Write a Compiler (in Python)

David Beazley
http://www.dabeaz.com

July 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 3.6 (Anaconda recommended)
  - llvmlite and clang
  - SLY

Compilers

Source Code

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

compiler

Executable

`a.out`

- Example: C, C++, Java, Go

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
```

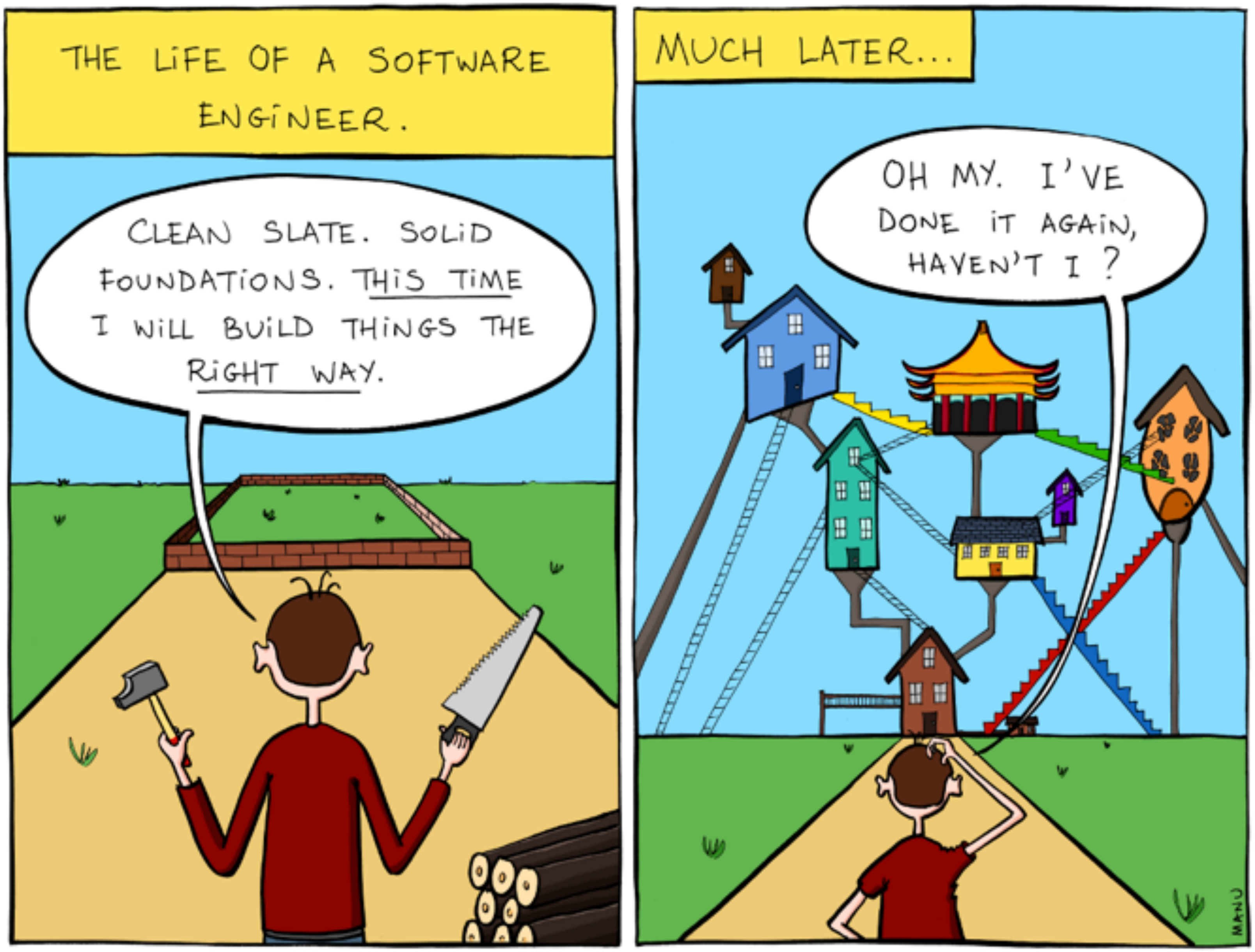
- Translation to a different language
- Example: Compilation to Javascript

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

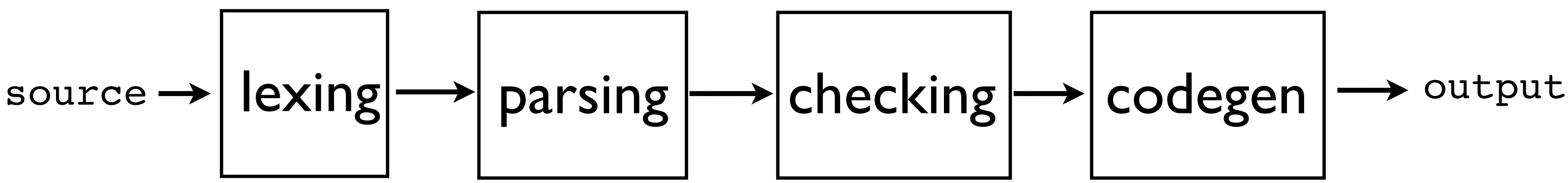
- It's one of the most complicated programming projects you will ever undertake
- Tricky issues at every turn
- Involves just about every topic in computer science (algorithms, hardware, etc.)
- Difficult software design (lots of parts)
- Few programmers dare to do it



http://www.bonkersworld.net

Behind the Scenes

- Compilers usually consist of many stages



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

Lexing

- Splits input text into tokens

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

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

- Detects illegal symbols

```
b = 40 * $5
```

Illegal Character

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

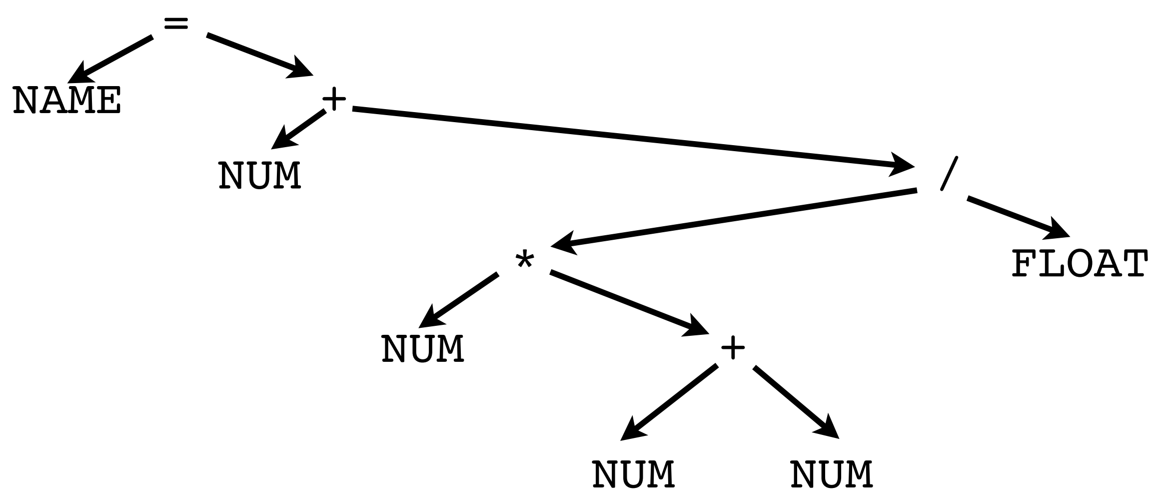
Parsing

- 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

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

- Example: + operator

```
        +
      /   \
   LHS     RHS
```

```
1. LHS and RHS must be the same type
2. If different types, must be convertible to same type
```

Code Generation

- Generation of output code:

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

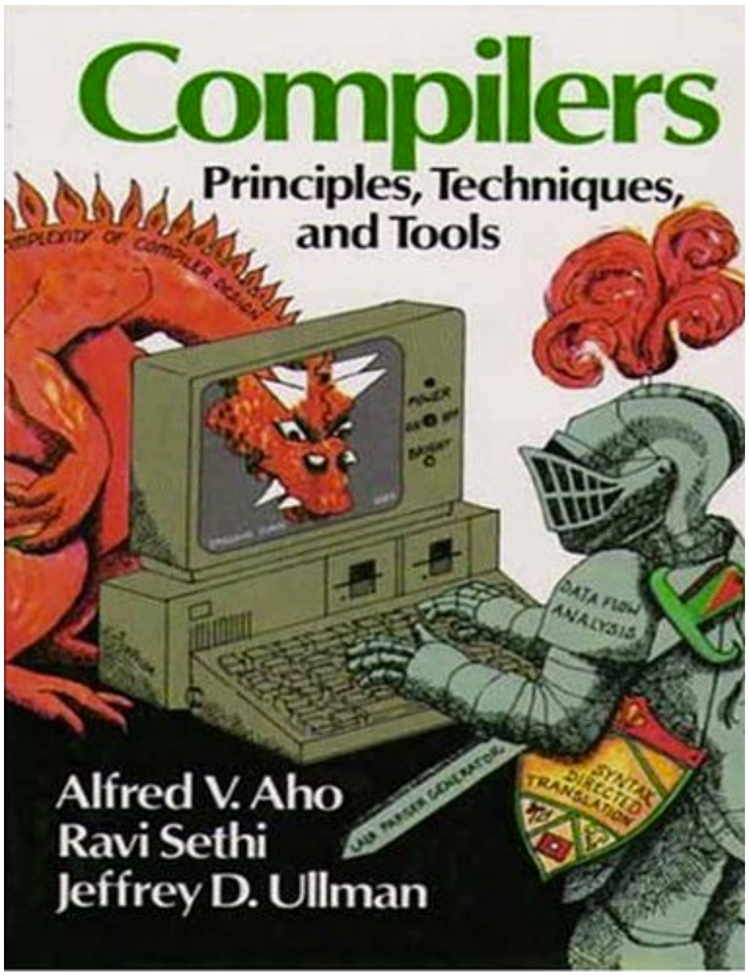
```
LOAD R1, 40
LOAD R2, 20
LOAD R3, 2
LOAD R4, 3
ADD  R3, R4, R3  ; R3 = (2+3)
MUL  R2, R3, R2  ; R2 = 20*(2+3)
LOAD R3, 37.5
DIV  R2, R3, R2  ; R2 = 20*(2+3)/37.5
ADD  R1, R2, R1  ; R1 = 40+20*(2+3)/37.5
STORE R1, "b"
```

- Note: Might generate other kinds of output

Why Write a Compiler?

- Doing so will demystify a huge amount of detail about how computers and languages work
- Confidence : If you can write a compiler, chances are you can code just about <u>anything</u> (few tasks are ever *that* insane)

Books



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

Teaching Compilers

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

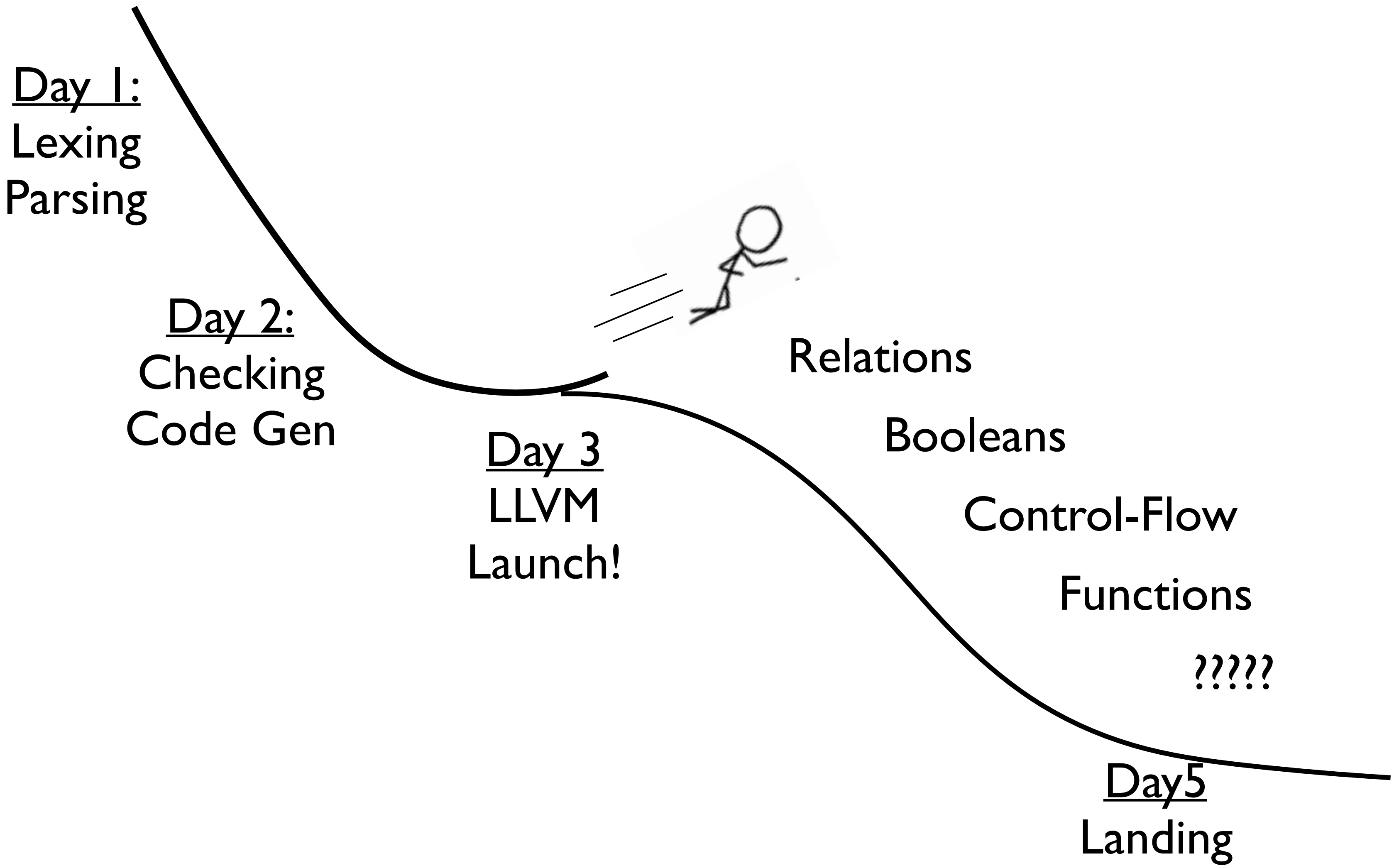
Our Approach

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

Project Overview



Exercises

- Each project is preceded by a short exercise in which important concepts are discussed
- Workflow for each project part will be:
  - Short Discussion (10-30 mins)
  - Short Exercise (10-60 mins)
  - Long Project (several hours)

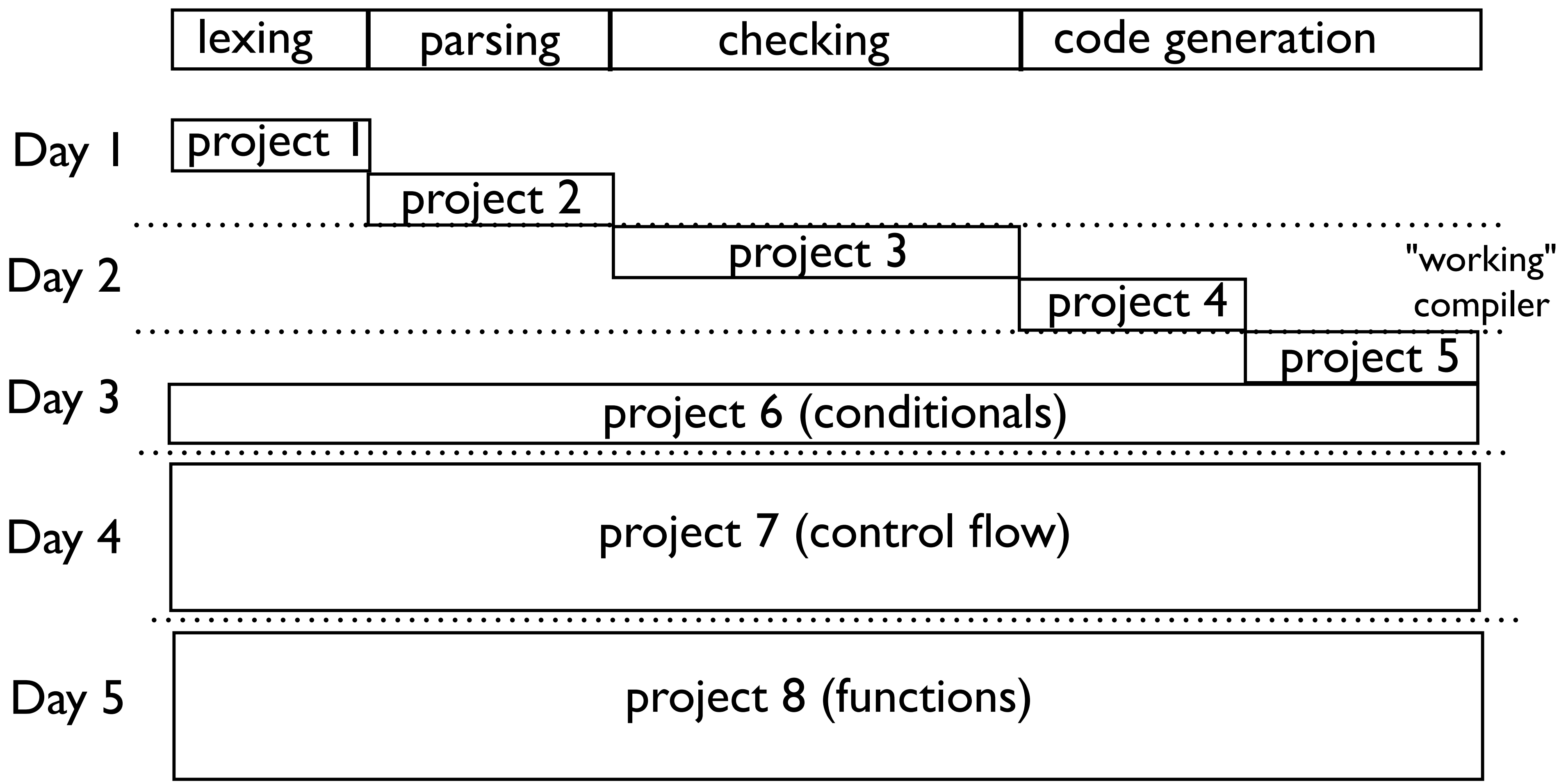
Tips

- We are going to be writing a lot of code.
- 2500-3500 lines of Python.
- I will be guiding you and giving you code fragments that point you in the right direction
- A minimal solution has been written that you can consult if you need to
- However, I think you should code it yourself

Making Progress

- Parts of the project are tricky
- It's not always necessary to solve all problems at once
- I will push you to more forward and come back to various problems later (it's okay)

Progress Overview



a much "better" compiler

Caution

- For success, you need as few distractions as possible (work, world cup, births, etc.)

Pace Yourself

- Take breaks, don't overdo it on food, etc.
- Get sleep (you'll need it by day 5)

Part 1

Lexing

Lexing in a Nutshell

- Convert input text into a token stream

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

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

- 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()
'1234'
>>> m.lastgroup
'NUM'
>>>
```

value ⟶ '1234'

type ⟶ 'NUM'

Linear Regex Scanning

- Must perform a linear text scan

start

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

- <u>ALL</u> characters must be matched
- Otherwise error:

start

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

Bad character "$"

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

- Longest matches must go 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:

notice the order

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

'<='  ⟶  [('LE','<=')]
```

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 1

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 1

Part 2

Parsing

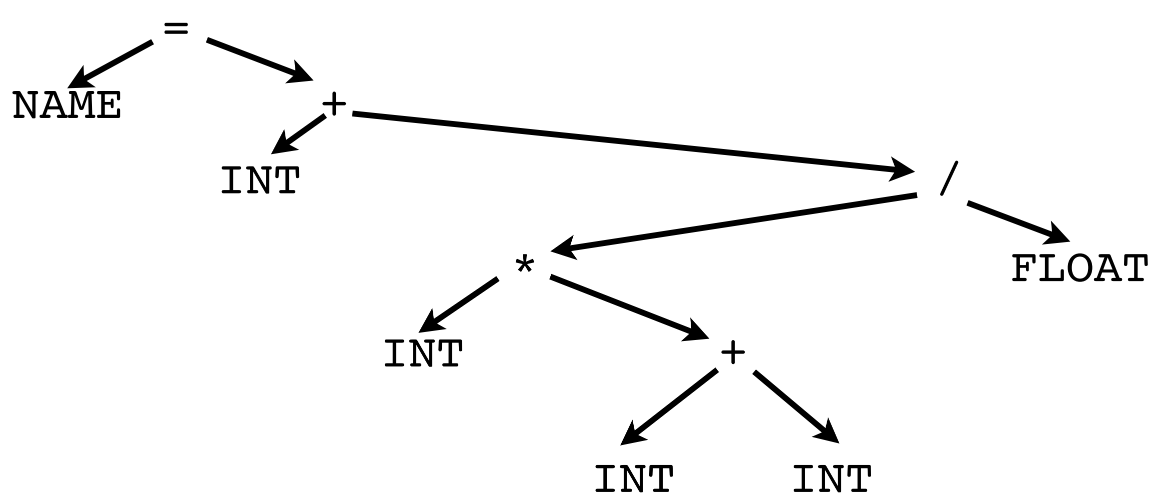
Parsing in a Nutshell

- Makes sure input is syntactically correct

```
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

Grammar Specification

- The language to be parsed must have a formal grammar specification

```
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).

Mini Exercise

- Write a grammar for Python assignment

```
assignment ::= lhs '=' rhs
```

Terminals/Nonterminals

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

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

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

nonterminals

terminals

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 ';'
```

```
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 ';'
```

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

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

Match!

