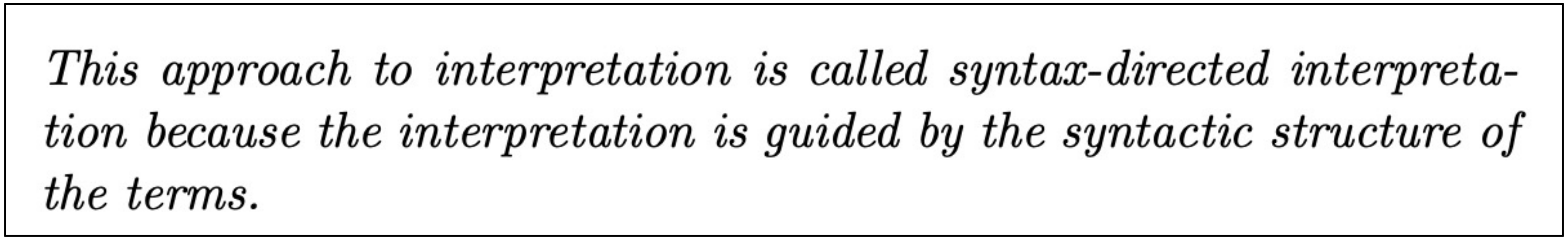
Syntax-Directed Interpretation

- We now have all the tools to build our first interpreter.
- We will extend Exp0 to Exp1 by allowing multi-symbol words
- We will build an interpreter for Exp1 using a technique called syntax-directed interpretation.



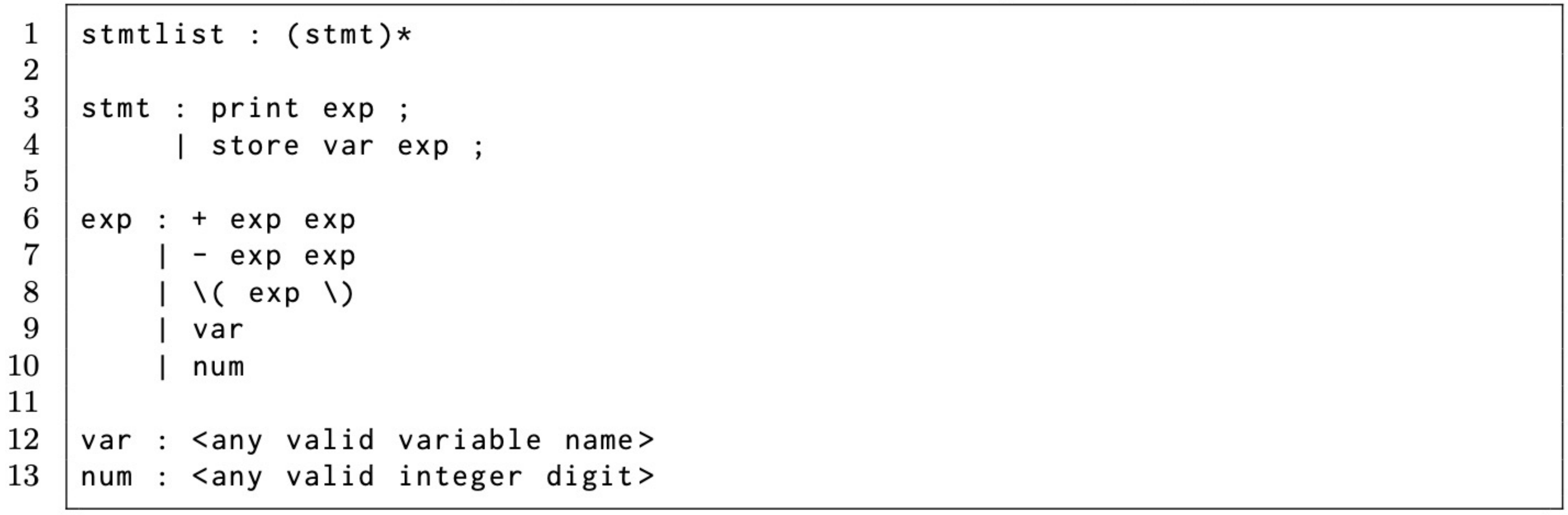
This approach to interpretation is called syntax-directed interpretation because the interpretation is guided by the syntactic structure of the terms.

Reading

- Chap 3

The Exp1 Language

- We extend the Exp0 language to create Exp1:
  - keywords that are longer than a single character
  - Variable names that conform to the normal variable names found in other programming languages: a single alpha character followed by zero or more alpha-numerical characters
  - Numbers that consist of more than one digit.

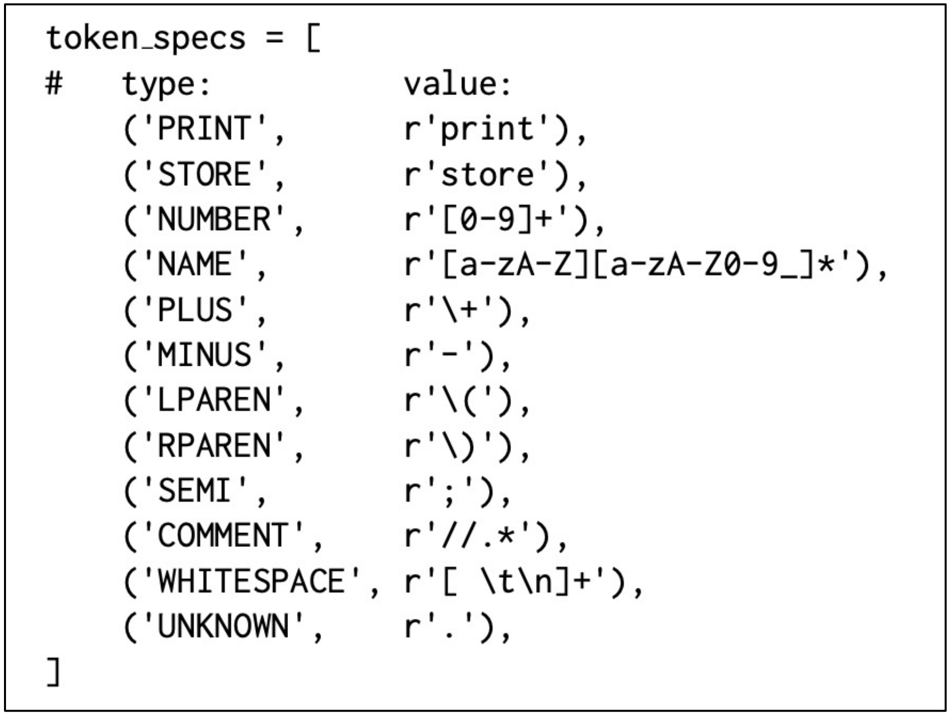


Listing 3.1: Grammar for the Exp1 language.

```
stmtlist : (stmt)*

stmt : print exp ;
     | store var exp ;

exp  : + exp exp
     | - exp exp
     | \( exp \)
     | var
     | num

var  : <any valid variable name>
num  : <any valid integer digit>
```

