expr '-' term  
      | expr '*' term  
      | expr '/' term  
      | term
```

```
term ::= '(' expr ')' '  
      | INT  
      | FLOAT  
      | NAME
```



```
expr ::= expr '+' term  
      | expr '-' term  
      | term
```

```
term ::= term '*' factor  
      | term '/' factor  
      | factor
```

```
factor ::= '(' expr ')' '  
      | INT  
      | FLOAT  
      | NAME
```

- Splitting into different precedence levels

Commentary

- Writing unambiguous grammars is hard
- There are techniques for refactoring
- There are mathematical proofs
- Not really our main focus here
- Be aware that it's an issue

Exercise 2

Project 2

LR Parsing

(Illustrated)

LR Parsing

- Three basic components:
 - A stack of grammar symbols and values.
 - Two operators: shift, reduce
 - An underlying state machine.
- Example

LR Example: Step I

stack

input

x = 3 + 4 * 5 \$end

Action:

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > factor(self, p)

LR Example: Step I

stack

input

x = 3 + 4 * 5 \$end

Action:

Grammar

- (1) assign : **NAME** EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > factor(self, p)

LR Example: Step 1

stack

NAME
'x'

input

= 3 + 4 * 5 \$end

Action: shift

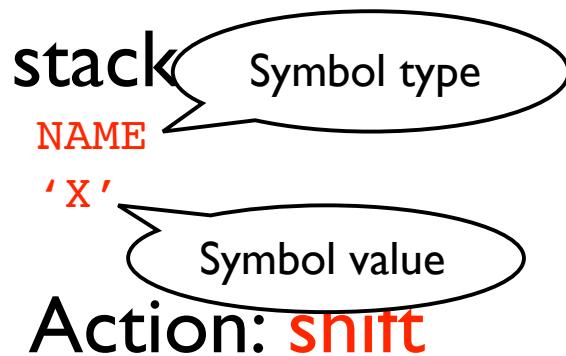
Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > factor(self, p)

LR Example: Step 1



input

= 3 + 4 * 5 \$end

Grammar

- (1) assign : **NAME** EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > factor(self, p)

LR Example: Step 2

stack

NAME
'x'

input

= 3 + 4 * 5 \$end

Action:

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > factor(self, p)

LR Example: Step 2

stack

NAME
'x'

input

= 3 + 4 * 5 \$end

Action:

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > factor(self, p)

LR Example: Step 2

stack

NAME EQUALS
'x' '='

input

3 + 4 * 5 \$end

Action: shift

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > factor(self, p)

LR Example: Step 3

stack

NAME EQUALS
'x' '='

input

3 + 4 * 5 \$end

Action:

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > factor(self, p)

LR Example: Step 3

stack

NAME EQUALS
'x' '='

input

3 + 4 * 5 \$end

Action:

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > factor(self, p)

LR Example: Step 3

stack

NAME EQUALS NUMBER
'x' '=' 3

input

+ 4 * 5 \$end

Action: shift

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > factor(self, p)

LR Example: Step 3

stack

NAME EQUALS NUMBER
'x' '=' 3

Action: shift

input

+ 4 * 5 \$end

```
@_(r'\d+')
def NUMBER(self, t):
    t.value = int(t.value)
    return t
```

Grammar

SET Rules

- | | |
|-------------------------------|--------------------|
| (1) assign : NAME EQUALS expr | -> assign(self, p) |
| (2) expr : expr PLUS term | -> expr(self, p) |
| (3) expr MINUS term | |
| (4) term | |
| (5) term : term TIMES factor | -> term(self, p) |
| (6) term DIVIDE factor | |
| (7) factor | |
| (8) factor : NUMBER | -> factor(self, p) |

LR Example: Step 4

stack

NAME EQUALS NUMBER
'x' '=' 3

input

+ 4 * 5 \$end

Action:

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > factor(self, p)

LR Example: Step 4

stack

NAME EQUALS **NUMBER**
'x' '=' 3

input

+ 4 * 5 \$end

Action:

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER**

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > **factor(self, p)**

LR Example: Step 4

stack

NAME EQUALS **NUMBER**
'x' '=' 3

input

+ 4 * 5 \$end

Action: reduce using rule 8

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER**

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > **factor(self, p)**

LR Example: Step 4

stack

NAME EQUALS factor
'x' '=' None

input

+ 4 * 5 \$end

Action: reduce using rule 8

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER**

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > **factor(self, p)**

LR Example: Step 4

stack

NAME EQUALS factor
'x' '=' None

Action: reduce

input

+ 4 * 5 \$end

This is None because
factor() didn't do anything.
More later.

Grammar

(1) assign : NAME EQUALS expr

(2) expr : expr PLUS term

(3)

(4) @_('NUMBER')

def factor(self, p):
 pass

(7)

(8) factor : NUMBER

SLY Rules

-> assign(self, p)

-> expr(self, p)

-> term(self, p)

-> factor(self, p)

LR Example: Step 5

stack

NAME EQUALS factor
'x' '=' None

input

+ 4 * 5 \$end

Action:

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > factor(self, p)

LR Example: Step 5

stack

NAME EQUALS **factor**
'x' '=' **None**

input

+ 4 * 5 \$end

Action:

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) **term** : term TIMES factor
- (6) | term DIVIDE factor
- (7) | **factor**
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > **term(self, p)**
- > factor(self, p)

LR Example: Step 5

stack

NAME EQUALS **factor**
'x' '=' **None**

input

+ 4 * 5 \$end

Action: **reduce using rule 7**

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) **term** : term TIMES factor
- (6) | term DIVIDE factor
- (7) | **factor**
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > **term(self, p)**
- > factor(self, p)

LR Example: Step 5

stack

NAME EQUALS **term**
'x' '=' None

input

+ 4 * 5 \$end

Action: **reduce using rule 7**

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) **term** : term TIMES factor
- (6) | term DIVIDE factor
- (7) | **factor**
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > **term(self, p)**
- > factor(self, p)

LR Example: Step 6

stack

NAME EQUALS term
'x' '=' None

input

+ 4 * 5 \$end

Action:

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > factor(self, p)

LR Example: Step 6

stack

NAME EQUALS **term**
'x' '=' **None**

input

+ 4 * 5 \$end

Action:

Grammar

- (1) assign : NAME EQUALS expr
- (2) **expr** : expr PLUS term
- (3) | expr MINUS term
- (4) | **term**
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > **expr(self, p)**
- > term(self, p)
- > factor(self, p)

LR Example: Step 6

stack

NAME EQUALS **term**
'x' '=' **None**

input

+ 4 * 5 \$end

Action: **reduce using rule 4**

Grammar

- (1) assign : NAME EQUALS expr
- (2) **expr** : expr PLUS term
- (3) | expr MINUS term
- (4) | **term**
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > **expr(self, p)**
- > term(self, p)
- > factor(self, p)

LR Example: Step 6

stack

NAME EQUALS **expr**
'x' '=' None

input

+ 4 * 5 \$end

Action: **reduce using rule 4**

Grammar

- (1) assign : NAME EQUALS expr
- (2) **expr** : expr PLUS term
- (3) | expr MINUS term
- (4) | **term**
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > **expr(self, p)**
- > term(self, p)
- > factor(self, p)

LR Example: Step 7

stack

NAME EQUALS expr	
'X' '='	None

input

+ 4 * 5 \$end

Action:

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > factor(self, p)

LR Example: Step 7

stack

NAME EQUALS expr
'x' '=' None

input

+ 4 * 5 \$end

Action: ???

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > factor(self, p)

LR Example: Step 7

stack

NAME EQUALS expr	
'x' '='	None

input

+ 4 * 5 \$end

Action: **shift**

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : **expr PLUS** term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > factor(self, p)

LR Example: Step 7

stack

NAME EQUALS expr **PLUS**
'x' '=' None '+'

input

4 * 5 \$end

Action: **shift**

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : **expr PLUS** term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > factor(self, p)

LR Example: Step 8

stack

NAME EQUALS expr PLUS
'X' '=' None '+'

input

← NUMBER

4 * 5 \$end

Action: shift

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > factor(self, p)

LR Example: Step 9

stack

NAME EQUALS expr PLUS NUMBER
'X' '=' None '+' 4

input

* 5 \$end

Action: reduce using rule 8

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER**

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > factor(self, p)

LR Example: Step 9

stack

NAME EQUALS expr PLUS factor
'X' '=' None '+' None

input

* 5 \$end

Action: reduce using rule 7

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) **term** : term TIMES factor
- (6) | term DIVIDE factor
- (7) | **factor**
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > **term(self, p)**
- > factor(self, p)

LR Example: Step 10

stack

NAME EQUALS expr PLUS term
'X' '=' None '+' None

input

TIMES

* 5 \$end

Action: shift

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > factor(self, p)

LR Example: Step 11

stack

NAME EQUALS expr PLUS term TIMES
'X' '=' None '+' None '*' ←

input

NUMBER

5 \$end

Action: shift

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > factor(self, p)

LR Example: Step 12

stack

NAME EQUALS expr PLUS term TIMES NUMBER
'x' '=' None '+' None '*' 5

input

\$end

Action: reduce using rule 8

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER**

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > **factor(self, p)**

LR Example: Step 13

stack

NAME EQUALS expr PLUS **term TIMES factor**
'x' '=' None '+' None '*' None

input

\$end

Action: reduce using rule 5

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor**
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)**
- > factor(self, p)

LR Example: Step 14

stack

NAME EQUALS **expr PLUS term**
'x' '=' None '+' None

input

\$end

Action: reduce using rule 2

Grammar

- (1) assign : NAME EQUALS expr
- (2) **expr** : **expr PLUS term**
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > **expr(self, p)**
- > term(self, p)
- > factor(self, p)

LR Example: Step 15

stack

```
NAME EQUALS expr  
'X'    '='      None
```

input

\$end

Action: reduce using rule 1

Grammar

```
(1) assign : NAME EQUALS expr  
(2) expr   : expr PLUS term  
(3)       | expr MINUS term  
(4)       | term  
(5) term   : term TIMES factor  
(6)       | term DIVIDE factor  
(7)       | factor  
(8) factor : NUMBER
```

SLY Rules

```
-> assign(self, p)  
-> expr(self, p)  
  
-> term(self, p)  
  
-> factor(self, p)
```

LR Example: Step 16

stack

assign
None

input

\$end

Action: Done.

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

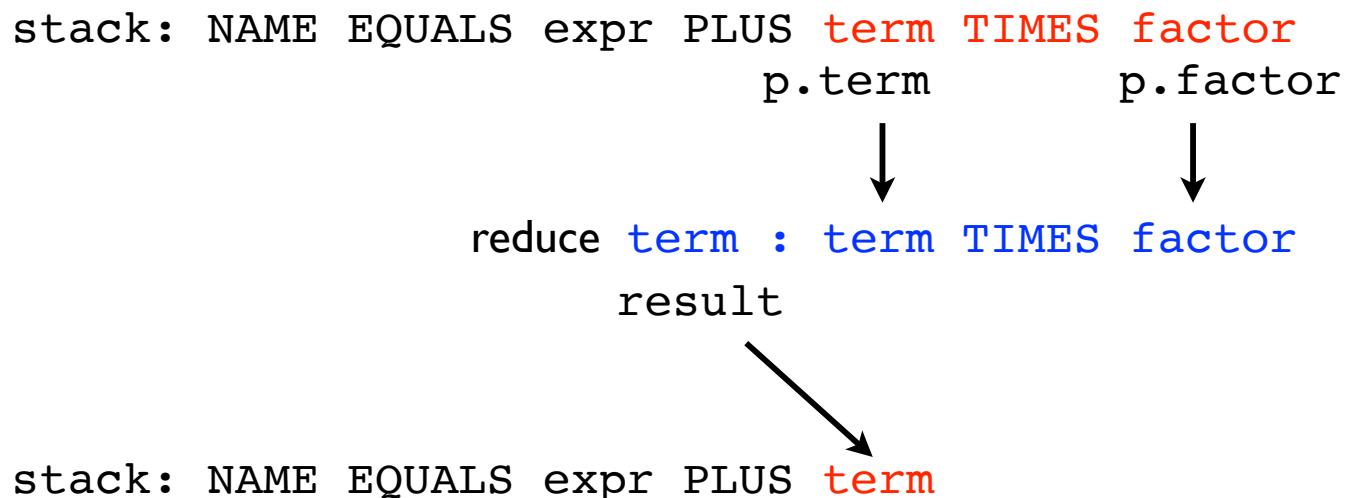
- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > factor(self, p)

SLY Rule Execution

- Rules are executed during reduction

```
@_('term TIMES factor')
def term(self, p):
    ...
    return result
```

- Parameter p refers to values on stack



Example: Calculator

```
@_( 'NAME EQUALS expr' )
def assign(self, p):
    print('Assigning', p.NAME, 'value', p.expr)

@_( 'expr PLUS term' )
def expr(self, p):
    return p.expr + p.term

@_( 'term TIMES factor' )
def term(self, p):
    return p.term * p.factor

@_( 'NUMBER' )
def factor(self, p):
    return p.NUMBER
```

LR Example: Step 4

stack

NAME EQUALS **NUMBER**
'x' '=' **3**

input

+ 4 * 5 \$end

Action:

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > factor(self, p)

LR Example: Step 4

stack

NAME EQUALS **NUMBER**
'x' '=' **3**

input

+ 4 * 5 \$end

Action:

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER**

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > **factor(self, p)**

LR Example: Step 4

stack

NAME EQUALS **NUMBER**
'x' '=' 3

input

+ 4 * 5 \$end

Action: reduce using rule 8

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER**

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > **factor(self, p)**

LR Example: Step 4

stack

NAME EQUALS factor
'x' '=' 3

input

+ 4 * 5 \$end

Action: reduce using rule 8

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS term
- (3) | expr MINUS term
- (4) | term
- (5) term : term TIMES factor
- (6) | term DIVIDE factor
- (7) | factor
- (8) factor : NUMBER**

SLY Rules

- > assign(self, p)
- > expr(self, p)
- > term(self, p)
- > **factor(self, p)**

LR Example: Step 4

stack

NAME EQUALS factor
'x' '=' 3

Action: **reduce**

input

+ 4 * 5 \$end

This retains its value because
of assignment in factor().

Grammar

(1) assign : NAME EQUALS expr

(2) expr : expr PLUS term

(3)

(4)

@_('NUMBER')

def factor(self, p):
 return p.NUMBER

(7)

(8) **factor : NUMBER**

SLY Rules

-> assign(self, p)

-> expr(self, p)

-> term(self, p)

-> **factor(self, p)**

Example: Parse Tree

```
@_( 'NAME EQUALS expr' )
def assign(self, p):
    return ('ASSIGN', p.NAME, p.expr)

@_( 'expr PLUS term' )
def expr(self, p):
    return ('+', p.expr, p.term)

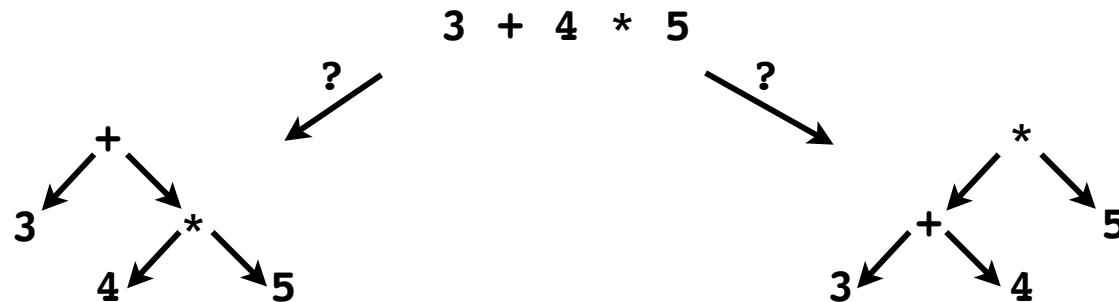
@_( 'term TIMES factor' )
def term(self, p):
    return ('*', p.term, p.factor)

@_( 'NUMBER' )
def factor(self, p):
    return ('NUM', p.NUMBER)
```

Ambiguous Grammars

```
@_('NAME EQUALS expr')
def assign(self, p):
    ...
    ...

@_('expr PLUS expr',
   'expr MINUS expr',
   'expr TIMES expr',
   'expr DIVIDE expr')
def expr(self, p):
    return (p[1], p.expr0, p.expr1)
```



Ambiguous Grammars

- Multiple possible parse trees
- Is reported as a “shift/reduce conflict”

Generating LALR parsing table...
16 shift/reduce conflicts

- May also get “reduce/reduce conflict”
- Probably most mysterious aspect of SLY

Shift/Reduce Conflict Explained

stack

NAME EQUALS expr PLUS expr

input

* 5 \$end

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS expr
- (3) | expr MINUS expr
- (4) | expr TIMES expr
- (5) | expr DIVIDE expr
- (6) | NUMBER

Possible Actions:

Shift/Reduce Conflict Explained

stack

NAME EQUALS **expr PLUS expr**

input

* 5 \$end

Grammar

- (1) assign : NAME EQUALS expr
- (2) **expr** : **expr PLUS expr**
- (3) | expr MINUS expr
- (4) | expr TIMES expr
- (5) | expr DIVIDE expr
- (6) | NUMBER

Possible Actions:

reduce using rule 2

Shift/Reduce Conflict Explained

stack

NAME EQUALS **expr PLUS expr**

input

* 5 \$end

NAME EQUALS expr
NAME EQUALS expr TIMES
NAME EQUALS expr TIMES NUMBER
NAME EQUALS expr TIMES expr
NAME EQUALS expr

Grammar

- (1) assign : NAME EQUALS expr
- (2) **expr** : **expr PLUS expr**
- (3) | expr MINUS expr
- (4) | expr TIMES expr
- (5) | expr DIVIDE expr
- (6) | NUMBER

Possible Actions:

reduce using rule 2

Shift/Reduce Conflict Explained

stack

NAME EQUALS expr PLUS **expr**

input

* 5 \$end

NAME EQUALS expr
NAME EQUALS expr TIMES
NAME EQUALS expr TIMES NUMBER
NAME EQUALS expr TIMES expr
NAME EQUALS expr

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS expr
- (3) | expr MINUS expr
- (4) | **expr TIMES expr**
- (5) | expr DIVIDE expr
- (6) | NUMBER

Possible Actions:

reduce using rule 2
shift TIMES

Shift/Reduce Conflict Explained

stack

NAME EQUALS expr PLUS **expr**

NAME EQUALS expr
NAME EQUALS expr TIMES
NAME EQUALS expr TIMES NUMBER
NAME EQUALS expr TIMES expr
NAME EQUALS expr

input

* 5 \$end

NAME EQUALS expr PLUS expr TIMES
NAME EQUALS expr PLUS expr TIMES NUMBER
NAME EQUALS expr PLUS expr TIMES expr
NAME EQUALS expr PLUS expr
NAME EQUALS expr

Grammar

- (1) assign : NAME EQUALS expr
- (2) expr : expr PLUS expr
- (3) | expr MINUS expr
- (4) | **expr TIMES expr**
- (5) | expr DIVIDE expr
- (6) | NUMBER

Possible Actions:

reduce using rule 2
shift TIMES

Shift/reduce resolution

- Default action is to always shift
- Can sometimes control with precedence

```
class MyParser(Parser):  
    ...  
    precedence = (  
        ('left','PLUS','MINUS'),  
        ('left','TIMES','DIVIDE'),  
    )  
  
    @_('expr PLUS expr',  
        'expr MINUS expr',  
        'expr TIMES expr',  
        'expr DIVIDE expr')  
    def expr(self, p):  
        return BinOp(p[1], p.expr0, p.expr1)
```

Reduce/Reduce Conflict Explained

stack

NAME EQUALS NUMBER

input

\$end

Grammar

- (1) assign : NAME EQUALS expr
- (2) | NAME EQUALS NUMBER
- (3) expr : expr PLUS expr
- (4) | expr MINUS expr
- (5) | expr TIMES expr
- (6) | expr DIVIDE expr
- (7) | NUMBER

Possible Actions:

Reduce/Reduce Conflict Explained

stack

NAME EQUALS NUMBER

input

\$end

Grammar

- (1) **assign** : NAME EQUALS expr
- (2) | **NAME EQUALS NUMBER**
- (3) expr : expr PLUS expr
- (4) | expr MINUS expr
- (5) | expr TIMES expr
- (6) | expr DIVIDE expr
- (7) | NUMBER

Possible Actions:

reduce using rule 2

Reduce/Reduce Conflict Explained

stack

assign

input

\$end

Grammar

- (1) **assign** : NAME EQUALS expr
- (2) | NAME EQUALS NUMBER
- (3) expr : expr PLUS expr
- (4) | expr MINUS expr
- (5) | expr TIMES expr
- (6) | expr DIVIDE expr
- (7) | NUMBER

Possible Actions:

reduce using rule 2

Reduce/Reduce Conflict Explained

stack

assign

NAME EQUALS NUMBER

input

\$end

Grammar

- (1) assign : NAME EQUALS expr
- (2) | NAME EQUALS NUMBER
- (3) **expr** : expr PLUS expr
- (4) | expr MINUS expr
- (5) | expr TIMES expr
- (6) | expr DIVIDE expr
- (7) | **NUMBER**

Possible Actions:

- reduce using rule 2
- reduce using rule 7

Reduce/Reduce Conflict Explained

stack

assign

NAME EQUALS expr

input

\$end

Grammar

- (1) assign : NAME EQUALS expr
- (2) | NAME EQUALS NUMBER
- (3) expr : expr PLUS expr
- (4) | expr MINUS expr
- (5) | expr TIMES expr
- (6) | expr DIVIDE expr
- (7) | NUMBER

Possible Actions:

- reduce using rule 2
- reduce using rule 7

Error handling/recovery

- Syntax errors first fed through error()

```
def error(self, p):
    print('Syntax error')
```

- Then an ‘error’ symbol is shifted onto stack
- Stack is unwound until error is consumed

Error recovery

stack

NAME EQUALS expr PLUS expr
'X' '=' 3 '+' 4

input

5 \$end

Grammar

- (1) assign : NAME EQUALS expr
- (2) | NAME EQUALS error
- (3) expr : expr PLUS expr
- (4) | expr MINUS expr
- (5) | expr TIMES expr
- (6) | expr DIVIDE expr
- (7) | NUMBER

Error recovery

stack

NAME EQUALS expr PLUS expr
'X' '=' 3 '+' 4

input

5 \$end
↑
syntax error

Grammar

- (1) assign : NAME EQUALS expr
- (2) | NAME EQUALS error
- (3) expr : expr PLUS expr
- (4) | expr MINUS expr
- (5) | expr TIMES expr
- (6) | expr DIVIDE expr
- (7) | NUMBER

Error recovery

stack

NAME EQUALS expr PLUS expr
'X' '=' 3 '+' 4

input

5 \$end
↑
syntax error

```
def error(self, p):  
    print('Syntax error')
```

Grammar

- (1) assign : NAME EQUALS expr
- (2) | NAME EQUALS error
- (3) expr : expr PLUS expr
- (4) | expr MINUS expr
- (5) | expr TIMES expr
- (6) | expr DIVIDE expr
- (7) | NUMBER

Error recovery

stack

NAME EQUALS expr PLUS expr **error**
'X' '=' 3 '+' 4

input

\$end

Grammar

- (1) assign : NAME EQUALS expr
- (2) | NAME EQUALS error
- (3) expr : expr PLUS expr
- (4) | expr MINUS expr
- (5) | expr TIMES expr
- (6) | expr DIVIDE expr
- (7) | NUMBER

Error recovery

stack

NAME EQUALS expr PLUS **error**
'x' '=' 3 '+'

input

\$end

Grammar

- (1) assign : NAME EQUALS expr
- (2) | NAME EQUALS error
- (3) expr : expr PLUS expr
- (4) | expr MINUS expr
- (5) | expr TIMES expr
- (6) | expr DIVIDE expr
- (7) | NUMBER

Error recovery

stack

NAME EQUALS expr **error**
'X' '=' 3

input

\$end

Grammar

- (1) assign : NAME EQUALS expr
- (2) | NAME EQUALS error
- (3) expr : expr PLUS expr
- (4) | expr MINUS expr
- (5) | expr TIMES expr
- (6) | expr DIVIDE expr
- (7) | NUMBER

Error recovery

stack

NAME EQUALS **error**
'X' '='

input

\$end

Grammar

- (1) assign : NAME EQUALS expr
- (2) | NAME EQUALS error
- (3) expr : expr PLUS expr
- (4) | expr MINUS expr
- (5) | expr TIMES expr
- (6) | expr DIVIDE expr
- (7) | NUMBER

Error recovery

stack

NAME EQUALS error
'X' '='

input

\$end

Grammar

- (1) **assign** : NAME EQUALS expr
- (2) | NAME EQUALS error
- (3) **expr** : expr PLUS expr
- (4) | expr MINUS expr
- (5) | expr TIMES expr
- (6) | expr DIVIDE expr
- (7) | NUMBER