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 ';'
```

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

Bottom-Up Parsing

- Example:

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

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

Bottom-Up Parsing

- Example:

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

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

Bottom-Up Parsing

- Example:

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


                expr ::= 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;
```

Bottom-Up Parsing

- Example:

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

```
                    expr ::= INT
                           |
                           v
               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;
```

Bottom-Up Parsing

- Example:

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

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

```
               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;
```

Bottom-Up Parsing

- Example:

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

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

                  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;
```

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.

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)

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 horrible!

Syntax Directed Translation

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

Grammar	Actions
`assignment ::= NAME '=' expr ';'`	`vars[NAME] = expr.val`
$expr_0$ `::=` $expr_1$ `'+'` $expr_2$	$expr_0$`.val =` $expr_1$`.val +` $expr_2$`.val`
`|` $expr_1$ `'-'` $expr_2$	$expr_0$`.val =` $expr_1$`.val -` $expr_2$`.val`
`|` $expr_1$ `'*'` $expr_2$	$expr_0$`.val =` $expr_1$`.val *` $expr_2$`.val`
`|` $expr_1$ `'/'` $expr_2$	$expr_0$`.val =` $expr_1$`.val /` $expr_2$`.val`
`|` `'('` $expr_1$ `')'`	$expr_0$`.val =` $expr_1$`.val`
`| INT`	`expr.val = INT.val`
`| FLOAT`	`expr.val = FLOAT.val`
`| NAME`	`expr.val = vars[NAME]`

- There is a propagation effect

Example

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

Goal

```
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 ';'
('a')    (2)     (3)     (4)
```

Tokens

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

Example

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

```
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 ';'
('a')    (2)     (3)     (4)
```

Tokens

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

Example

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

```
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;
```

Example

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

```
NAME '=' expr '+' expr ';'              expr.val = INT.val
('a')    (6)      (4)
                   ↑
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;
```

Example

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

```
NAME '=' expr ';'                       expr0.val = expr1.val + expr2.val
('a')    (10)

NAME '=' expr '+' expr ';'              expr.val = INT.val
('a')    (6)      (4)

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;
```

Example

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

assignment ↑ vars[NAME] = expr.val

NAME **'='** **expr** **';'**
('a') (10)

$expr_0.val = expr_1.val + expr_2.val$

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

expr.val = INT.val

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

$expr_0.val = expr_1.val * expr_2.val$

```
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

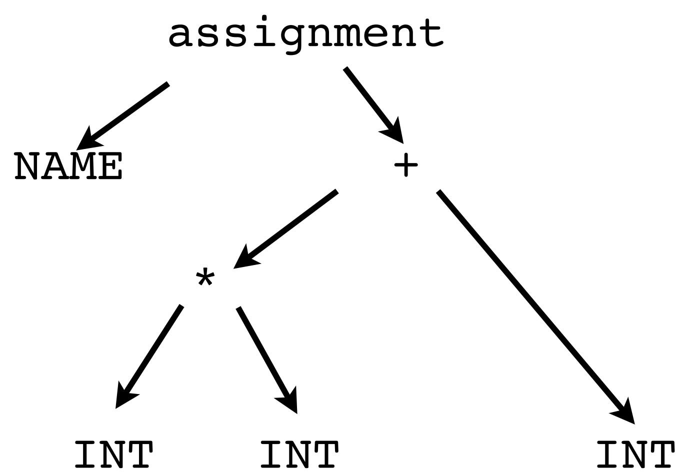
```
input text   : a = 2 * 3 + 4;
```

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

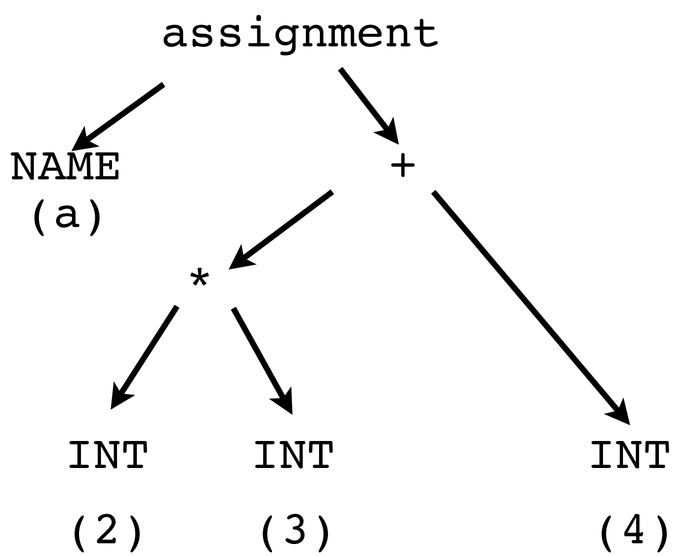
- Actions build "nodes"

Grammar	Actions
`assignment ::= NAME '=' expr ';'`	`assignment = ('store',NAME,expr)`
$expr_0$ `::=` $expr_1$ `'+'` $expr_2$	$expr_0$ `= ('+',` $expr_1$, $expr_2$`)`
`|` $expr_1$ `'-'` $expr_2$	$expr_0$ `= ('-',` $expr_1$, $expr_2$`)`
`|` $expr_1$ `'*'` $expr_2$	$expr_0$ `= ('*',` $expr_1$, $expr_2$`)`
`|` $expr_1$ `'/'` $expr_2$	$expr_0$ `= ('/',` $expr_1$, $expr_2$`)`
`|` `'('` $expr_1$ `')'`	$expr_0$ = $expr_1$
`|` `INT`	`expr = INT.val`
`|` `FLOAT`	`expr = FLOAT.val`
`|` `NAME`	`expr = ('load', NAME)`

- Example: nested tuples

AST Representation

- Nodes represented by tuples, classes, etc.




```
('assign','a',
   ('+',
       ('*', 2, 3),
       4
   )
)
```

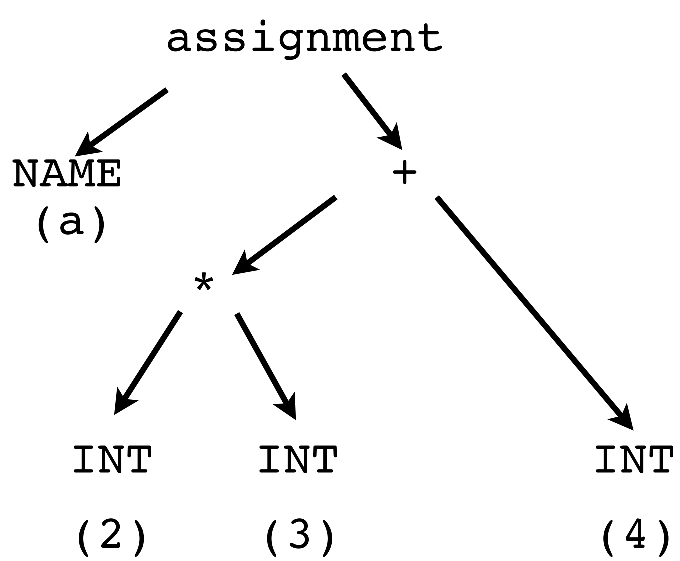
DEMO

- Tree construction with SLY

AST Representation

- Nodes represented by tuples, classes, etc.

```
class Assignment:
   ...
class Binop:
   ...
class Number:
   ...
```

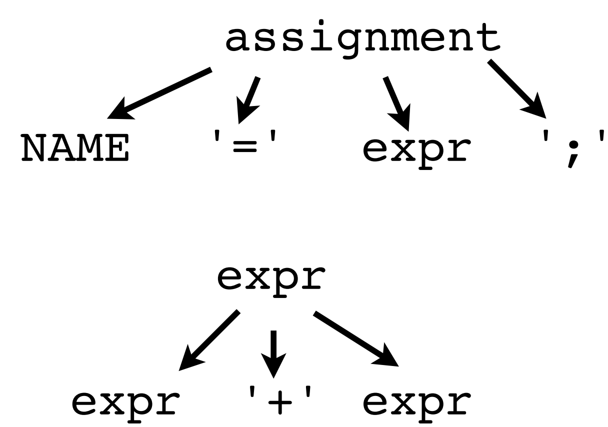



```
Assignment('a',
   Binop('+',
      Binop('*',
            Number(2),
            Number(3)),
      Number(4)
   )
)
```

Parse Trees

- Syntax trees are constructed during the application of grammar rules
- Left side : Top
- Right side : Leaves

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

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



Parse Tree Construction

- Example: Bottom-up construction

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

Parse Tree Construction

- Example: Bottom-up construction

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

```
expr ::= INT
```

```
expr
 ↓
INT
```

Parse Tree Construction

- Example: Bottom-up construction

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

```
expr ::= INT
```

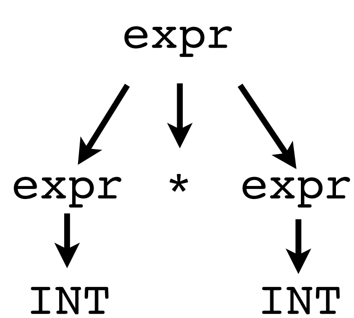
```
expr      expr
 ↓         ↓
INT       INT
```

Parse Tree Construction

- Example: Bottom-up construction

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

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

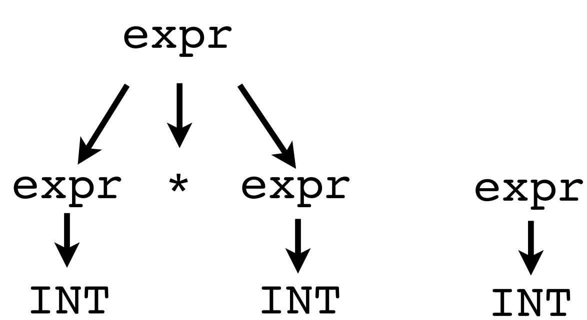


Parse Tree Construction

- Example: Bottom-up construction

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

```
expr ::= INT
```

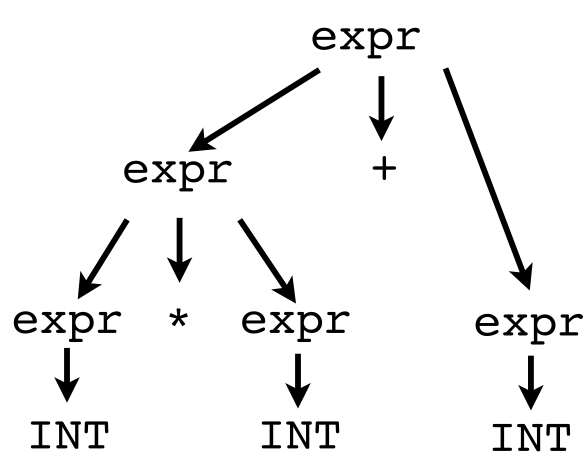


Parse Tree Construction

- Example: Bottom-up construction

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

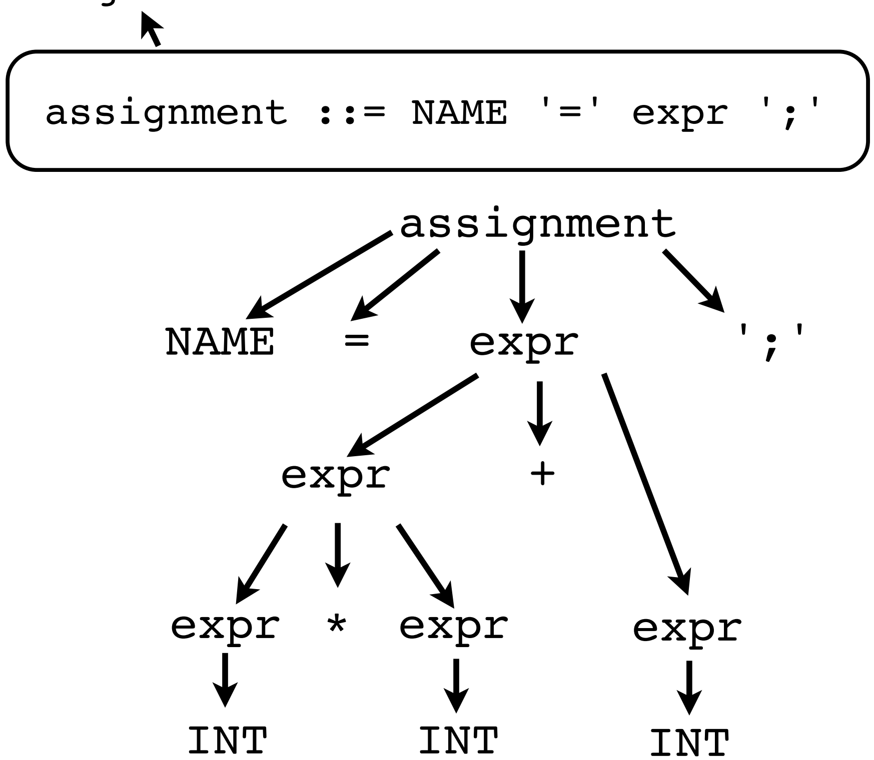
```
expr ::= expr '+' expr
```



Parse Tree Construction

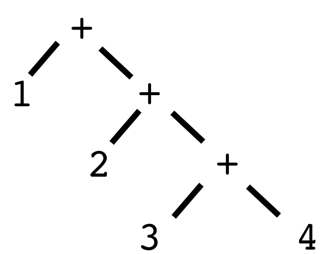
- Example: Bottom-up construction

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

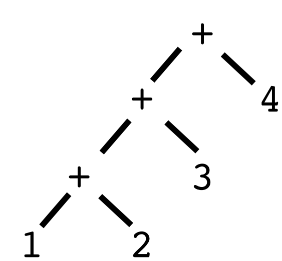


Problem : Ambiguity

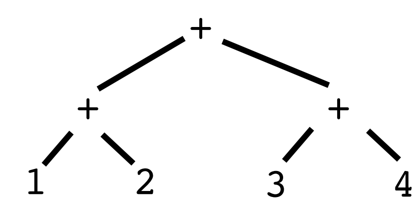
- Grammars must be unambiguous
- Consider: `1 + 2 + 3 + 4`
- There are many possible parses



or



or



- Ambiguity is bad!

Problem : Ambiguity

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

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

→

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

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

- This is forcing left associativity

Problem : Ambiguity

- Associativity example:

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

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

```
1 + 2 + 3 + 4

expr + term
(expr + term) + term
((expr + term) + term) + term
((term + term) + term) + term

((1 + 2) + 3) + 4
```

Notice how everything is grouping to the left

Problem : Precedence

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

```
          +
         / \
        *   4
       / \
      +   3
     / \
    1   2
```

wrong

```
        +
       / \
      +   4
     / \
    1   *
       / \
      2   3
```

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

Part 3

Type Checking

Types

- Programming languages have different kinds 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
>>>
```

Type Checking

- Verifying that actions found in the parse tree are valid (enforcing semantics)
- Much of it is common sense stuff
  - Can't read an undefined variable
  - Can't do operations (+,-,*,/) if not supported by the underlying datatype
  - Can't overwrite constants

Rule Specification

- Type checking rules are often attached to the grammar rules (informally)

```
assignment ::= name '=' expr ';'
    - name must be declared as variable
    - name.type == expr.type

expr ::= expr1 '+' expr2 ';'
    - expr1.type == expr.2type
    - '+' 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 specify types
  - Need a symbol table (to record info)
  - Must walk the AST and enforce rules

Type Specification

- Types have names:

```
int, float, string, bool, etc...
```

- Must have some kind of identification
- Must be comparable

```
int != float
```

Type Specification

- Types support different operators

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

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

- Checker will consult when validating

Type Specification

- Complexity: Mixed types

```
string * int -> string

'hello' * 2 -> 'hellohello'
```

- Complexity: Type Promotion

```
int + float
 ↓
float + float -> float

2 + 2.5
↓
2.0 + 2.5 -> 4.5
```

- Note: We are not addressing either problem

Symbol Tables

- Symbol table is a dictionary 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:

```
var a int;
var b int;
var c string;
```

symbol table

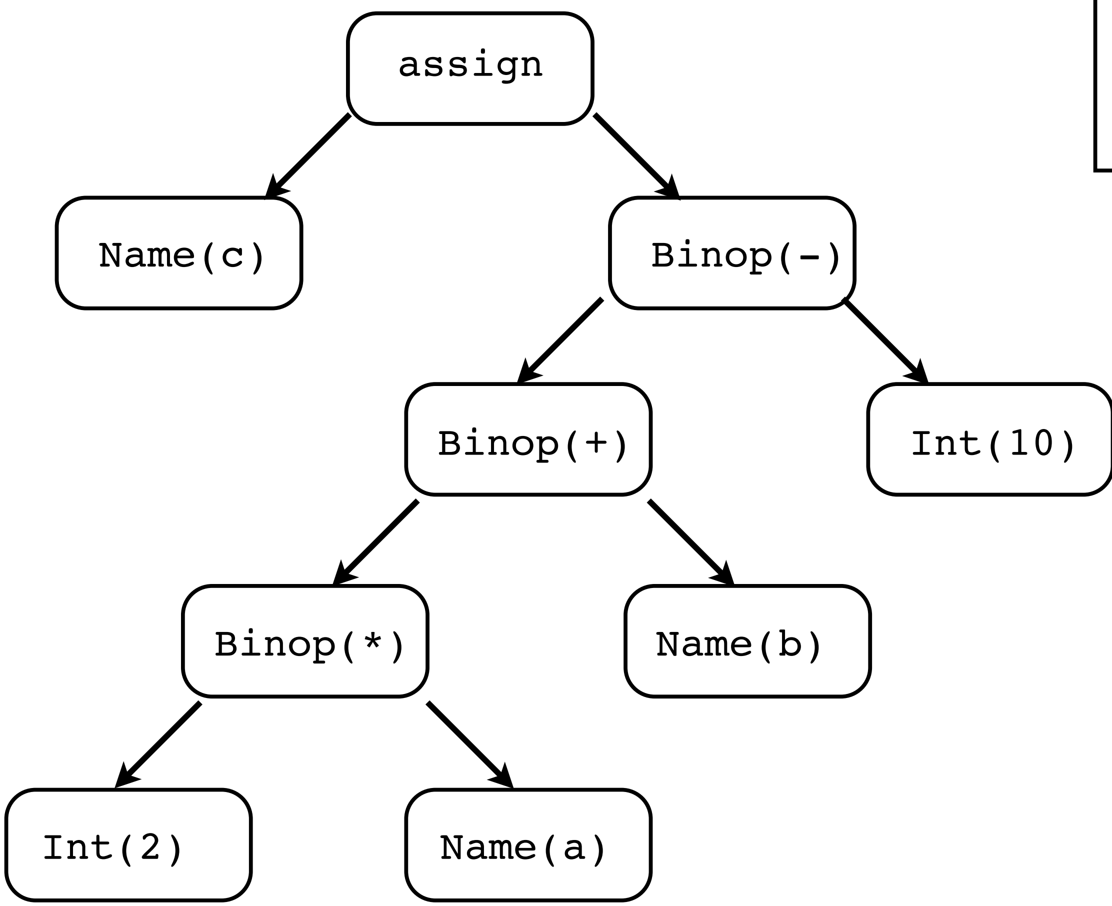
```
'a' : VarDecl(type="int"),
'b' : VarDecl(type="int"),
'c' : VarDecl(type="string")
```