Exp1 Lexer

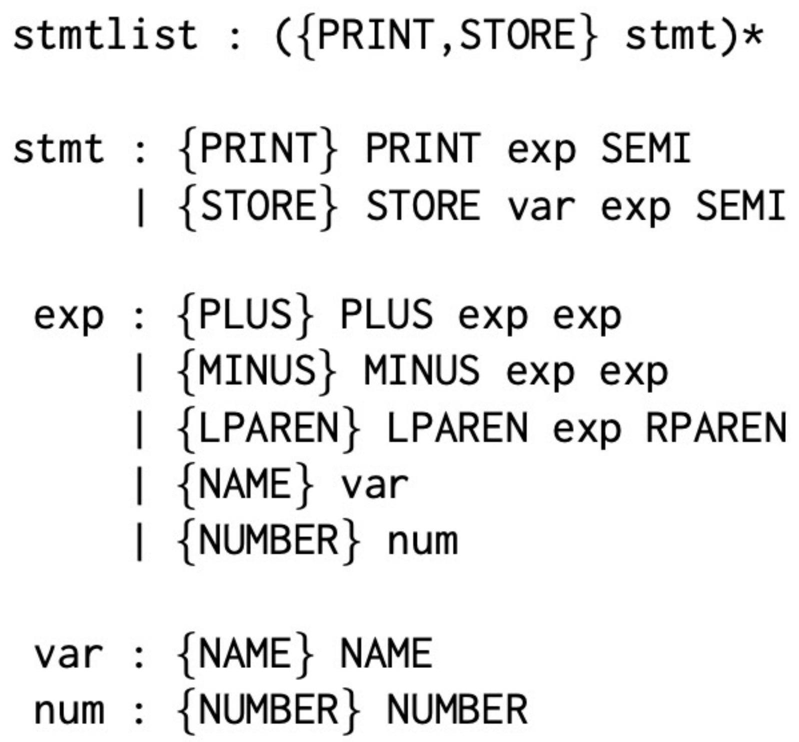
- The only thing that changes in the lexer between the Calc lexer and the Exp1 lexer is the token specification



```
token_specs = [
#    type:           value:
     ('PRINT',       r'print'),
     ('STORE',       r'store'),
     ('NUMBER',      r'[0-9]+'),
     ('NAME',        r'[a-zA-Z][a-zA-Z0-9_]*'),
     ('PLUS',        r'\+'),
     ('MINUS',       r'-'),
     ('LPAREN',      r'\('),
     ('RPAREN',      r'\)'),
     ('SEMI',        r';'),
     ('COMMENT',     r'//.*'),
     ('WHITESPACE', r'[ \t\n]+'),
     ('UNKNOWN',     r'.'),
]
```

Exp1 Grammar

- Rewriting the grammar in terms of tokens and lookahead sets.



```
stmtlist : ({PRINT,STORE} stmt)*

stmt : {PRINT} PRINT exp SEMI
     | {STORE} STORE var exp SEMI

 exp : {PLUS} PLUS exp exp
     | {MINUS} MINUS exp exp
     | {LPAREN} LPAREN exp RPAREN
     | {NAME} var
     | {NUMBER} num

 var : {NAME} NAME
 num : {NUMBER} NUMBER
```

The Parser

```
def stmtlist(stream):
  while stream.pointer().type in ['PRINT','STORE']:
    stmt(stream)
  return
```

```
def stmt(stream):
    token = stream.pointer()
    if token.type in ['PRINT']:
        stream.match('PRINT')
        exp(stream)
        stream.match('SEMI')
        return
    elif token.type in ['STORE']:
        stream.match('STORE')
        var(stream)
        exp(stream)
        stream.match('SEMI')
        return
    else:
        raise SyntaxError("stmt: syntax error
                                    .format(token.value))
```

```
def exp(stream):
    token = stream.pointer()
    if token.type in ['PLUS']:
        stream.match('PLUS')
        exp(stream)
        exp(stream)
        return
    elif token.type in ['MINUS']:
        stream.match('MINUS')
        exp(stream)
        exp(stream)
        return
    elif token.type in ['LPAREN']:
        stream.match('LPAREN')
        exp(stream)
        stream.match('RPAREN')
        return
    elif token.type in ['NAME']:
        var(stream)
        return
    elif token.type in ['NUMBER']:
        num(stream)
        return
    else:
        raise SyntaxError("exp: syntax error at {}"
```

```
def var(stream):
    token = stream.pointer()
    if token.type in ['NAME']:
        stream.match('NAME')
        return
    else:
        raise SyntaxError("var: syntax error at {}"
                                .format(token.value))
```

```
def num(stream):
    token = stream.pointer()
    if token.type in ['NUMBER']:
        stream.match('NUMBER')
        return
    else:
        raise SyntaxError("num: syntax error at {}"
                                    .format(token.value))
```

Testing the Parser

```
$ python3 exp1_parser.py
store x 1; print + x 1;
^D
parse successful
$
```

```
$ python3 exp1_parser.py
print 1 + 1;
^D
error: unexpected token PLUS while parsing, expected SEMI
$
```

Writing an Interpreter for Exp1

- Syntax-directed interpretation – we pass values along the parse tree in a bottom-up fashion
- Writing an interpreter for Exp1
  - We add code to the parser that interprets the values within the phrase structure of a program.
  - Observation: we need access to the token values during parsing in order to evaluate things like the values of numbers or the value of an addition.
  - Observation: interpretation always starts at the leaves.