```
@_('NAME EQUALS error')
def assign(self, p):
    print('Bad assignment')
```

Error recovery

stack

assign

input

\$end

Grammar

- (1) **assign** : NAME EQUALS expr
- (2) : NAME EQUALS error
- (3) expr : expr PLUS expr
- (4) : expr MINUS expr
- (5) : expr TIMES expr
- (6) : expr DIVIDE expr
- (7) : NUMBER

```
@_('NAME EQUALS error')
def assign(self, p):
    print('Bad assignment')
```

Debugging Output

- SLY can create a debugging file

```
class MyParser(Parser):  
    debugfile = 'parser.out'  
    ...
```

- Reading it involves voodoo magic
- Can help resolve parsing conflicts

Debugging Output

Grammar

```

Rule 1  statement -> NAME = expression
Rule 2  statement -> expression
Rule 3  expression -> expression + expression
Rule 4  expression -> expression - expression
Rule 5  expression -> expression * expression
Rule 6  expression -> expression / expression
Rule 7  expression -> NUMBER

```

Terminals, with rules where they appear

```

*          : 5
+          : 3
-          : 4
/          : 6
=          : 1
NAME       : 1
NUMBER     : 7
error      :

```

Nonterminals, with rules where they appear

```

expression   : 1 2 3 3 4 4 5 5 6 6
statement    : 0

```

Parsing method: LALR

state 0

```

(0) S' -> . statement
(1) statement -> . NAME = expression
(2) statement -> . expression
(3) expression -> . expression + expression
(4) expression -> . expression - expression
(5) expression -> . expression * expression
(6) expression -> . expression / expression
(7) expression -> . NUMBER

```

```

NAME        shift and go to state 1
NUMBER      shift and go to state 2

```

```

expression   shift and go to state 4
statement    shift and go to state 3

```

state 1

```

(1) statement -> NAME . = expression
=           shift and go to state 5

```

state 10

```

(1) statement -> NAME = expression .
(3) expression -> expression . + expression
(4) expression -> expression . - expression
(5) expression -> expression . * expression
(6) expression -> expression . / expression

```

```

$end        reduce using rule 1 (statement -> NAME = expression .)
+           shift and go to state 7
-           shift and go to state 6
*           shift and go to state 8
/           shift and go to state 9

```

state 11

```

(4) expression -> expression - expression .
(3) expression -> expression . + expression
(4) expression -> expression . - expression
(5) expression -> expression . * expression
(6) expression -> expression . / expression

```

```

! shift/reduce conflict for + resolved as shift.
! shift/reduce conflict for - resolved as shift.
! shift/reduce conflict for * resolved as shift.
! shift/reduce conflict for / resolved as shift.
$end        reduce using rule 4 (expression -> expression - expression .)
+           shift and go to state 7
-           shift and go to state 6
*           shift and go to state 8
/           shift and go to state 9
! +
! -          [ reduce using rule 4 (expression -> expression - expression .) ]
! *          [ reduce using rule 4 (expression -> expression - expression .) ]
! /          [ reduce using rule 4 (expression -> expression - expression .) ]

```

Debugging Output

```
...
state 11

(4) expression -> expression - expression .
(3) expression -> expression . + expression
(4) expression -> expression . - expression
(5) expression -> expression . * expression
(6) expression -> expression . / expression

! shift/reduce conflict for + resolved as shift.
! shift/reduce conflict for - resolved as shift.
! shift/reduce conflict for * resolved as shift.
! shift/reduce conflict for / resolved as shift.
$end      reduce using rule 4 (expression -> expression - expression .)
+
shift and go to state 7
-
shift and go to state 6
*
shift and go to state 8
/
shift and go to state 9

! +
[ reduce using rule 4 (expression -> expression - expression .) ]
! -
[ reduce using rule 4 (expression -> expression - expression .) ]
! *
[ reduce using rule 4 (expression -> expression - expression .) ]
! /
[ reduce using rule 4 (expression -> expression - expression .) ]
...
= shift and go to state 5
```

Part 3

Type Checking

Types

- Programming languages have different types of data and objects

```
a = 42          # int
b = 4.2         # float
c = "fortytwo" # str
d = [1,2,3]     # list
e = {'a':1,'b':2} # dict
...

```

- Each type has different capabilities

```
>>> a - 10
32
>>> c - "ten"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'st
>>>
```

What is a Type?

- Ultimately relates to representation of objects

```
int a = 42;  
short b = 42;  
long c = 42;  
float d = 4.2;
```

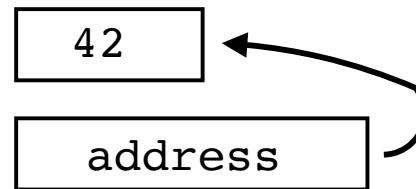
00	00	00	2a				
00	2a						
00	00	00	00	00	00	00	2a
40	10	cc	cc	cc	cc	cc	cd

- It directly relates to how raw data gets handled during computation
- Low-level operations on hardware.
- CPUs have limited capabilities

Derived Types

- Pointers/References

```
int a = 42;
```



```
int *b = &a;
```

- Arrays

```
int items[4];
```



- Structures

```
struct Point {  
    int x;  
    int y;  
}
```



Complexity: Composition

- Types combine together in arbitrary ways

```
int a;      // Base type
int *b;     // Pointer to int
int **c;    // Pointer to pointer to int
int d[4];   // Array of int
int *d[4]; // Array of pointers to int
```

- Might also have qualifiers (const, mutable, etc.)
- Comment: This gets quite complicated. How do you structure and reason about it?
- The topic of a different course

Complexity: Composition

```
class BaseType:  
    def __init__(self, name):  
        self.name = name  
  
class PointerTo:  
    def __init__(self, type):  
        self.type = type  
  
class ArrayOf:  
    def __init__(self, type, size):  
        self.type = type  
        self.size = size  
  
/* int *[10] */  
ArrayOf(PointerTo(BaseType('int')), 10)
```

Type Checking

- Verifying that actions found in a program are valid (enforcing semantics)
- Most of it is common sense
 - Can't read an undefined variable
 - Can't do operations (+,-,*,/) if not supported by the underlying datatype
 - Can't overwrite immutable data
 - Array indices must be integers

Dynamic Typing

- Type rules are enforced at run-time
- All objects carry type-information in execution

```
>>> a = 42
>>> a.__class__
<class 'int'>
>>> a + 10
52
>>> a.__add__(10)
52
>>> a.__add__('hello')
NotImplemented
>>> a + 'hello'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int'
and 'str'
>>>
```

Static Typing

- Type rules are enforced at compile-time
- Code is annotated with explicit types

```
/* C */  
int fact(int n) {  
    int result = 1;  
    while (n > 0) {  
        result *= n;  
        n--;  
    }  
}
```

- Compiler executes a "proof of correctness"
- Types are "discarded" during execution

Note

- Most programming languages involve a mix of both techniques (static/dynamic)
- Compiler does as much as it can
- Certain checks may be forced to run-time
- Example: Array-bounds checking

Rule Specification

- Type checking rules are associated with the grammar rules (informal notation)

```
assignment ::= name '=' expr ';'  
           - name must be declared as variable  
           - name.type == expr.type  
  
expr ::= expr1 '+' expr2 ';'  
       - expr1.type == expr2.type  
       - '+' operator must be supported  
Set: expr.type = result type
```

- Type checking phase of compiler involves walking the AST and enforcing the rules

How to Type Check

- A Few Basic Requirements:
 - Must be able to specify types
 - Need a symbol table (to record info)
 - Must walk the AST and enforce rules

Type Specification

- Types are "labels" that get attached to names:

```
float x
int fact(int n);
string name;
```

- Types have names (there is syntax for typing it)
- Keep in mind that the "type" could be much more complex (pointers, arrays, structs, etc.).

Type Specification

- Types must be comparable

```
int != float
```

- A major part of type-checking is finding type-mismatches in the code (comparing types)
- Might be simple name comparison
- Might be much more complex

Type Specification

- Types have different capabilities

```
int:  
    binary_ops = { '+', '-', '*', '/' },  
    unary_ops = { '+', '-' }
```

```
string:  
    binary_ops = { '+' },  
    unary_ops = {}
```

- Checker will consult when validating

Symbol Tables

- Symbol table is a mapping that records information about identifiers in a program

```
symbols = {  
    'a'      : Variable(),  
    'fact'   : Function(),  
    ...  
}
```

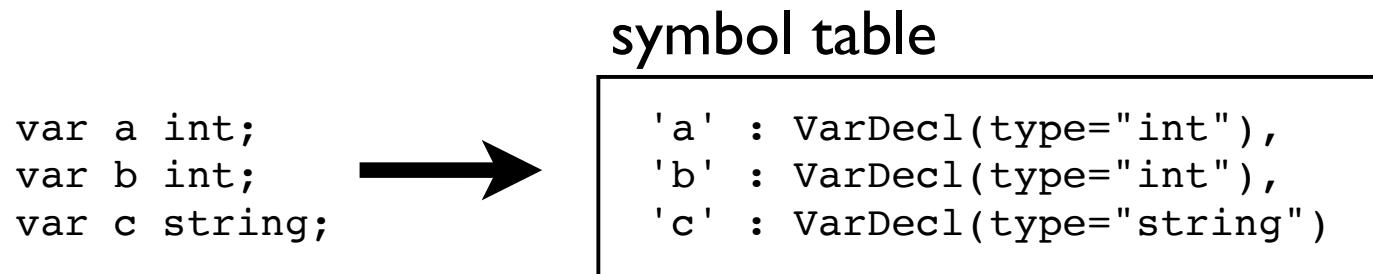
- Whenever a name is encountered, symbol table is consulted to get more information about it
- Symbol table records the "known knowns" about the program

AST Walking

- Generally a depth-first traversal of the AST
- Symbol table gets updated and consulted as you go along
- AST is annotated to propagate information
- Errors reported (if any)

Example:

- Example: Variable declarations:

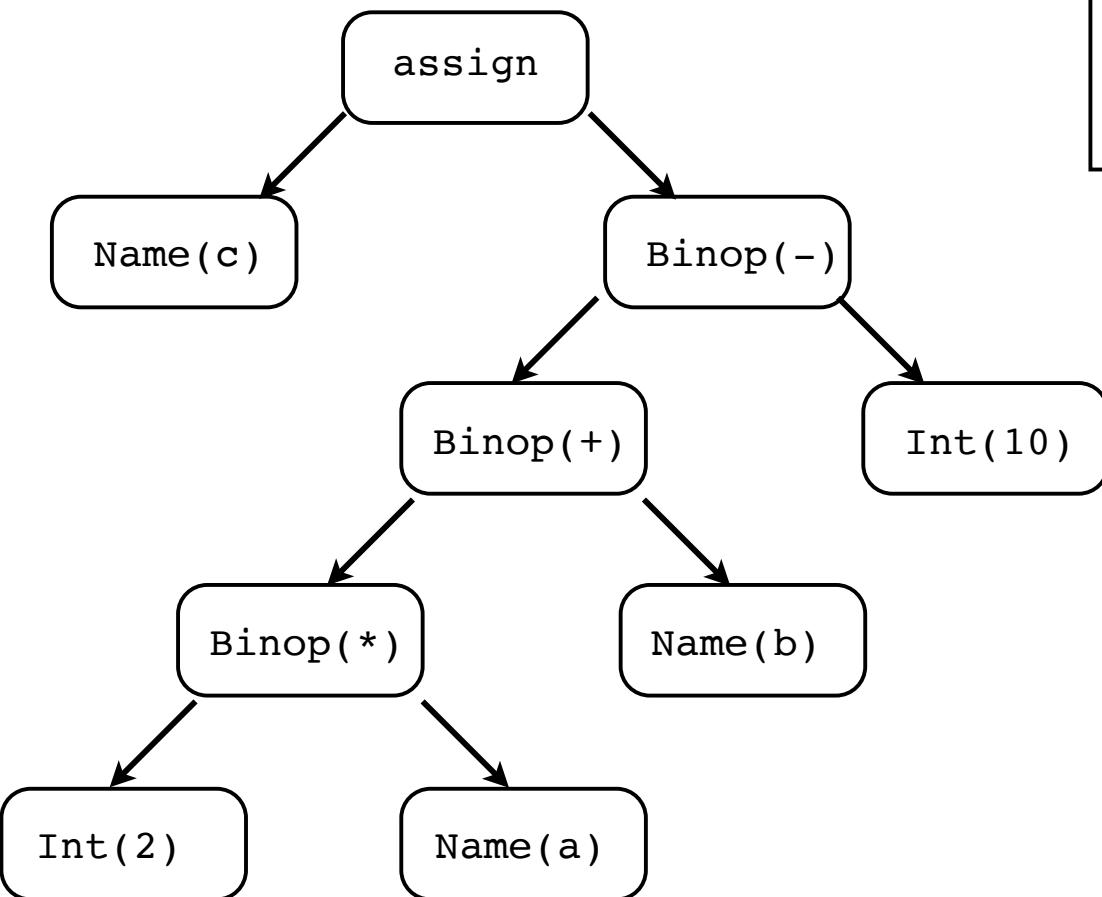


- The purpose of declarations is to establish type (and scope)
- Now, consider the parsing of this statement

c = 2*a + b - 10;

AST Annotation

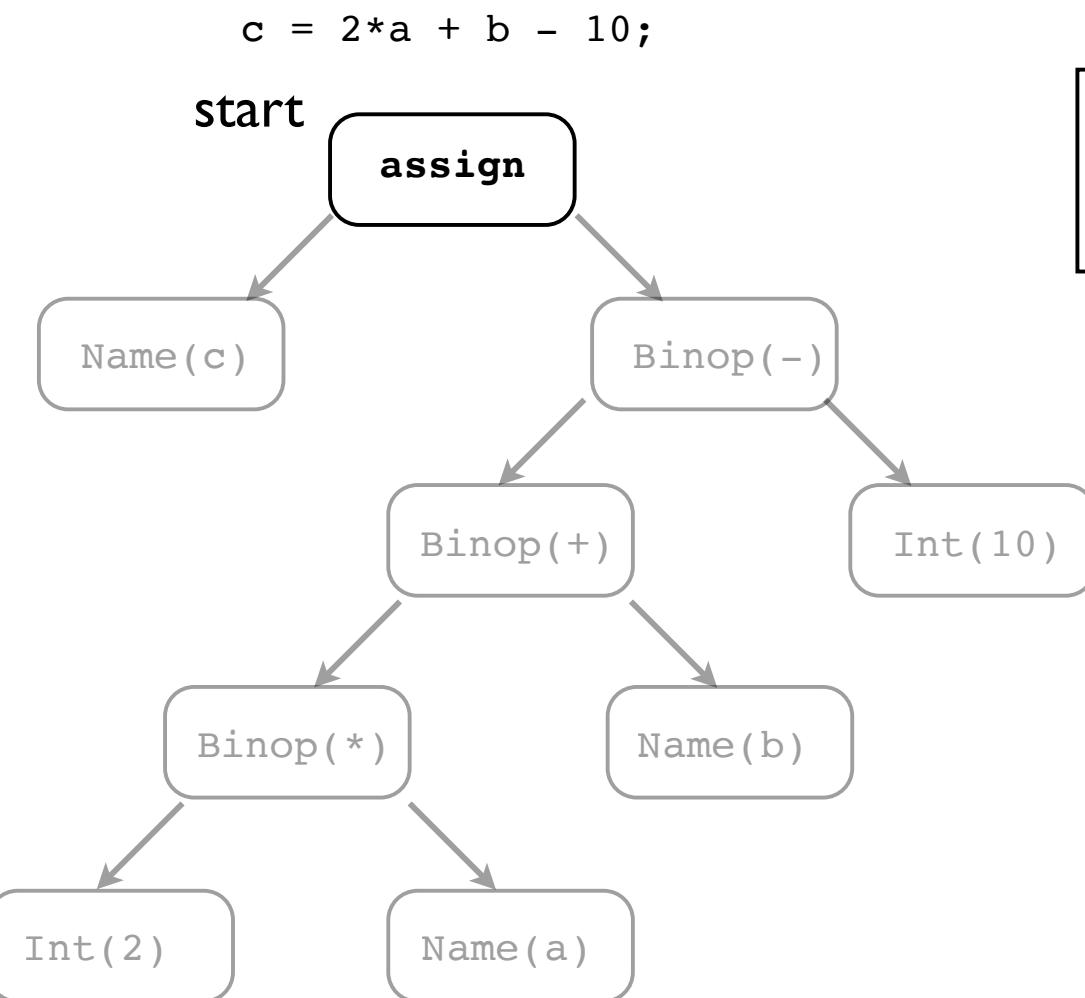
c = 2*a + b - 10;



symbol table

```
'a' : VarDecl(type="int"),  
'b' : VarDecl(type="int"),  
'c' : VarDecl(type="string")
```

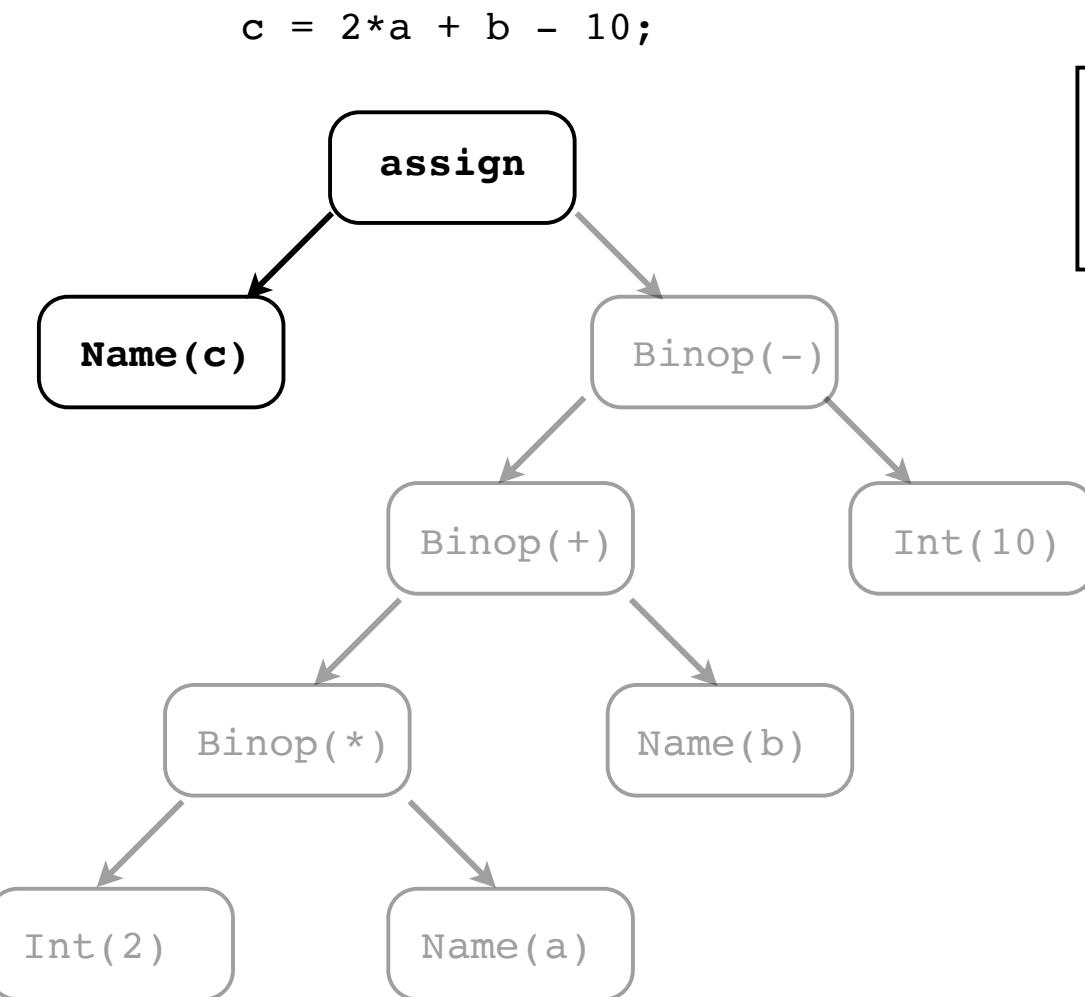
AST Annotation



symbol table

```
'a' : VarDecl(type="int"),  
'b' : VarDecl(type="int"),  
'c' : VarDecl(type="string")
```

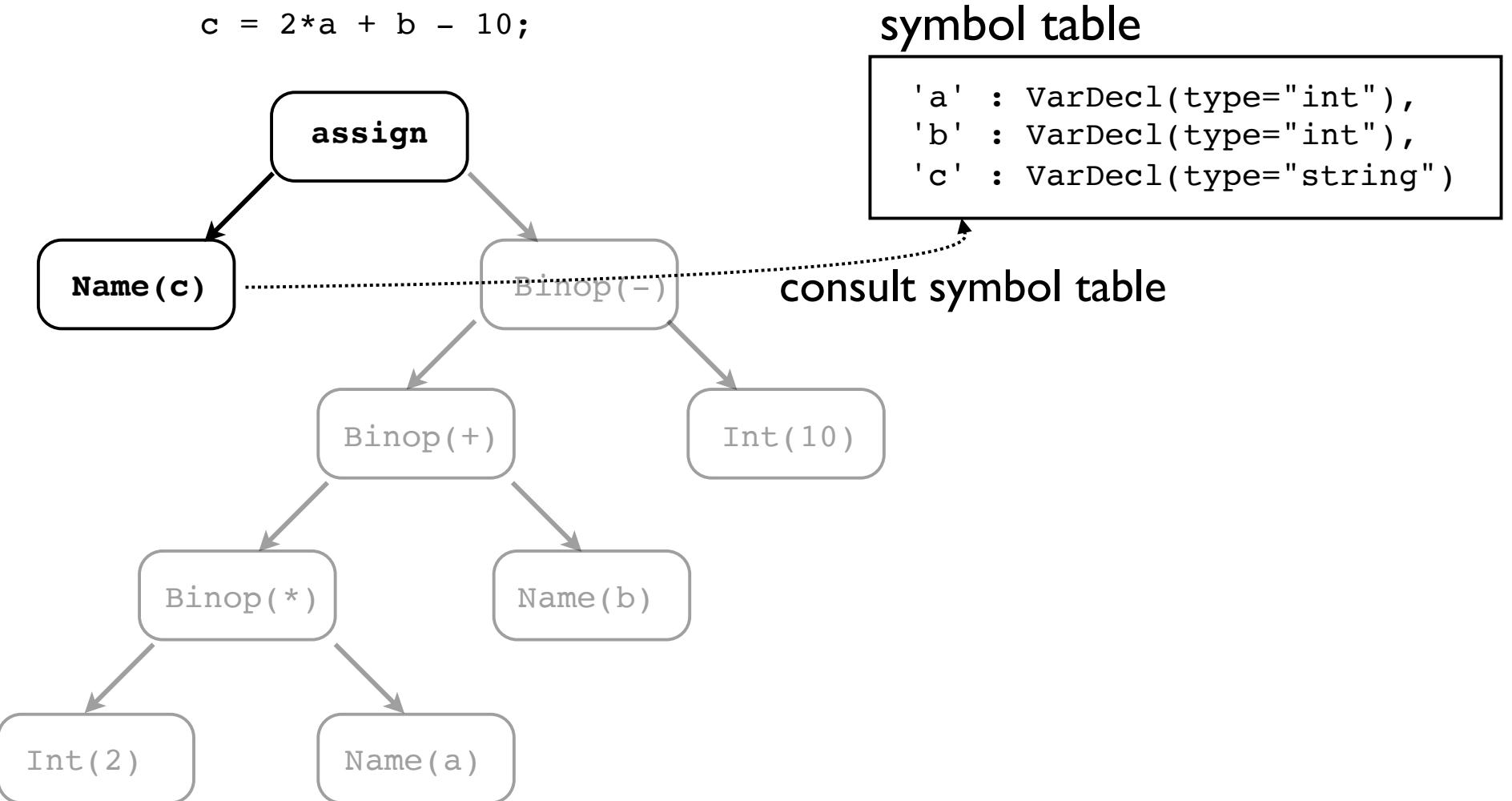
AST Annotation



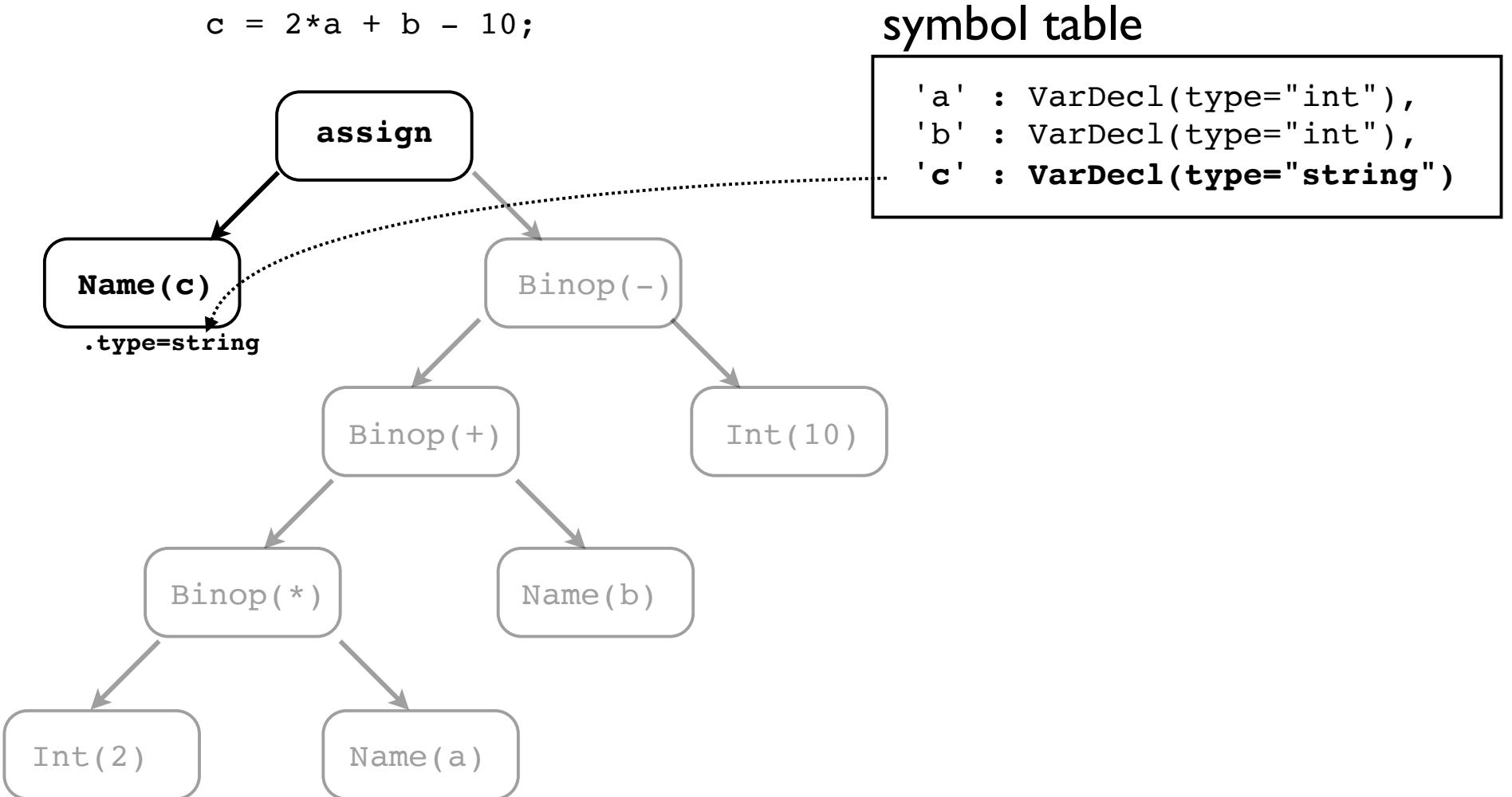
symbol table