- Now, consider the parsing of this statement

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

AST Annotation

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

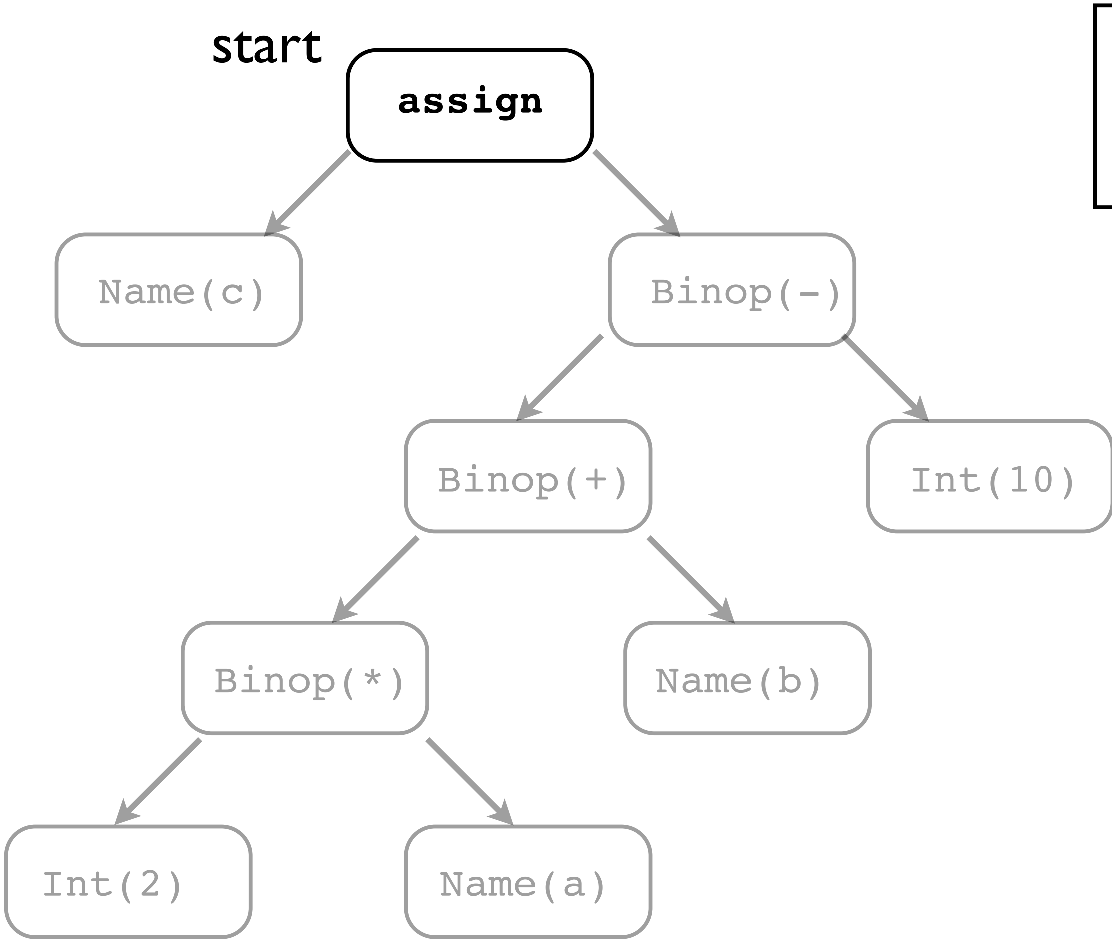


symbol table

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

AST Annotation

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

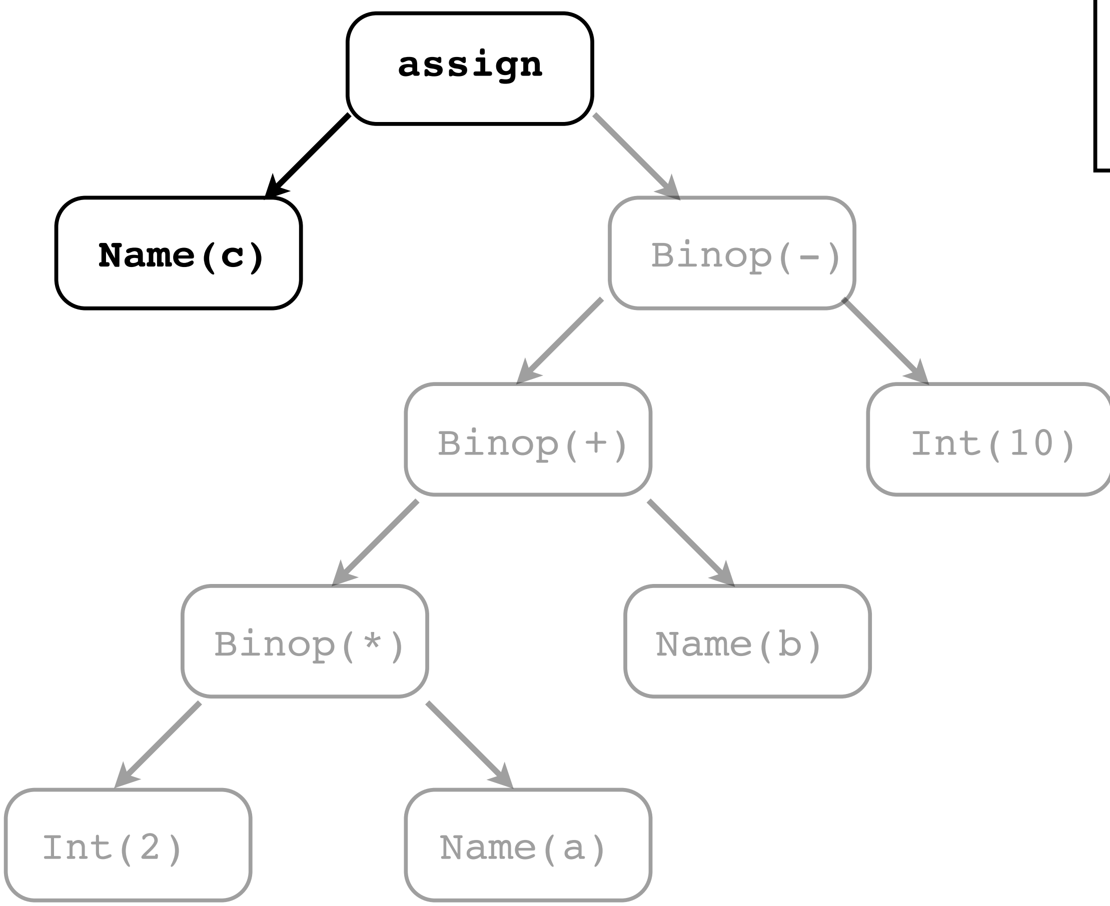


symbol table

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

AST Annotation

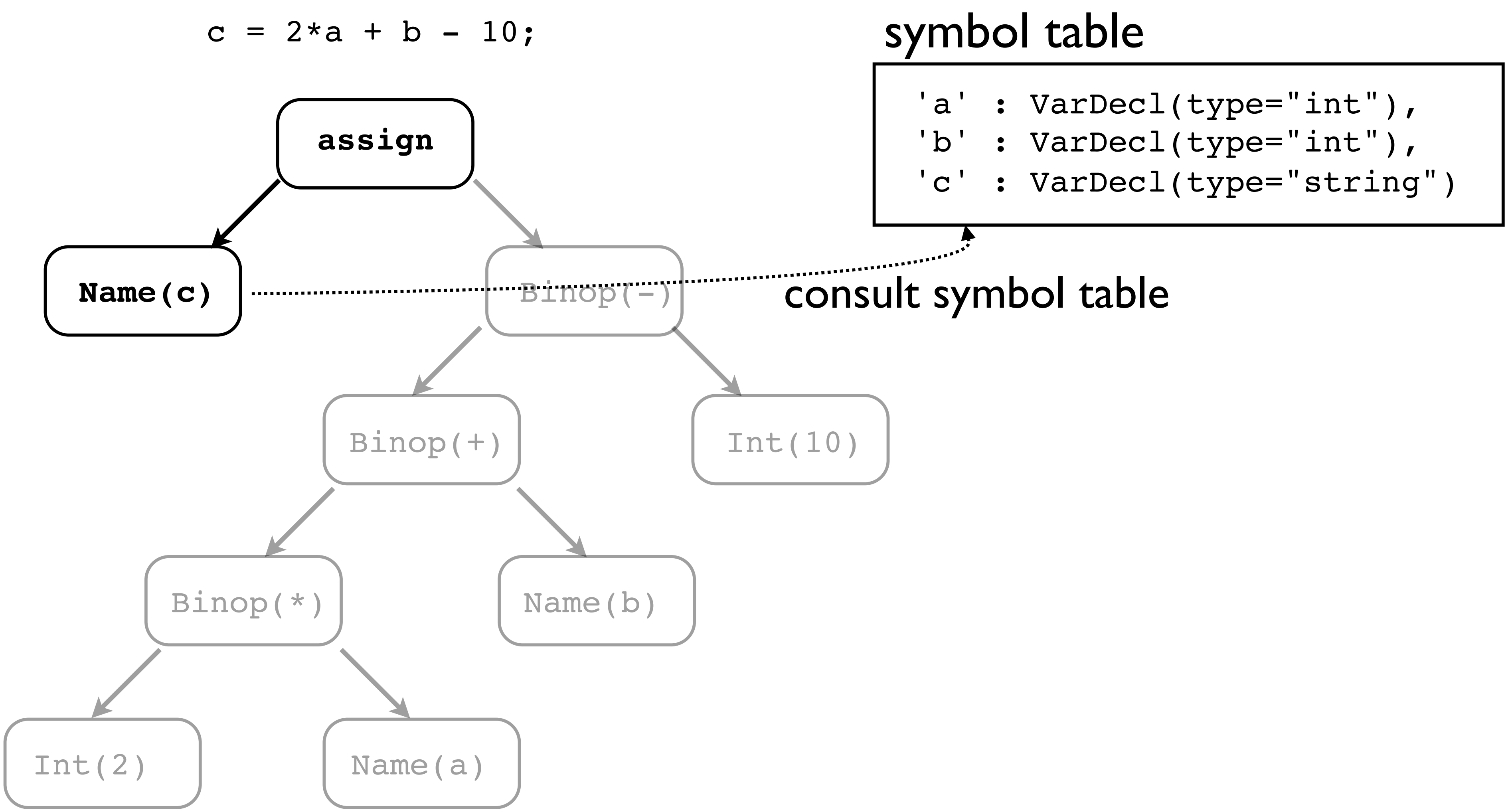
```
c = 2*a + b - 10;
```



symbol table

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

AST Annotation

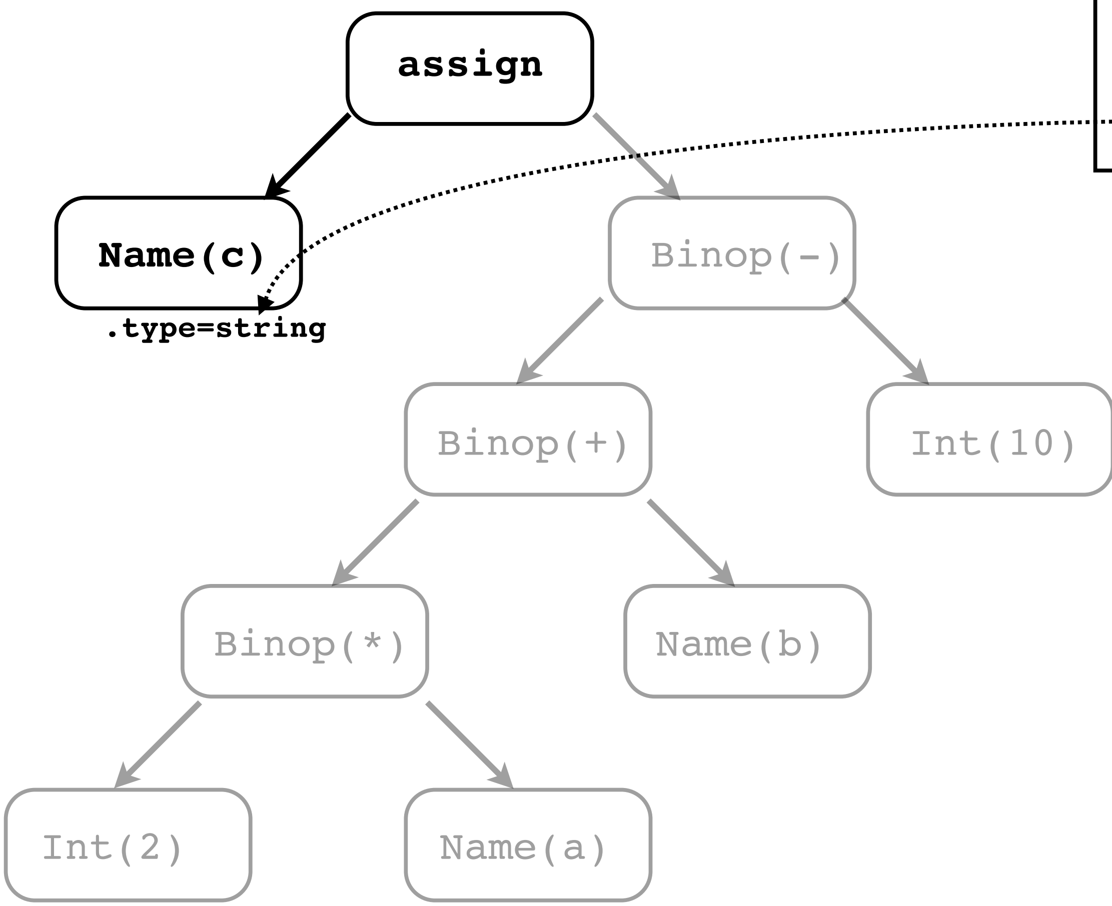


AST Annotation

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

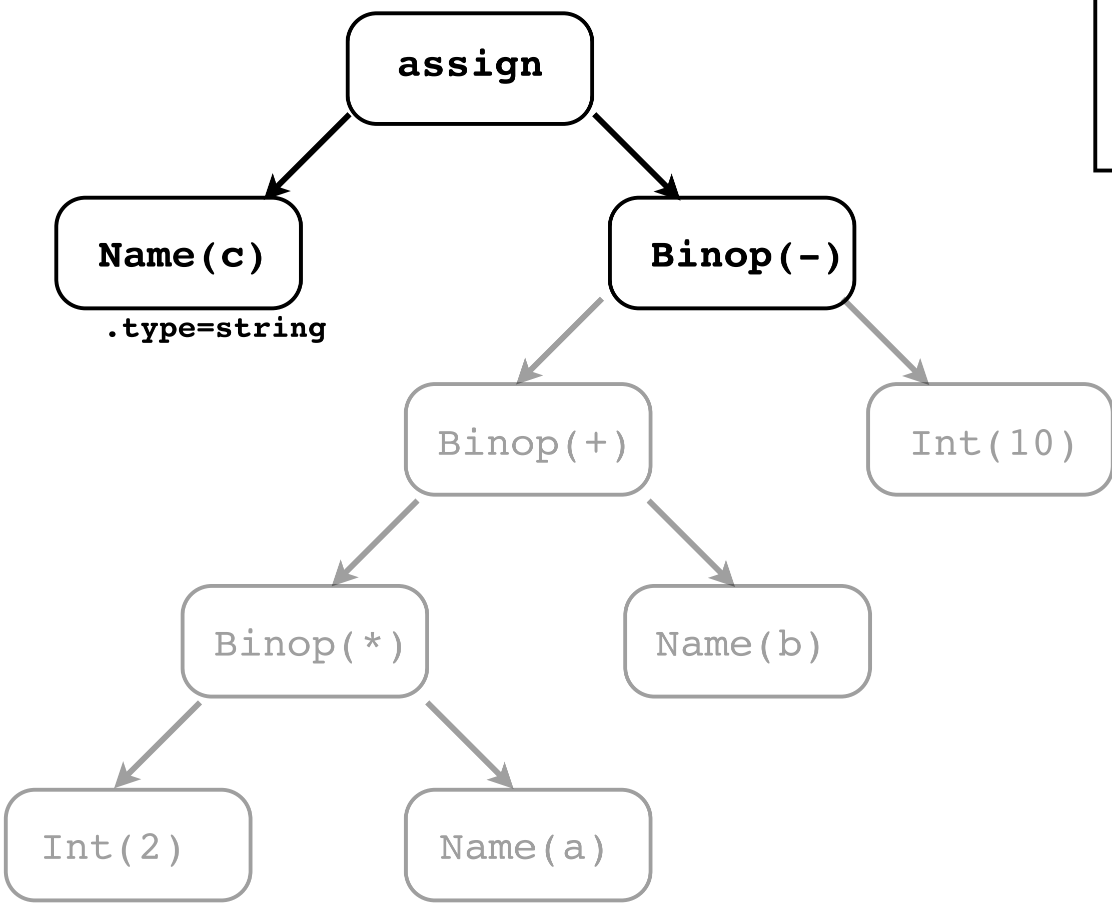
symbol table

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



AST Annotation

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

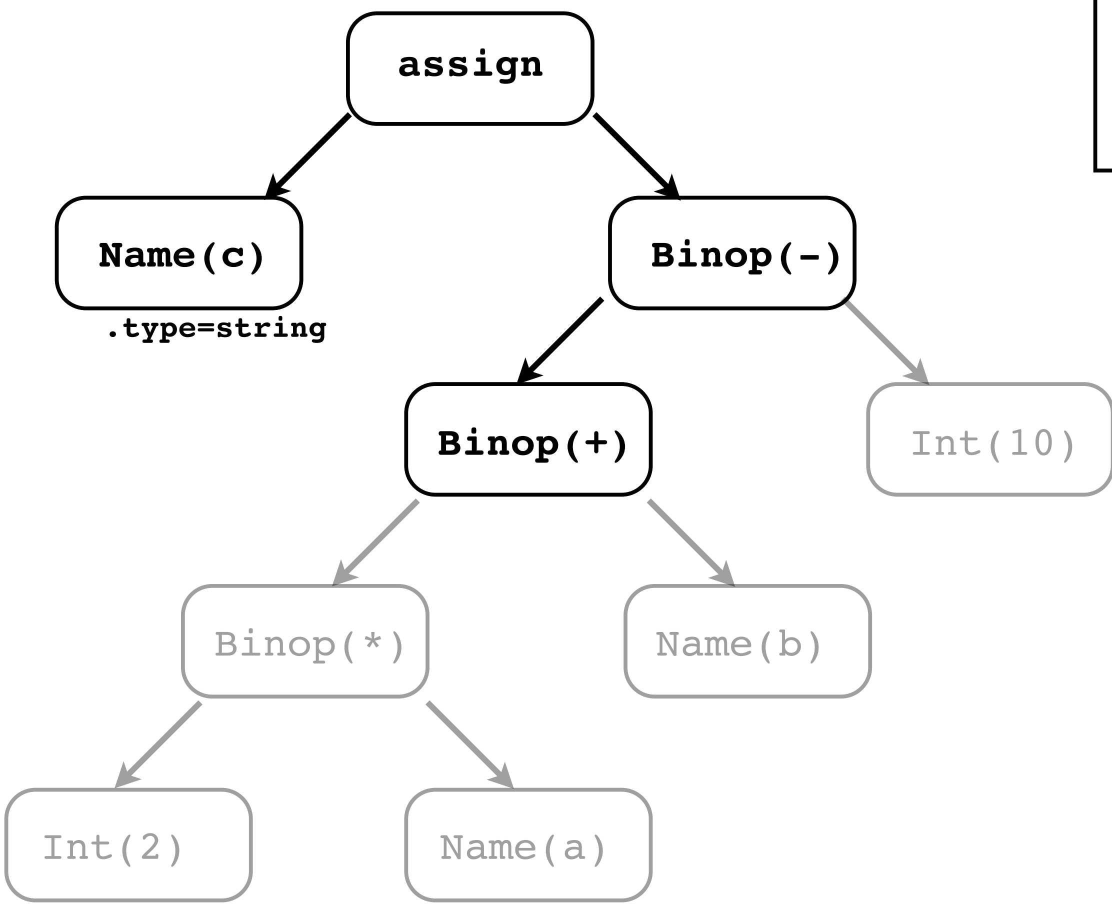


symbol table

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

AST Annotation

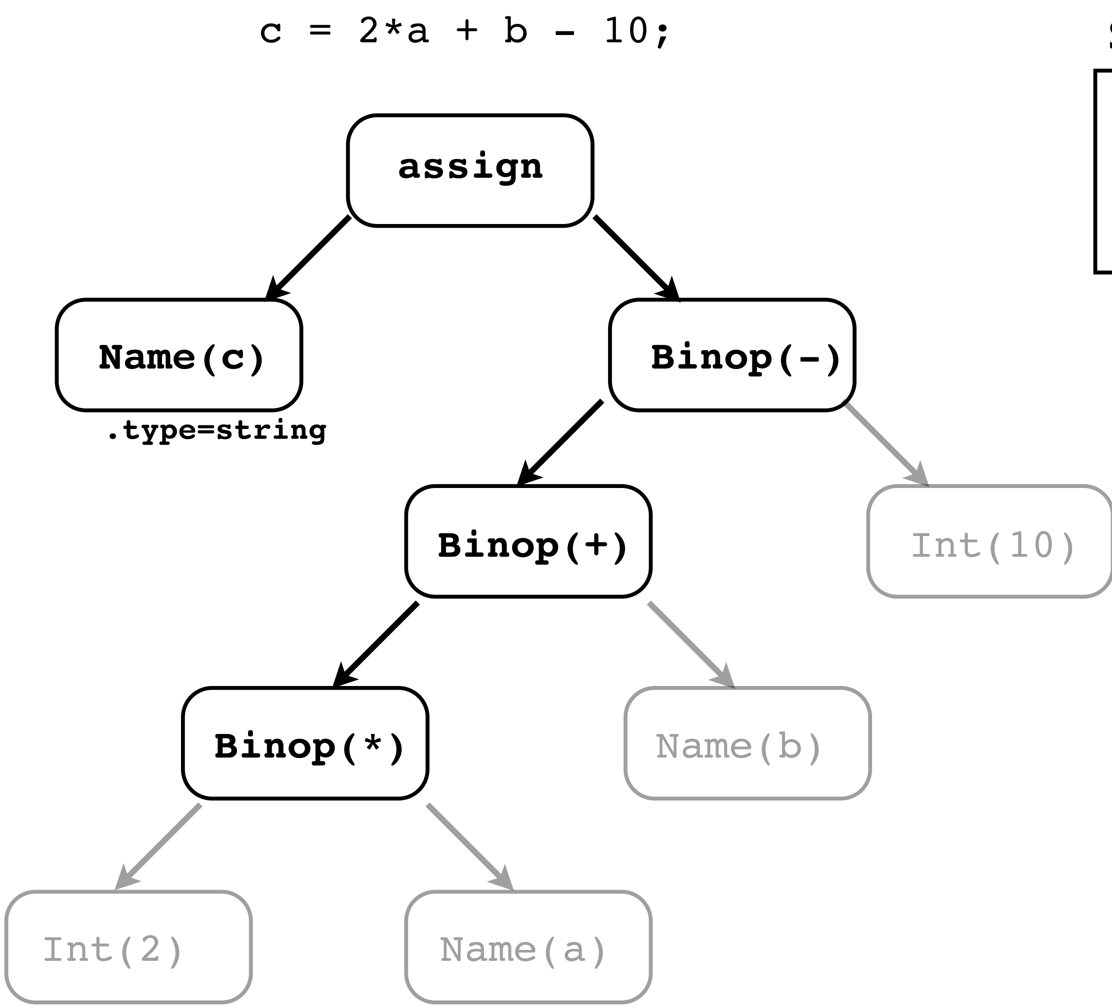
```
c = 2*a + b - 10;
```



symbol table

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

AST Annotation



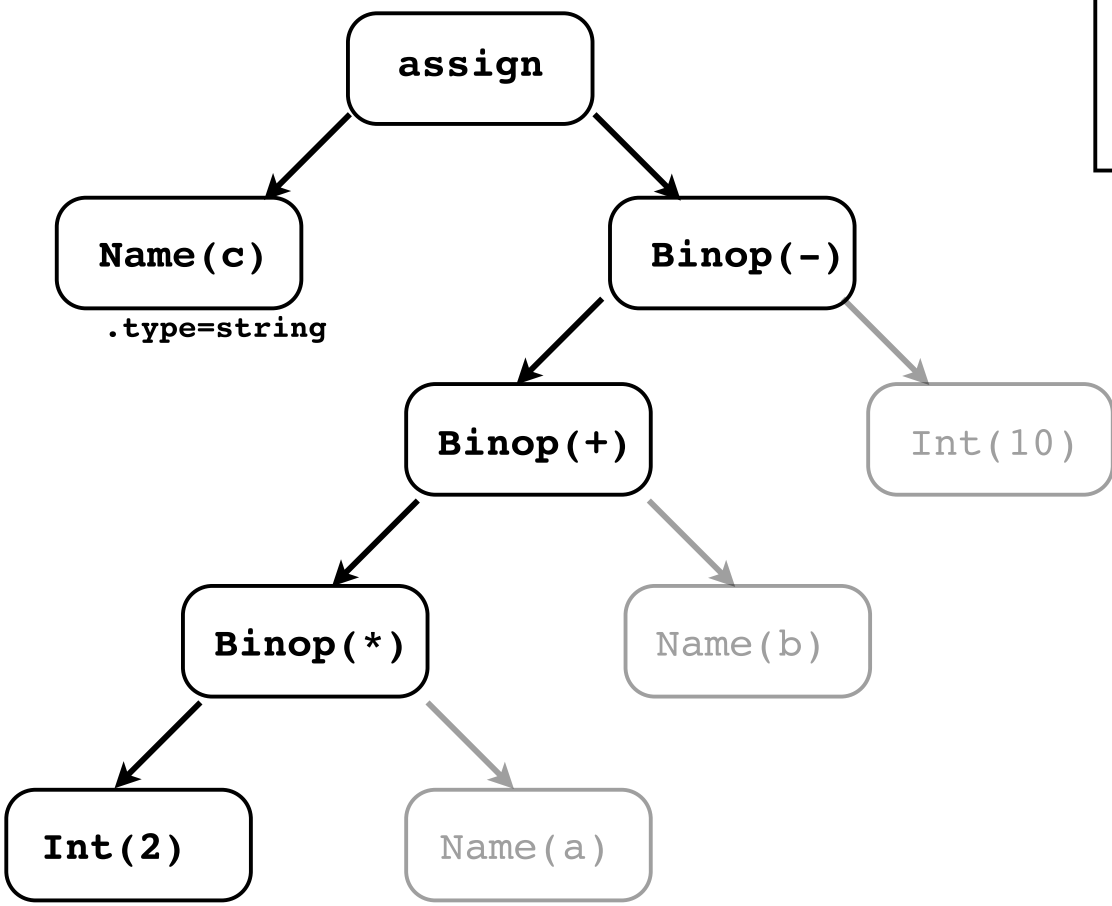
symbol table