Writing an Interpreter for Exp1

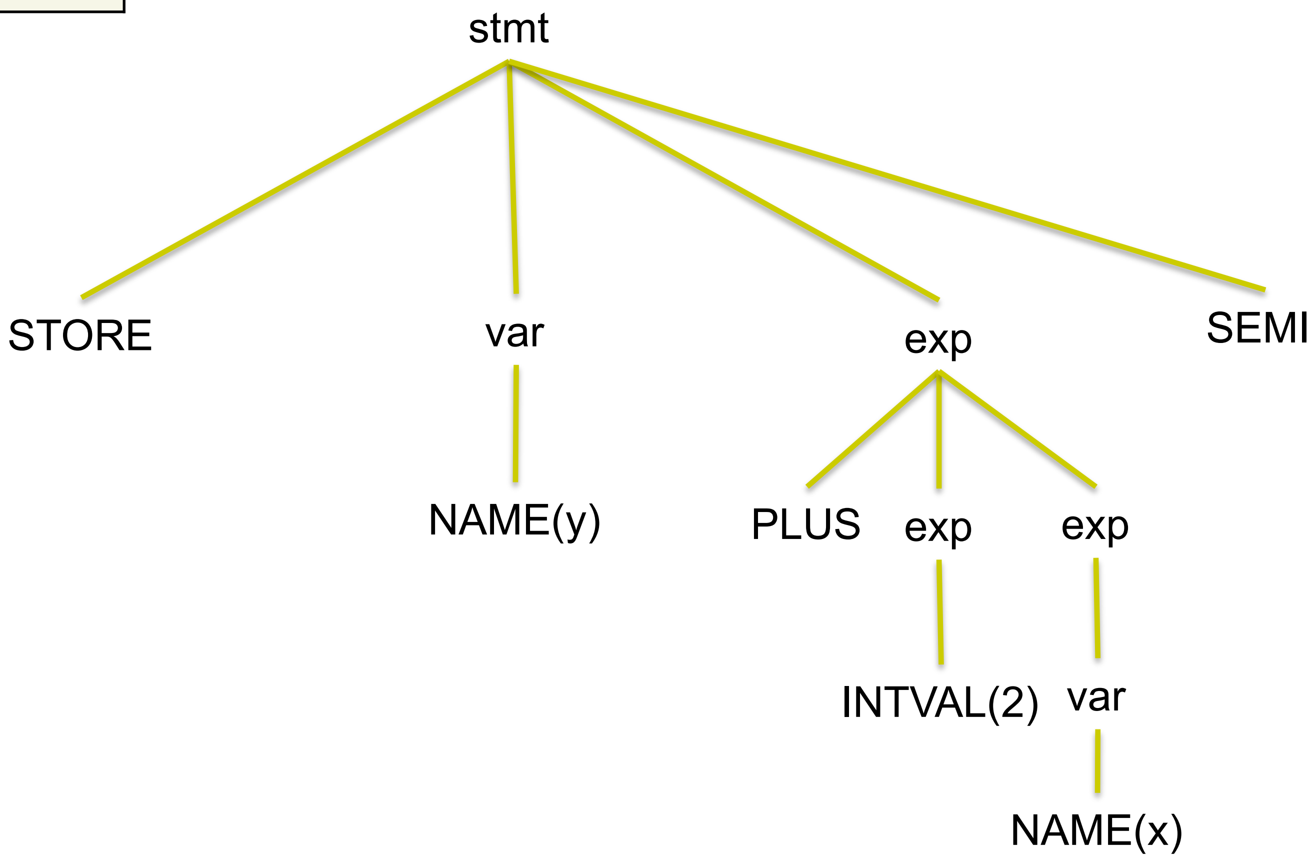
- Consider the following Exp1 program:

```
store y + 2 x ;
```

- Assumption: x has the value 3.

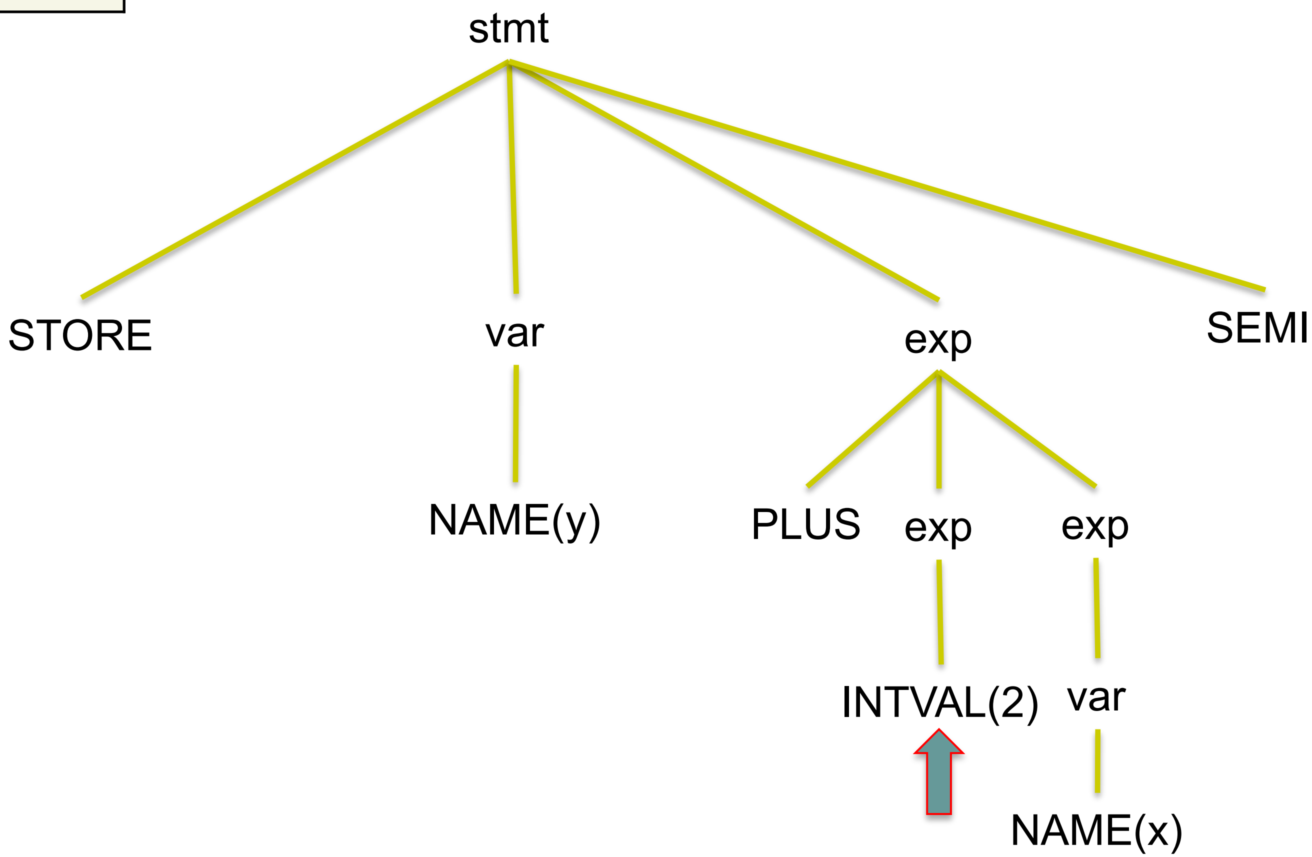
Symbol Table	
x	3
y	???

```
Action: start
```