```
'a' : VarDecl(type="int"),  
'b' : VarDecl(type="int"),  
'c' : VarDecl(type="string")
```

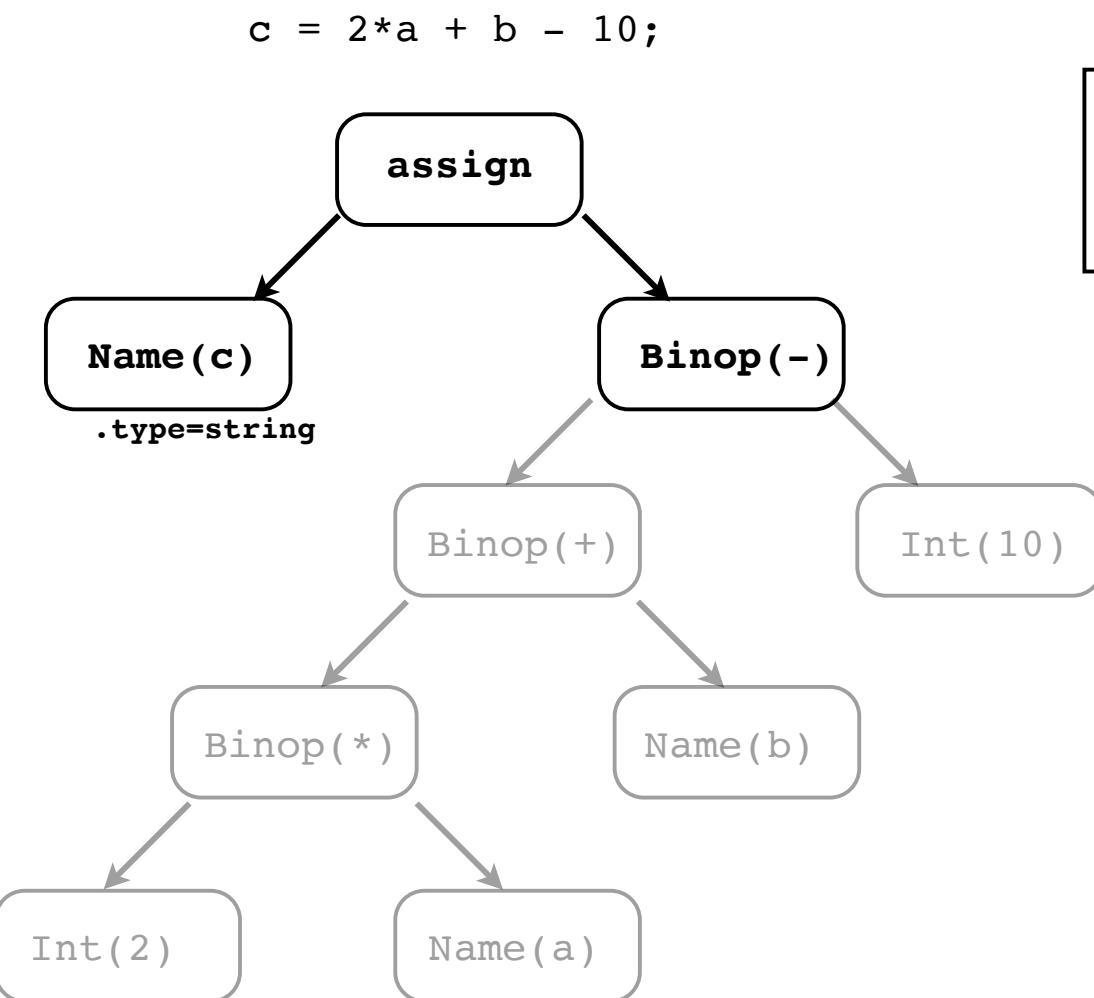
AST Annotation



AST Annotation



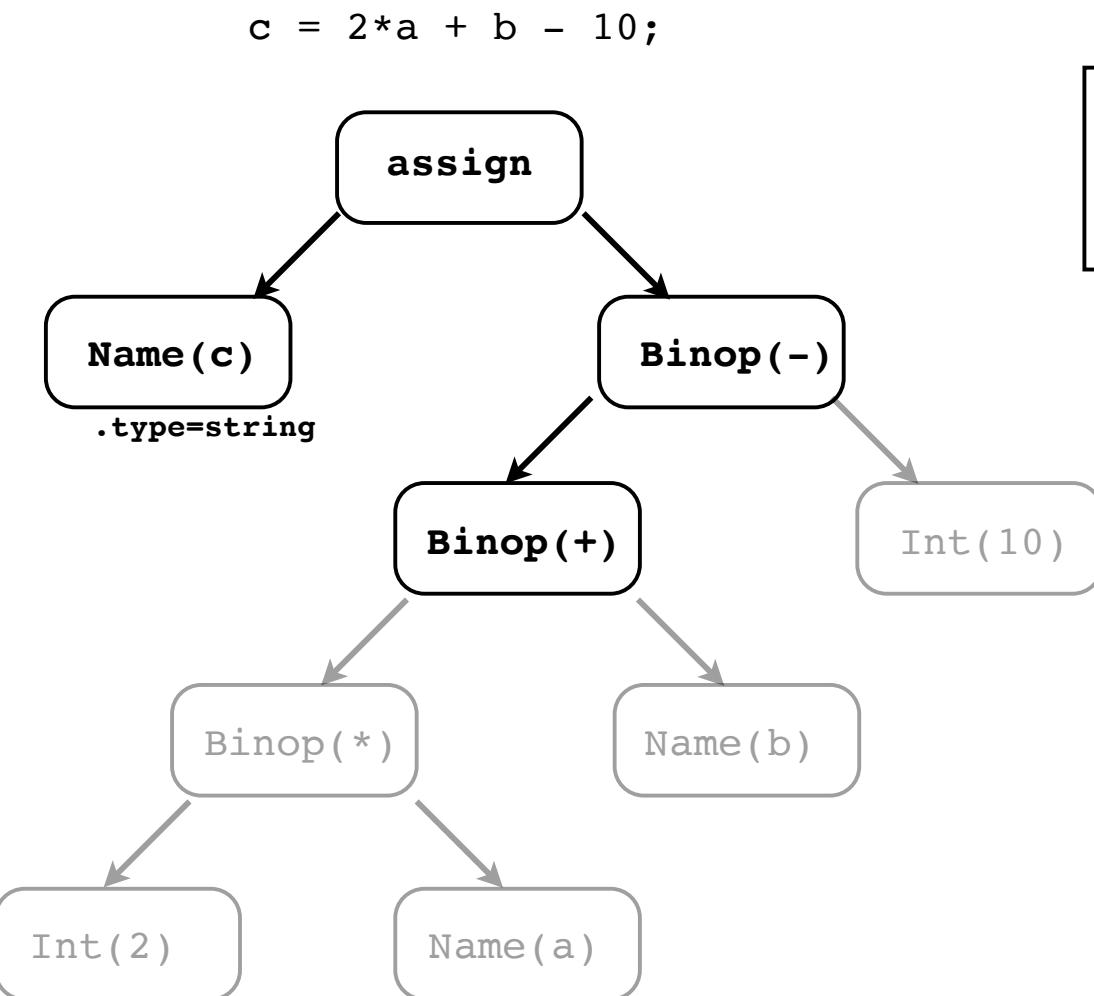
AST Annotation



symbol table

```
'a' : VarDecl(type="int"),  
'b' : VarDecl(type="int"),  
'c' : VarDecl(type="string")
```

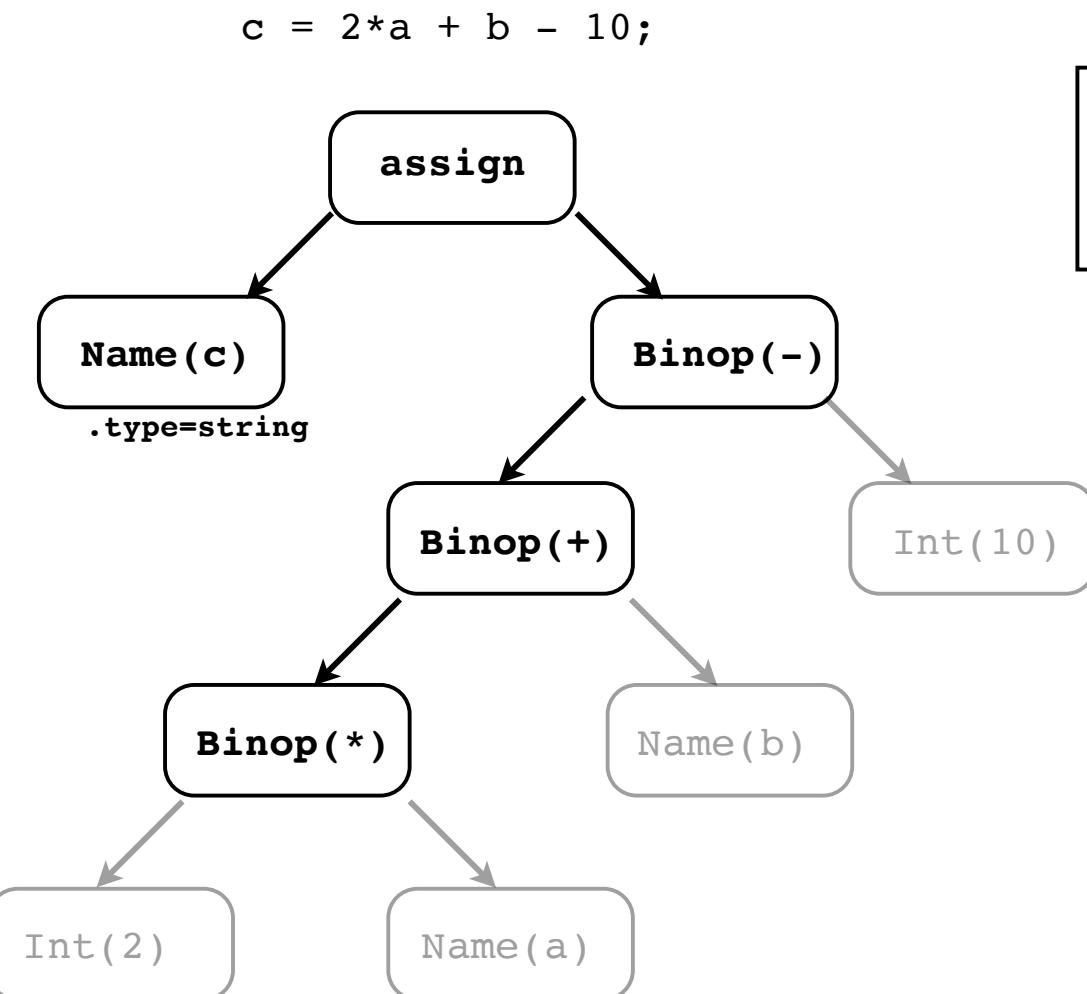
AST Annotation



symbol table

```
'a' : VarDecl(type="int"),  
'b' : VarDecl(type="int"),  
'c' : VarDecl(type="string")
```

AST Annotation



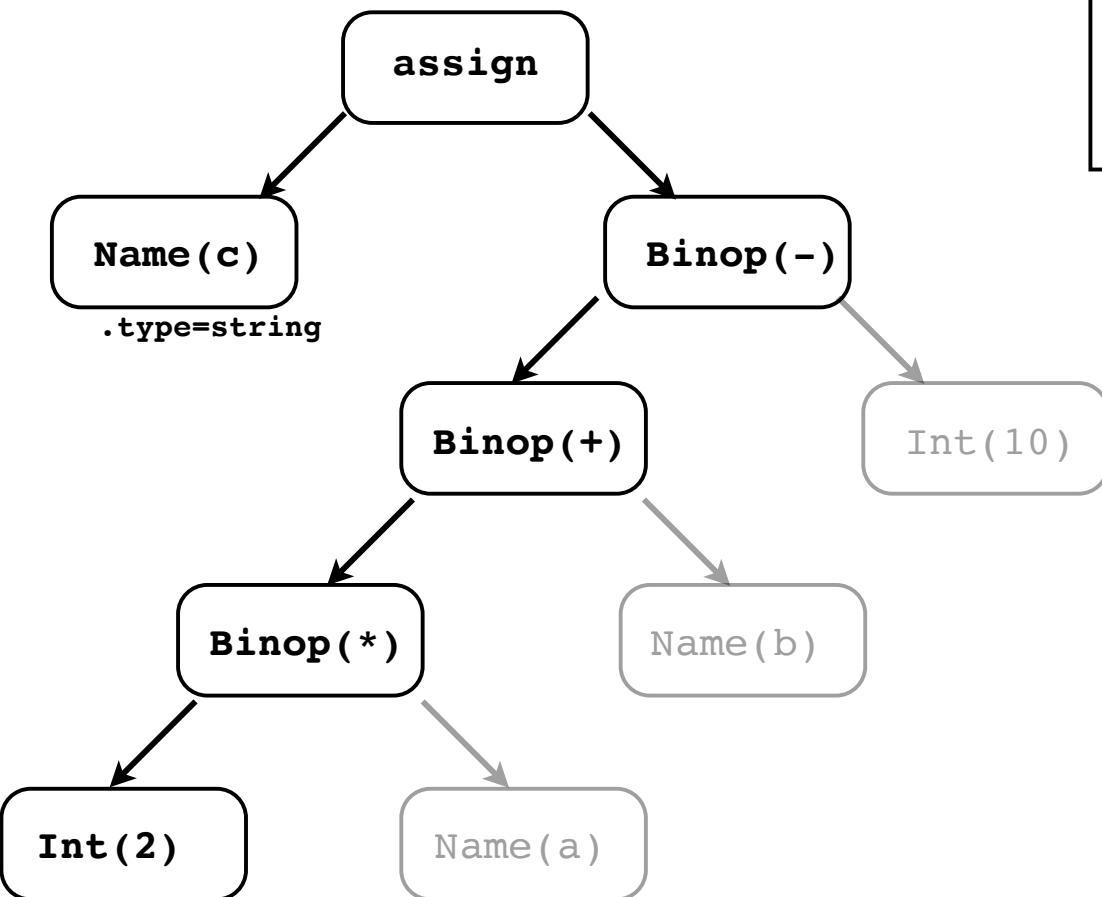
symbol table

```
'a' : VarDecl(type="int"),  
'b' : VarDecl(type="int"),  
'c' : VarDecl(type="string")
```

Note: depth-first traversal

AST Annotation

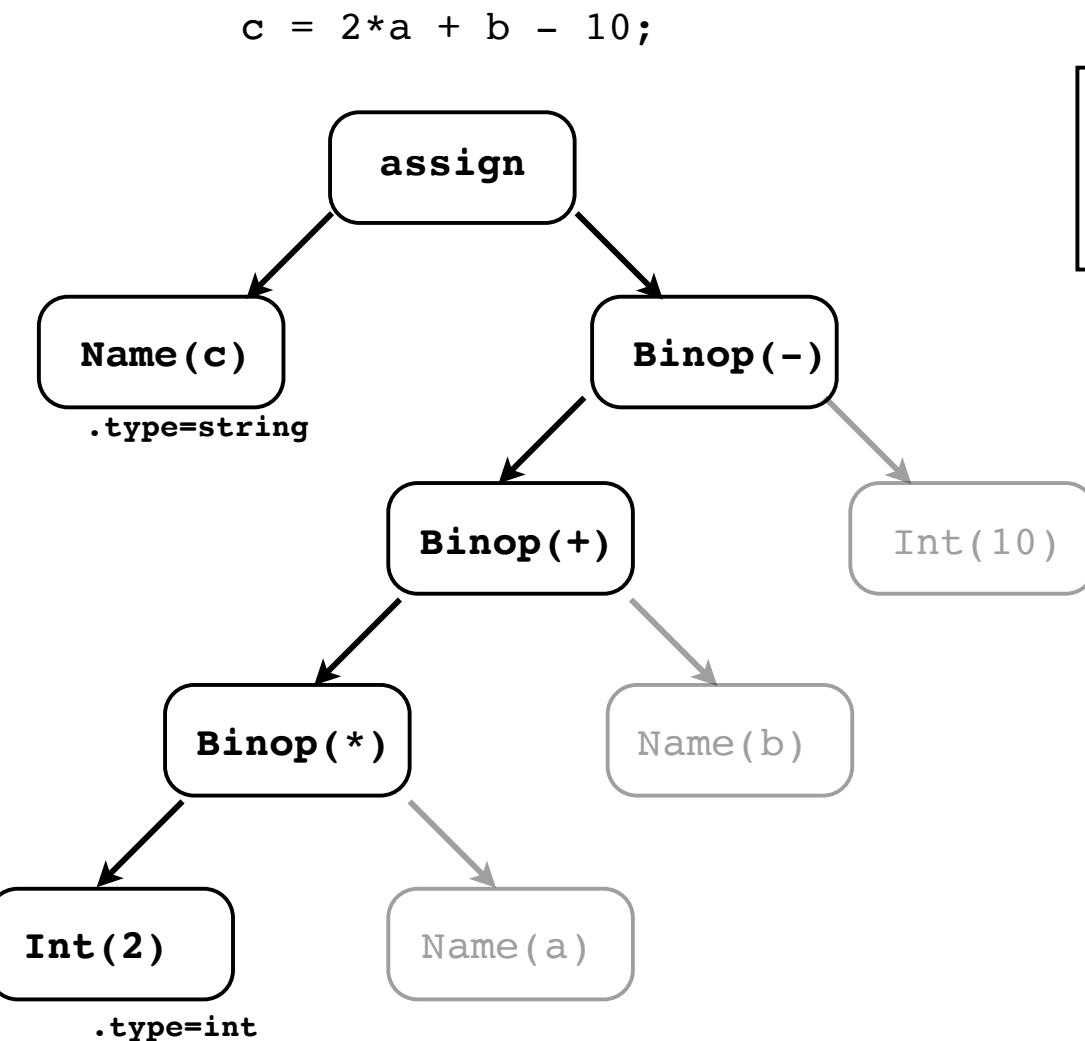
c = 2*a + b - 10;



symbol table

```
'a' : VarDecl(type="int"),  
'b' : VarDecl(type="int"),  
'c' : VarDecl(type="string")
```

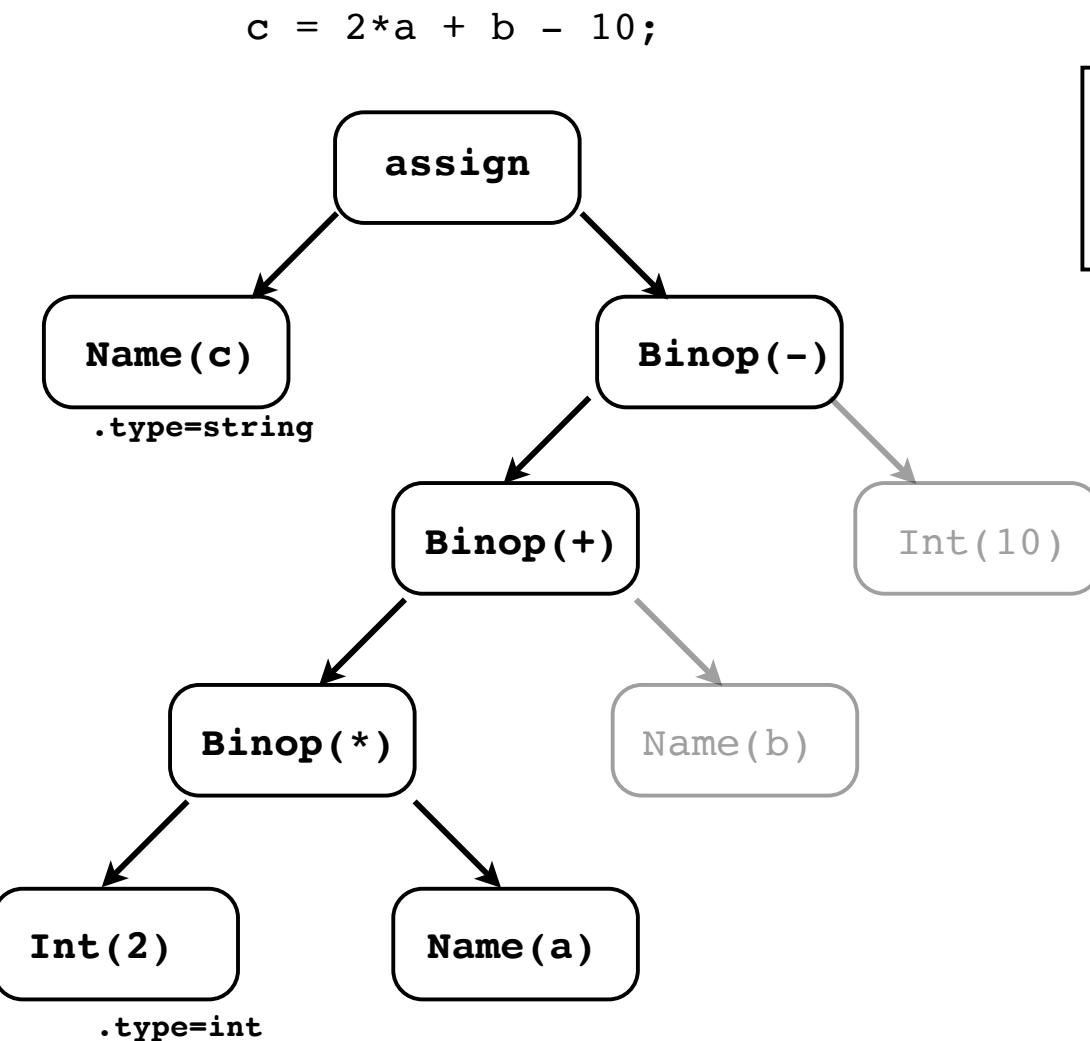
AST Annotation



symbol table