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

Note: depth-first traversal

AST Annotation

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

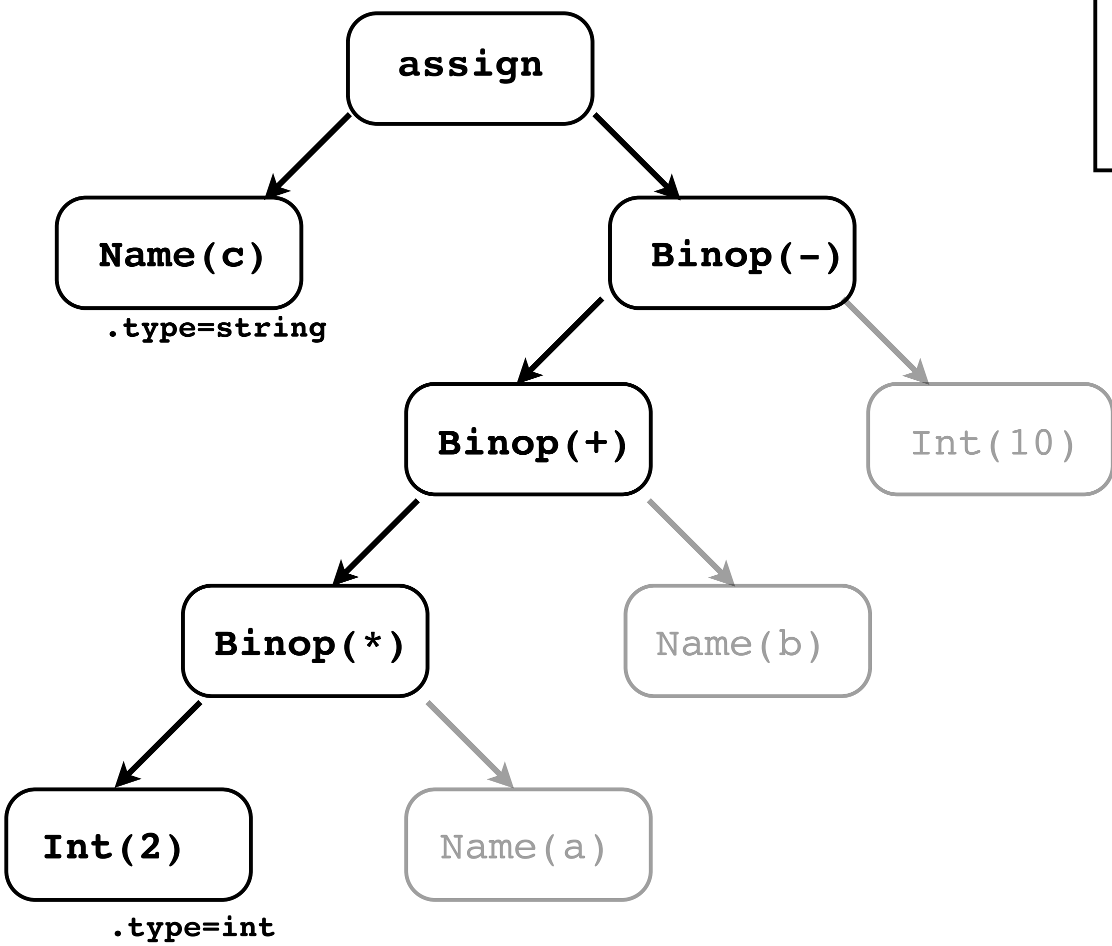


symbol table

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

AST Annotation

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

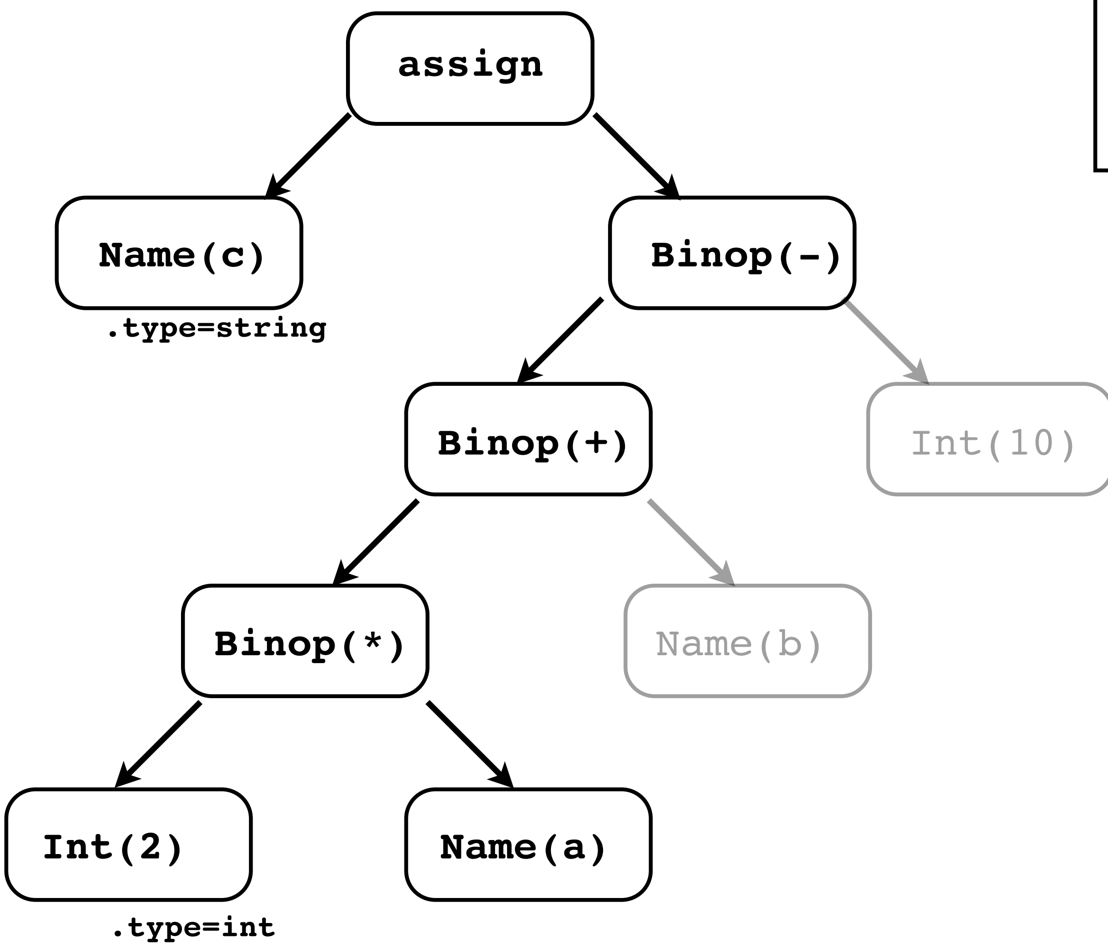


symbol table

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

AST Annotation

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

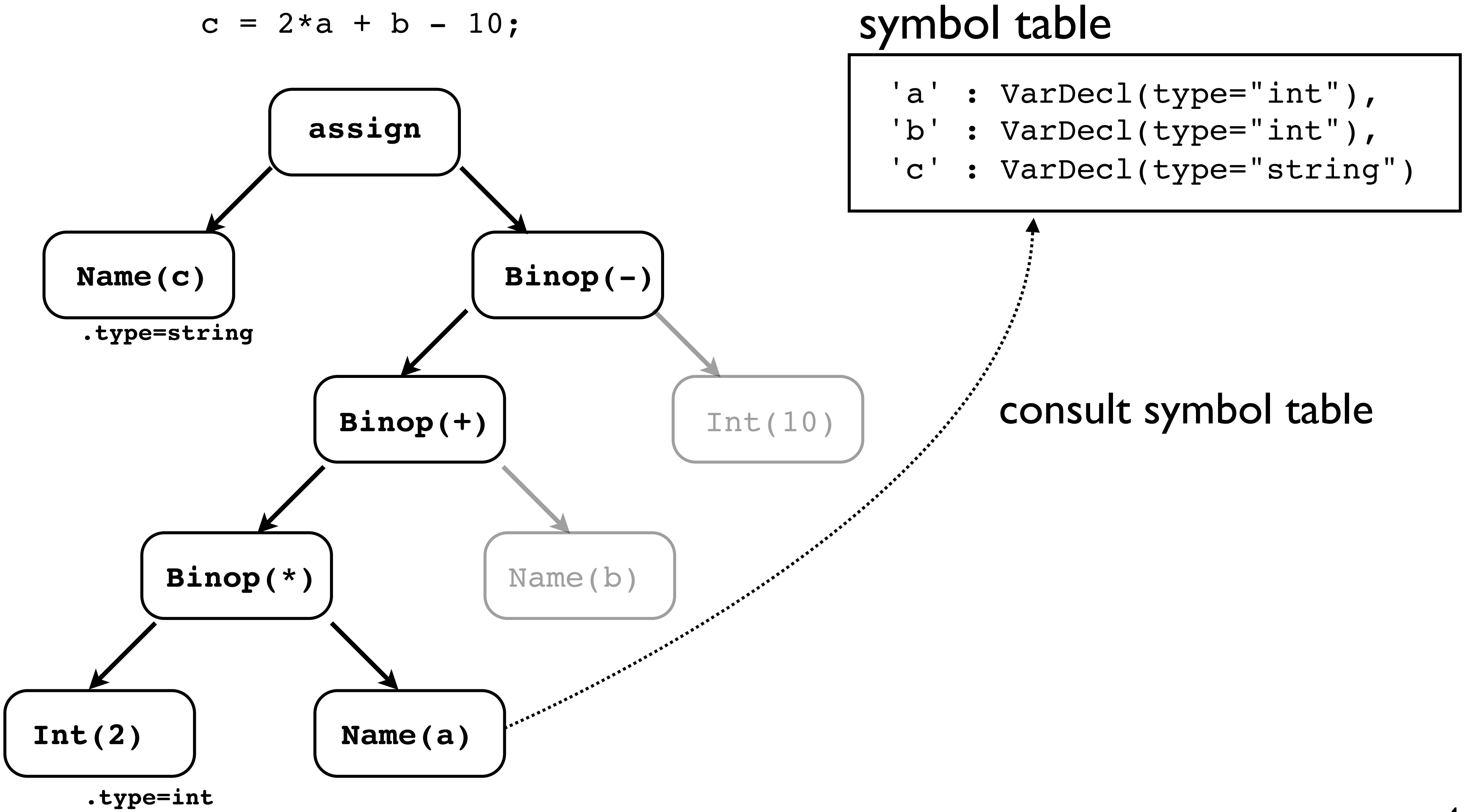


symbol table

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

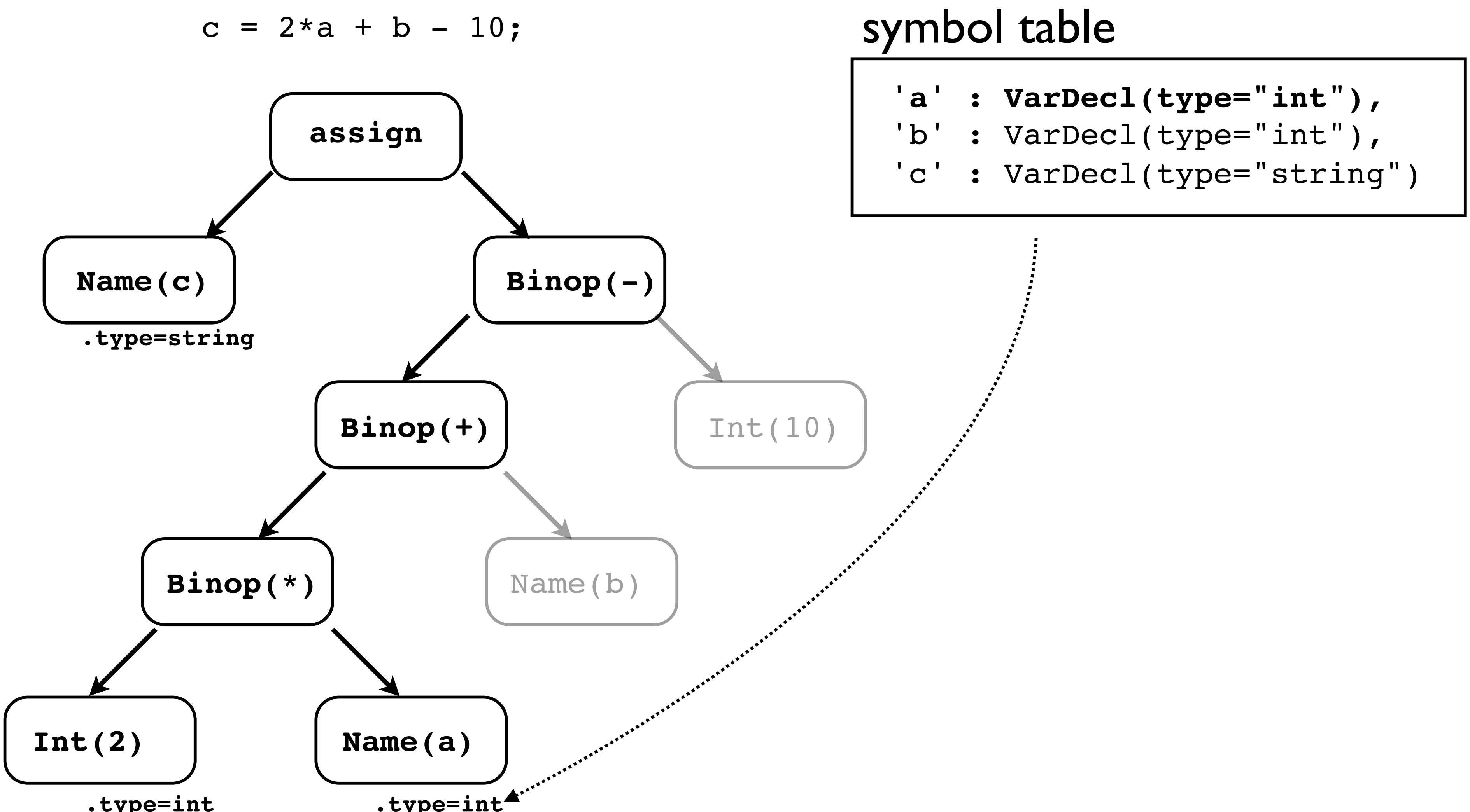
AST Annotation

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



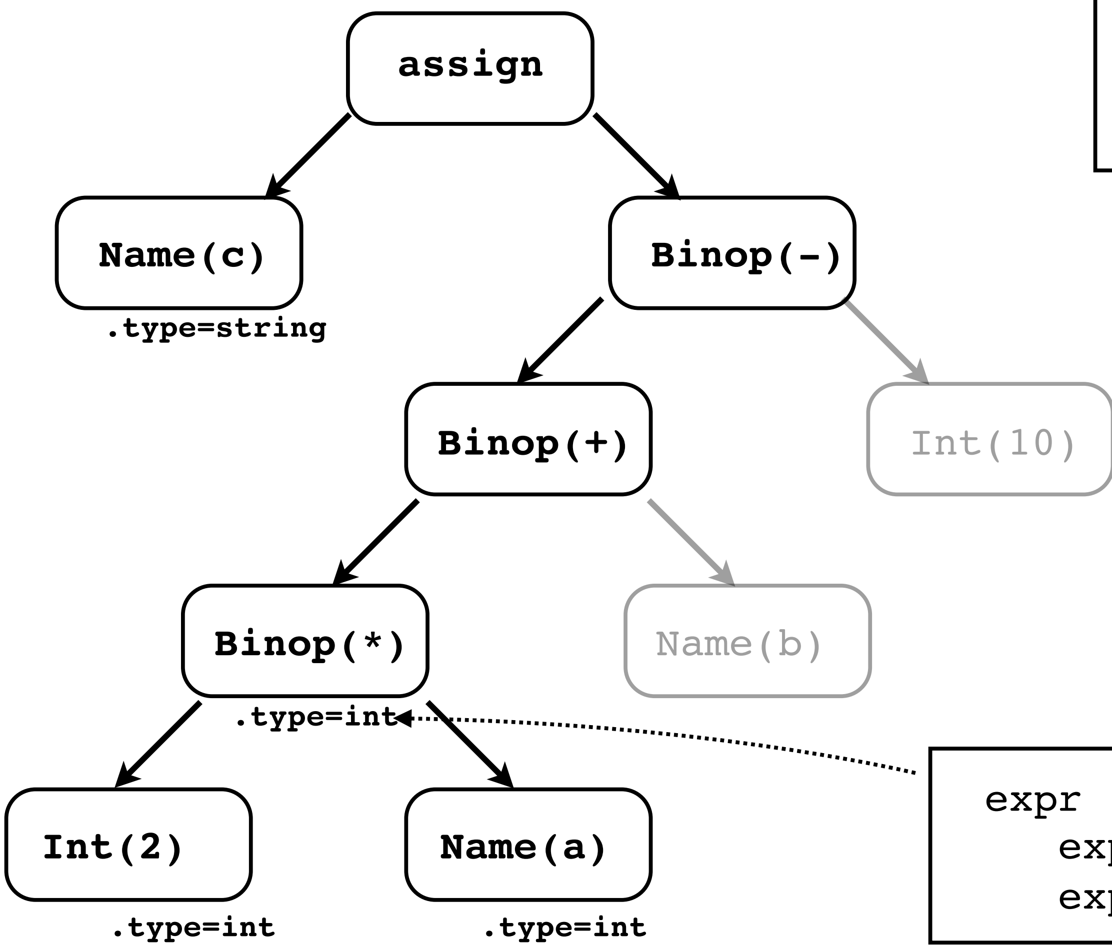
AST Annotation

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



AST Annotation

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



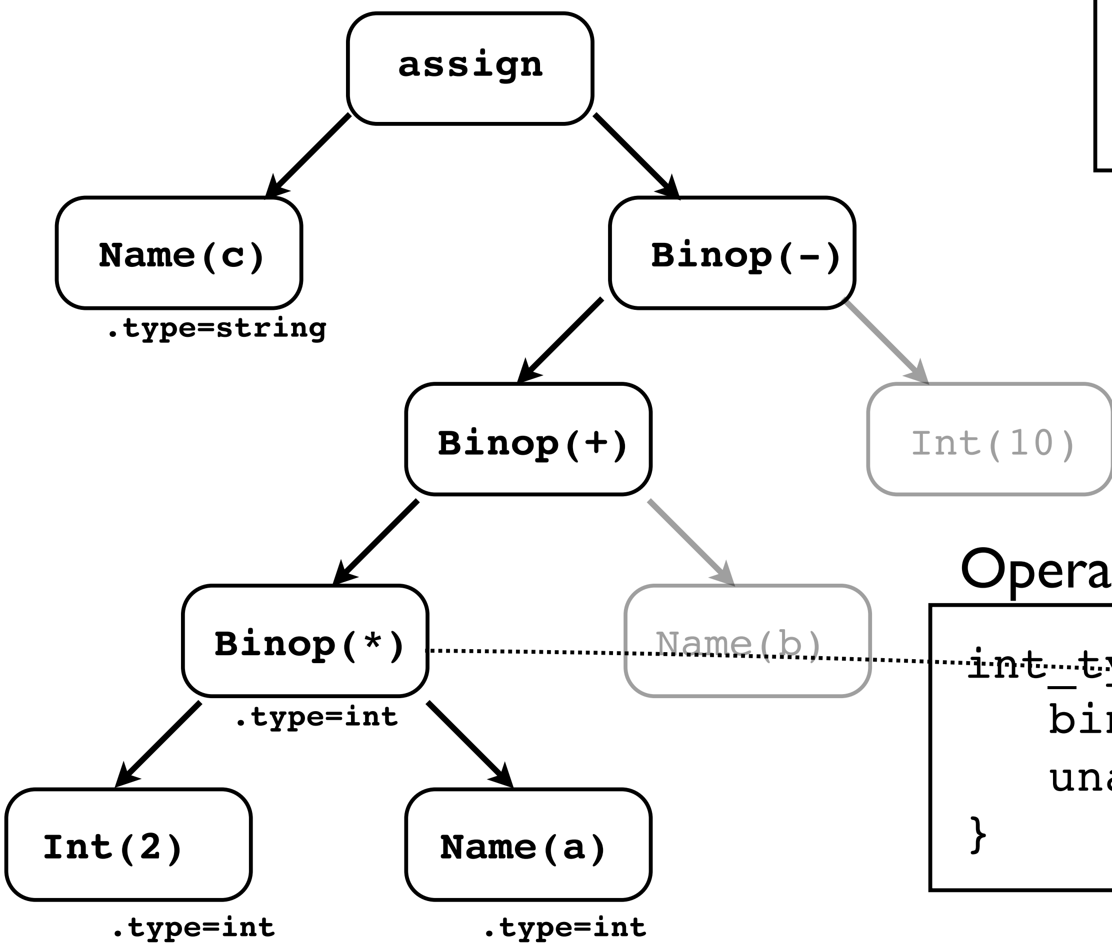
symbol table

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

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

AST Annotation

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



symbol table

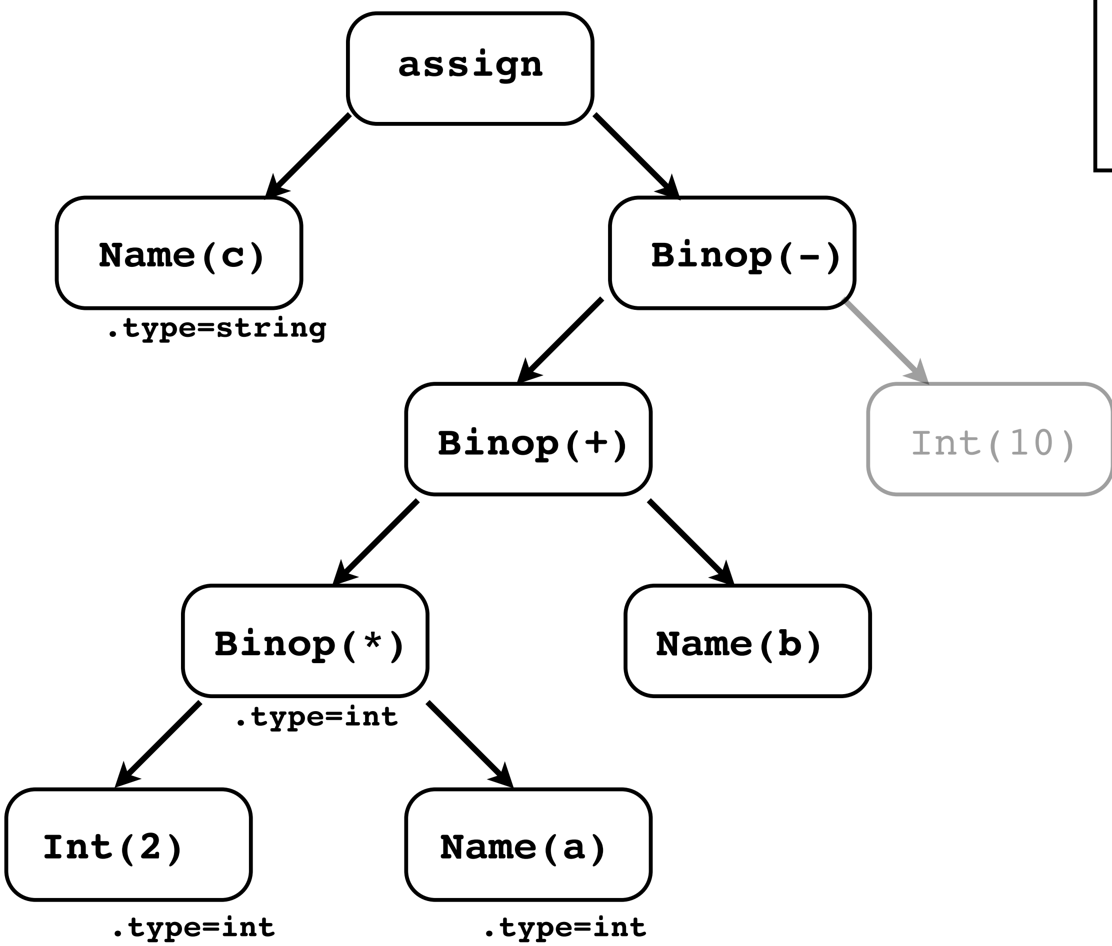
```
'a' : VarDecl(type="int"),
'b' : VarDecl(type="int"),
'c' : VarDecl(type="string")
```