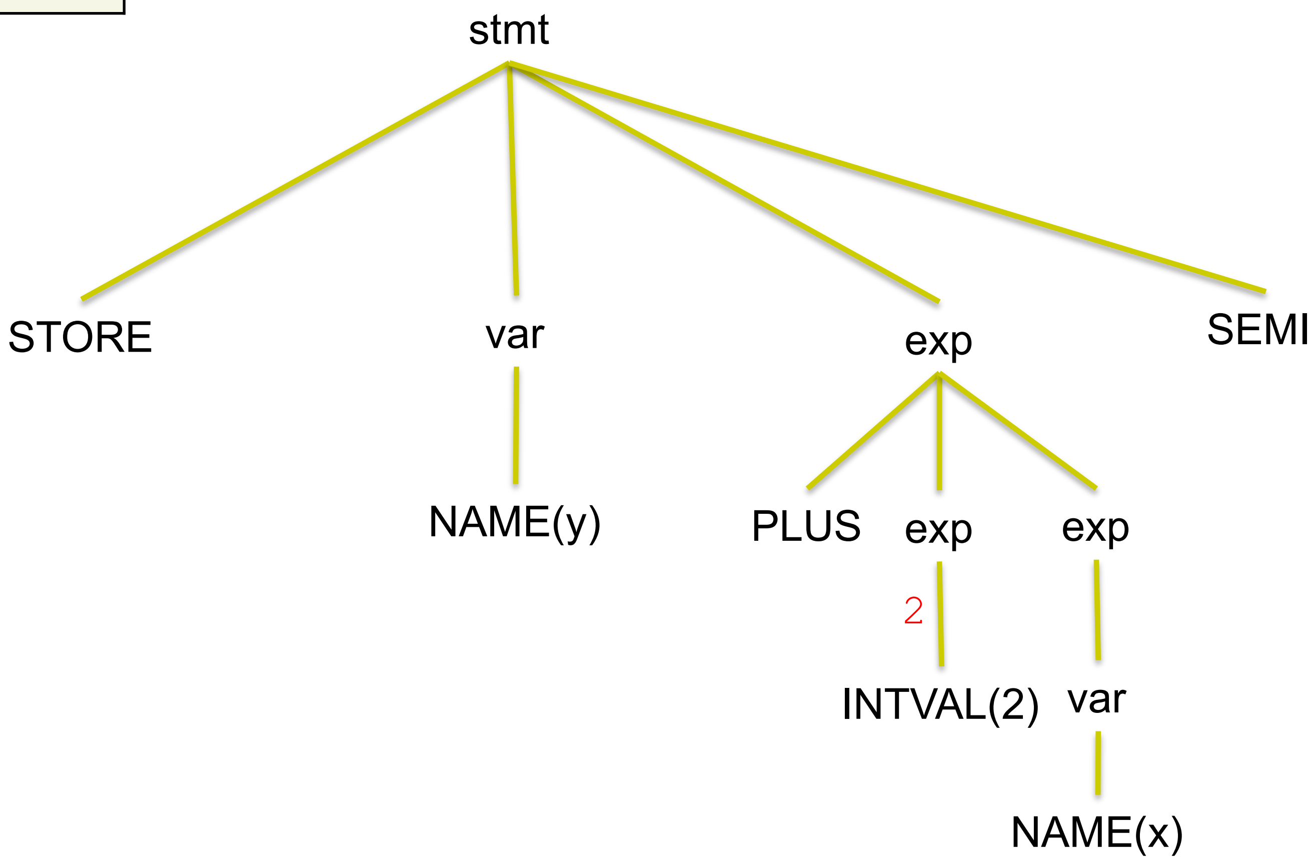
Symbol Table	
x	3
y	???

Action: interpret INTVAL



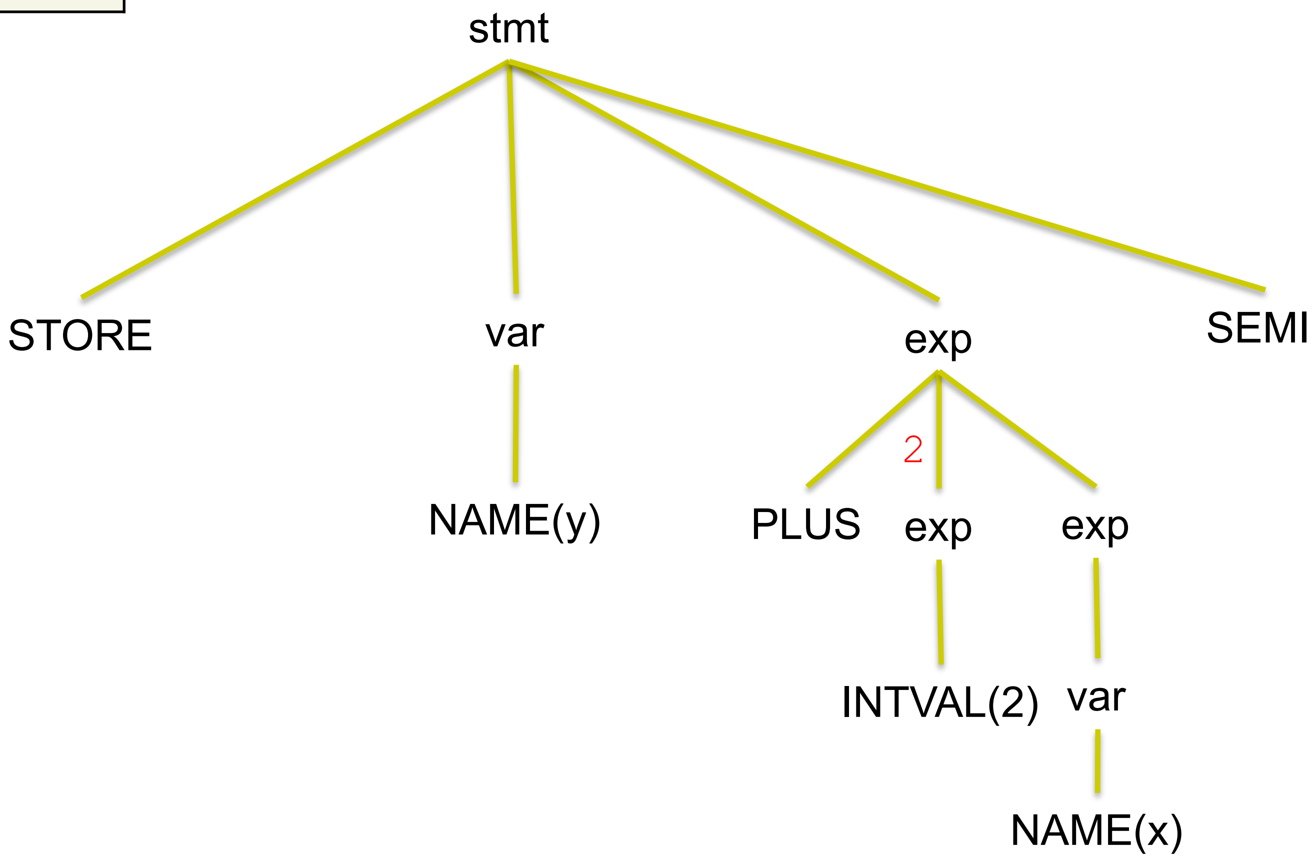
Symbol Table	
x	3
y	???