```
'a' : VarDecl(type="int"),  
'b' : VarDecl(type="int"),  
'c' : VarDecl(type="string")
```

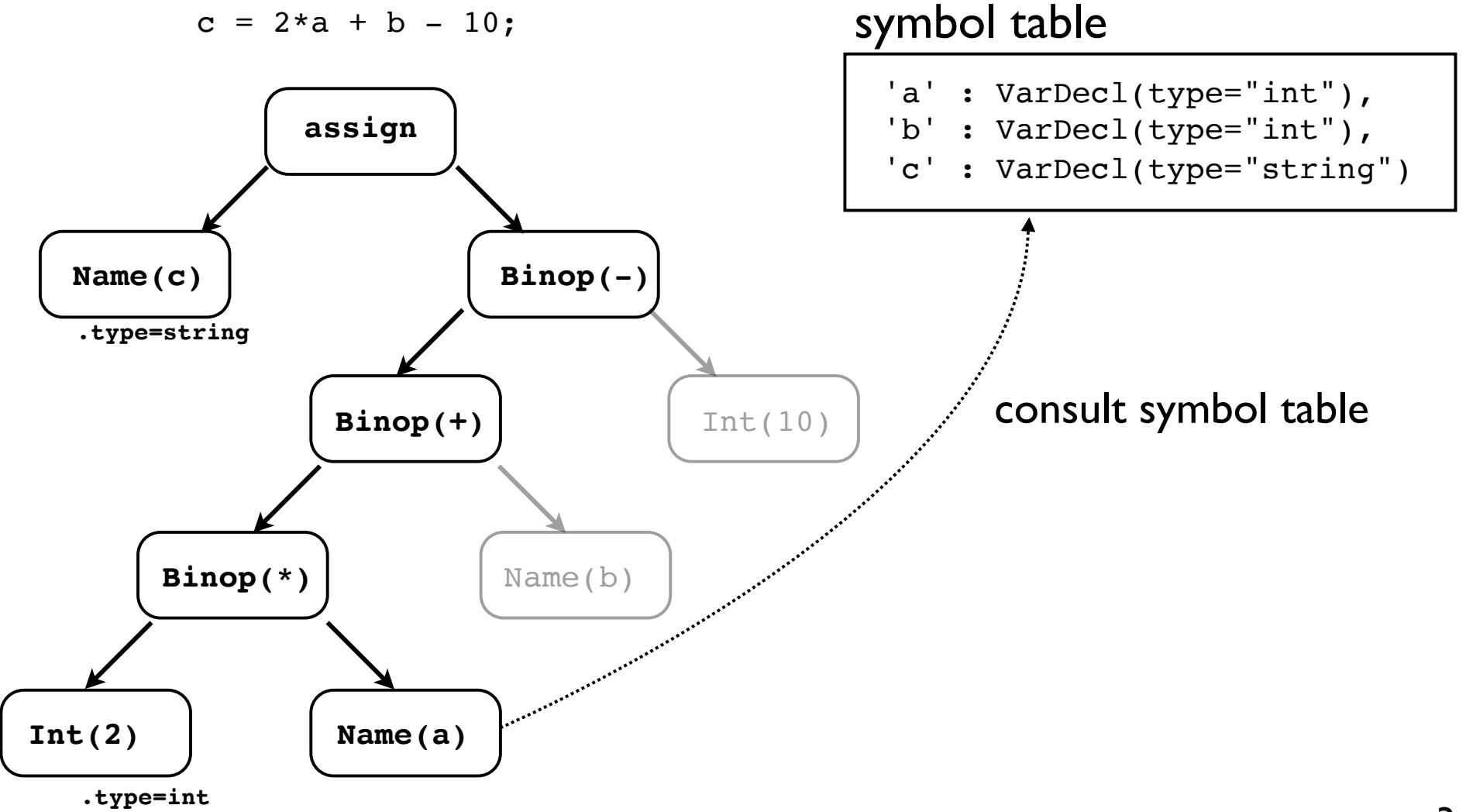
AST Annotation



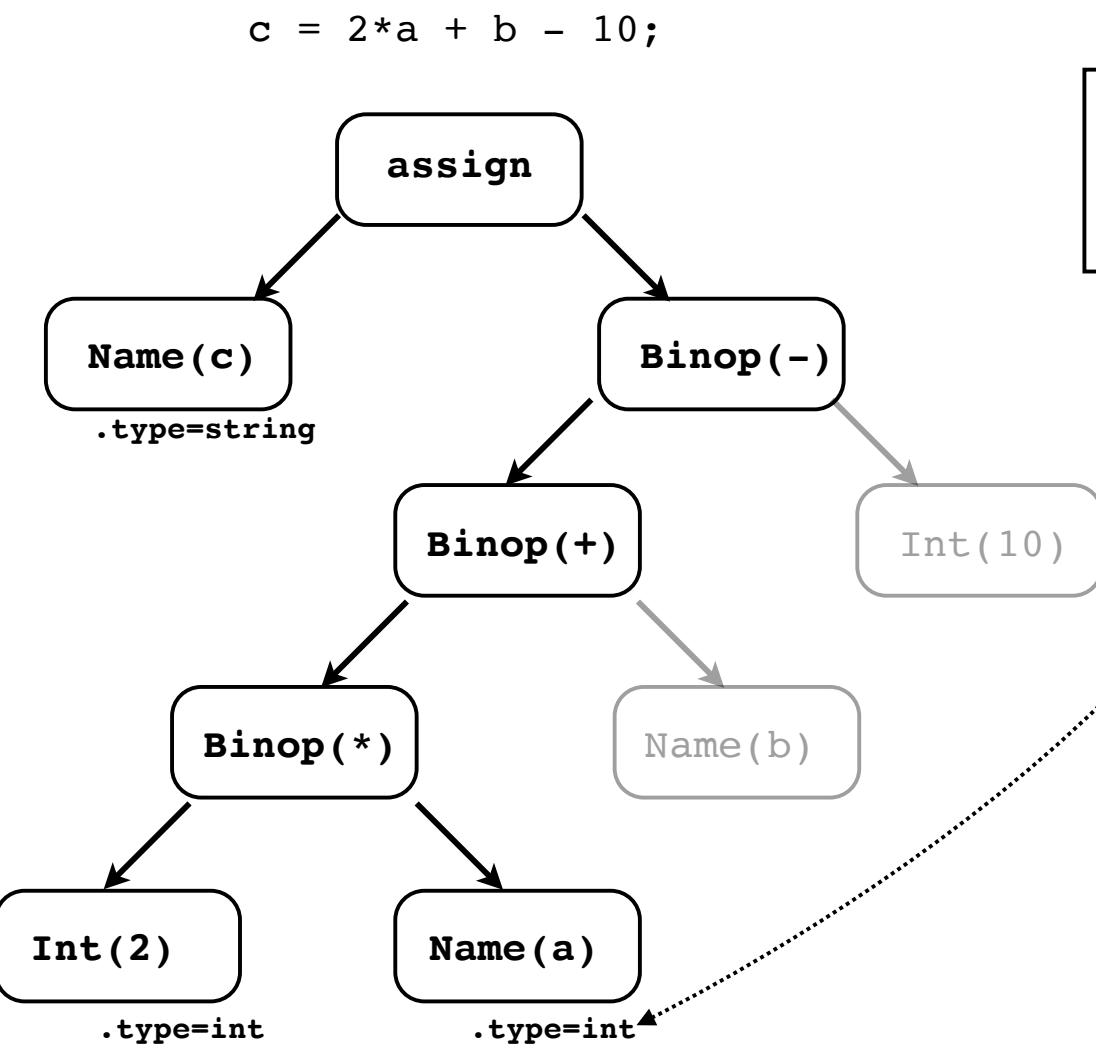
symbol table

```
'a' : VarDecl(type="int"),  
'b' : VarDecl(type="int"),  
'c' : VarDecl(type="string")
```

AST Annotation



AST Annotation

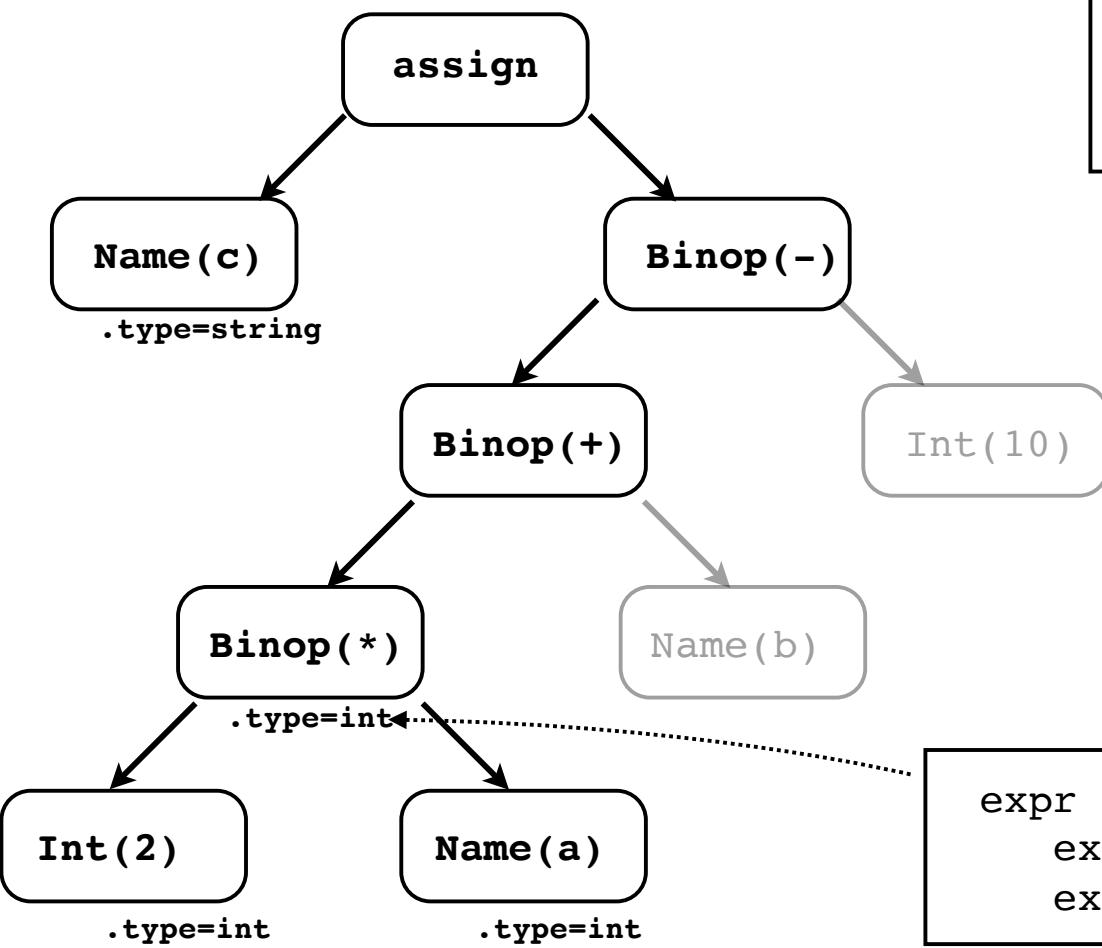


symbol table

```
'a' : VarDecl(type="int"),  
'b' : VarDecl(type="int"),  
'c' : VarDecl(type="string")
```

AST Annotation

c = 2*a + b - 10;

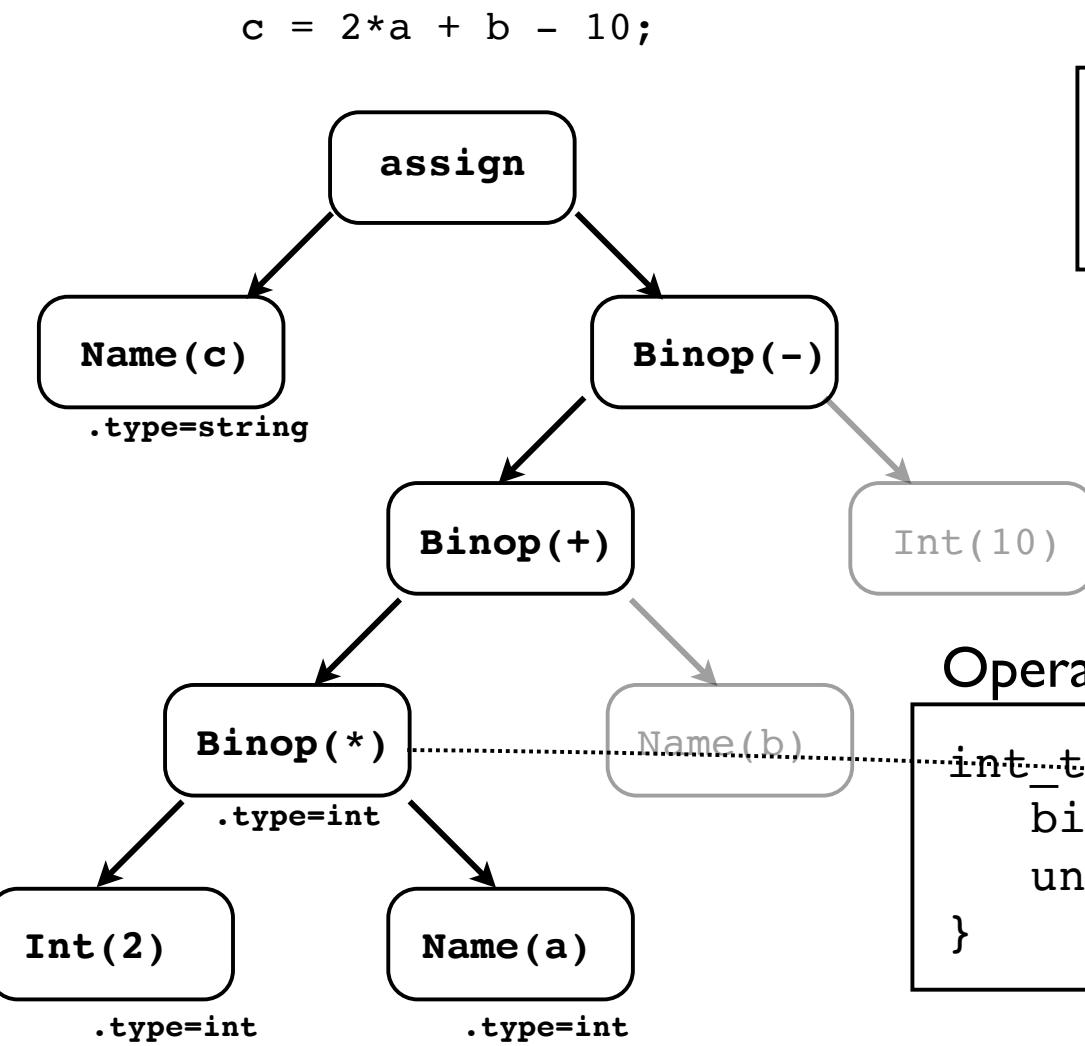


symbol table

```
'a' : VarDecl(type="int"),  
'b' : VarDecl(type="int"),  
'c' : VarDecl(type="string")
```

```
expr := expr1 '*' expr2  
expr1.type == expr2.type?  
expr.type = expr1.type
```

AST Annotation



symbol table

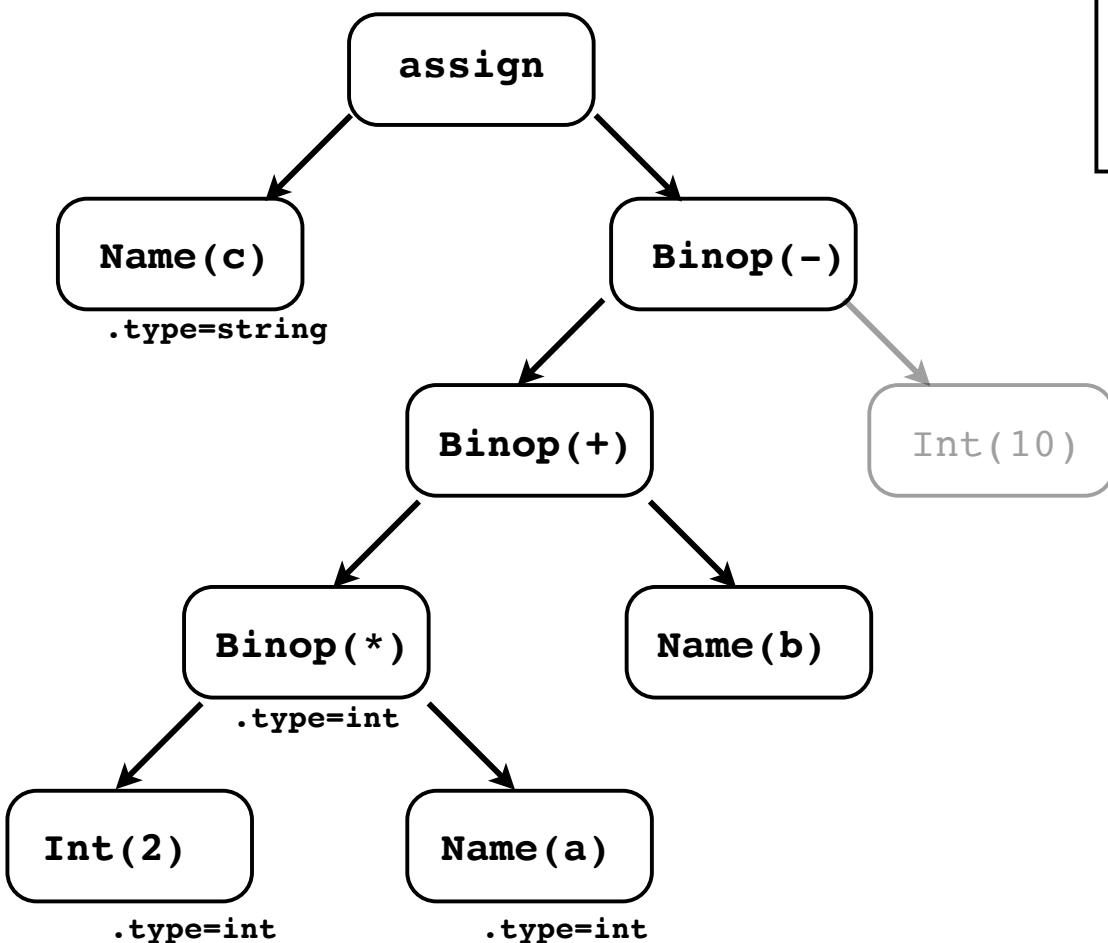
```
'a' : VarDecl(type="int"),  
'b' : VarDecl(type="int"),  
'c' : VarDecl(type="string")
```

Operator checked against type spec

```
int_type = Type(  
    binary_ops = { '+', '-' },  
    unary_ops = { '+', '-' }  
)
```

AST Annotation

c = 2*a + b - 10;

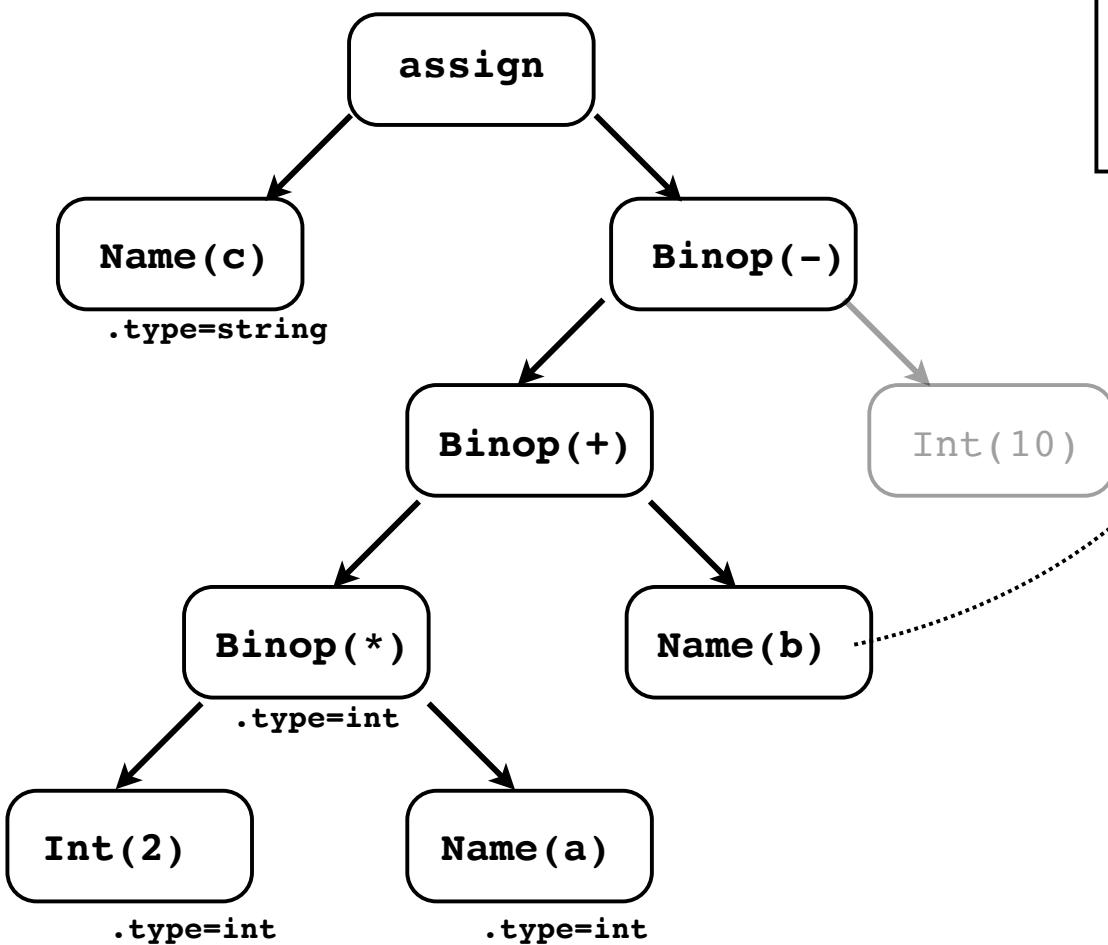


symbol table