Operator checked against type spec

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

AST Annotation

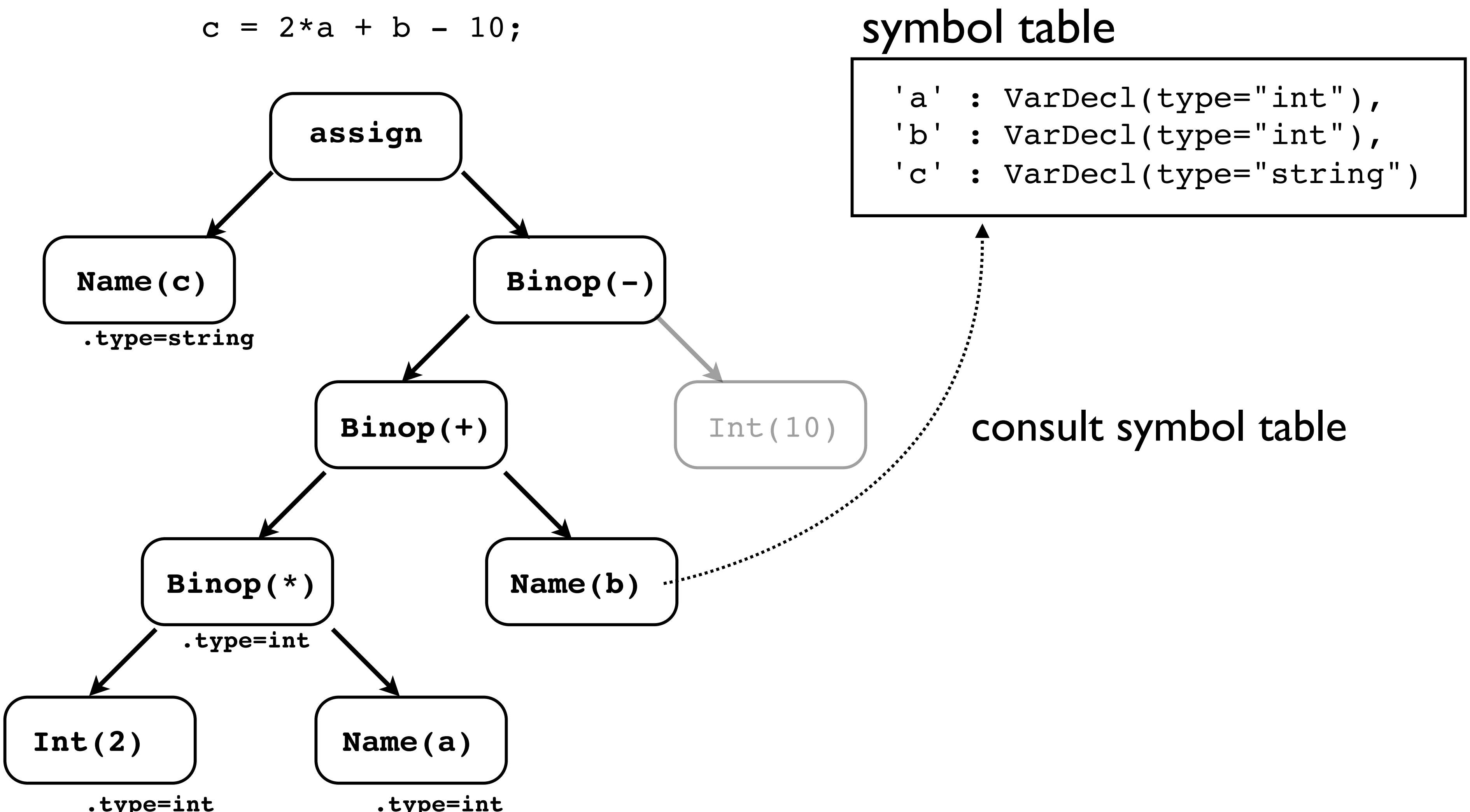
```
c = 2*a + b - 10;
```



symbol table

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

AST Annotation

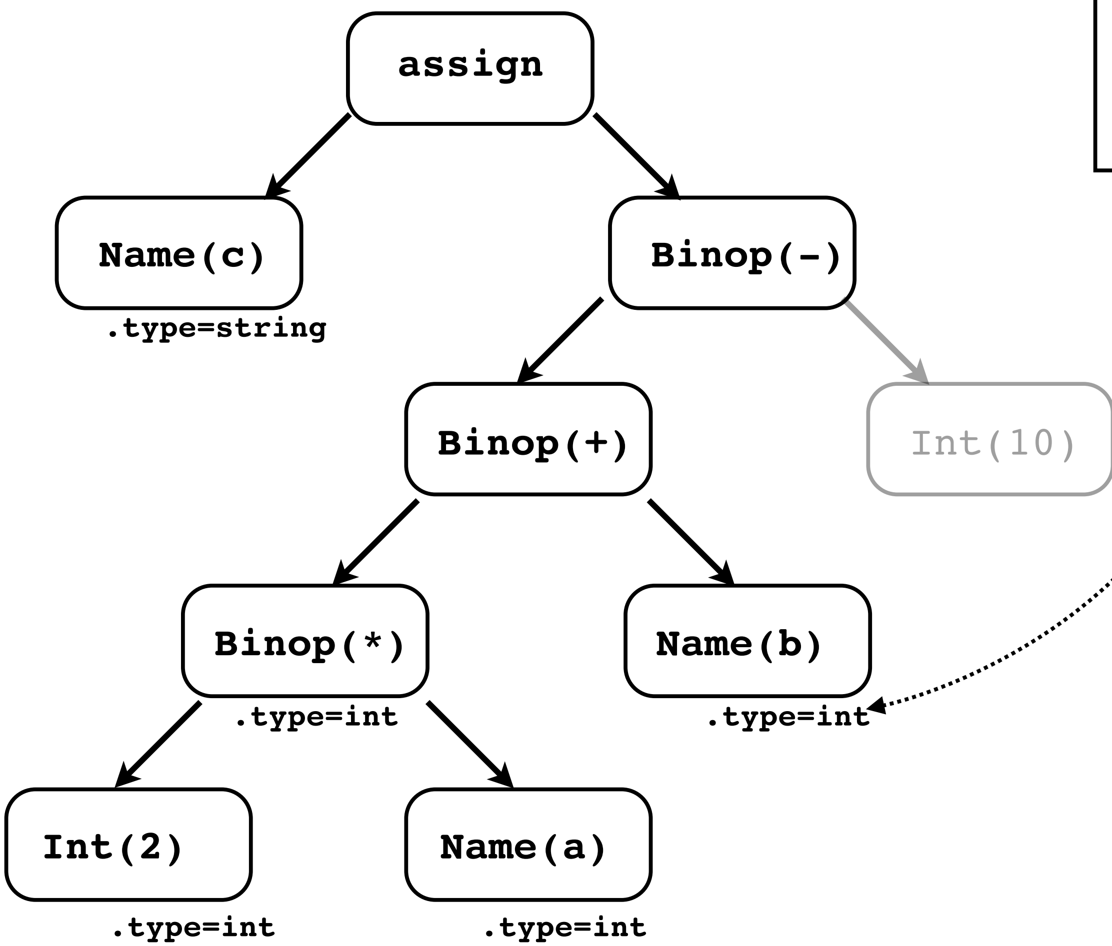


AST Annotation

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

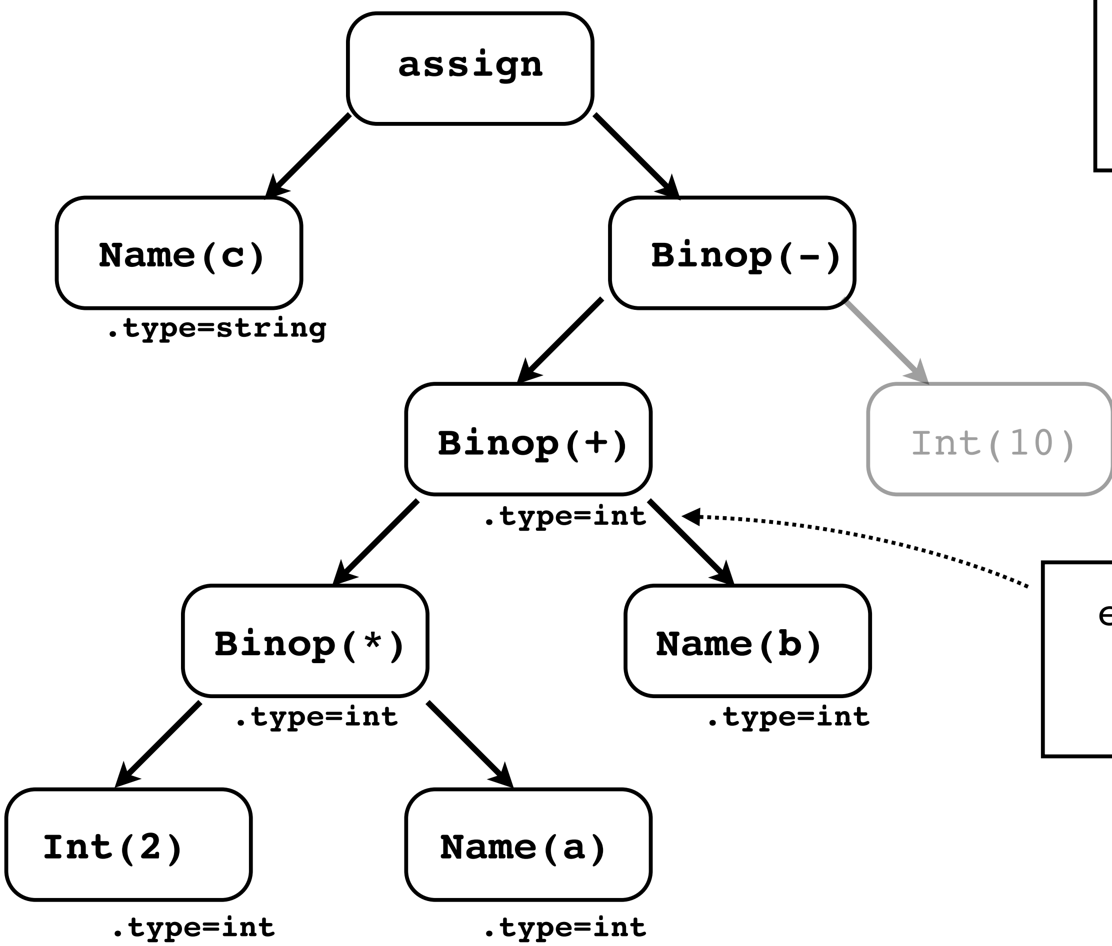
symbol table

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



AST Annotation

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



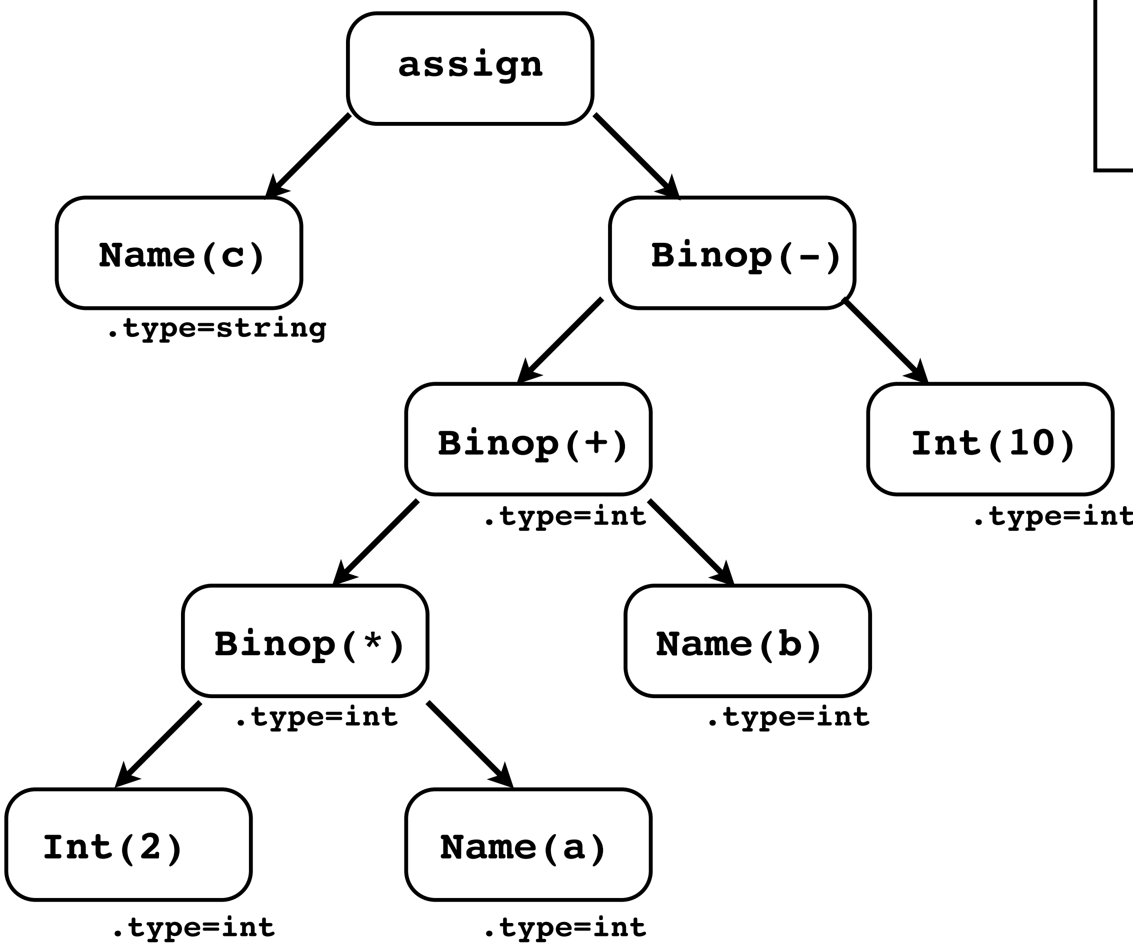
symbol table

```
'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

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

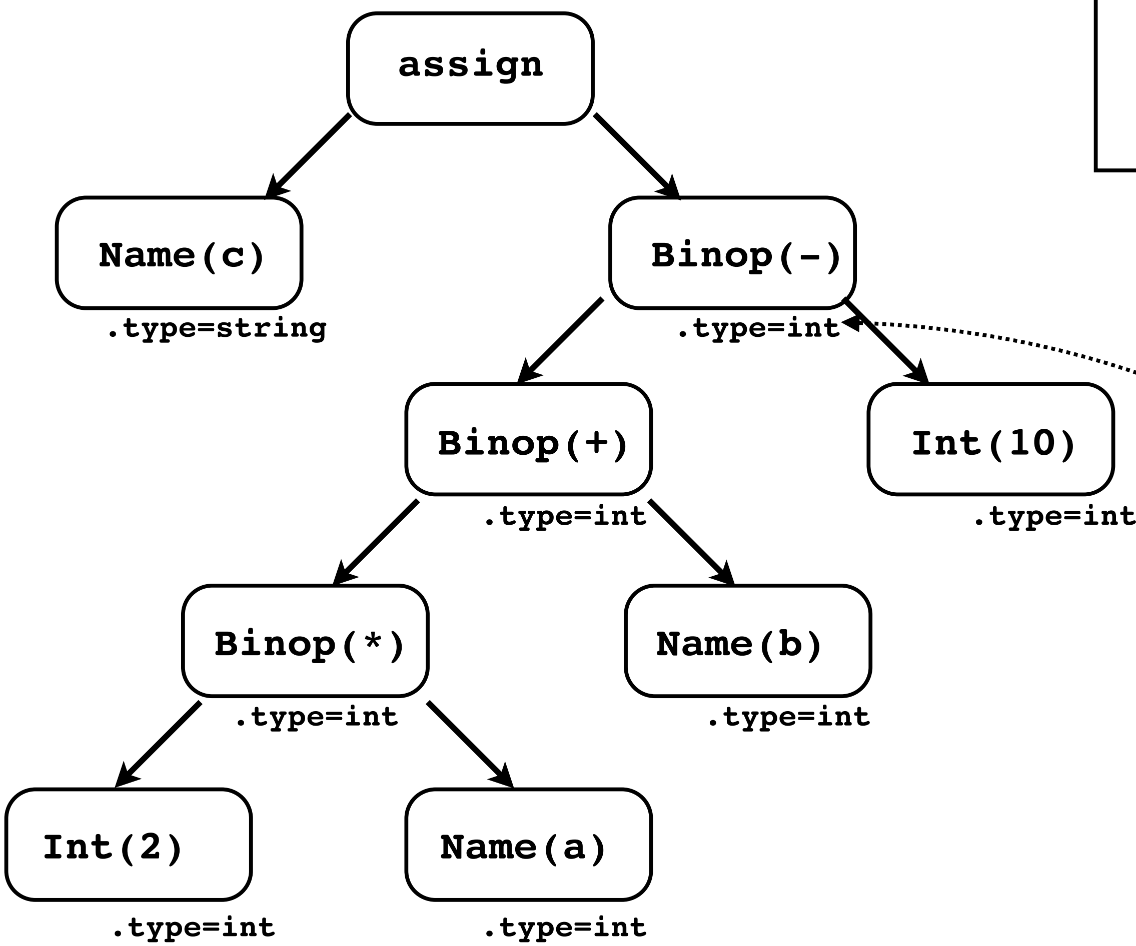


symbol table

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

AST Annotation

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



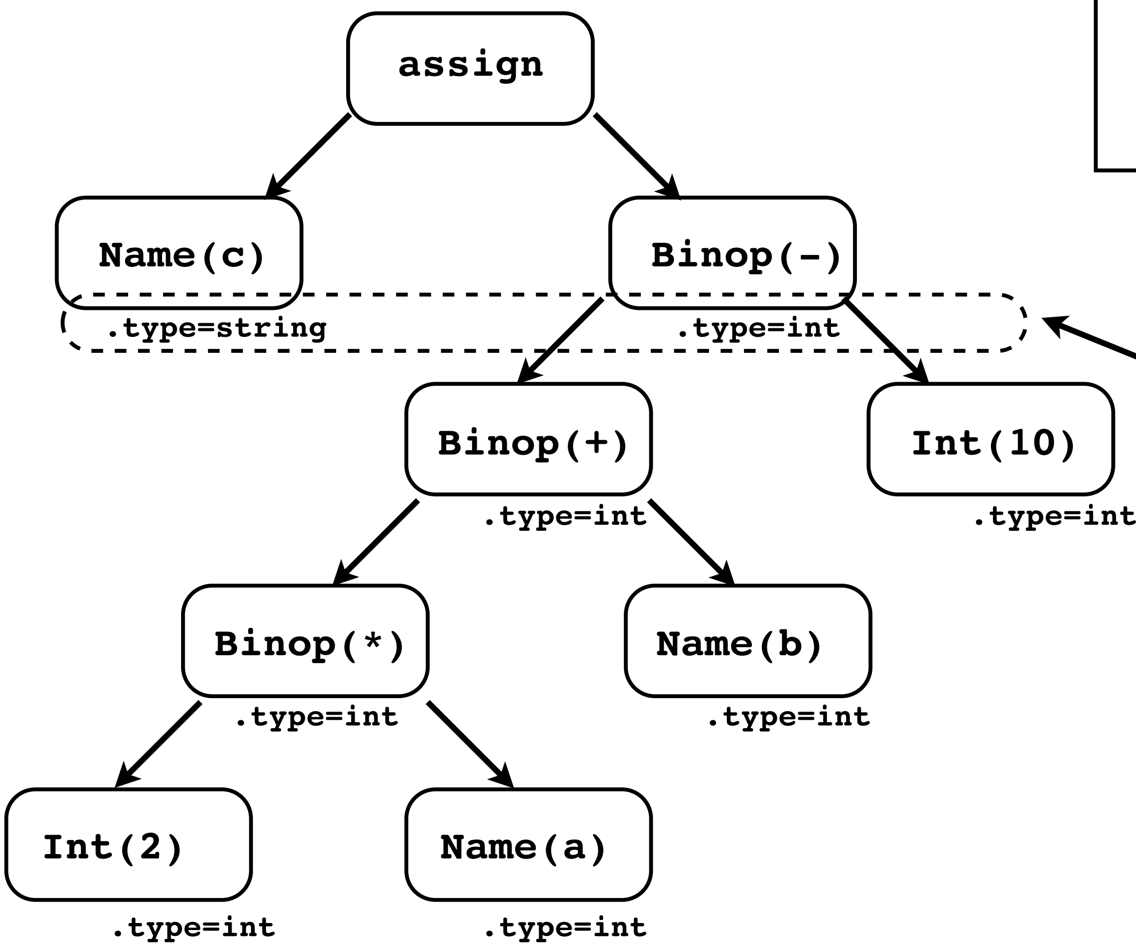
symbol table