`Action: propagate`



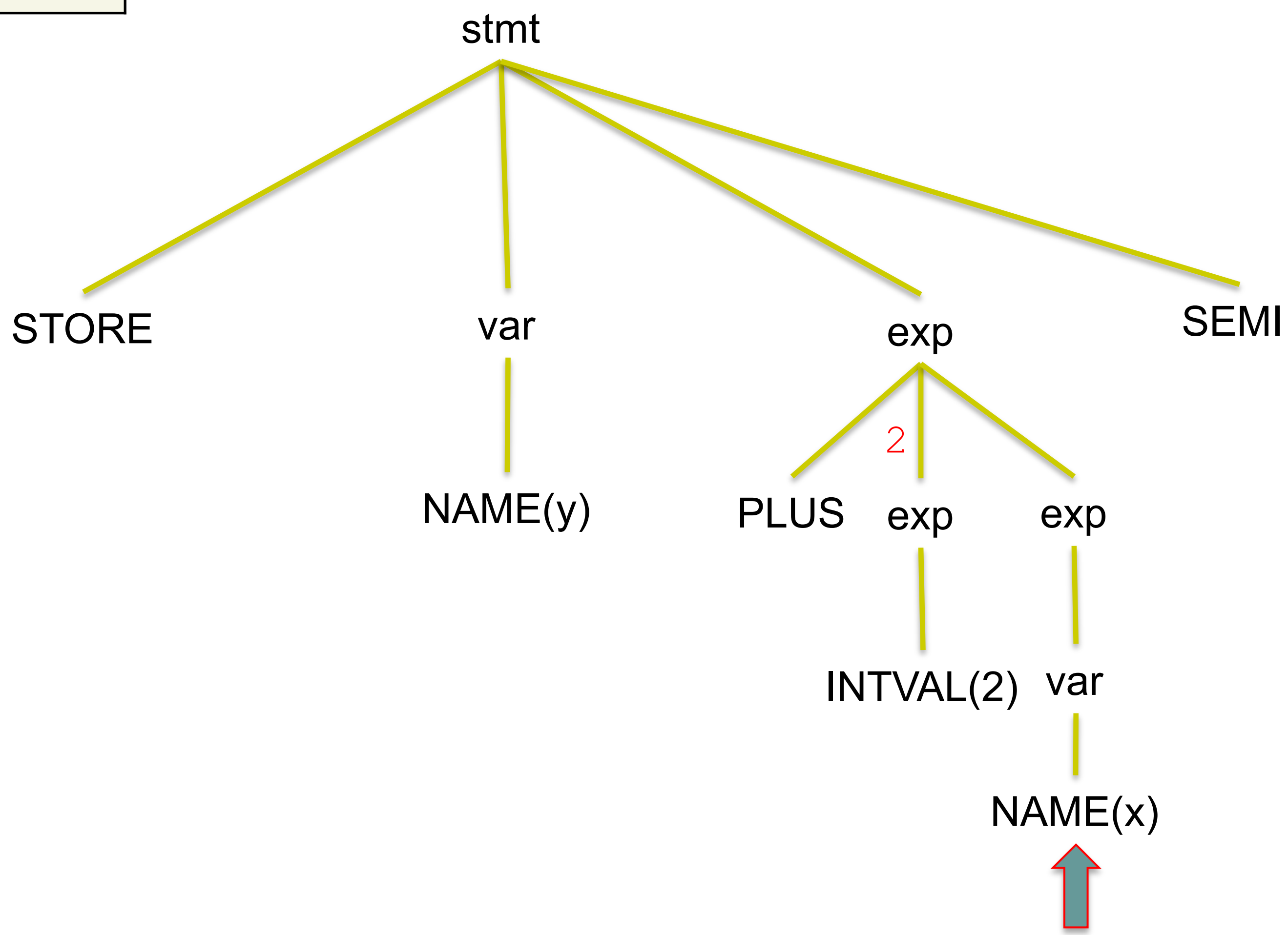
Symbol Table	
x	3
y	???

Action: propagate



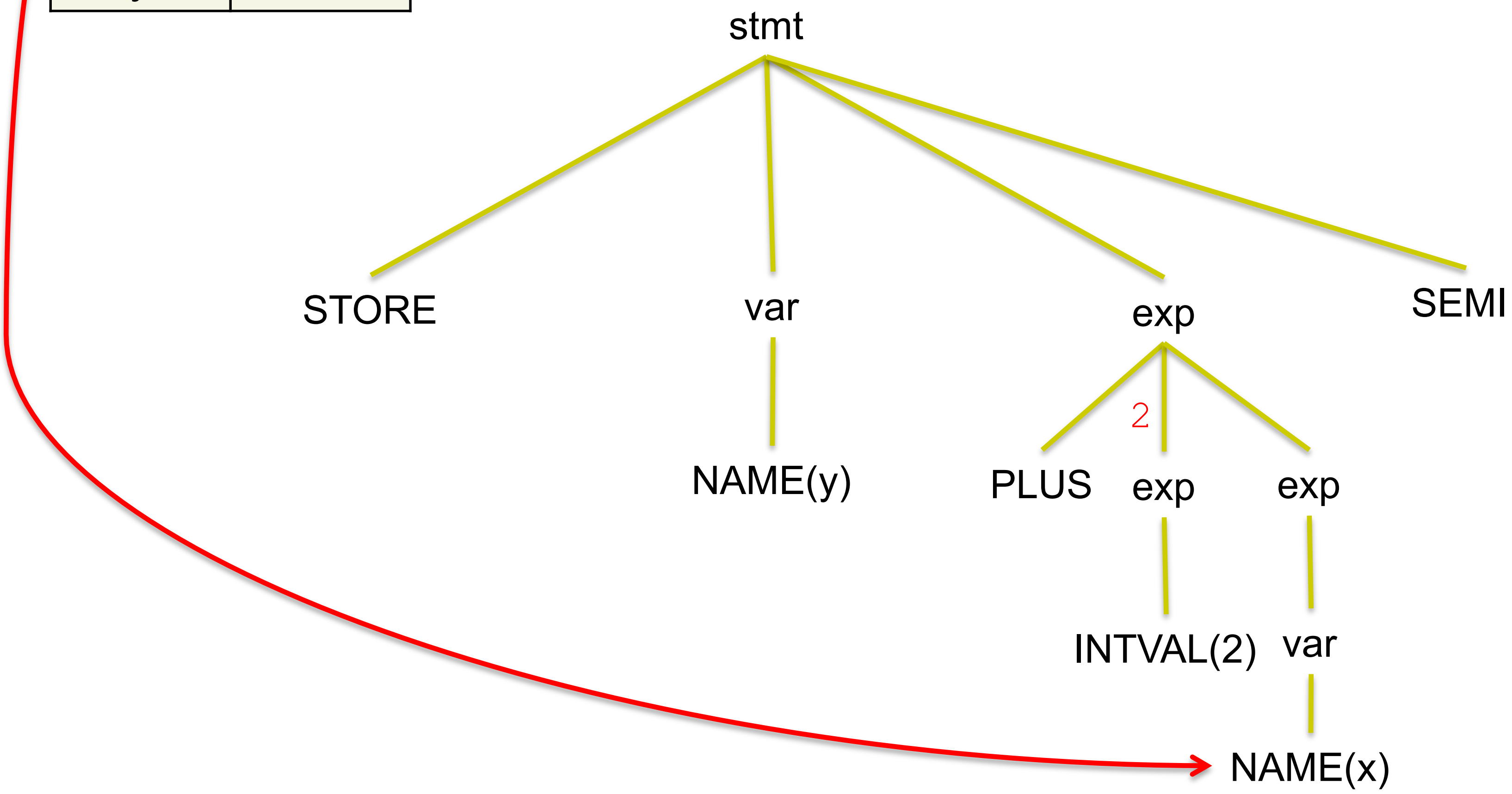
Symbol Table	
x	3
y	???