```
'a' : VarDecl(type="int"),  
'b' : VarDecl(type="int"),  
'c' : VarDecl(type="string")
```

AST Annotation

c = 2*a + b - 10;



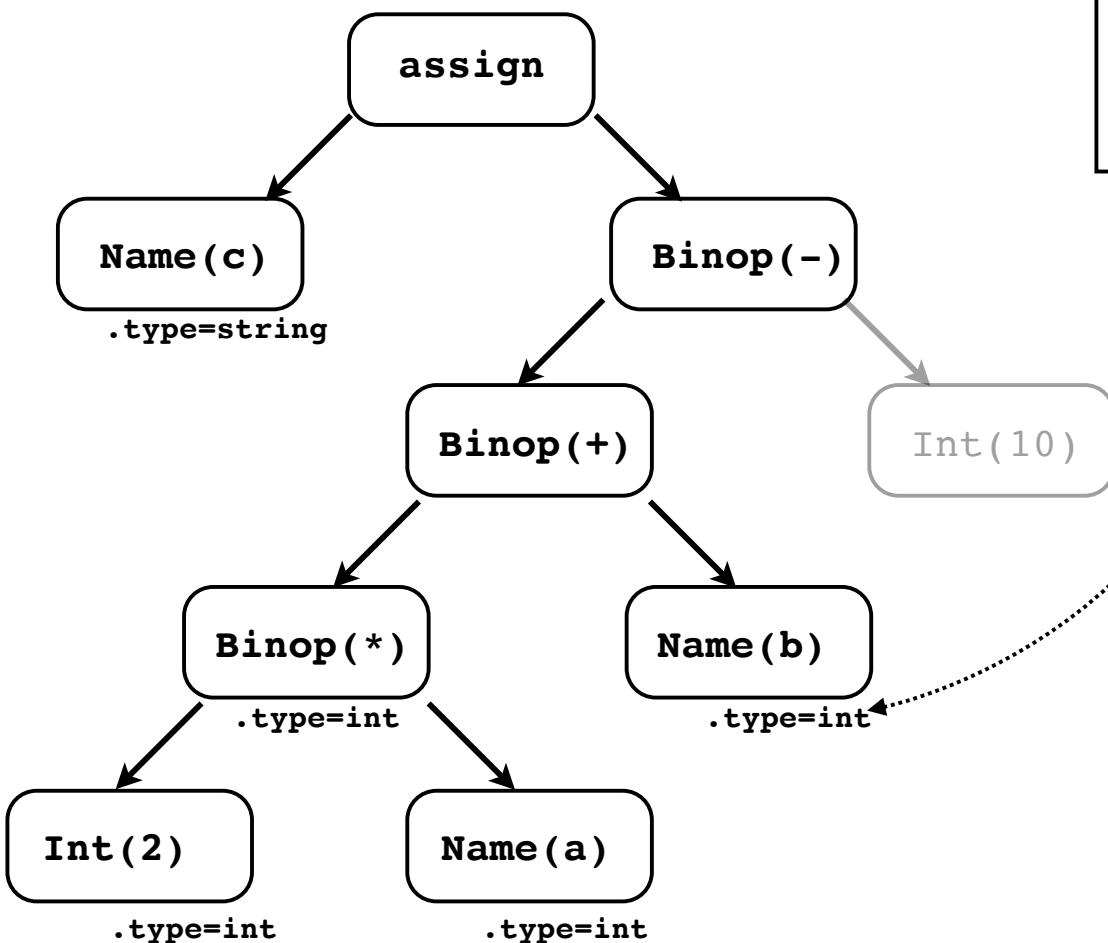
symbol table

```
'a' : VarDecl(type="int"),  
'b' : VarDecl(type="int"),  
'c' : VarDecl(type="string")
```

consult symbol table

AST Annotation

c = 2*a + b - 10;

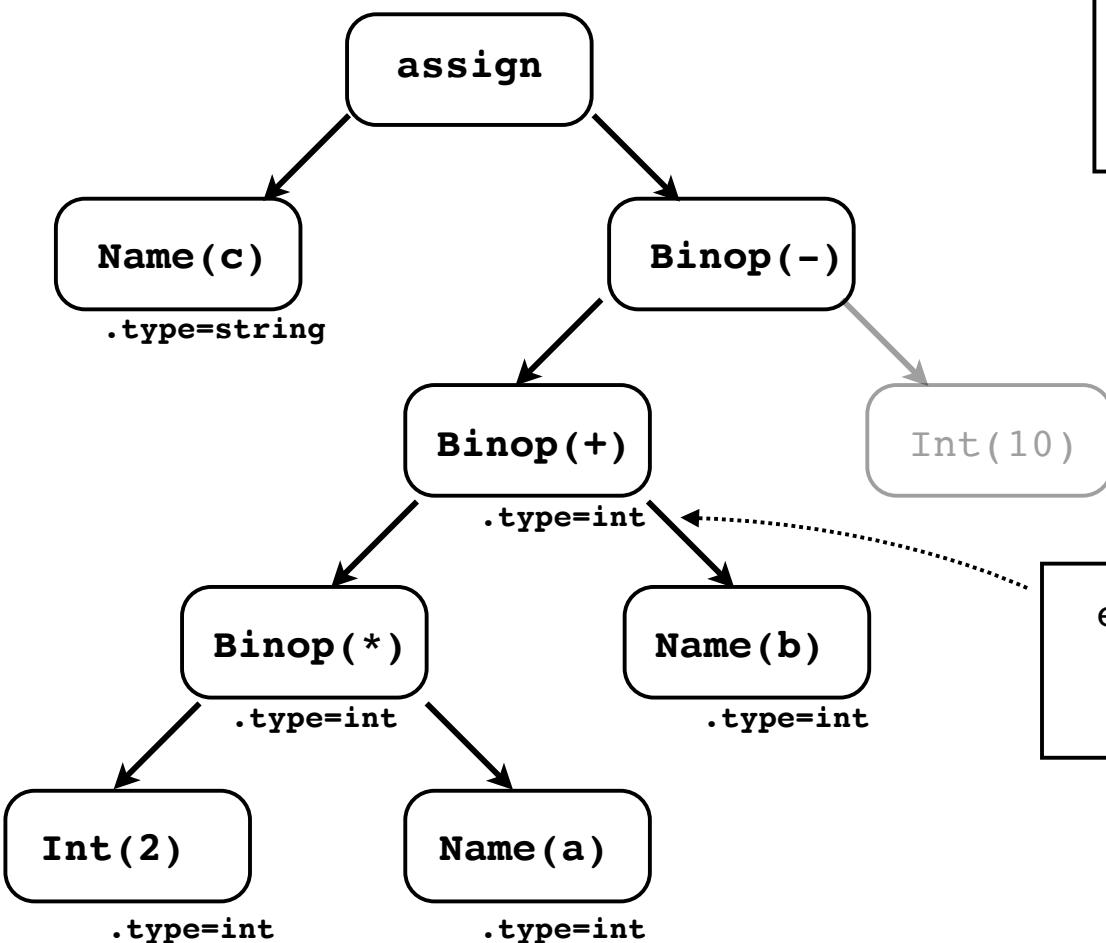


symbol table

```
'a' : VarDecl(type="int"),  
'b' : VarDecl(type="int"),  
'c' : VarDecl(type="string")
```

AST Annotation

c = 2*a + b - 10;

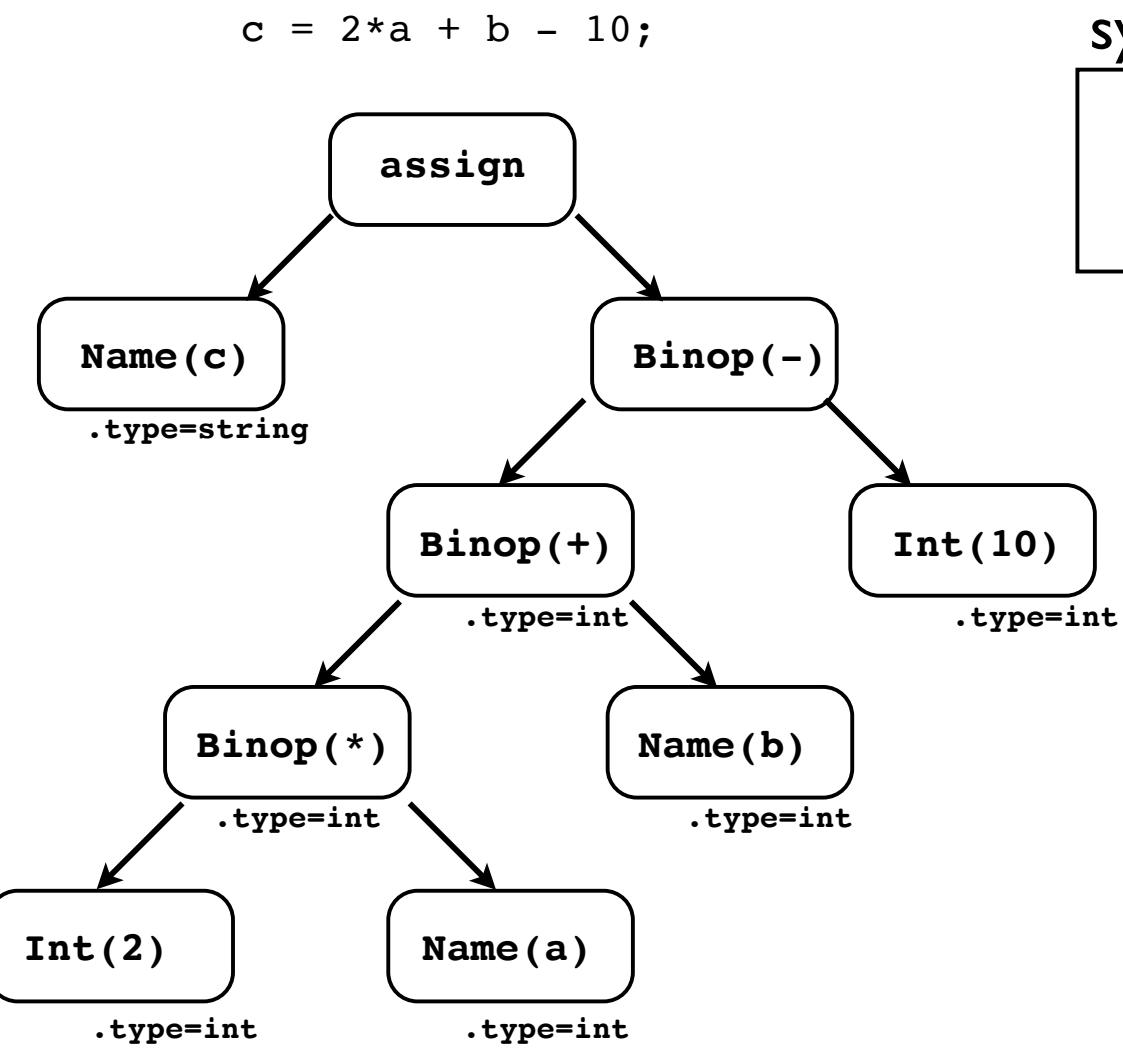


symbol table

```
'a' : VarDecl(type="int"),  
'b' : VarDecl(type="int"),  
'c' : VarDecl(type="string")
```

```
expr := expr1 '+' expr2  
expr1.type == expr2.type?  
expr.type = expr1.type
```

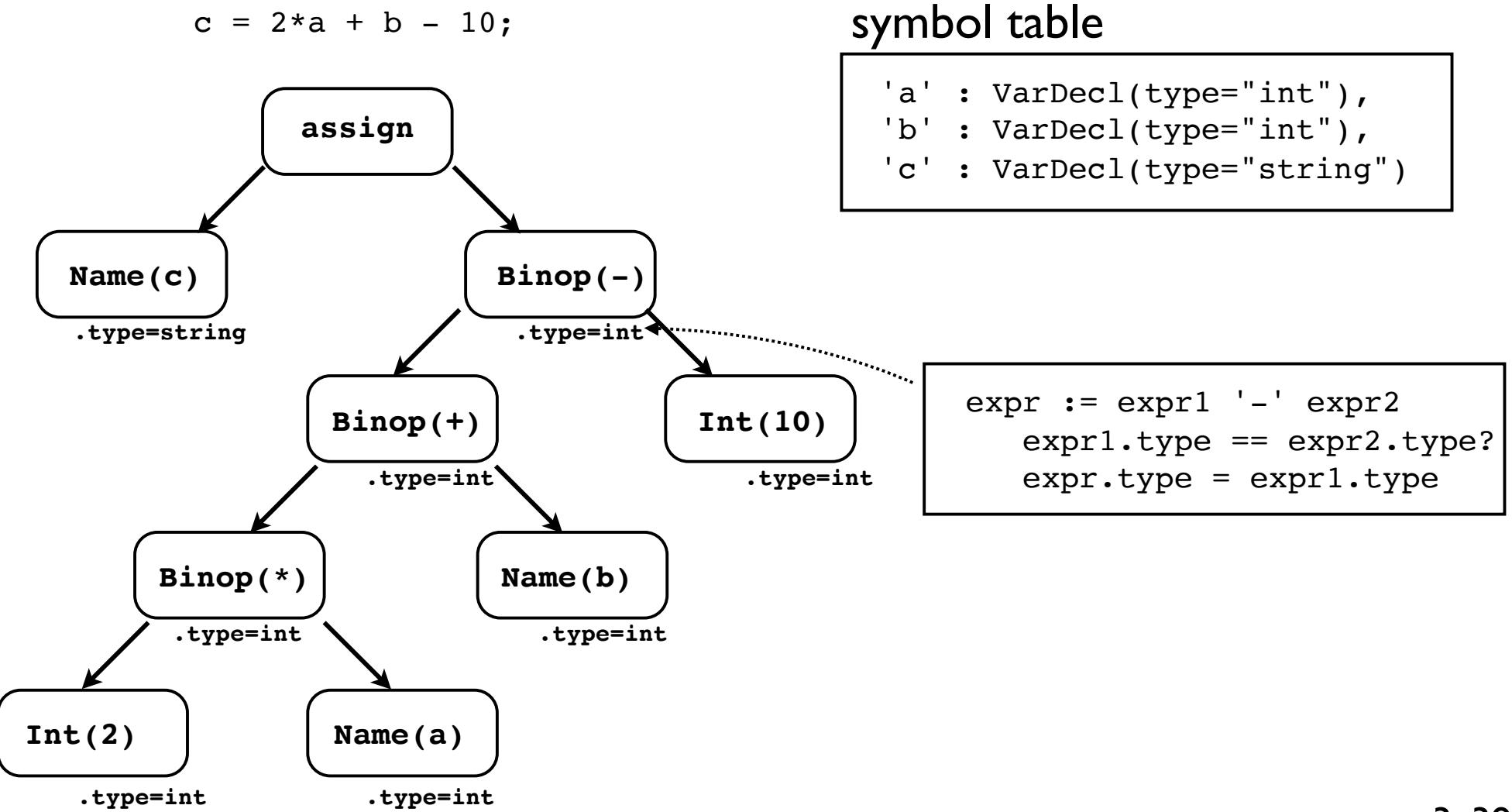
AST Annotation



symbol table

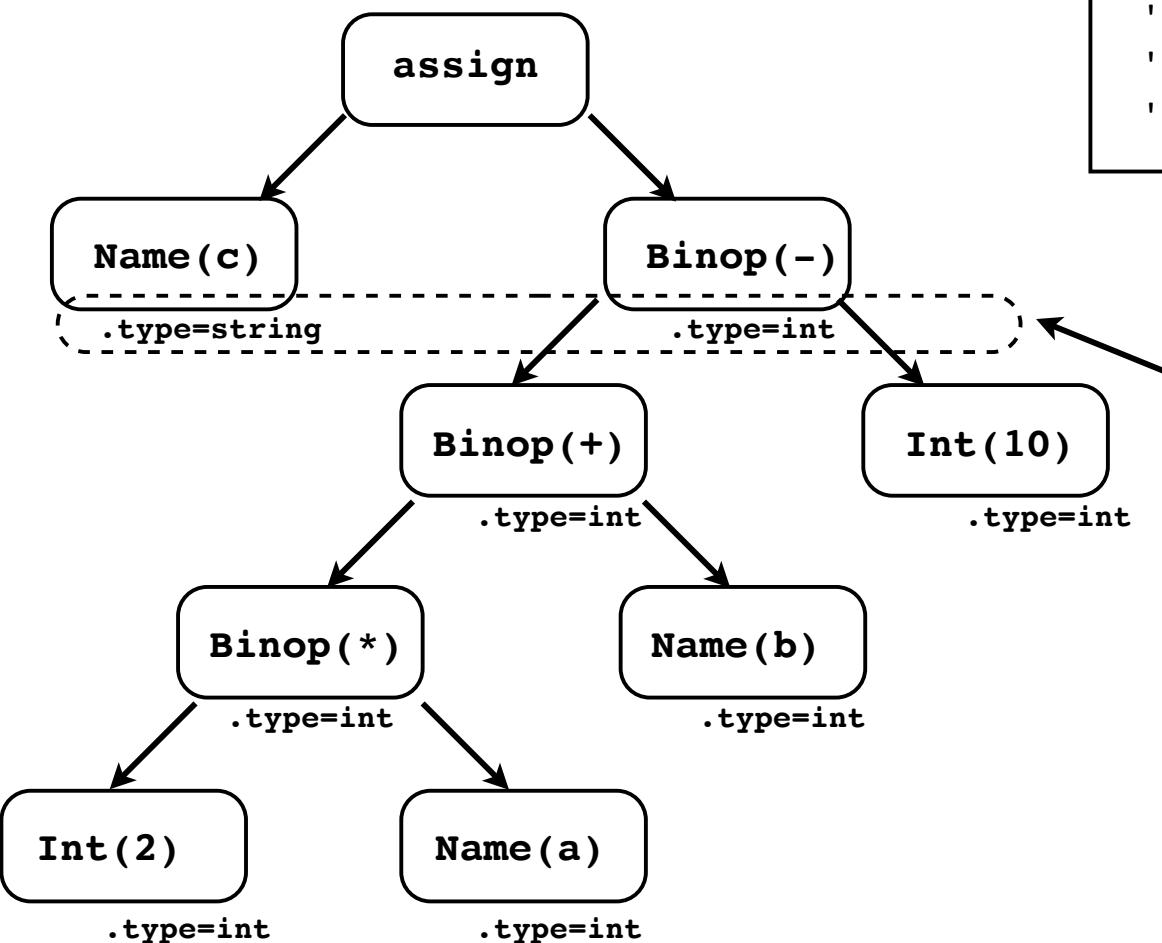
```
'a' : VarDecl(type="int"),  
'b' : VarDecl(type="int"),  
'c' : VarDecl(type="string")
```

AST Annotation



AST Annotation

```
c = 2*a + b - 10;
```

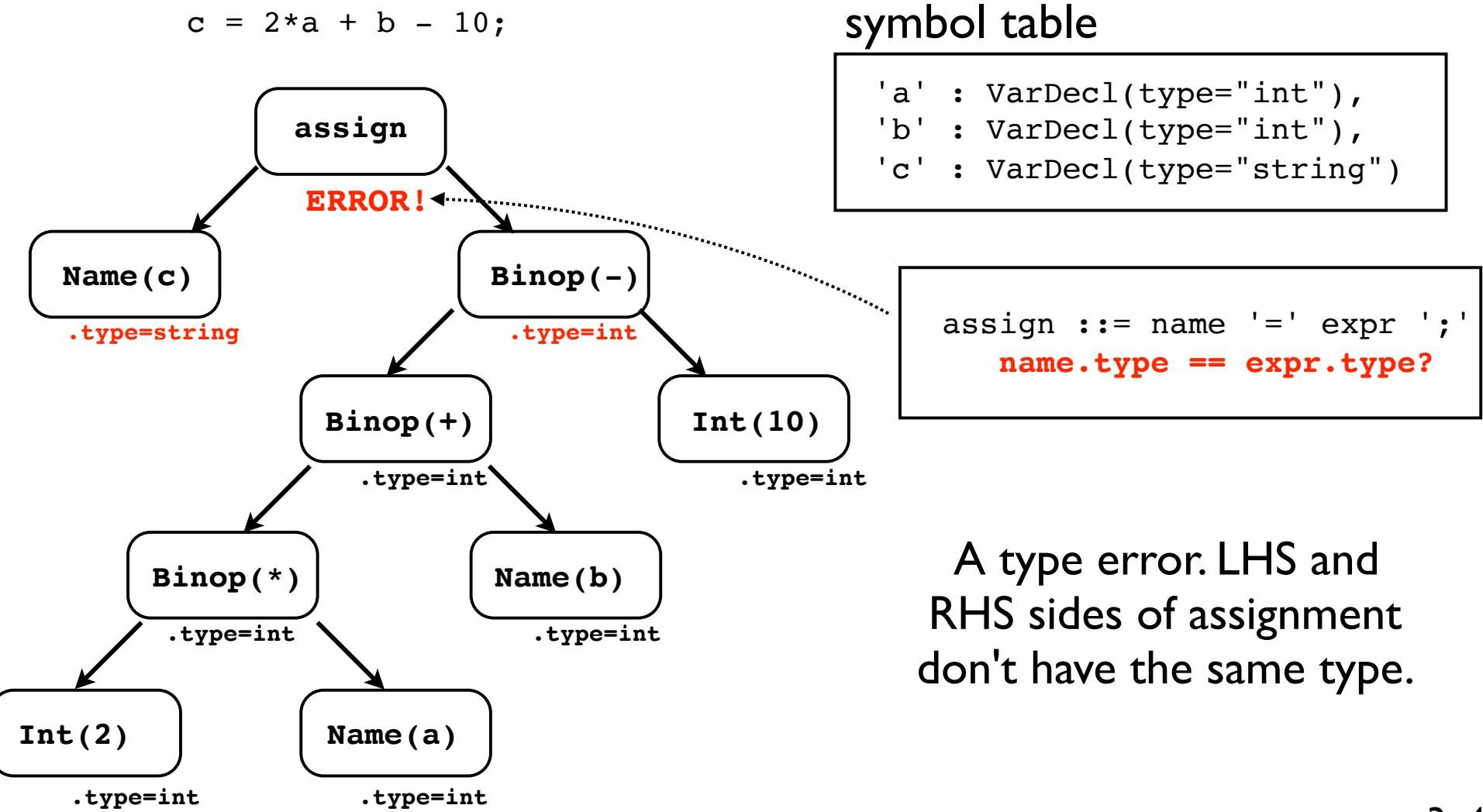


symbol table