```
'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

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



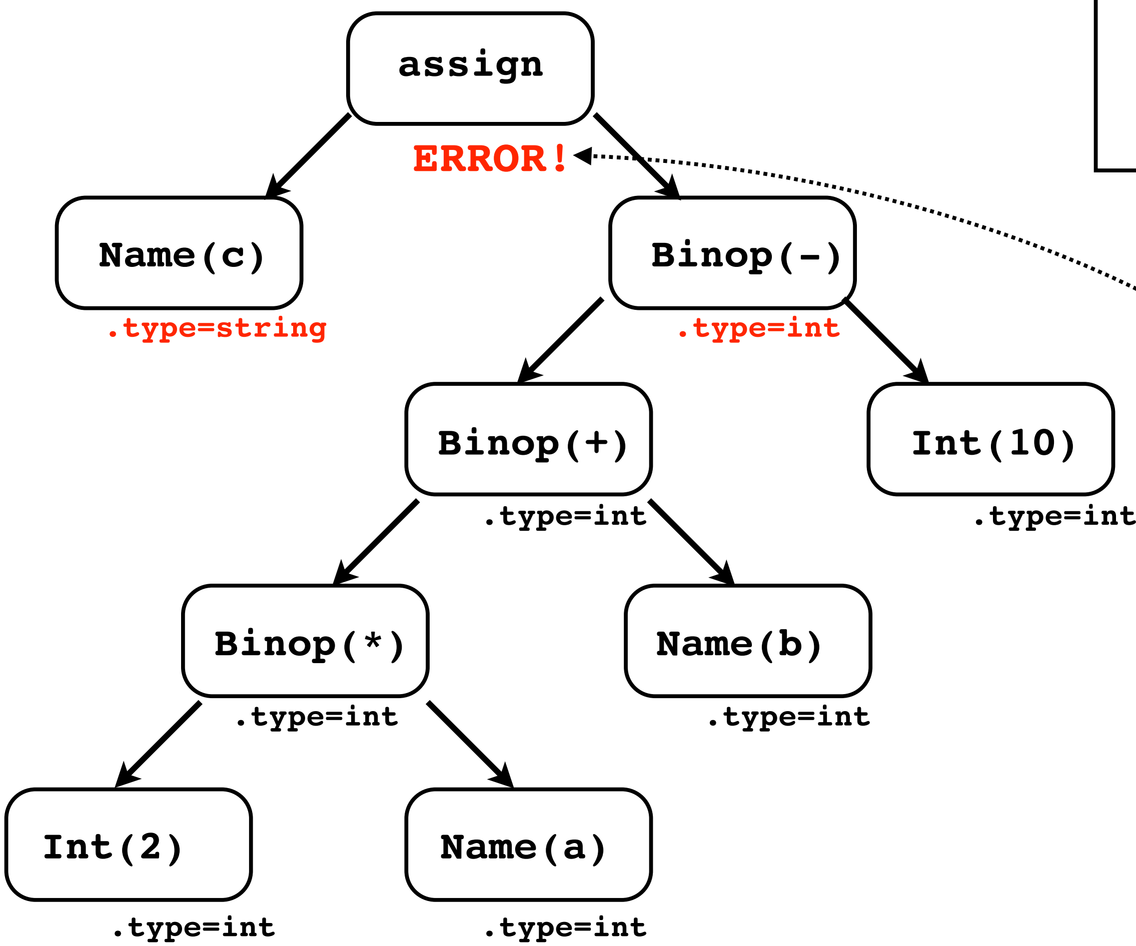
symbol table

```
'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

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



symbol table

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

```
assign ::= name '=' expr ';'
    name.type == expr.type?
```

A type error. LHS and RHS sides of assignment don't have the same type.

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

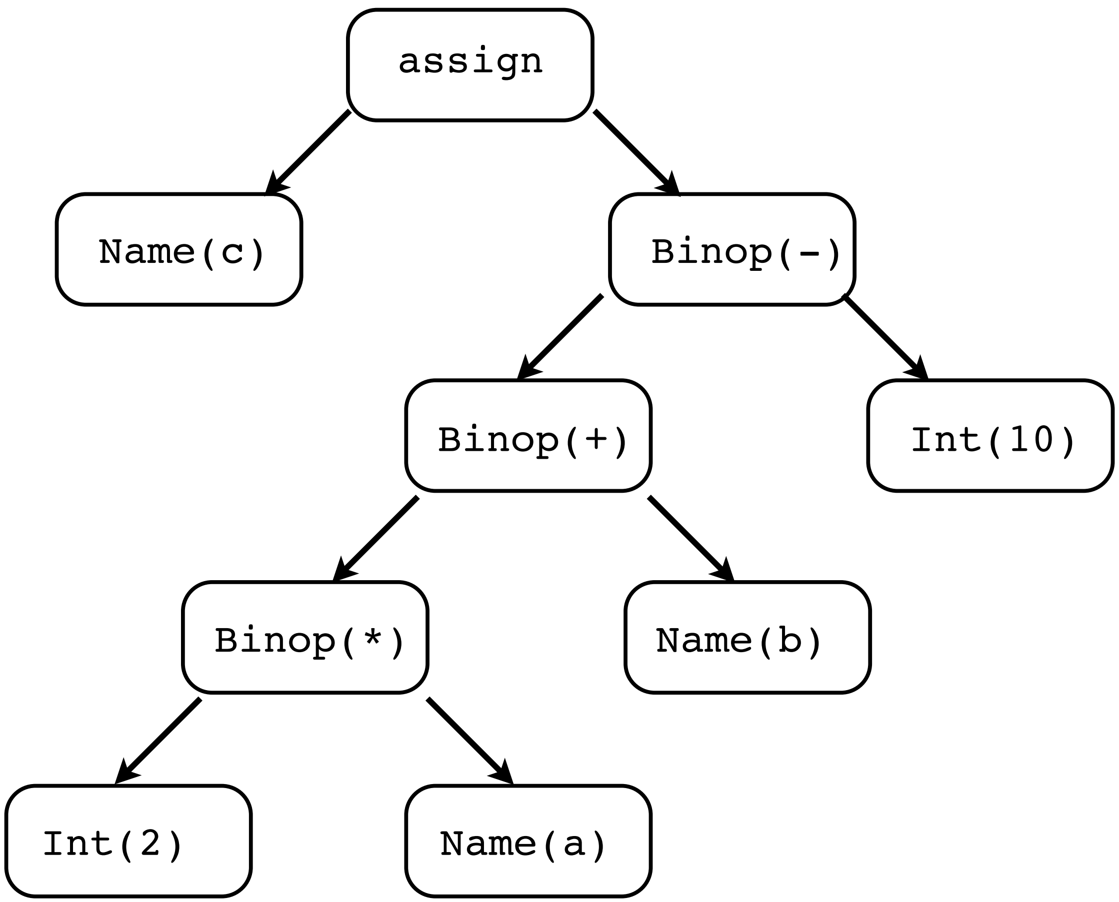
- Example: Compute: 2 + 3 * 4

Instructions	Stack
`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;
```

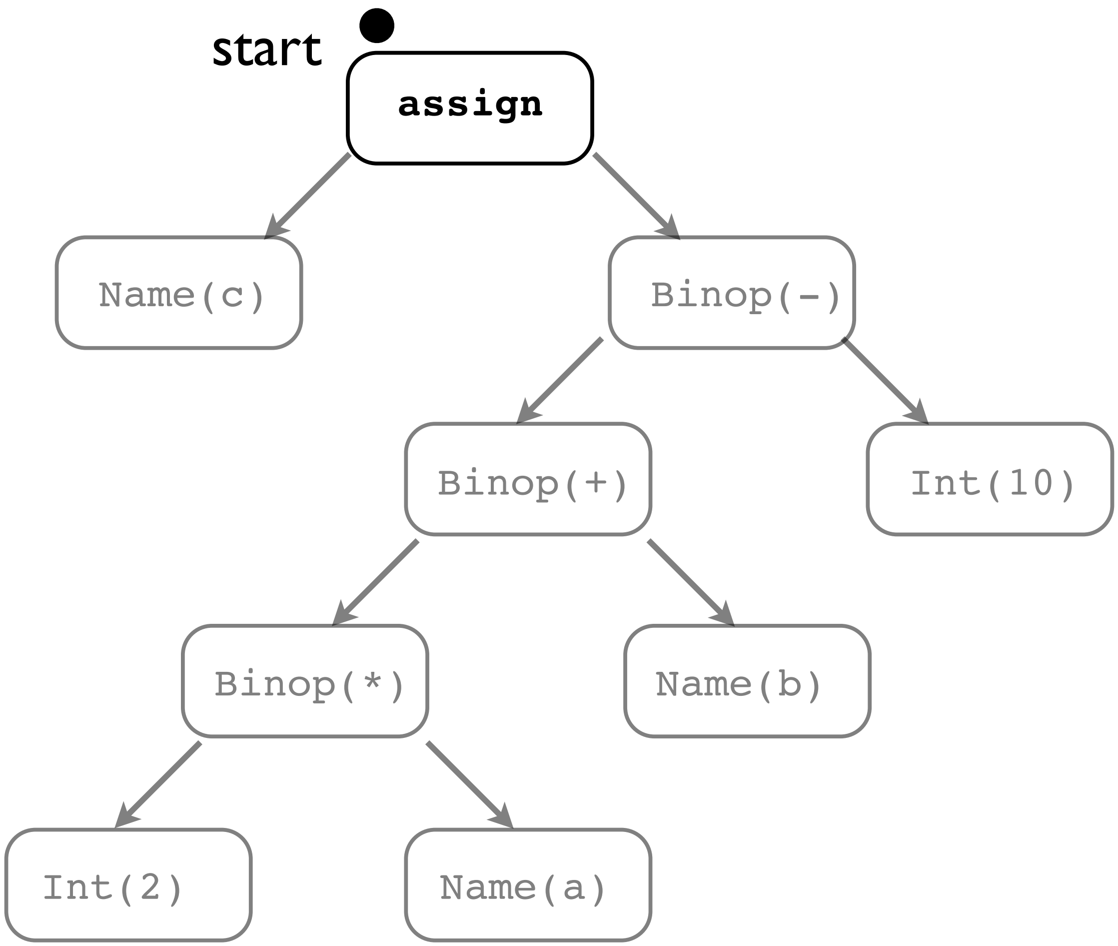


Instructions:

Compute Stack

Code Generation

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

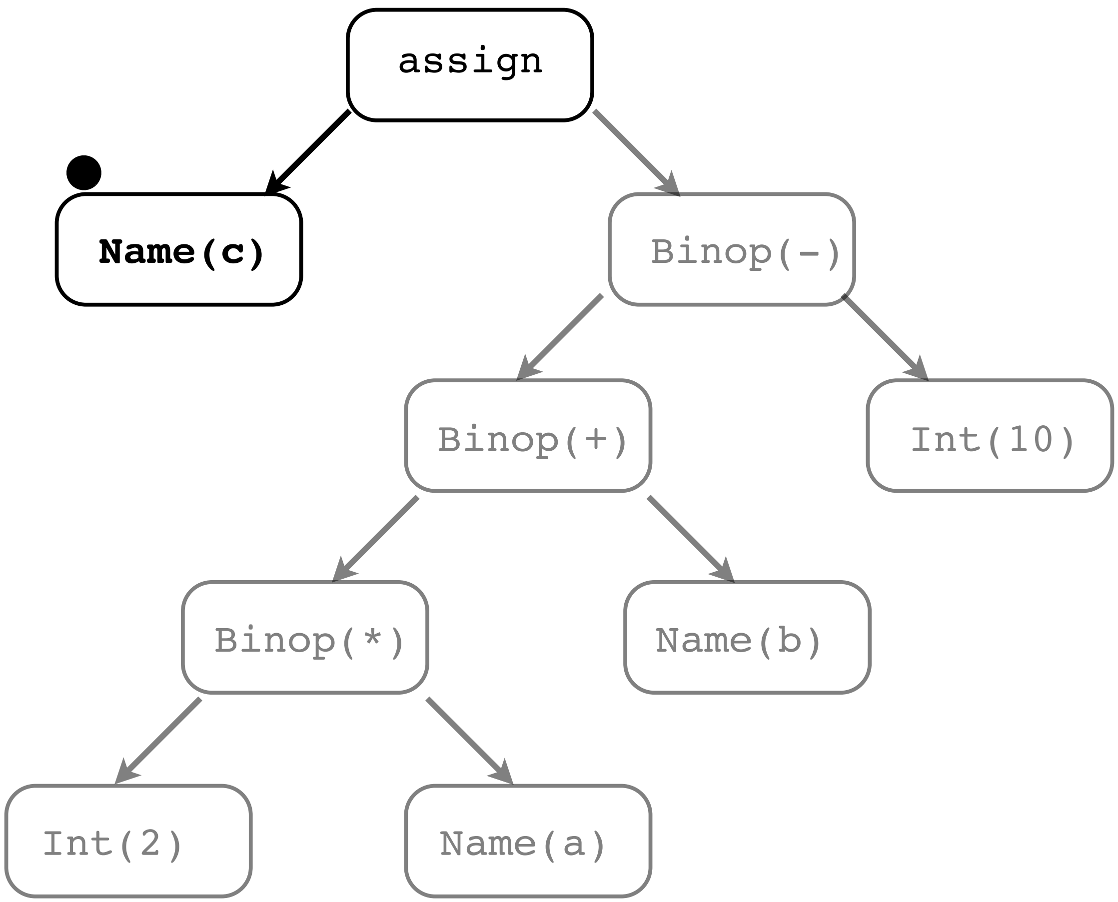


Instructions:

Compute Stack

Code Generation

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



Instructions:

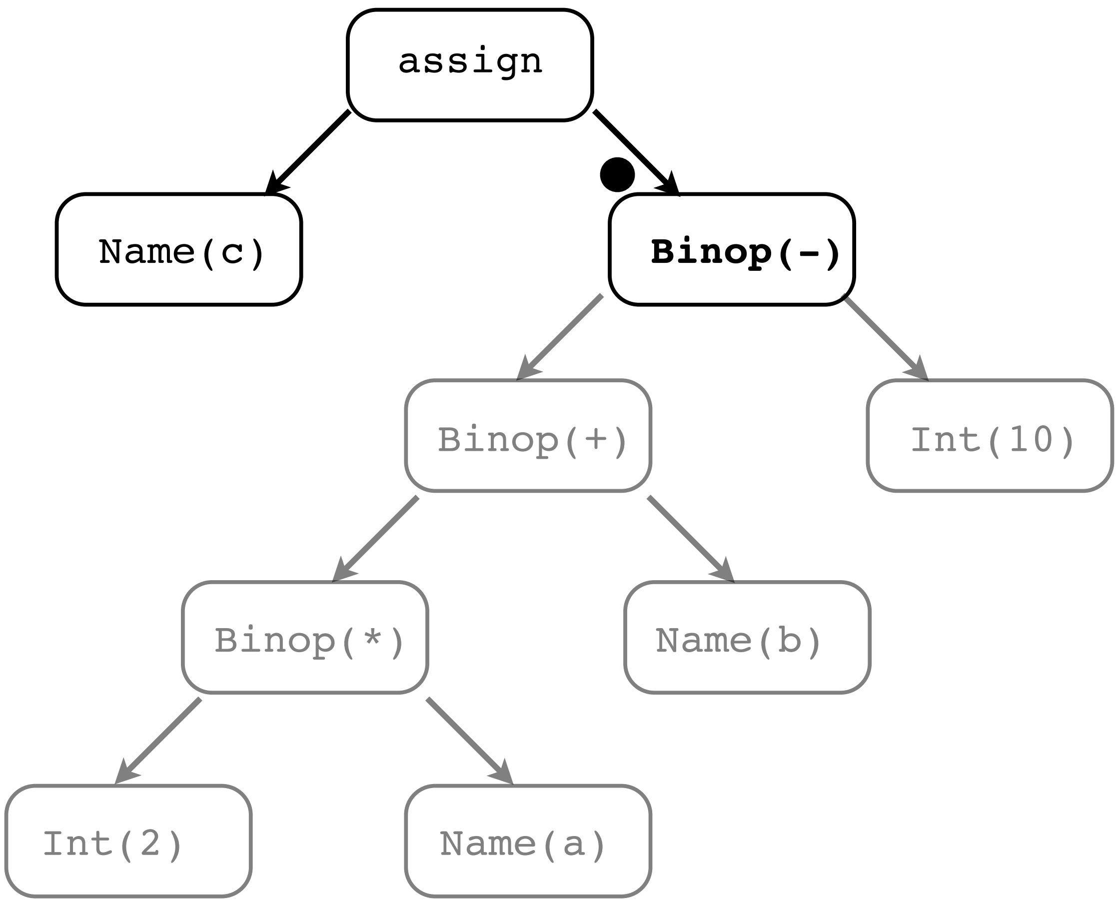
```
1. PUSH LOCATION(c)
```

Compute Stack

```
LOCATION(c)
```

Code Generation

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



Instructions:

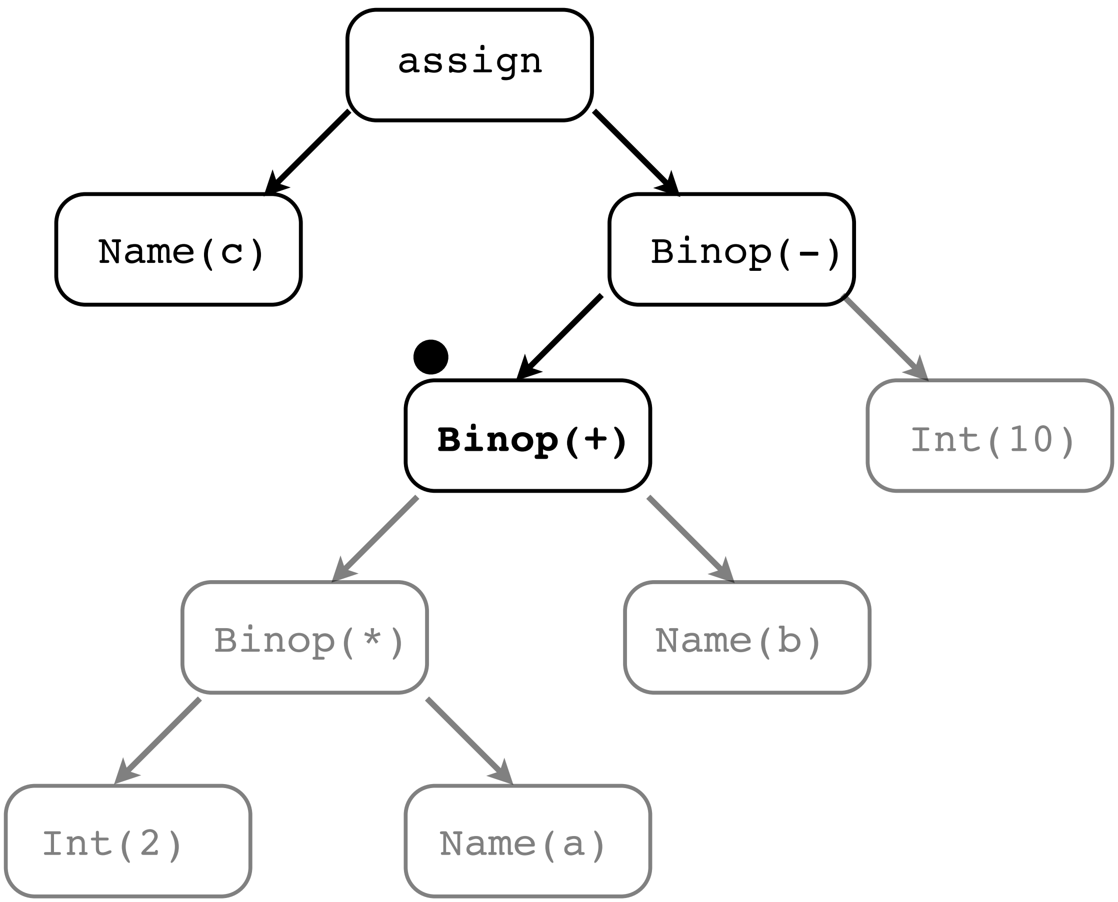
```
1. PUSH LOCATION(c)
```

Compute Stack

```
LOCATION(c)
```

Code Generation

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



Instructions:

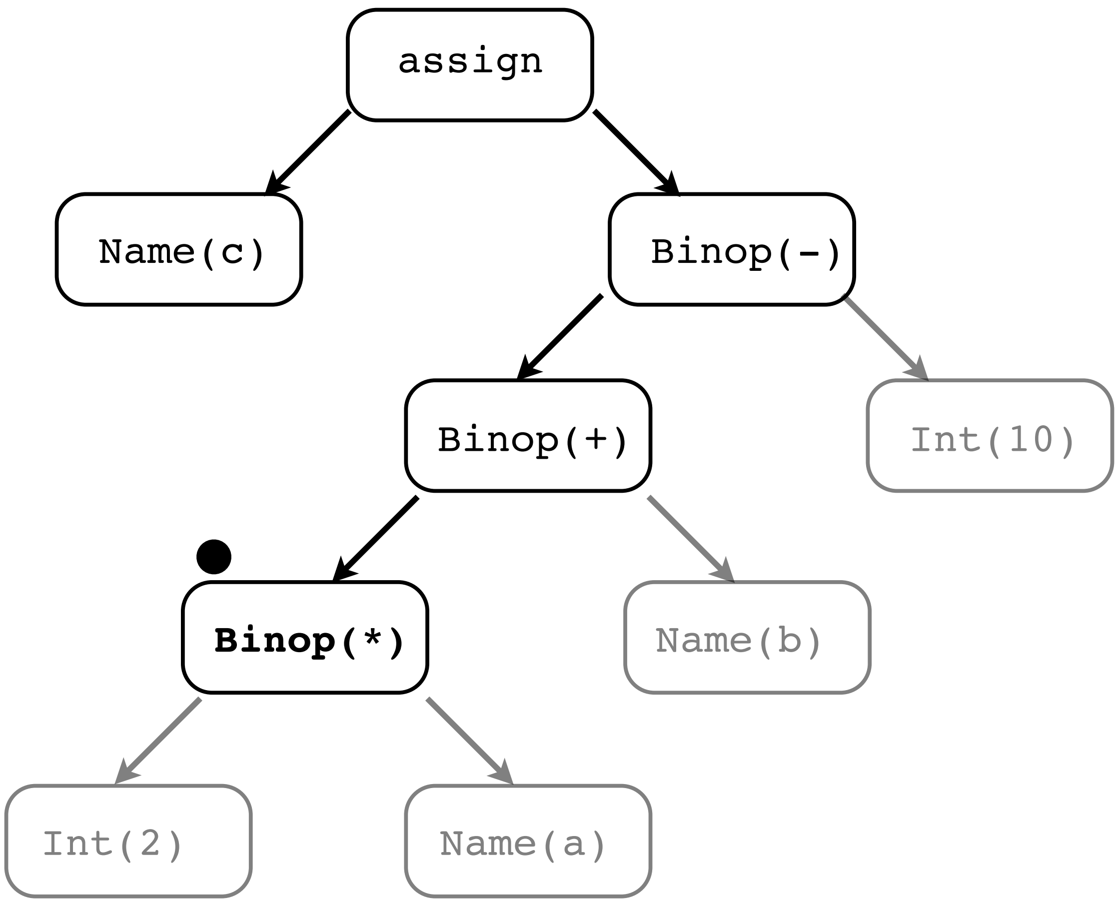
```
1. PUSH LOCATION(c)
```

Compute Stack

```
LOCATION(c)
```

Code Generation

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



Instructions:

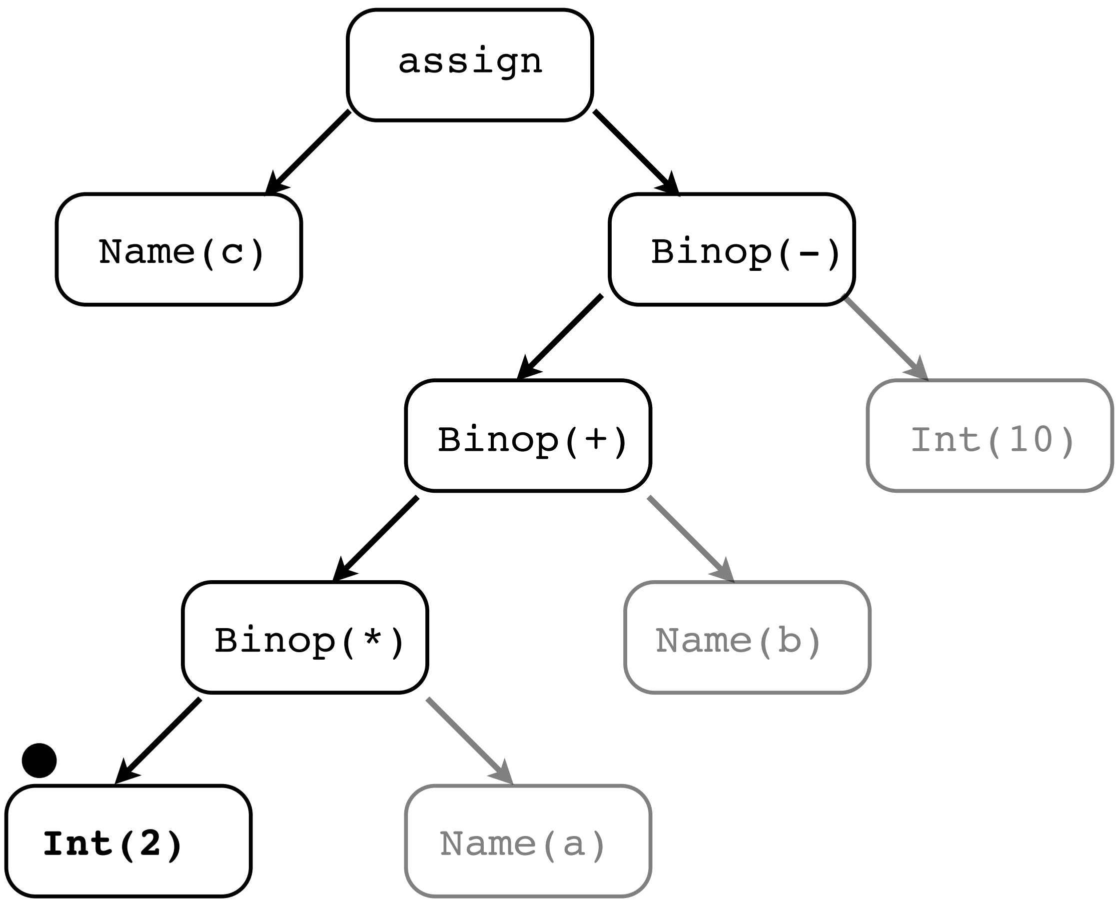
```
1. PUSH LOCATION(c)
```

Compute Stack

```
LOCATION(c)
```

Code Generation

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



Instructions:

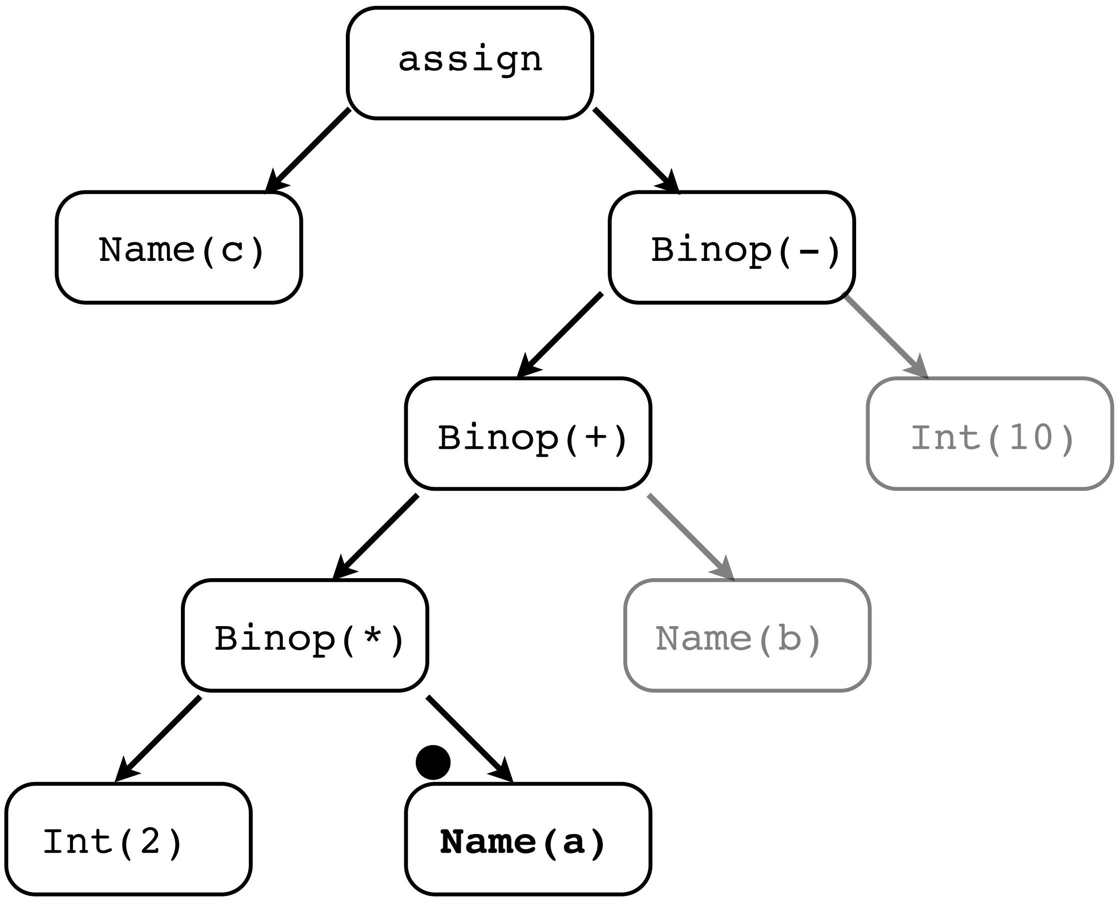
```
1. PUSH LOCATION(c)
2. PUSH 2
```

Compute Stack

```
LOCATION(c)
2
```

Code Generation

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



Instructions:

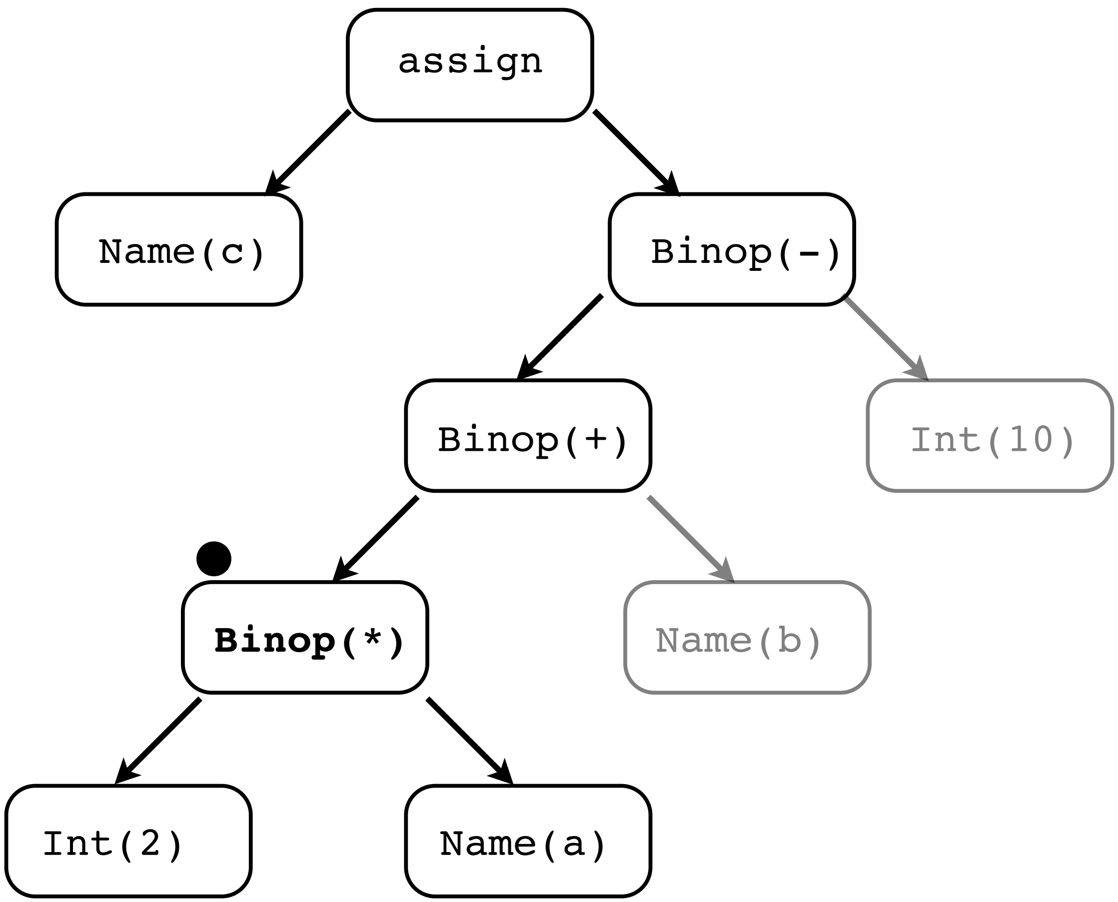
```
1. PUSH LOCATION(c)
2. PUSH 2
3. PUSH LOCATION(a)
4. LOAD
```

Compute Stack

```
LOCATION(c)
2
a
```

Code Generation

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



Instructions:

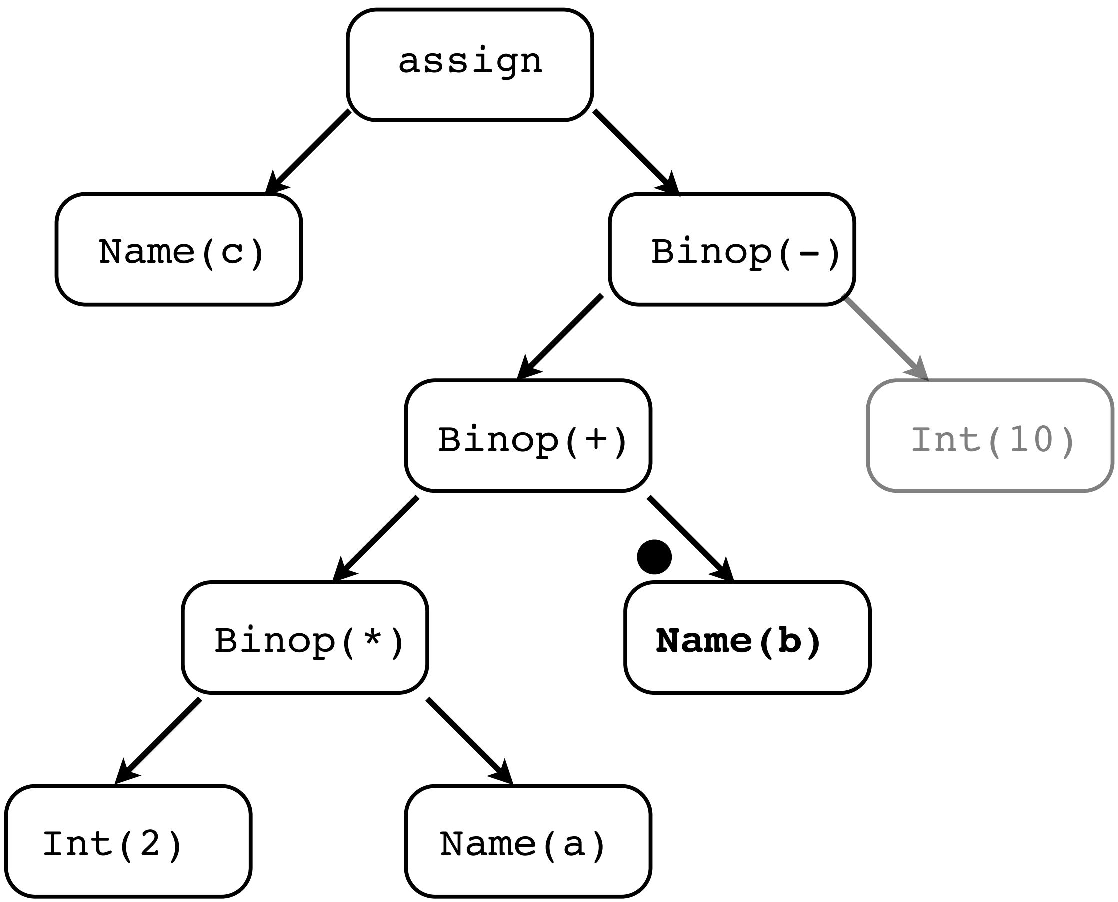
```
1. PUSH LOCATION(c)
2. PUSH 2
3. PUSH LOCATION(a)
4. LOAD
5. MUL
```

Compute Stack

```
LOCATION(c)
2 * a
```

Code Generation

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



Instructions:

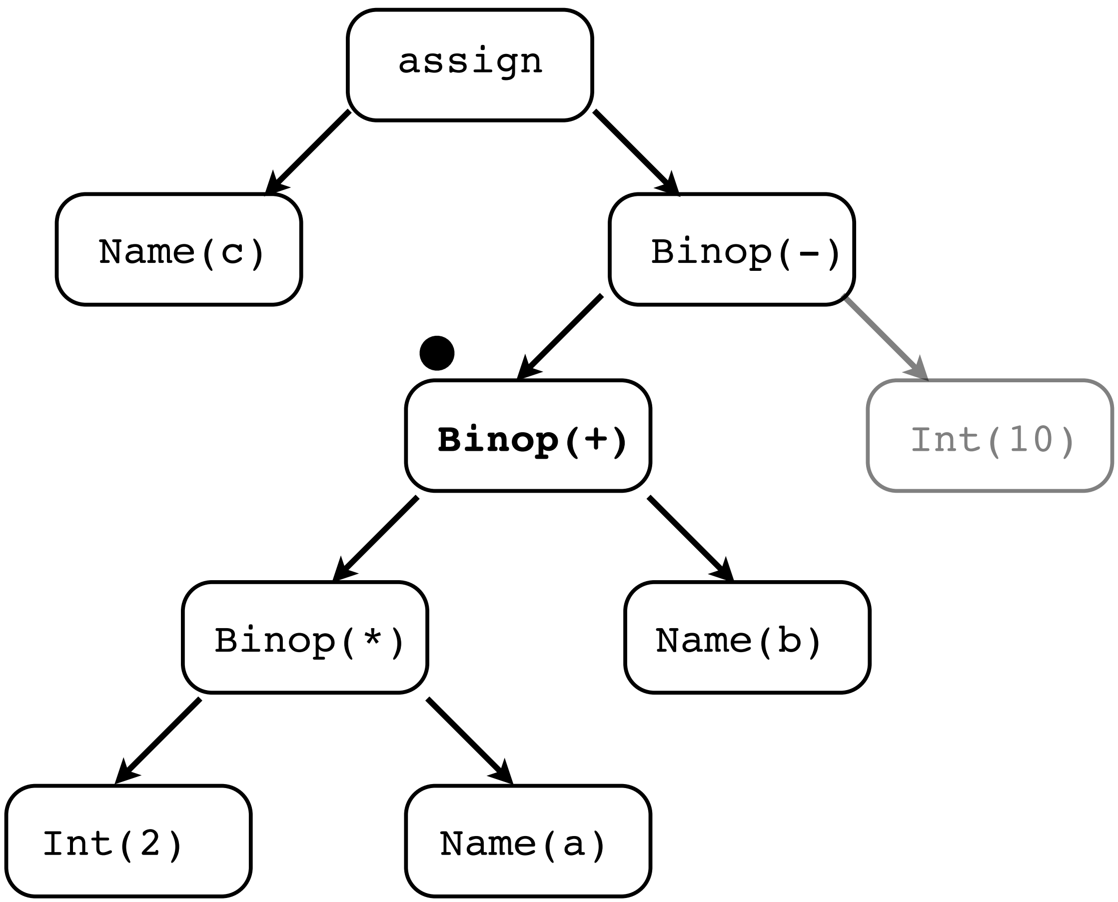
```
1. PUSH LOCATION(c)
2. PUSH 2
3. PUSH LOCATION(a)
4. LOAD
5. MUL
6. PUSH LOCATION(b)
7. LOAD
```

Compute Stack

```
LOCATION(c)
2 * a
b
```

Code Generation

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



Instructions:

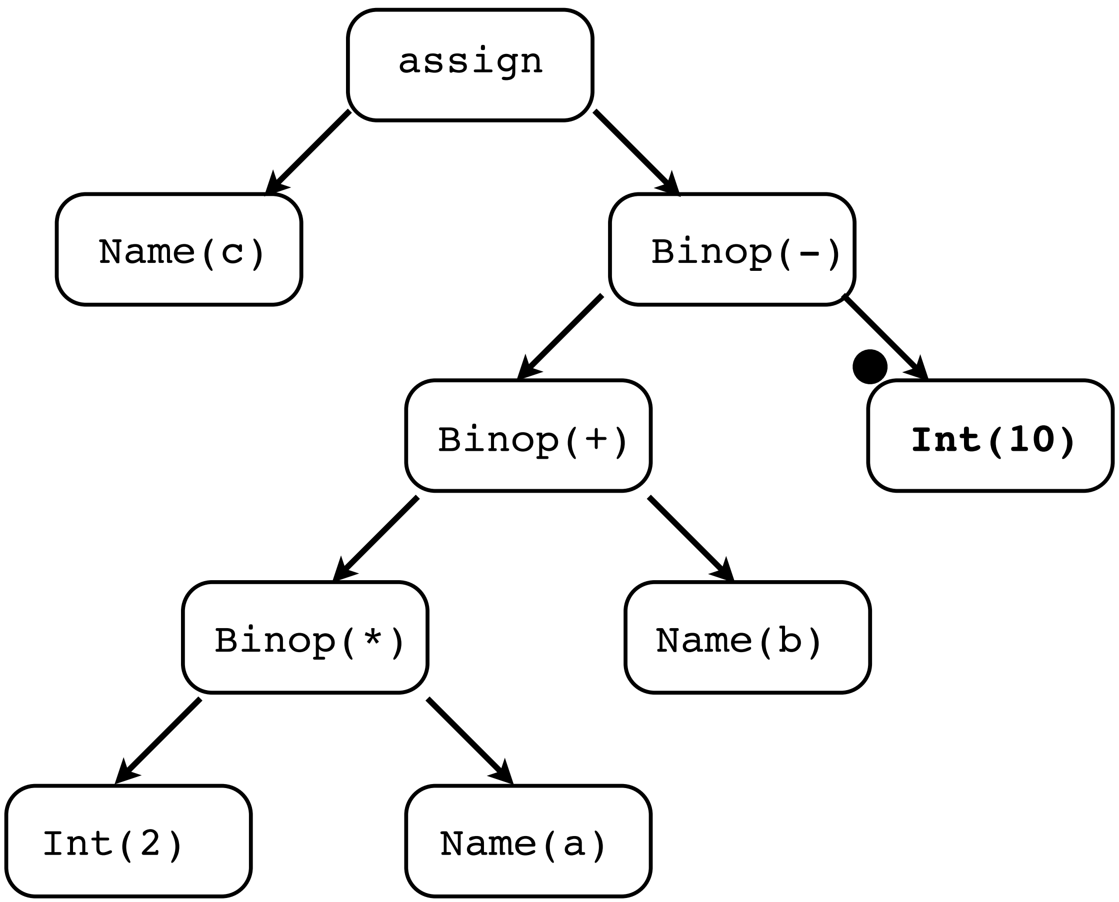
```
1. PUSH LOCATION(c)
2. PUSH 2
3. PUSH LOCATION(a)
4. LOAD
5. MUL
6. PUSH LOCATION(b)
7. LOAD
8. ADD
```