`Action: interpret NAME`



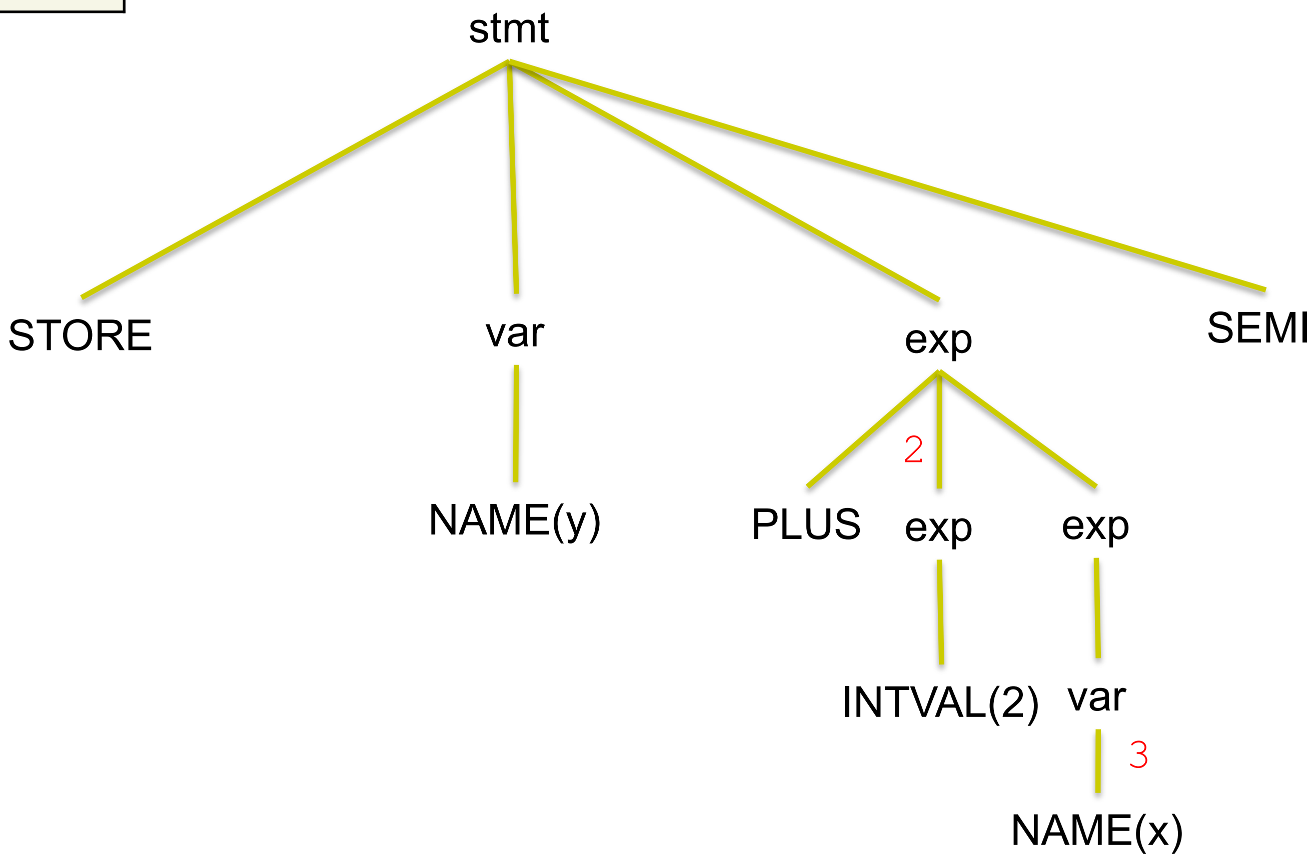
Symbol Table	
x	3
y	???

Action: read symbol table



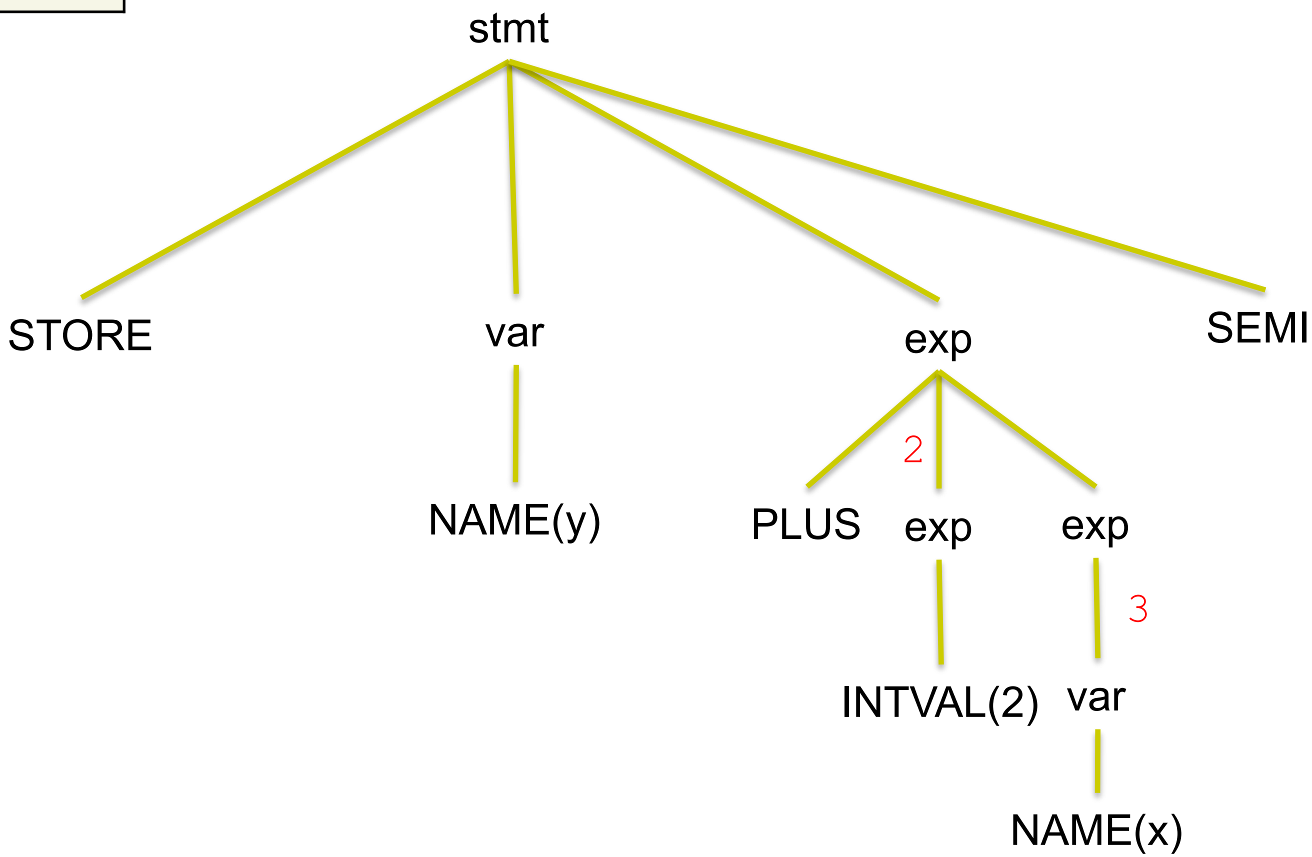
Symbol Table	
x	3
y	???