```
'a' : VarDecl(type="int"),  
'b' : VarDecl(type="int"),  
'c' : VarDecl(type="string")
```

Observe how type information has propagated up the AST towards the assignment

AST Annotation



Commentary

- At high-level, type checking "makes sense"
- You're enforcing the common coding rules that you already know as a programmer
- Also: requires extreme attention to detail
- Obscure corner cases. Language lawyering.

Exercise & Project 3

Part 4

Code Generation

Let's Make Code

- Eventually a compiler has to make some output code
 - Assembly code
 - C code
 - Virtual machine instructions
- How do you do it?
- Walk the AST and emit code

Example : Stack Machine

- Many virtual machines (including Python) are based on a stack architecture
- General idea
 - Operands get pushed onto stack
 - Operators consume stack items
- Like RPN HP Calculators

Example : Stack Machine

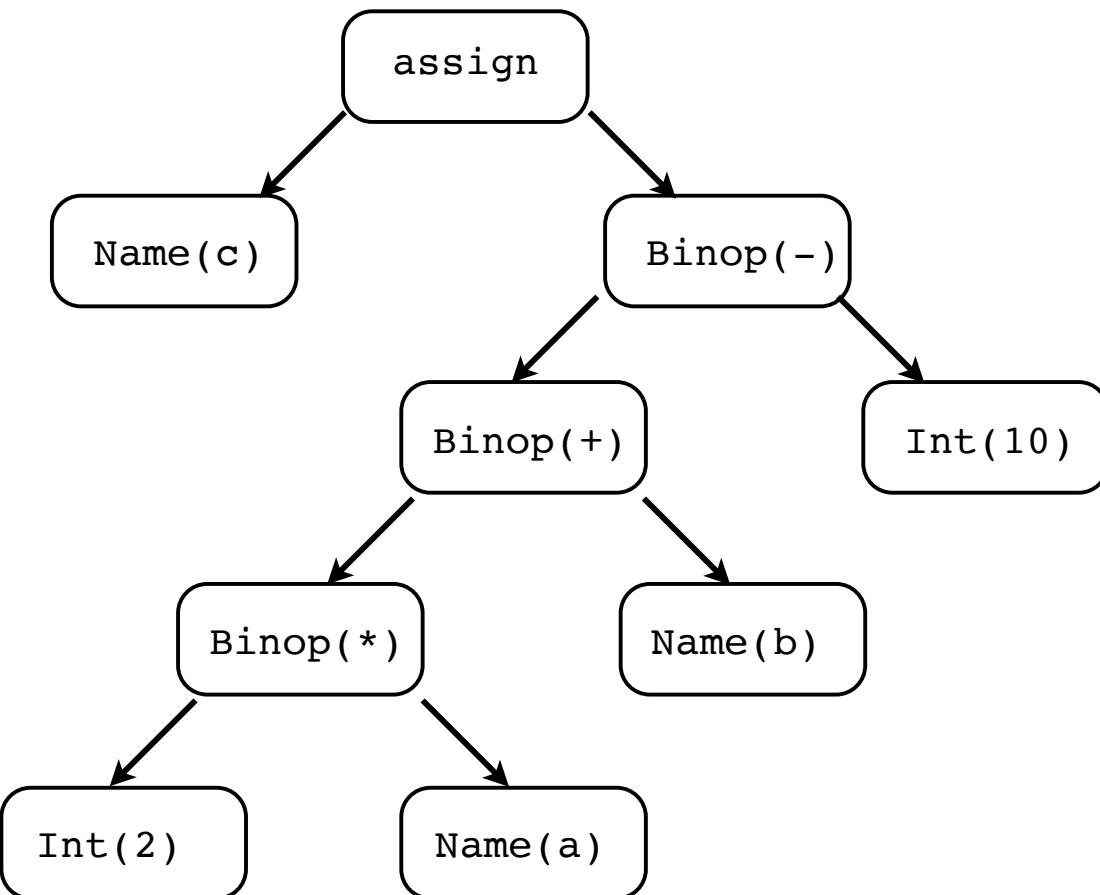
- Example: Compute: $2 + 3 * 4$

<u>Instructions</u>	<u>Stack</u>
PUSH 2	[2]
PUSH 3	[2, 3]
PUSH 4	[2, 3, 4]
MUL	[2, 12]
ADD	[14]

- Let's turn an AST into code

Code Generation

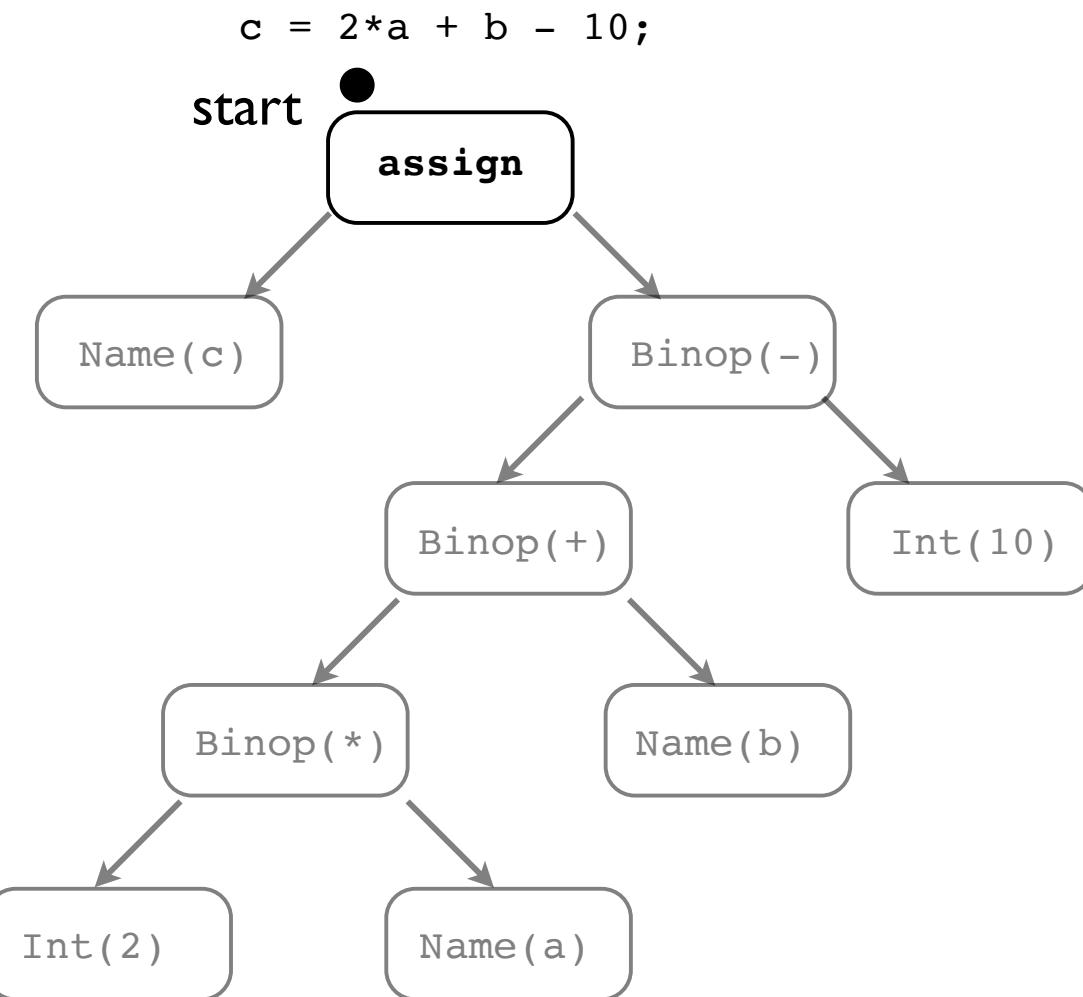
c = 2*a + b - 10;



Instructions:

Compute Stack

Code Generation

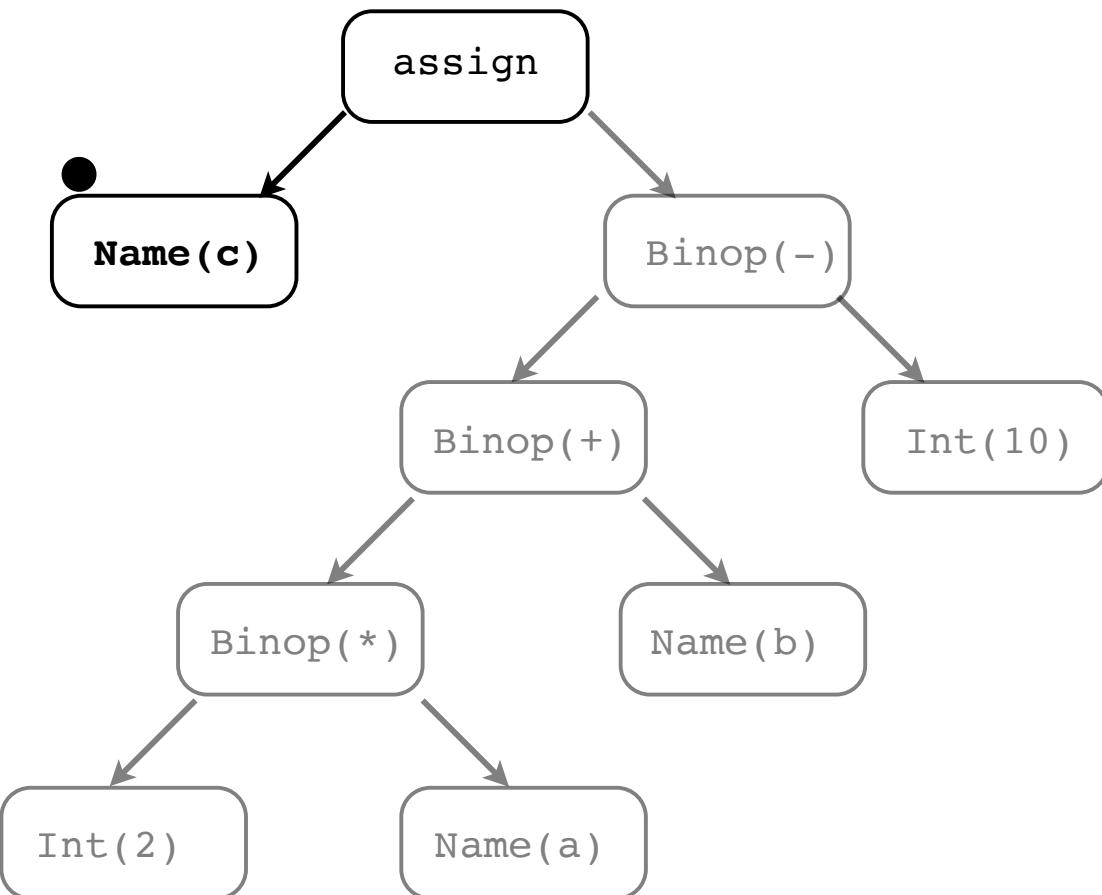


Instructions:

Compute Stack

Code Generation

c = 2*a + b - 10;



Instructions:

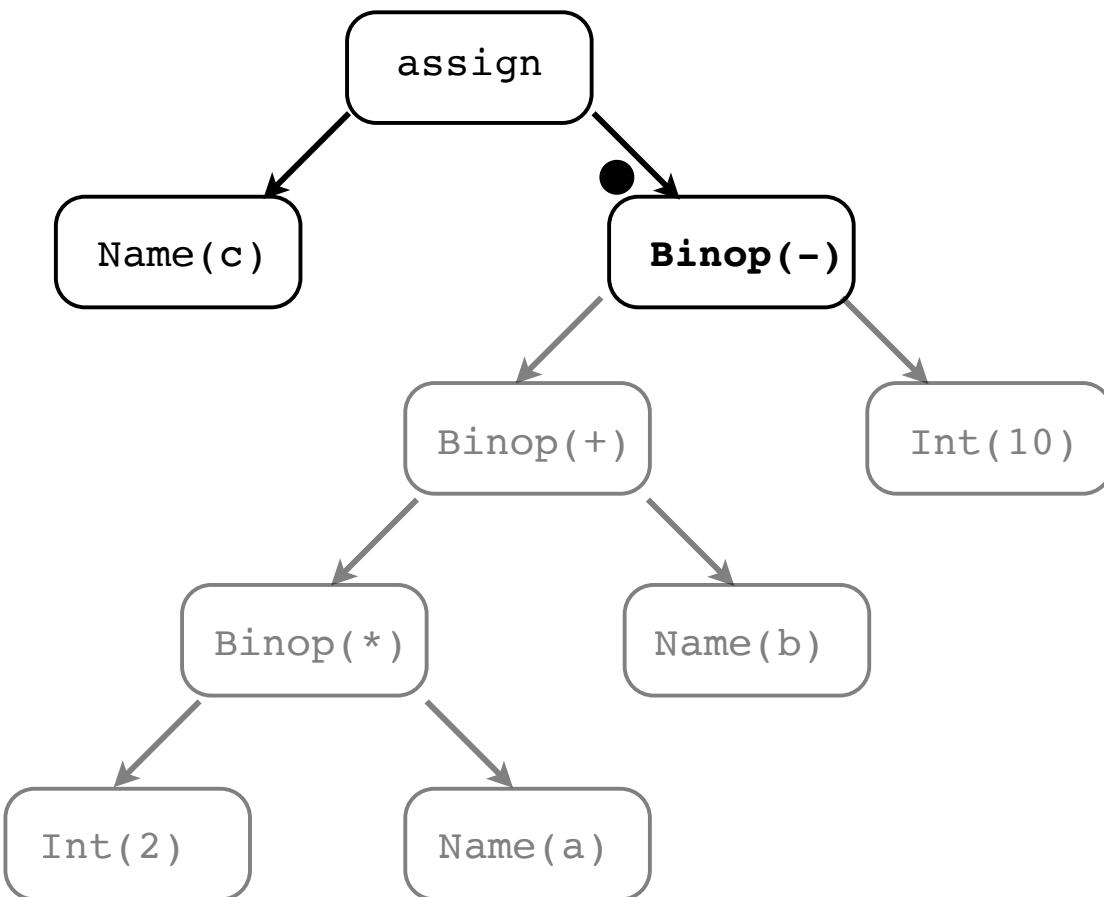
1. PUSH LOCATION(c)

Compute Stack

LOCATION(c)

Code Generation

c = 2*a + b - 10;



Instructions:

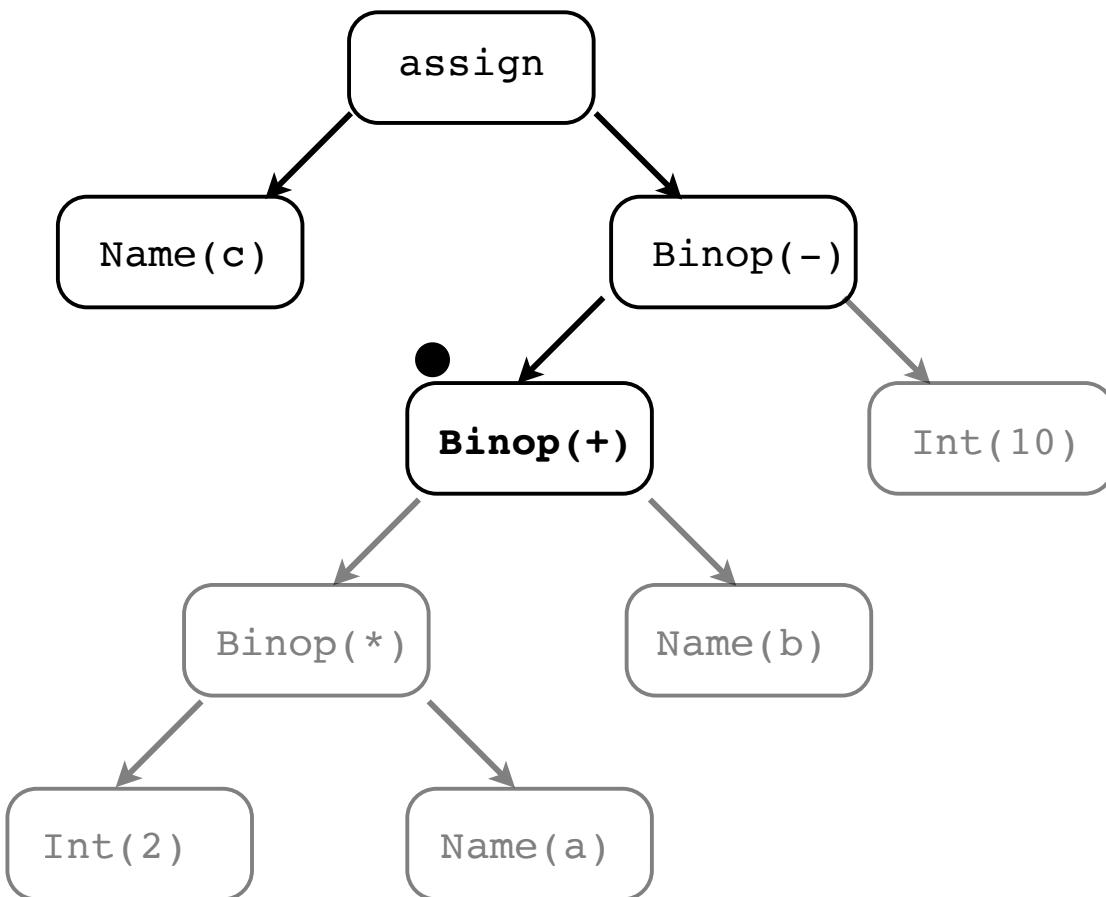
1. PUSH LOCATION(c)

Compute Stack

LOCATION(c)

Code Generation

c = 2*a + b - 10;



Instructions:

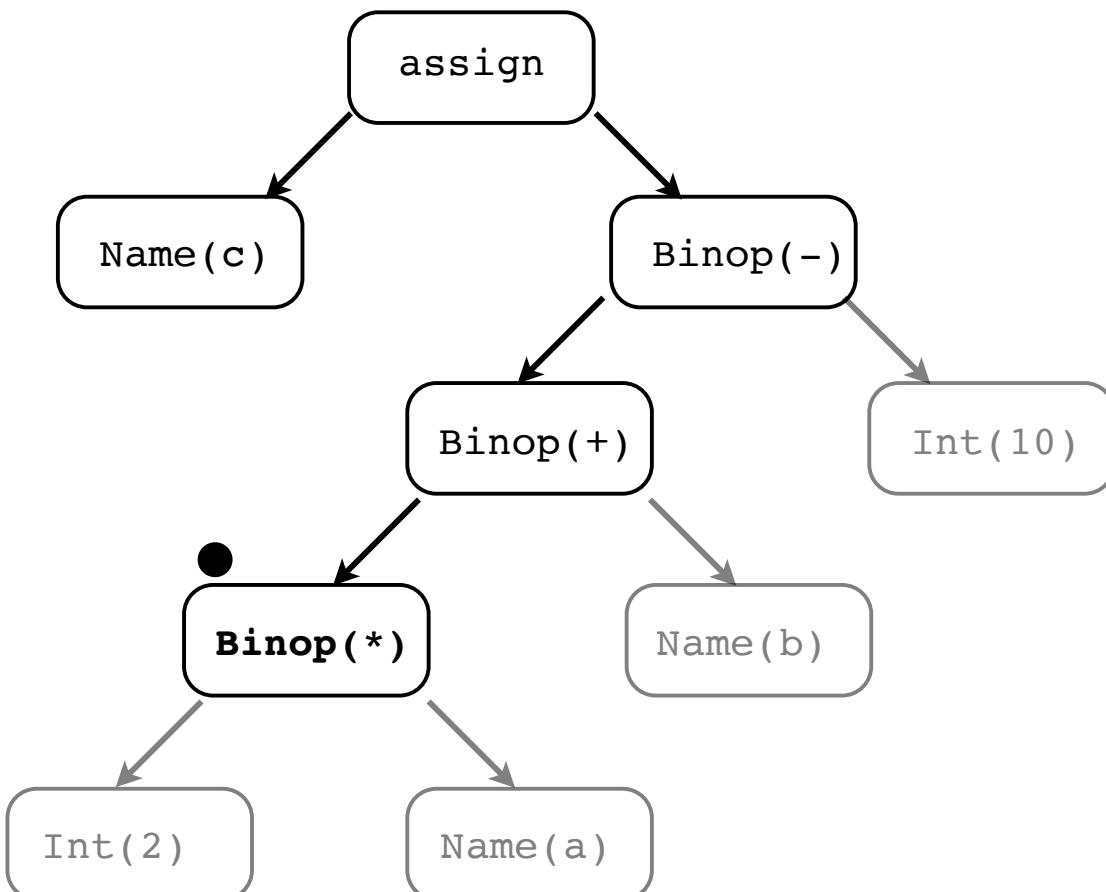
1. PUSH LOCATION(c)

Compute Stack

LOCATION(c)

Code Generation

c = 2*a + b - 10;



Instructions:

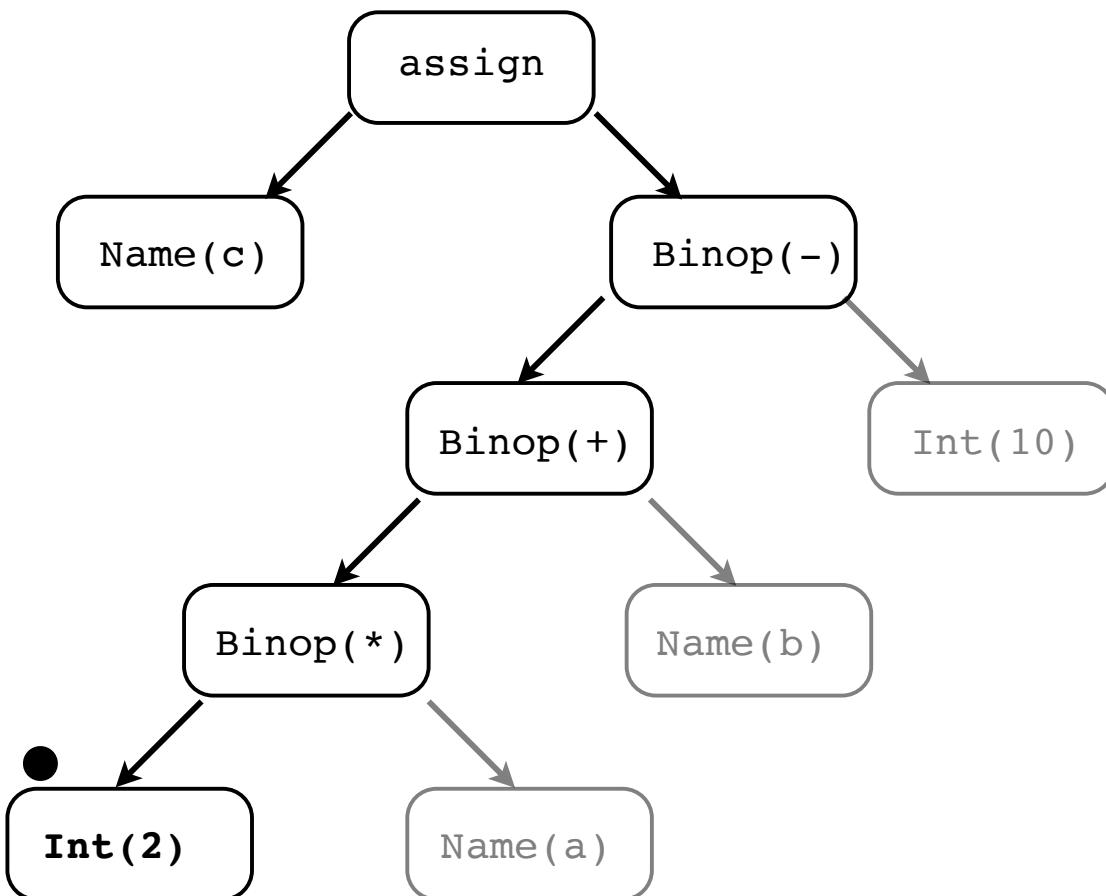
1. PUSH LOCATION(c)

Compute Stack

LOCATION(c)

Code Generation

c = 2*a + b - 10;



Instructions:

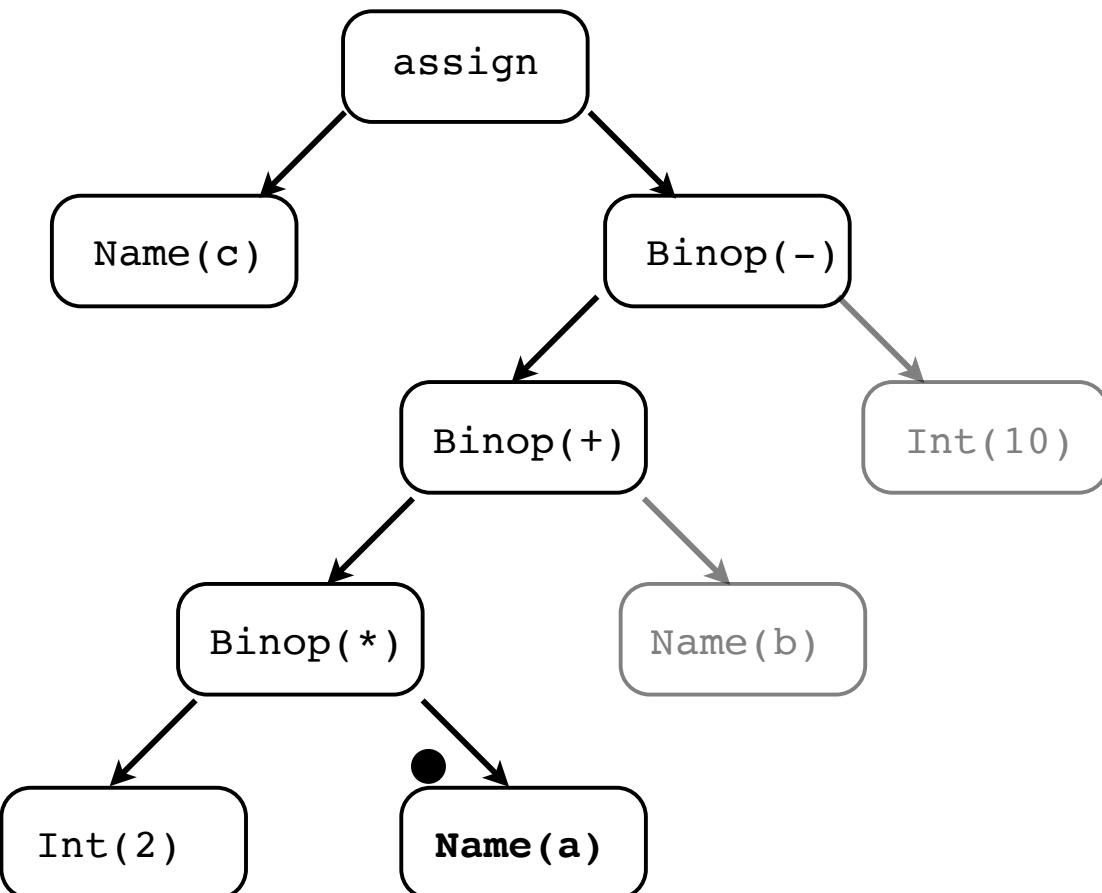
1. PUSH LOCATION(c)
2. PUSH 2

Compute Stack

LOCATION(c)
2

Code Generation

c = 2*a + b - 10;



Instructions:

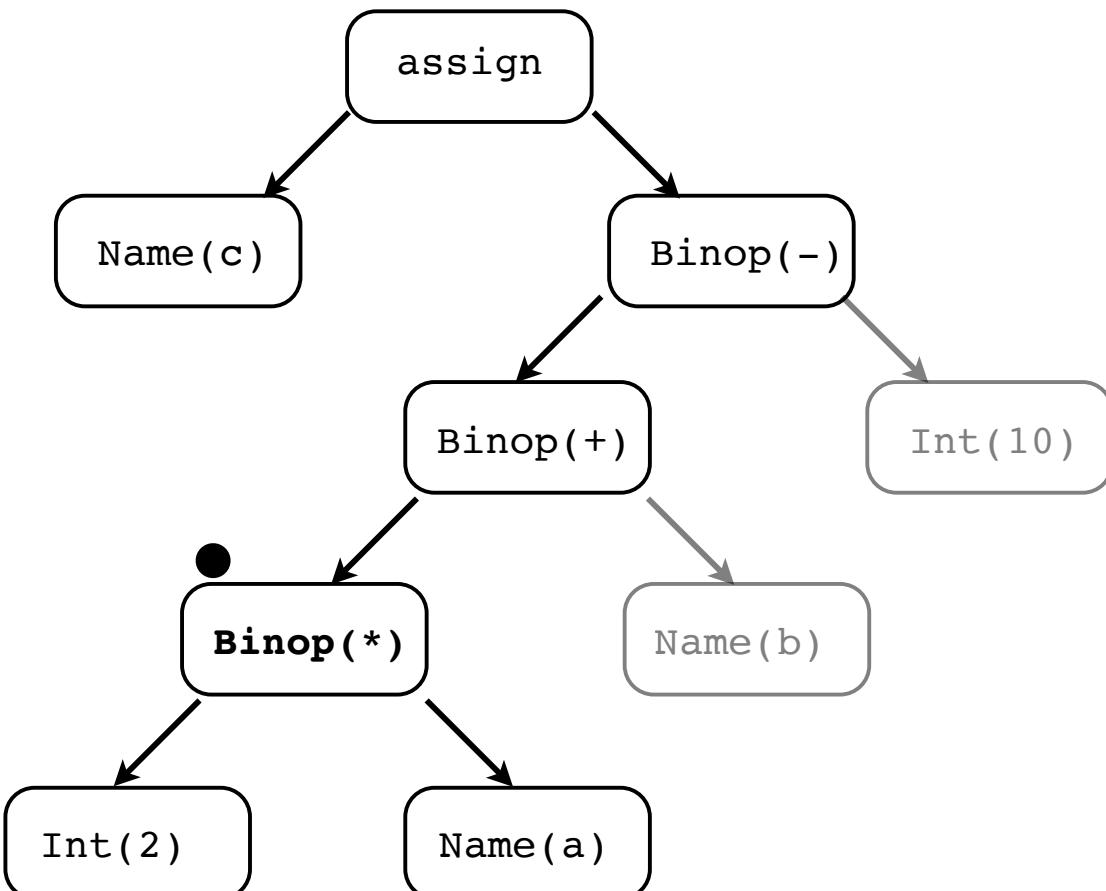
1. PUSH LOCATION(c)
2. PUSH 2
3. PUSH LOCATION(a)
4. LOAD

Compute Stack

LOCATION(c)
2
a

Code Generation

c = 2*a + b - 10;



Instructions:

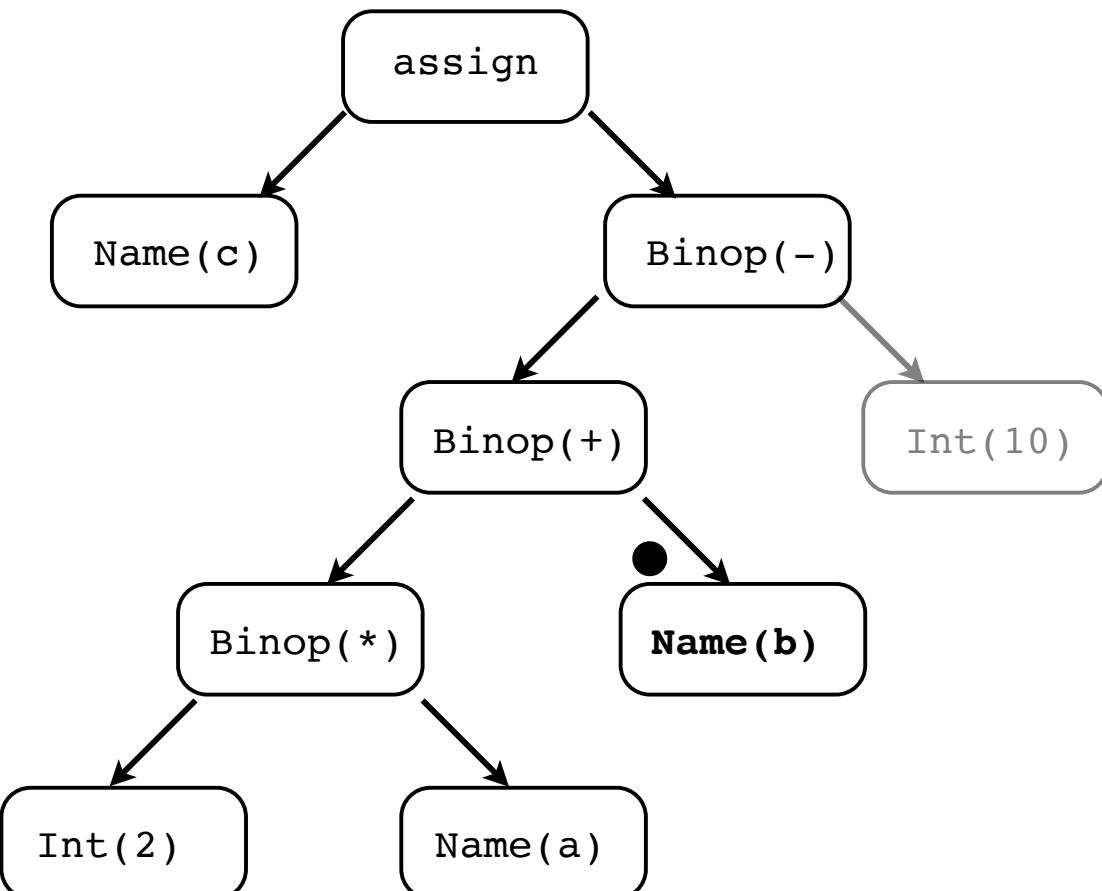
1. PUSH LOCATION(c)
2. PUSH 2
3. PUSH LOCATION(a)
4. LOAD
5. MUL

Compute Stack

LOCATION(c)
2 * a

Code Generation

c = 2*a + b - 10;



Instructions:

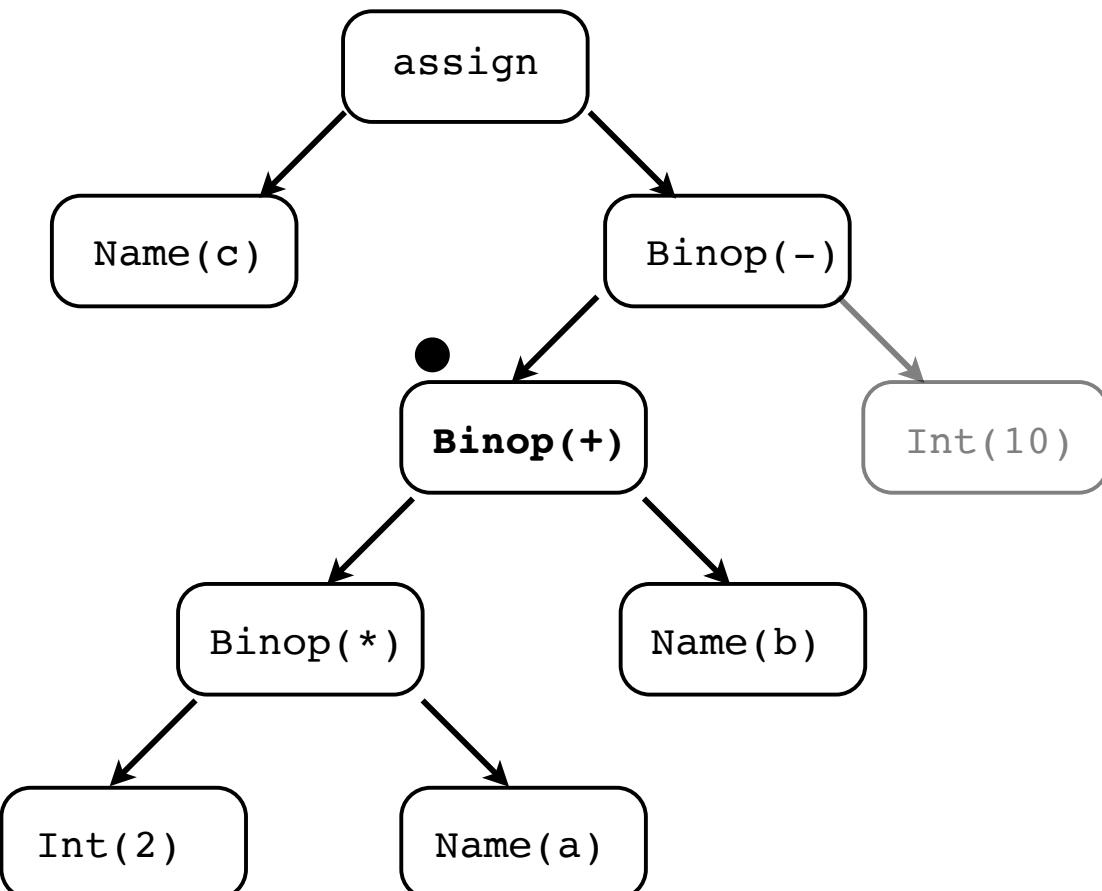
1. PUSH LOCATION(c)
2. PUSH 2
3. PUSH LOCATION(a)
4. LOAD
5. MUL
6. PUSH LOCATION(b)
7. LOAD

Compute Stack

LOCATION(c)
2 * a
b

Code Generation

c = 2*a + b - 10;



Instructions:

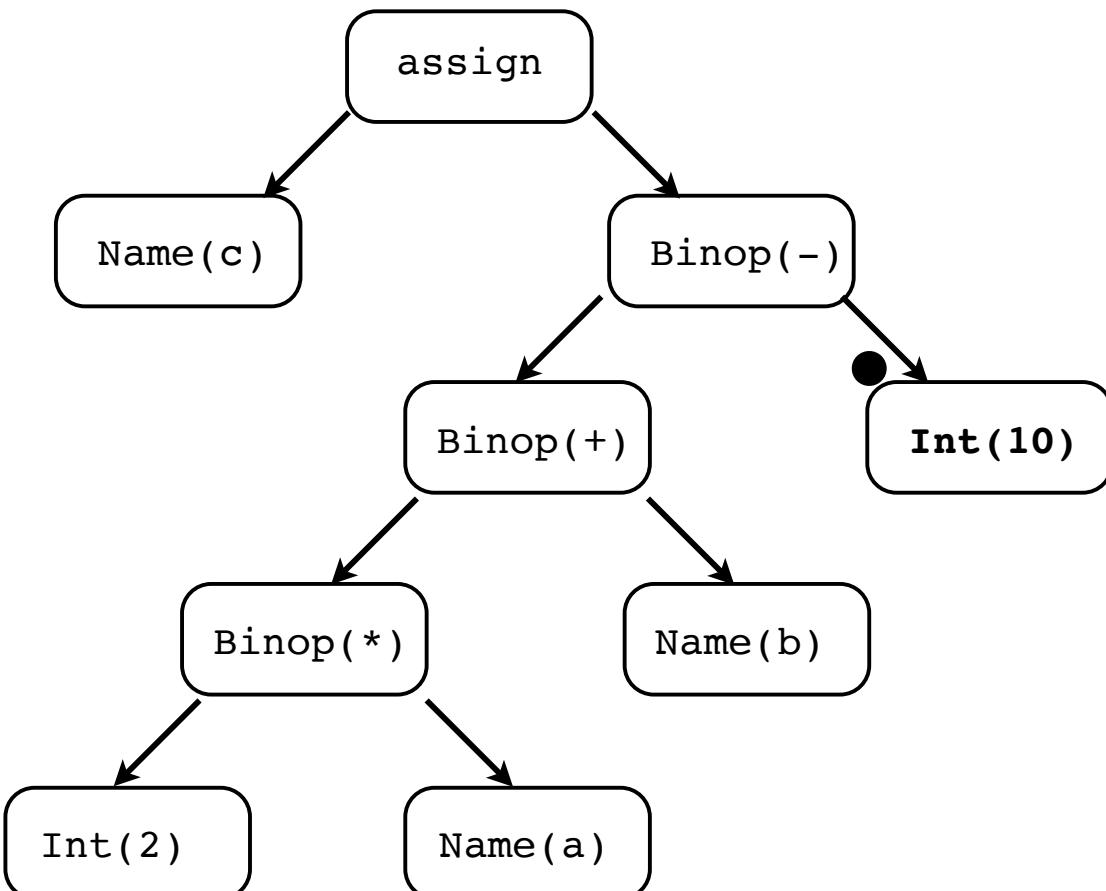
1. PUSH LOCATION(c)
2. PUSH 2
3. PUSH LOCATION(a)
4. LOAD
5. MUL
6. PUSH LOCATION(b)
7. LOAD
8. ADD

Compute Stack

LOCATION(c)
2 * a + b

Code Generation

c = 2*a + b - 10;



Instructions:

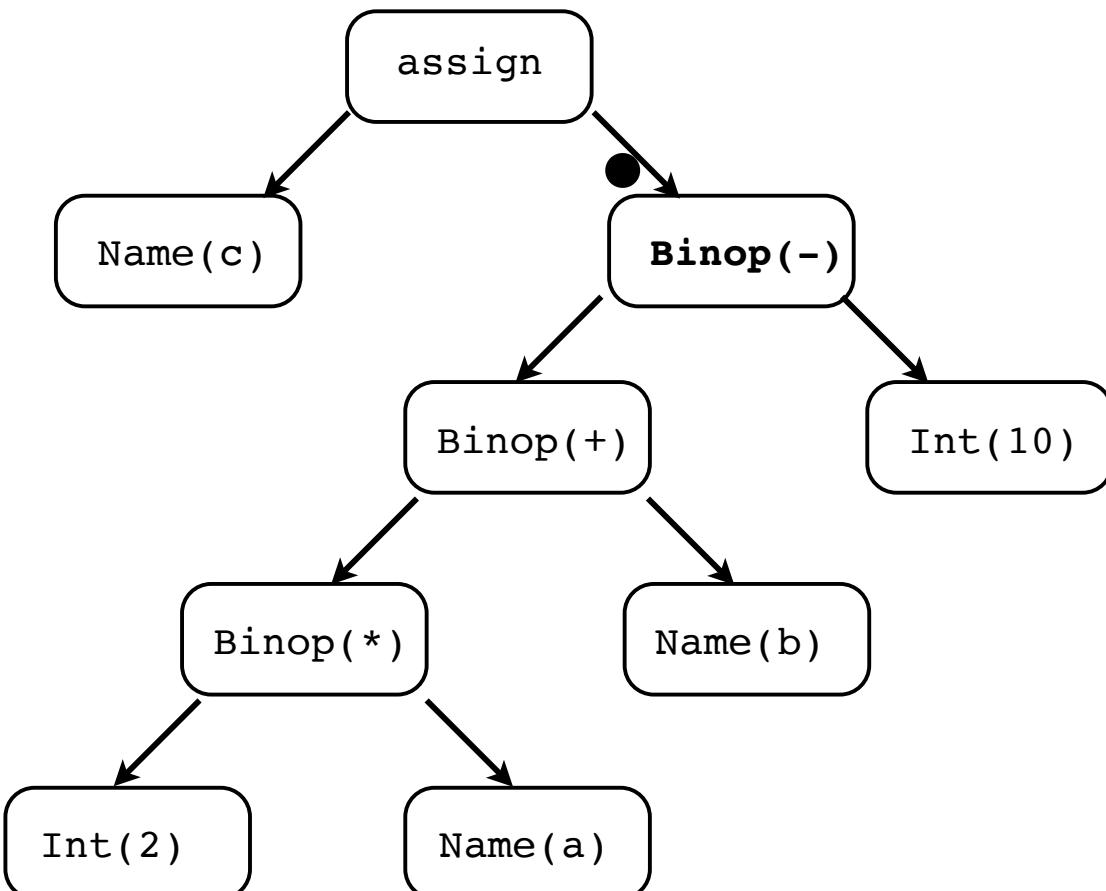
1. PUSH LOCATION(c)
2. PUSH 2
3. PUSH LOCATION(a)
4. LOAD
5. MUL
6. PUSH LOCATION(b)
7. LOAD
8. ADD
9. PUSH 10

Compute Stack

LOCATION(c)
2 * a + b
10

Code Generation

c = 2*a + b - 10;



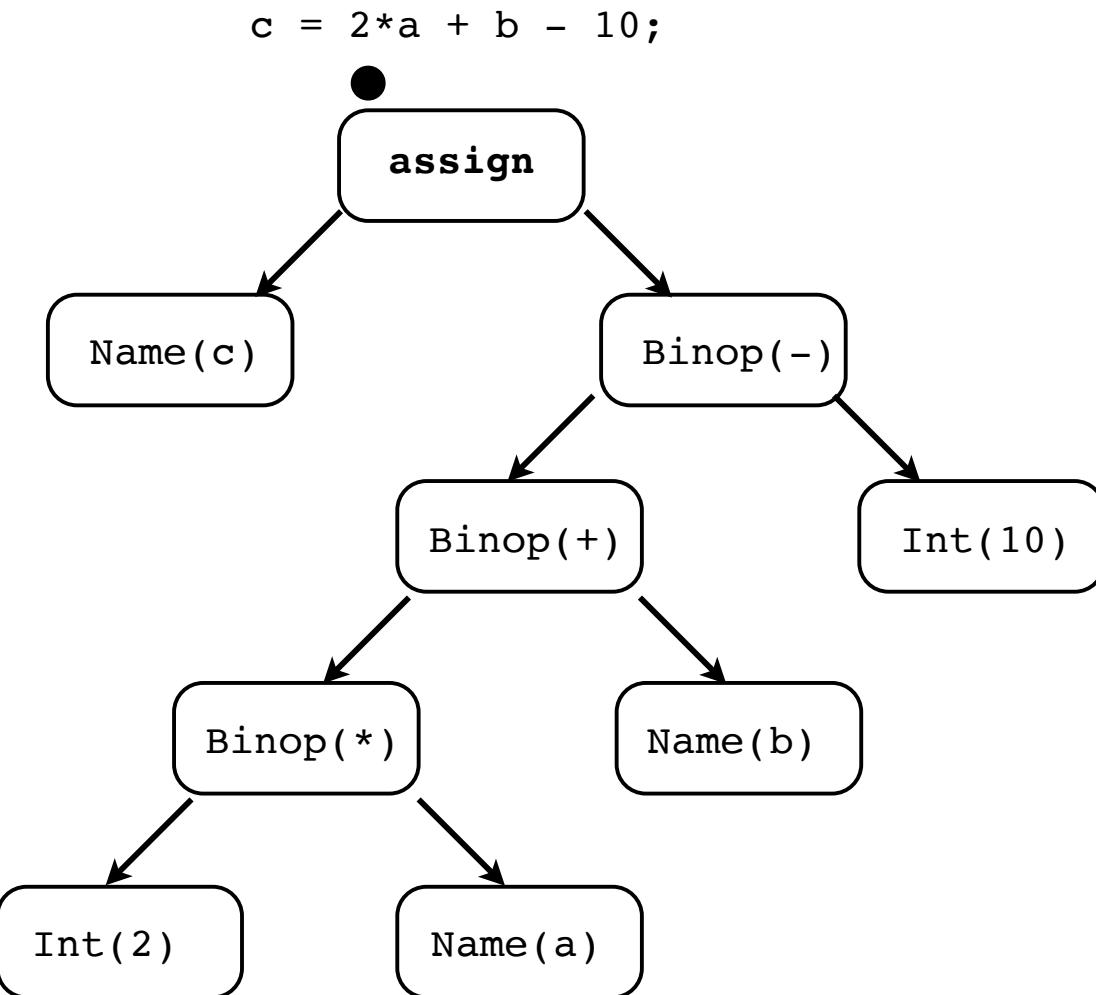
Instructions:

1. PUSH LOCATION(c)
2. PUSH 2
3. PUSH LOCATION(a)
4. LOAD
5. MUL
6. PUSH LOCATION(b)
7. LOAD
8. ADD
9. PUSH 10
10. SUB

Compute Stack

LOCATION(c)
2 * a + b - 10

Code Generation



Instructions:

1. PUSH LOCATION(c)
2. PUSH 2
3. PUSH LOCATION(a)
4. LOAD
5. MUL
6. PUSH LOCATION(b)
7. LOAD
8. ADD
9. PUSH 10
10. SUB
11. STORE

Compute Stack

Demo: Stack Machine

Commentary

- Actual instructions may vary
- Depends entirely on what the target is
- Overall idea is the same though
- Depth-first traversal where leaves push data onto stack and inner nodes perform operations on the stack

Demo: Register Machine

Intermediate Code

- Compilers often generate an abstract intermediate code instead of directly emitting low-level instructions
- Intermediate code is sort of a generic machine code
- Easier to analyze and translate

Three-Address Code

- A common IR where most instructions are just tuples (opcode,src1,src2,target)