Compute Stack

```
LOCATION(c)
2 * a + b
```

Code Generation

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



Instructions:

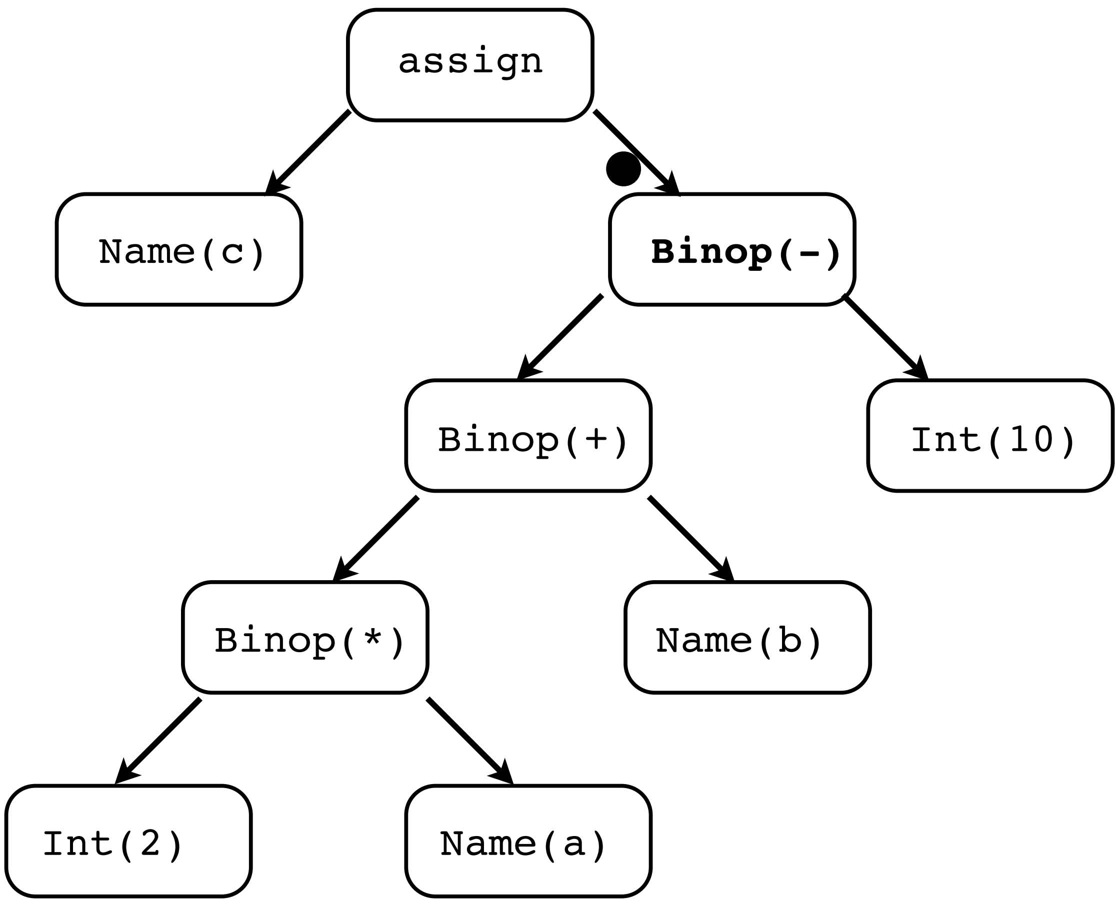
```
1. PUSH LOCATION(c)
2. PUSH 2
3. PUSH LOCATION(a)
4. LOAD
5. MUL
6. PUSH LOCATION(b)
7. LOAD
8. ADD
9. PUSH 10
```

Compute Stack

```
LOCATION(c)
2 * a + b
10
```

Code Generation

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



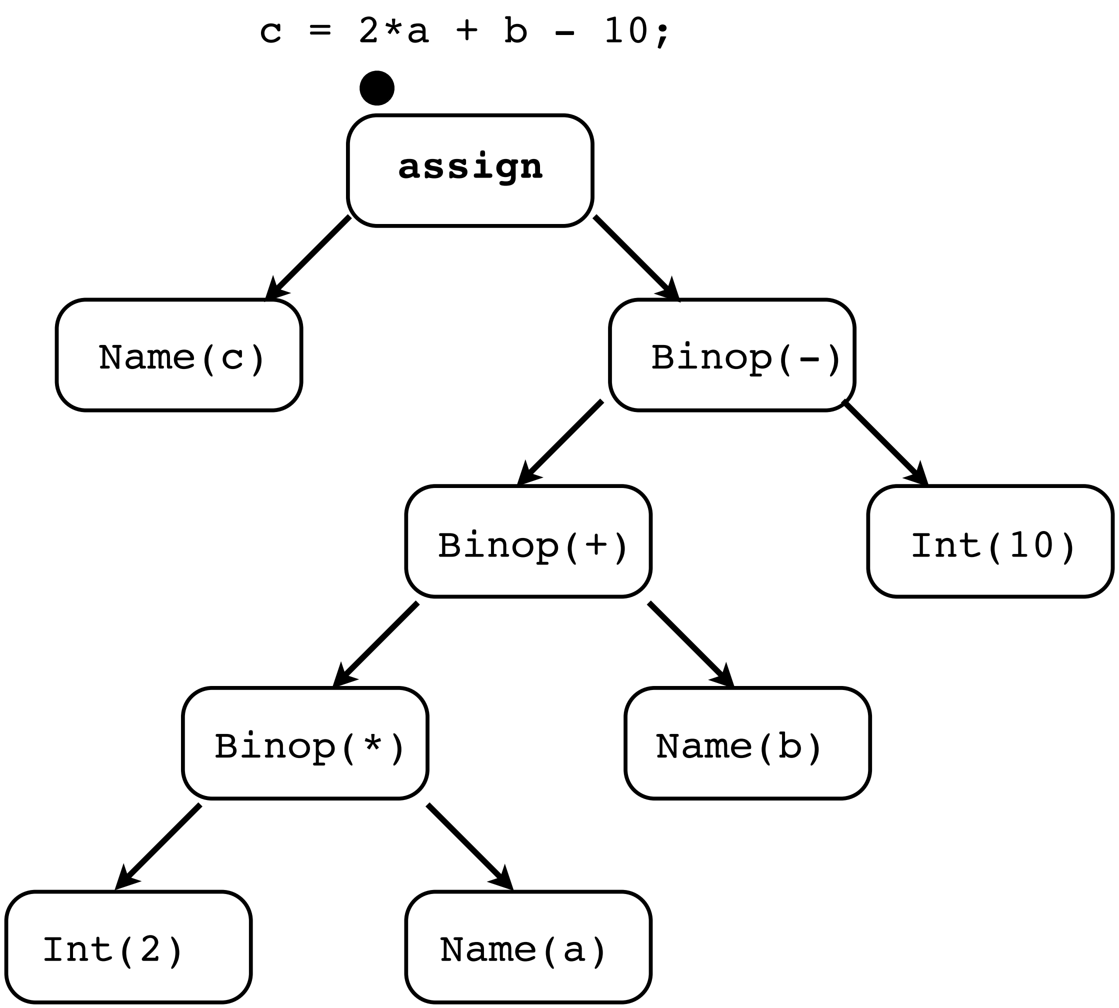
Instructions:

```
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

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
- Simple 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
```

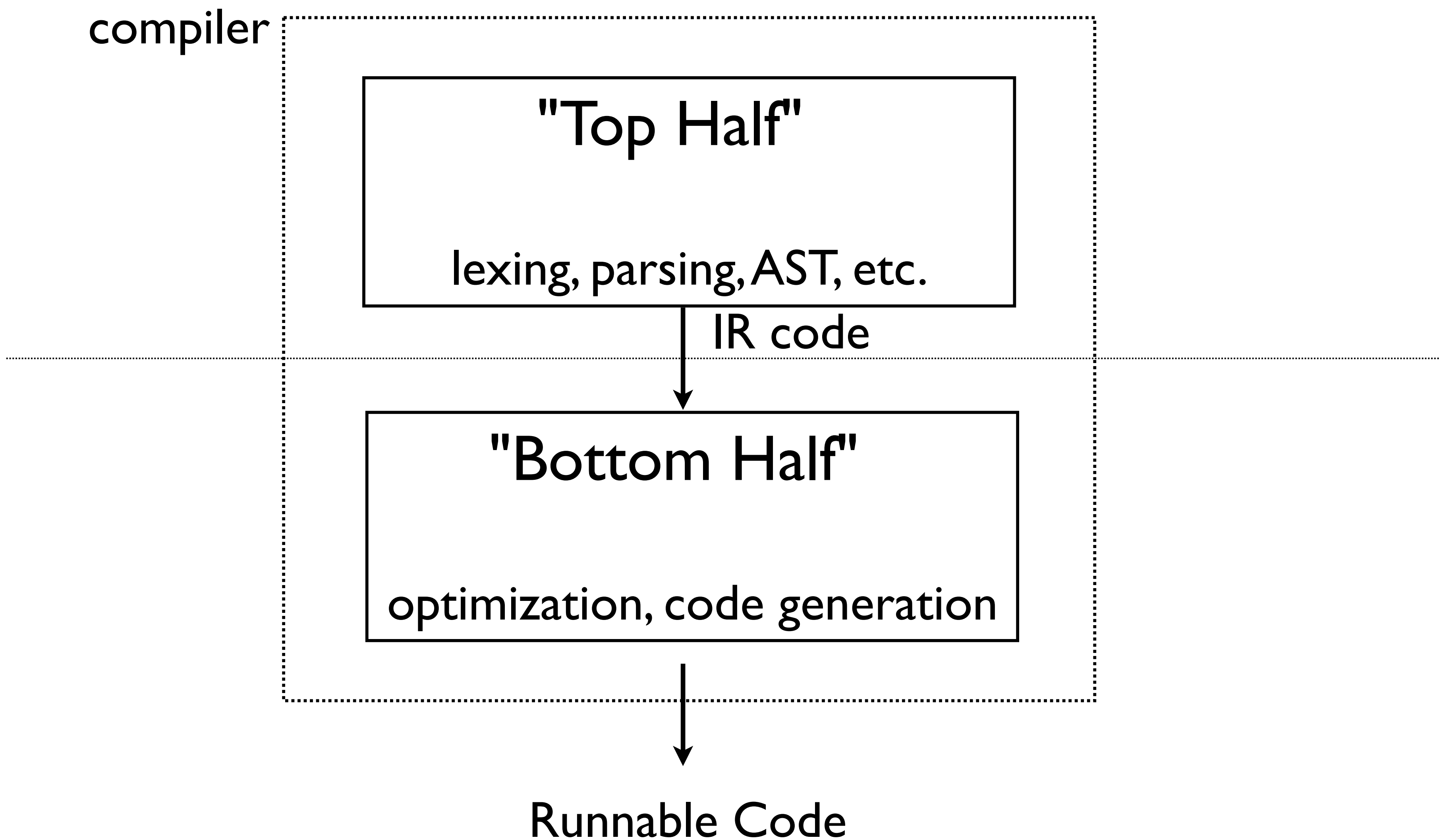
→

```
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



SSA Code

- Single Static Assignment
- A variant of 3-address code
    - Can never assign variables more than once
    - Assignments always go to new vars

Exercise and Project 4-5

Part 5

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

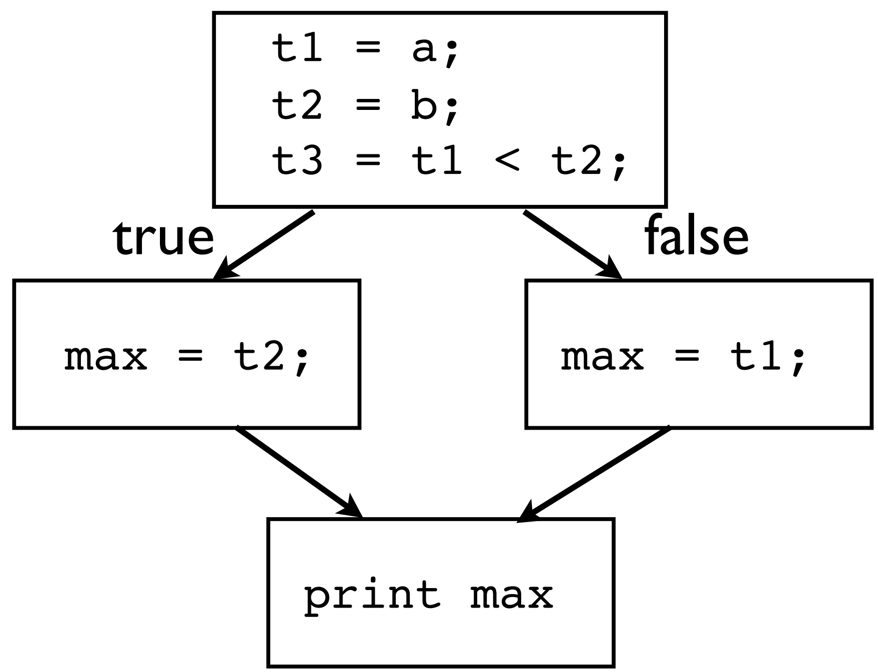
```
var a int = 2;
var b int = 3;
var c int = a + b;
print(2*c);
...
```

- 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

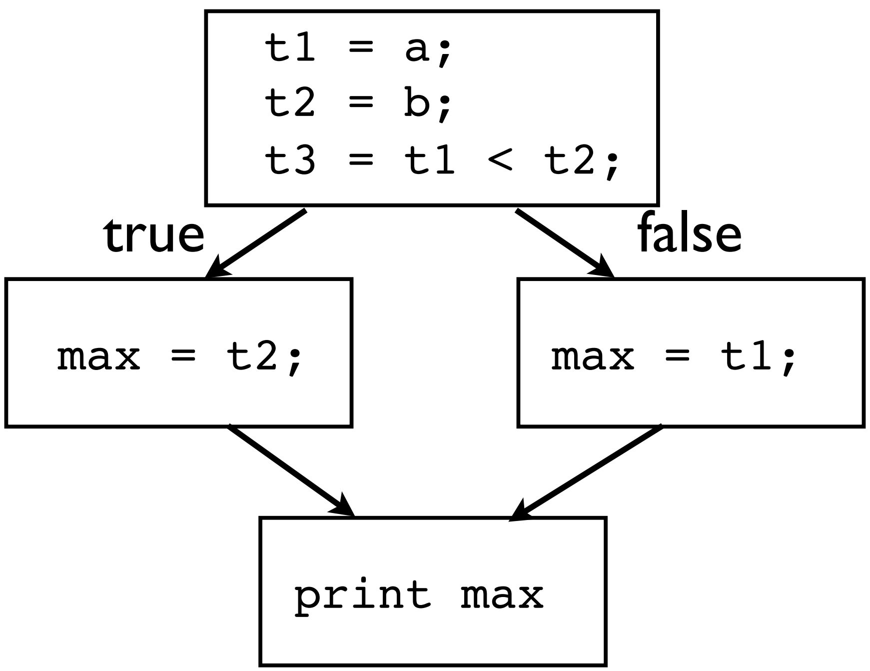
```
var a int = 2;
var b int = 3;
var max int;

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
}
```

<u>current block</u>

...

Implementation Example

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

<u>current block</u>

```
...
('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')

('block', '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')

('block', 'b2')
statements1
('jump', 'b4')

('block', 'b3')
statements2
('jump', 'b4')
```

Implementation Example

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

Create merge block

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

<u>current block</u>

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

('block', 'b2')
statements1
('jump', 'b4')

('block', 'b3')
statements2
('jump', 'b4')

('block', 'b4')
...
```

Complexity: Return

- Handling of the return statement

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

- Notice: no return statement if False
- Compiler would need to check for it

Complexity: Break

- break

```
func spam(x int) int {
    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

Exercise and Project 7

Part 6

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)

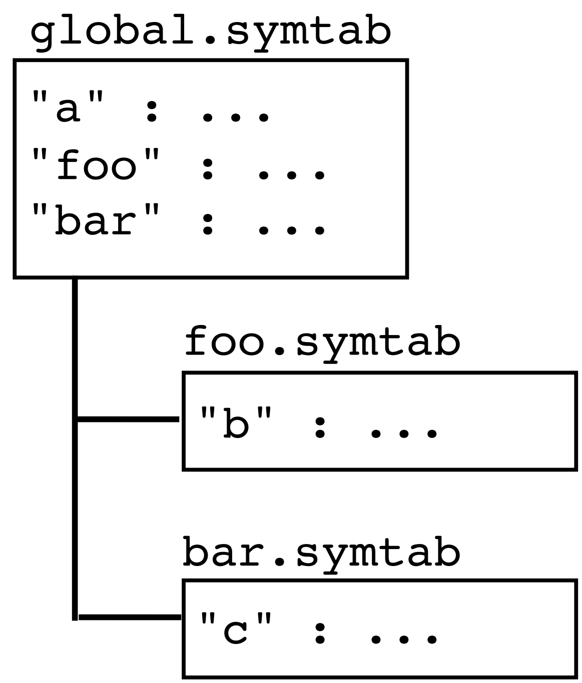
```
int a = 1;              / Global
int foo() {
    int n = 0;          / Local. Visible in entire func
    while (n < 10) {
        int x = 2;      / Block. Visible only in 'while'
        ...
    }
    printf("%d\n",x); / Error. x not defined
}
```

- 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)
```

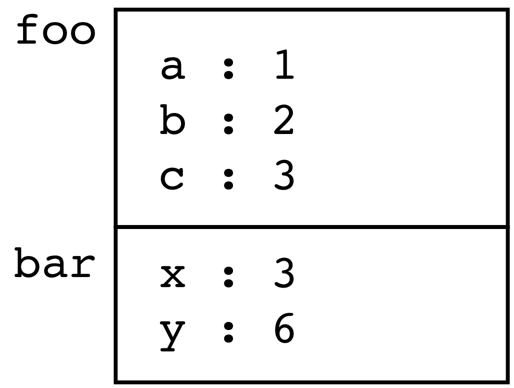
Activation Frames

```
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
```

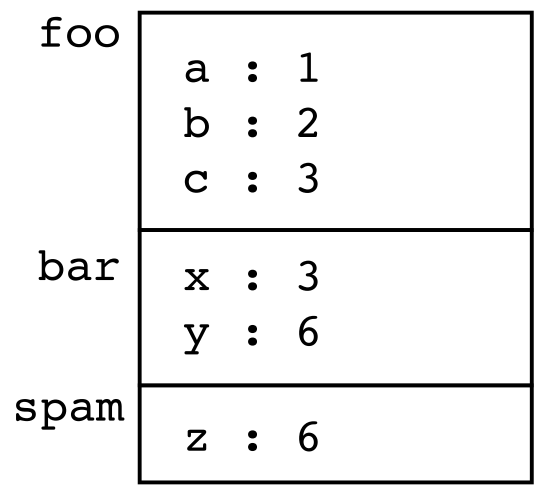
Activation Frames

```
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_FuncDecla
  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_IfStateme
  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 obtain 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):
    z = x + y
    return z
```

(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
```

creates

```
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

```
x : 1
y : 2
return : None
```

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

<u>Semantic Issue: What does the frame contain?</u>

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
```

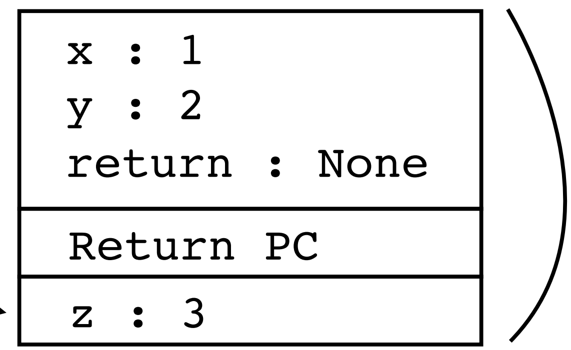
```
x : 1
y : 2
return : None
-------------
Return PC
```

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
```



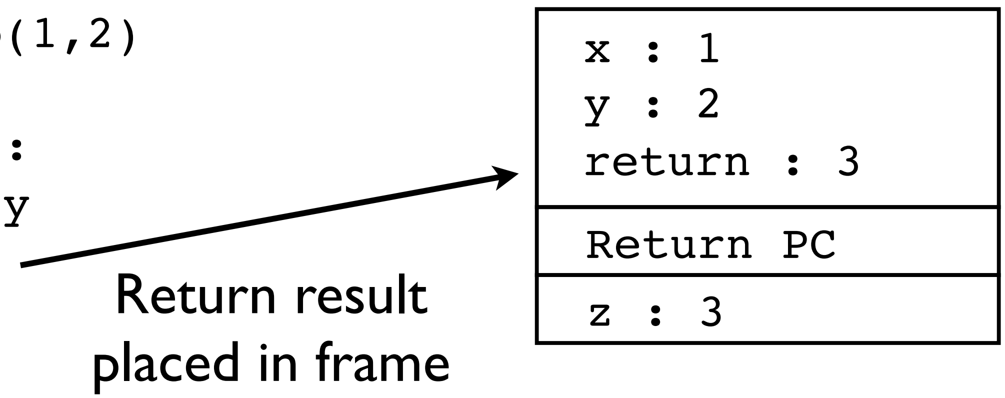
complete frame

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
```



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

```
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)

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

caller destroys remaining frame on assignment of result

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
```

0	x : 1
1	y : 2
2	return : None
3	Return PC
4	z : 3

complete frame

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

Frame Example

- Python Disassembly

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

>>> import dis
>>> dis.dis(foo)
  2           0 LOAD_FAST                0 (x)
              3 LOAD_FAST                1 (y)
              6 BINARY_ADD
              7 STORE_FAST               2 (z)

  3          10 LOAD_FAST                2 (z)
             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)
```

foo

```
a : 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)
```

```
bar  | a : 0 |
```

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

- Note: Python does <u>not</u> 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