`Action: propagate`



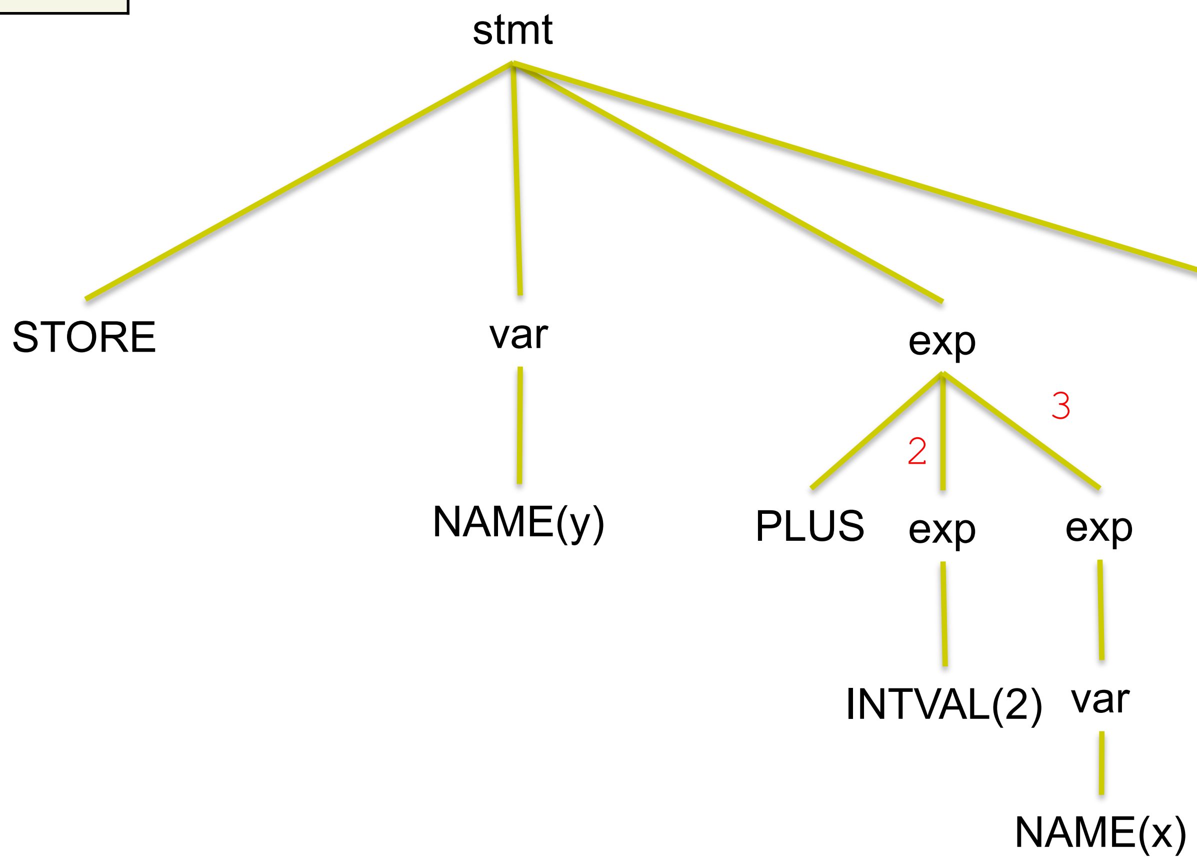
Symbol Table	
x	3
y	???

Action: propagate



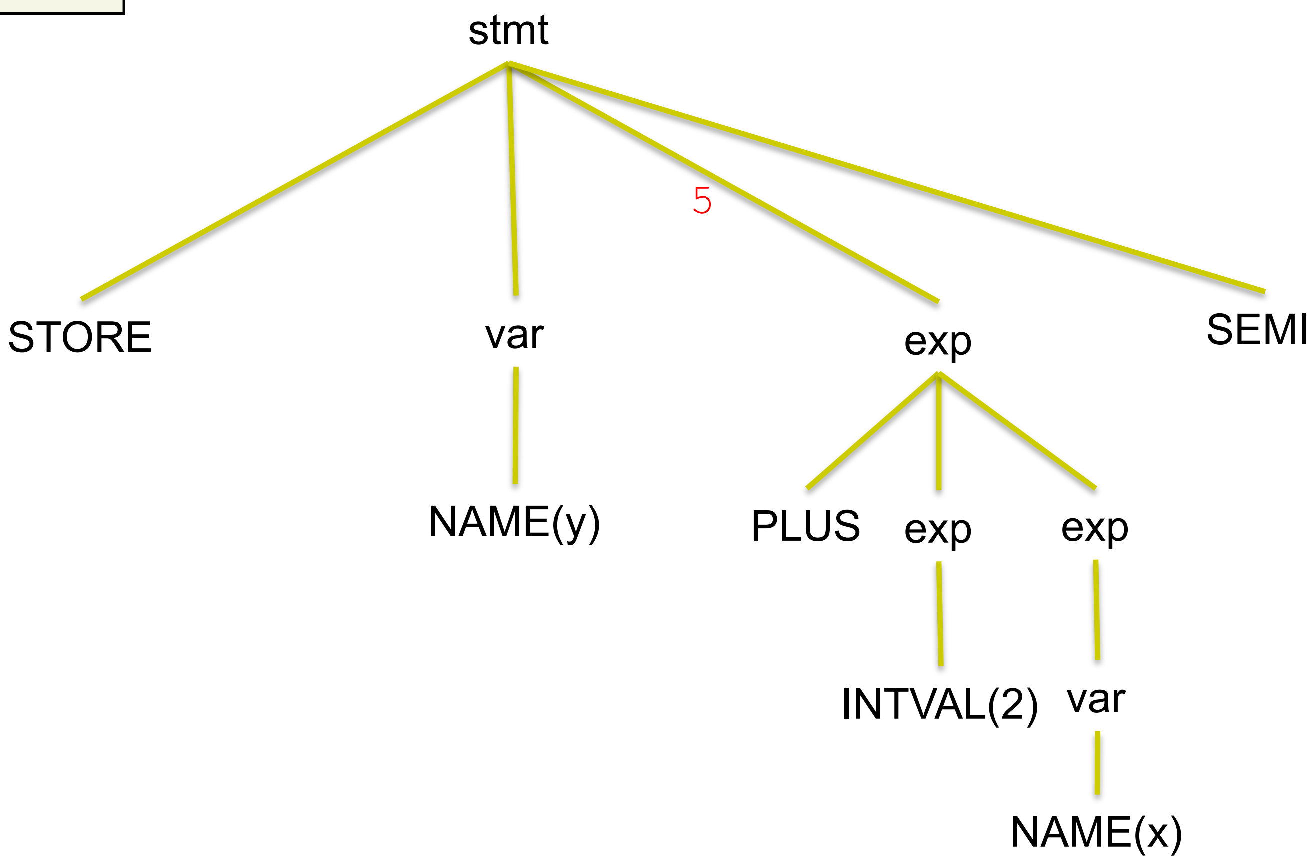
Symbol Table	
x	3
y	???