```
( 'ADD' ,a,b,c)      # c = a + b  
( 'SUB' ,x,y,z)      # z = x - y  
( 'LOAD' ,a,b)       # b = a
```

- Closely mimics machine code on CPUs

Three-Address Code

- Example of three-address code IR

c = 2*a + b - 10

t1 = 2	('LOADI', 2, 't1')
t2 = a	('LOADVAR', 'a', 't2')
t3 = t1 * t2	('MUL', 't1', 't2', 't3')
t4 = b	('LOADVAR', 'b', 't4')
t5 = t3 + t4	('ADD', 't3', 't4', 't5')
t6 = 10	('LOADI', 10, 't6')
t7 = t5 - t6	('SUB', 't5', 't6', 't7')
c = t7	('STOREVAR', 't7', 'c')

Demo: 3AC

Optimization

- A lot of compiler optimization techniques involve analysis of 3AC IR
- Example: peephole optimization

```
t1 = 2  
t2 = a  
t3 = t1 * t2  
t4 = b  
t5 = t3 + t4  
t6 = 10  
t7 = t5 - t6  
c = t7  
t8 = 10  
t9 = c  
t10 = t8 * t9  
d = t10
```



```
t1 = 2  
t2 = a  
t3 = t1 * t2  
t4 = b  
t5 = t3 + t4  
t6 = 10  
t7 = t5 - t6  
c = t7  
t10 = t6 * t7  
d = t10
```

Optimization

- Example: Subexpression elimination

$$(x+y)/2 + (x+y)/4$$

```
t1 = x  
t2 = y  
t3 = t1 + t2  
t4 = 2  
t5 = t3 / t4
```

```
t6 = x  
t7 = y  
t8 = t1 + t2  
t9 = 4  
t10 = t8 / t9
```



```
t1 = x  
t2 = y  
t3 = t1 + t2  
t4 = 2  
t5 = t3 / t4
```

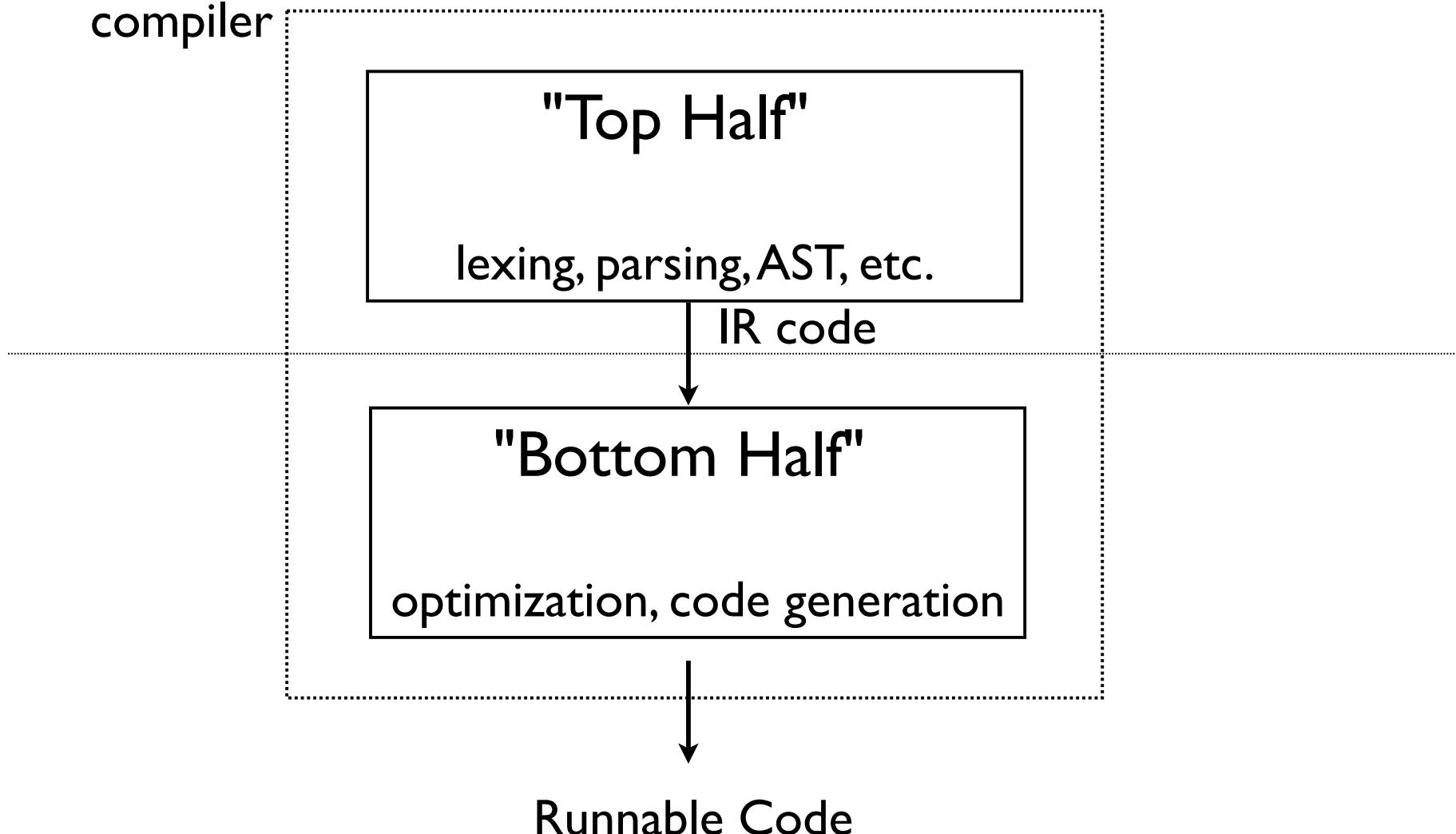
```
t9 = 4  
t10 = t3 / t9
```

Commentary

- 3AC IR is a very common
- Tends to simplify compiler implementation by separating compiler into two halves

Compiler Design

compiler



SSA Code

- Single Static Assignment
- A variant of 3-address code
 - Can never assign variables more than once
 - Assignments always go to new vars
- Imagine a register machine with infinite registers
- Registers never get overwritten (1-time use)

Exercise and Project 4-5

Part 6

Control Flow

Control Flow

- Programming languages have control-flow

```
if a < b:
```

```
    ...
```

```
else:
```

```
    ...
```

```
while a < b:
```

```
    ...
```

- Introduces branching to the underlying code

Relations

- First need relational operations

a < b
a <= b
a > b
a >= b
a == b
a != b

- And you need booleans

a and b
a or n
not a

Type System (Revisited)

- Relations add new complexity to type system
- Operators result different type than operands

```
a = 2  
b = 3
```

```
a < b      # int < int -> bool
```

- What is a truth value?

```
if a:          # Legal or not?  
    ...
```

- Both require thought in type system

Exercise and Project 6

Basic Blocks

- So far, we have focused on simple statements

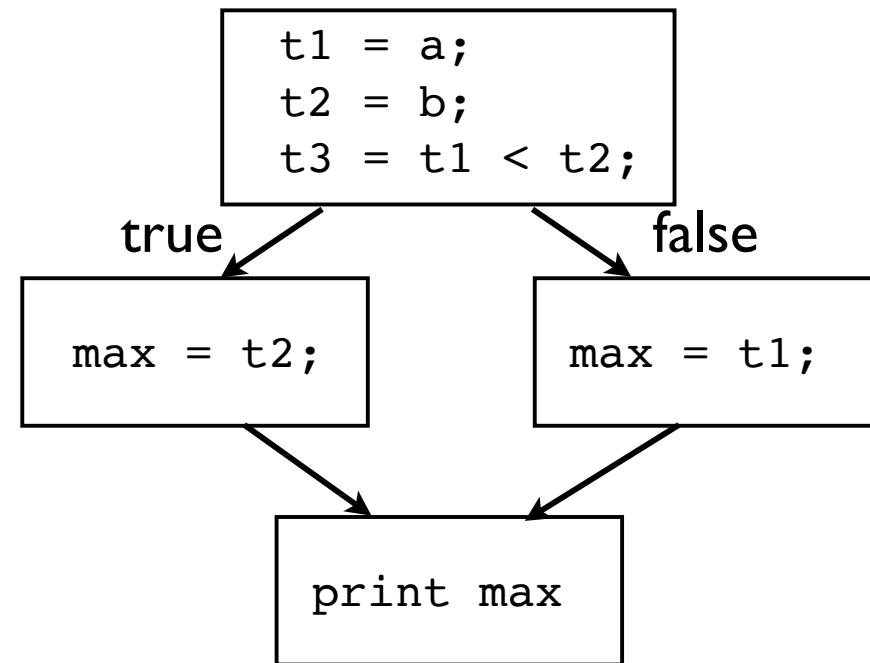
```
var a int = 2;  
var b int = 3;  
var c int = a + b;  
print(2*c);  
...
```

- A sequence of statements with no change in control-flow is known as a "basic block"

Control-Flow

- Control flow statements break code into basic blocks connected in a graph

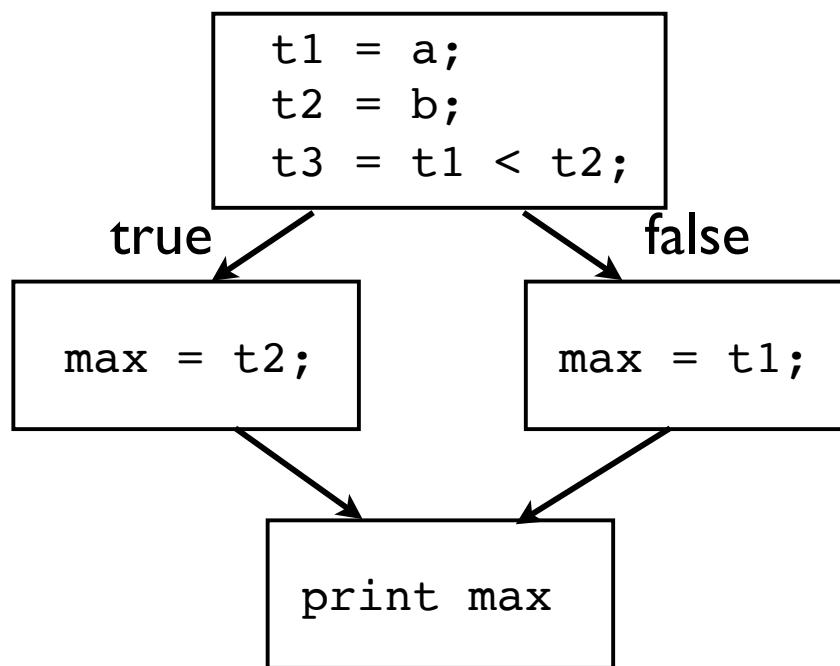
```
var a int = 2;  
var b int = 3;  
var max int;  
  
if a < b {  
    max = b;  
} else {  
    max = a;  
}  
  
print max;
```



- Control flow graph

Code Generation

- Code generation will build all of the blocks and link them with jump statements



```
b1: t1 = a;  
t2 = b;  
t3 = t1 < t2;  
jump_true t3, b2, b3  
  
b2: max = t2  
jump b4  
  
b3: max = t1  
jump b4  
  
b4: print max
```

Implementation

- Code generator must emit unique block labels
- Blocks must be linked by jump instructions
- Visit all code branches

Implementation Example

```
if a < b {  
    statements1  
} else {  
    statements2  
}
```

current block

...

Implementation Example

```
if a < b {  
    statements1  
} else {  
    statements2  
}
```

current block

```
...  
('load', 'a', 't1')  
('load', 'b', 't2')  
('lt', 't1', 't2', 't3')
```

Implementation Example

```
if a < b {  
    statements1  
} else {  
    statements2  
}
```

current block

```
...  
('load', 'a', 't1')  
('load', 'b', 't2')  
('lt', 't1', 't2', 't3')
```

Create labels

```
true_label = 'b2'  
false_label = 'b3'  
merge_label = 'b4'
```

Implementation Example

```
if a < b {  
    statements1  
} else {  
    statements2  
}
```

current block

```
...  
('load', 'a', 't1')  
('load', 'b', 't2')  
('lt', 't1', 't2', 't3')  
('jump_true', 't3', 'b2', 'b3')
```

Emit jump

```
true_label = 'b2'  
false_label = 'b3'  
merge_label = 'b4'
```

Implementation Example

```
if a < b {  
    statements1  
} else {  
    statements2  
}
```

Visit "true" branch

```
true_label = 'b2'  
false_label = 'b3'  
merge_label = 'b4'
```

current block

```
...  
('load', 'a', 't1')  
('load', 'b', 't2')  
('lt', 't1', 't2', 't3')  
('jump_true', 't3', 'b2', 'b3')
```

```
('label', 'b2')  
statements1  
('jump', 'b4')
```

Implementation Example

```
if a < b {  
    statements1  
} else {  
    statements2  
}
```

Visit "false" branch

```
true_label = 'b2'  
false_label = 'b3'  
merge_label = 'b4'
```

current block

```
...  
( 'load' , 'a' , 't1' )  
( 'load' , 'b' , 't2' )  
( 'lt' , 't1' , 't2' , 't3' )  
( 'jump_true' , 't3' , 'b2' , 'b3' )
```

```
( 'label' , 'b2' )
```

statements1

```
( 'jump' , 'b4' )
```

```
( 'label' , 'b3' )
```

statements2

```
( 'jump' , 'b4' )
```

Implementation Example

```
if a < b {  
    statements1  
} else {  
    statements2  
}
```

Create merge block

```
true_label = 'b2'  
false_label = 'b3'  
merge_label = 'b4'
```

current block

```
...  
( 'load' , 'a' , 't1' )  
( 'load' , 'b' , 't2' )  
( 'lt' , 't1' , 't2' , 't3' )  
( 'jump_true' , 't3' , 'b2' , 'b3' )
```

```
( 'label' , 'b2' )
```

```
statements1
```

```
( 'jump' , 'b4' )
```

```
( 'label' , 'b3' )
```

```
statements2
```

```
( 'jump' , 'b4' )
```

```
( 'label' , 'b4' )
```

```
...
```

Complexity: Break

- break

```
while x > 0 {  
    if x == 5 {  
        break;  
    }  
    x = x - 1;  
}
```

- Alters control-flow inside while-loop
- Similar: continue statement

Complexity: Short-circuit

- Consider boolean expressions

```
def spam():
    print('Spam')

a = 2
b = 3
if a < b or spam():
    pass
```

- Right operand not evaluated if left is true.
- A hidden control-flow change in evaluation

Complexity: Return

- Handling of the return statement

```
func spam(x int) int {  
    if x > 0 {  
        return x + 1;  
    }  
}
```

- Notice: no return statement on false
- Compiler would need to check for it

Complexity: Return

- Note: Returns simplified if there is only one

```
func spam(x int) int {
    _result = 0;
    if x > 0 {
        _result = x + 1;
        goto _return;
    }

    _return:
        return _result;
}
```

- Notice: no return statement on false
- Compiler would need to check for it

Exercise and Project 7

Part 8

Functions

Functions

- Programming languages let you define functions

```
def add(x,y):
```

```
    return x+y
```

```
def countdown(n):
```

```
    while n > 0:
```

```
        print("T-minus",n)
```

```
        n -= 1
```

```
    print("Boom! ")
```

- Two problems:

- Scoping of identifiers

- Runtime implementation

Function Scoping

- Most languages use lexical scoping
- Pertains to visibility of identifiers

```
a = 13
def foo():
    b = 42
    print(a,b)          # a,b are visible

def bar():
    c = 13
    print(a,b)          # a,c are visible
                           # b is not visible
```

- Identifiers defined in enclosing source code context of a particular statement are visible

Python Scoping

- Python uses two-level scoping
 - Global scope (module-level)
 - Local scope (function bodies)

```
a = 13                      # Global
def foo():
    b = 42                  # Local
    print(a,b)
```

Block Scoping

- Some languages use block scoping (e.g., C)

```
int a = 1;                  / Global
int foo() {
    int n = 0;              / Local. Visible in entire func
    while (n < 10) {
        int x = 2;          / Block. Visible only in 'while'
        ...
    }
    printf("%d\n",x); / Error. x not defined
}
```

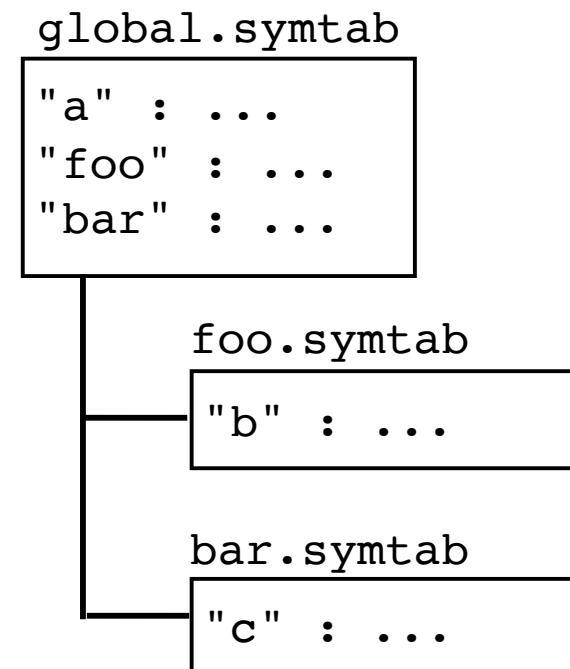
- Not in Python though...

Scope Implementation

- In the compiler: nested symbol tables

```
a = 13
def foo():
    b = 42
    print(a,b)
```

```
def bar():
    c = 13
    print(a,b)
```



- Symbol table lookup checks all parents

Function Runtime

- Each invocation of a function creates a new environment of local variables
- Known as an activation frame (or record)
- Activation frames make up the call stack

Activation Frames

```
def foo(a,b):  
    c = a+b  
    bar(c)  
  
def bar(x):  
    y = 2*x  
    spam(y)  
  
def spam(z):  
    return 10*z  
  
foo(1,2)
```

Activation Frames

```
def foo(a,b):  
    c = a+b  
    bar(c)
```

```
def bar(x):  
    y = 2*x  
    spam(y)
```

```
def spam(z):  
    return 10*z
```

```
foo(1,2)
```

foo

a	:	1
b	:	2
c	:	3

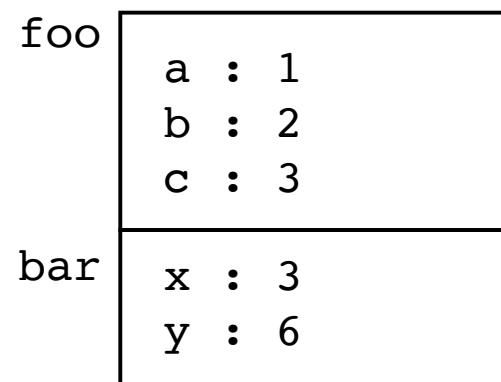
Activation Frames

```
def foo(a,b):  
    c = a+b  
    bar(c)
```

```
def bar(x):  
    y = 2*x  
    spam(y)
```

```
def spam(z):  
    return 10*z
```

```
foo(1,2)
```



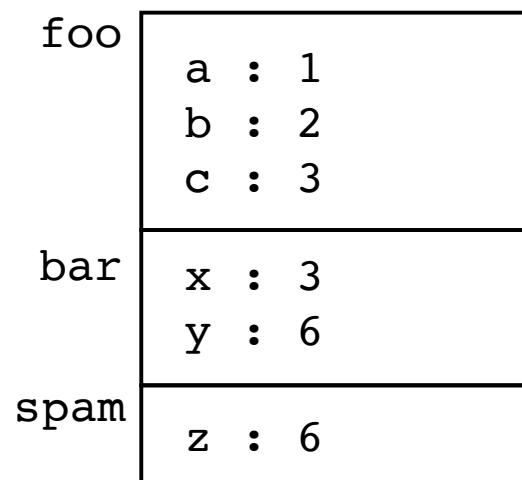
Activation Frames

```
def foo(a,b):
    c = a+b
    bar(c)

def bar(x):
    y = 2*x
    spam(y)

def spam(z):
    return 10*z

foo(1,2)
```



Activation Frames

- You see frames in tracebacks

```
File "expr.py", line 20, in <module>
    exprcheck.check_program(program)
File "/Users/beazley/Desktop/Compiler/compilers/exprcheck.py", line 410, in check_program
    checker.visit(node)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 238, in visit
    return visitor(node)
File "/Users/beazley/Desktop/Compiler/compilers/exprcheck.py", line 163, in visit_Program
    self.visit(node.statements)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 238, in visit
    return visitor(node)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 253, in generic_visit
    self.visit(item)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 238, in visit
    return visitor(node)
File "/Users/beazley/Desktop/Compiler/compilers/exprcheck.py", line 350, in visit_FuncDeclar
    self.visit(node.statements)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 238, in visit
    return visitor(node)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 253, in generic_visit
    self.visit(item)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 238, in visit
    return visitor(node)
File "/Users/beazley/Desktop/Compiler/compilers/exprcheck.py", line 303, in visit_IfStatemer
    self.visit(node.if_statements)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 238, in visit
    return visitor(node)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 253, in generic_visit
    self.visit(item)
```

Activation Frames

- You can inspect frames using `sys._getframe(n)`

```
import sys

def spam(a, b):
    c = a+b
    bar(c)

def bar(x):
    f = sys._getframe(1)
    print(f.f_locals)
```

- So called "frame hacking"

Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)      (caller)
```

```
def foo(x,y):          (callee)
    z = x + y
    return z
```

Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)
```

creates →

```
def foo(x,y):  
    z = x + y  
    return z
```

x : 1
y : 2
return : None

Caller is responsible for creating new frame and populating it with input arguments.

Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)
```

creates →

```
def foo(x,y):  
    z = x + y  
    return z
```

x : 1
y : 2
return : None

Semantic Issue: What does the frame contain?

Copies of the arguments? (Pass by value)

Pointers to the arguments? (Pass by reference)

Depends on the language

Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)
def foo(x,y):
    z = x + y
    return z
```

x : 1
y : 2
return : None
Return PC

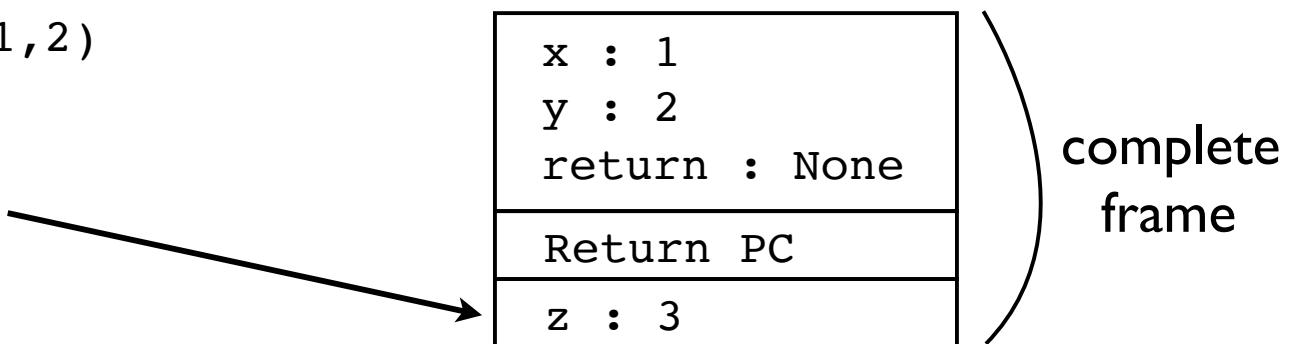
Return address (PC) recorded in the frame (so system knows how to get back to the caller upon return)

Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)
```

```
def foo(x,y):  
    z = x + y  
    return z
```



Local variables get added to the frame by the callee

Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)
```

```
def foo(x,y):  
    z = x + y  
    return z
```

Return result
placed in frame

x : 1
y : 2
return : 3
Return PC
z : 3

Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)
def foo(x,y):
    z = x + y
    return z
```

```
x : 1
y : 2
return : 3
```

callee destroys its part
of the frame on return

Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)
```

caller destroys
remaining frame on
assignment of result

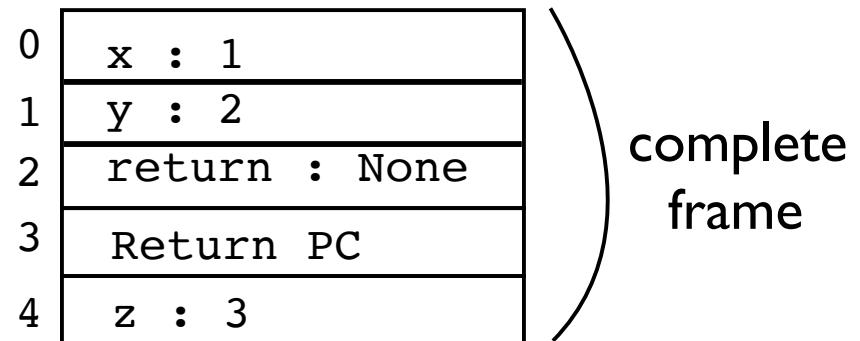
```
def foo(x,y):  
    z = x + y  
    return z
```

Frame Management

- Implementation Detail : Frame often organized as an array of numeric "slots"

```
result = foo(1,2)
```

```
def foo(x,y):  
    z = x + y  
    return z
```



- Slot numbers used in low-level instructions
- Determined at compile-time

ABIs

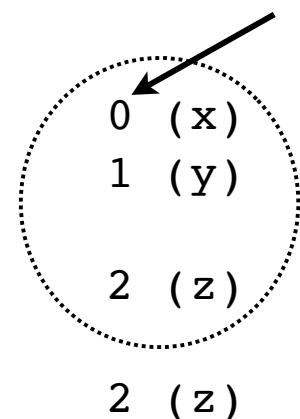
- Application Binary Interface
- A precise specification of function/procedure call semantics related to activation frames
- Language agnostic
- Critical part of creating programming libraries, DLLs, modules, etc.
- Different than an API (higher level)

Frame Example

- Python Disassembly

```
def foo(x,y):  
    z = x + y  
    return z  
  
>>> import dis  
>>> dis.dis(foo)  
 2           0 LOAD_FAST  
             3 LOAD_FAST  
             6 BINARY_ADD  
             7 STORE_FAST  
  
 3           10 LOAD_FAST  
             13 RETURN_VALUE  
  
>>>
```

numbers refer to "slots" in
the activation frame



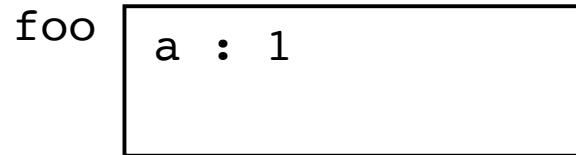
Tail Call Optimization

- Sometimes the compiler can eliminate frames

```
def foo(a):  
    ...  
    return bar(a-1)
```

```
def bar(a):  
    ...  
    return result
```

```
foo(1)
```

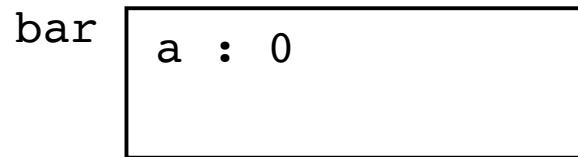


compiler detects that no
more statements follow

Tail Call Optimization

- Sometimes the compiler can eliminate frames

```
def foo(a):  
    ...  
    return bar(a-1)  
  
def bar(a):  
    ...  
    return result  
  
foo(1)
```



compiler reuses the same stack frame and just jumps to the next procedure (goto)

- Note: Python does not do this (although people often wish that it did)

Putting it Together

- Implementing functions involves two parts
 - Compile-time analysis/scoping
 - Runtime management of frames
- Multiple symbol tables (compile-time)
- Precise protocol for managing activation frames during execution (creation, variable locations, destruction, etc.).

Program Startup

- Most programs have an entry point
- Often called main()
- Must be written by the user

Program Startup

- Compiler generates a hidden startup/initialization function that calls main()

```
func main() int {
    // Written by the programmer
    ...
    return 0;
}

func __start() int {
    // Initialization (created by compiler)
    ...
    return main();
}
```

- Primary purpose is to initialize globals

Program Startup

- Initialization example:

```
var x int = v1;
var y int = v2;
...
func main() int {
    // Written by the programmer
    ...
    return 0;
}

func __start() int {
    // Initialization (created by compiler)
    x = v1;      // Setting of global variables
    y = v2;
    return main();
}
```

Compilation Steps

- Compiler processes each function, one at a time and creates basic blocks of IR code
- Compiler tracks global initialization steps
- Automatically create special startup/init function upon completion of all other code
- Final result is a collection of functions

Exercise and Project 8