Action: add



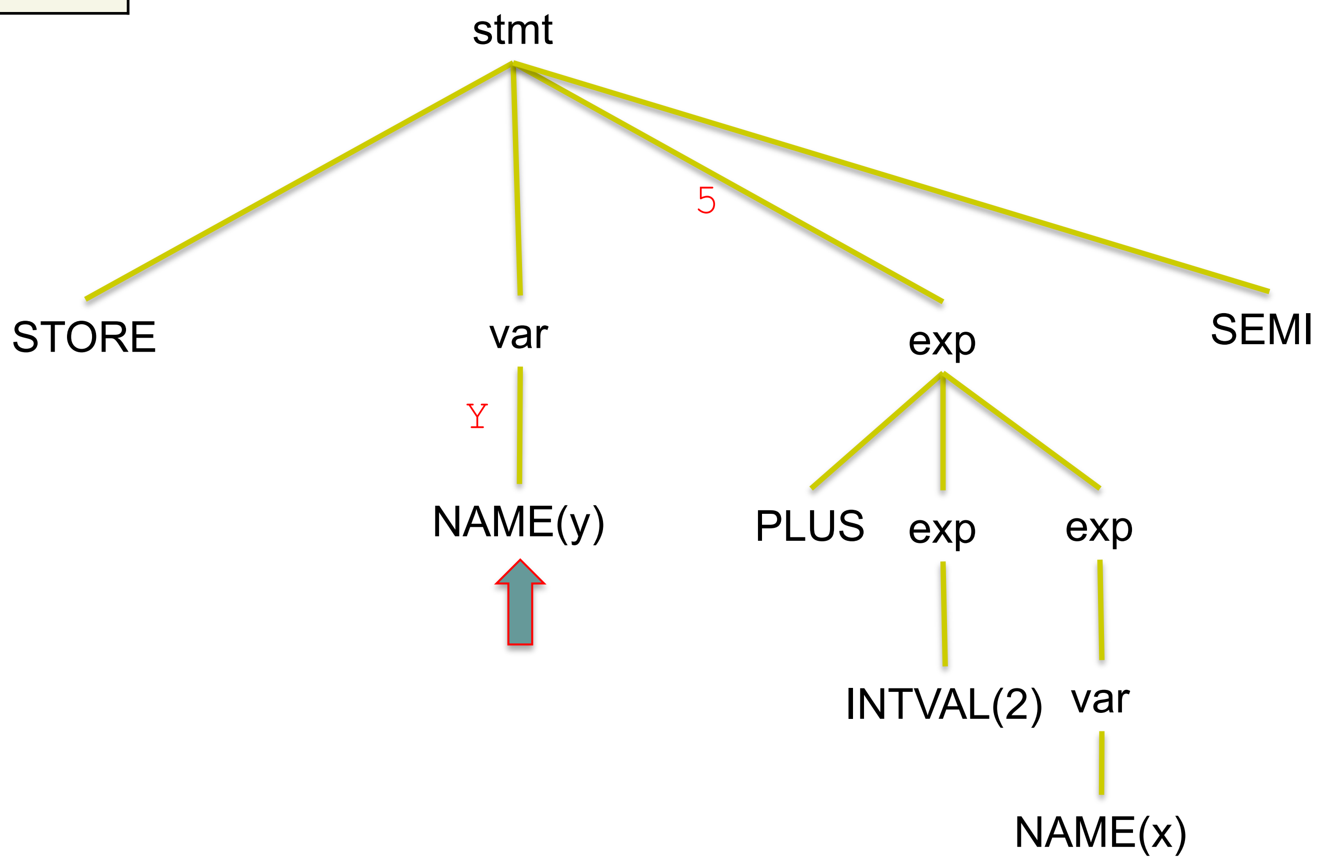
Symbol Table	
x	3
y	???

```
Action: propagate
```



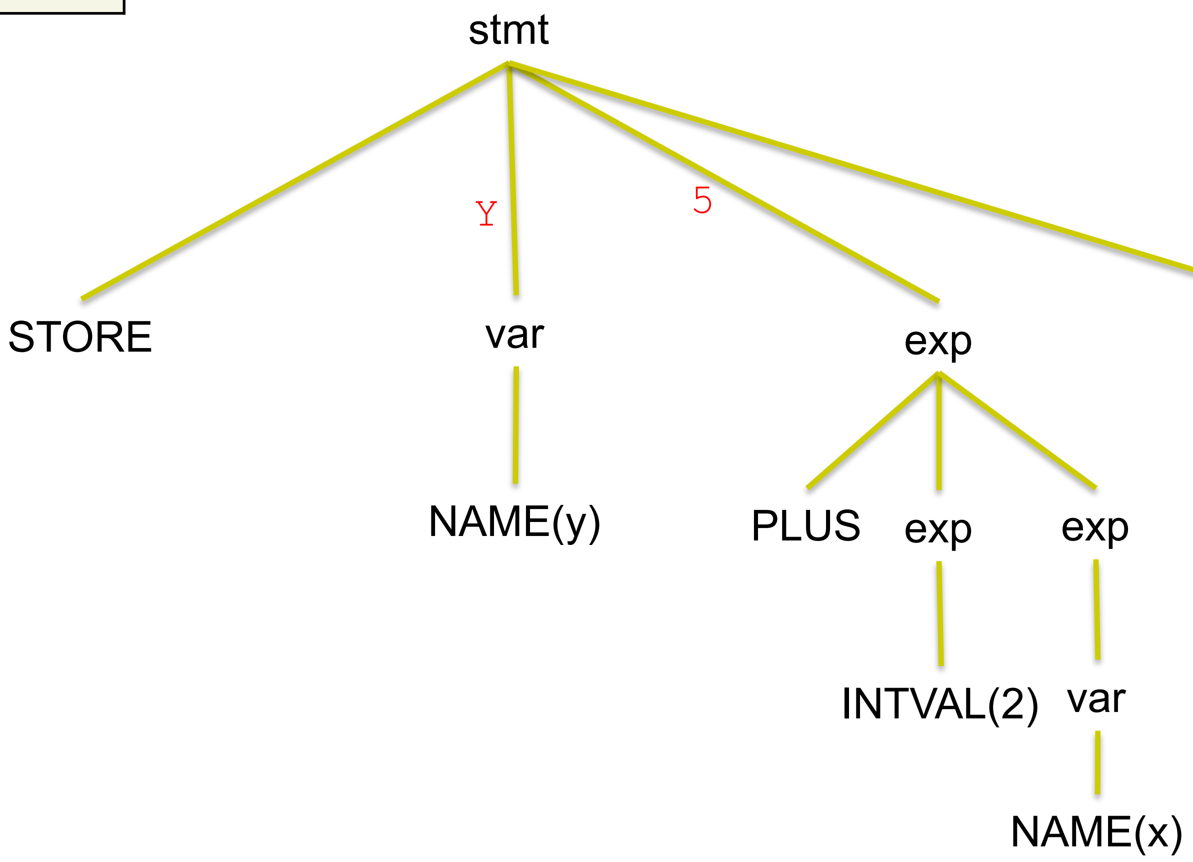
Symbol Table	
x	3
y	???

Action: eval name



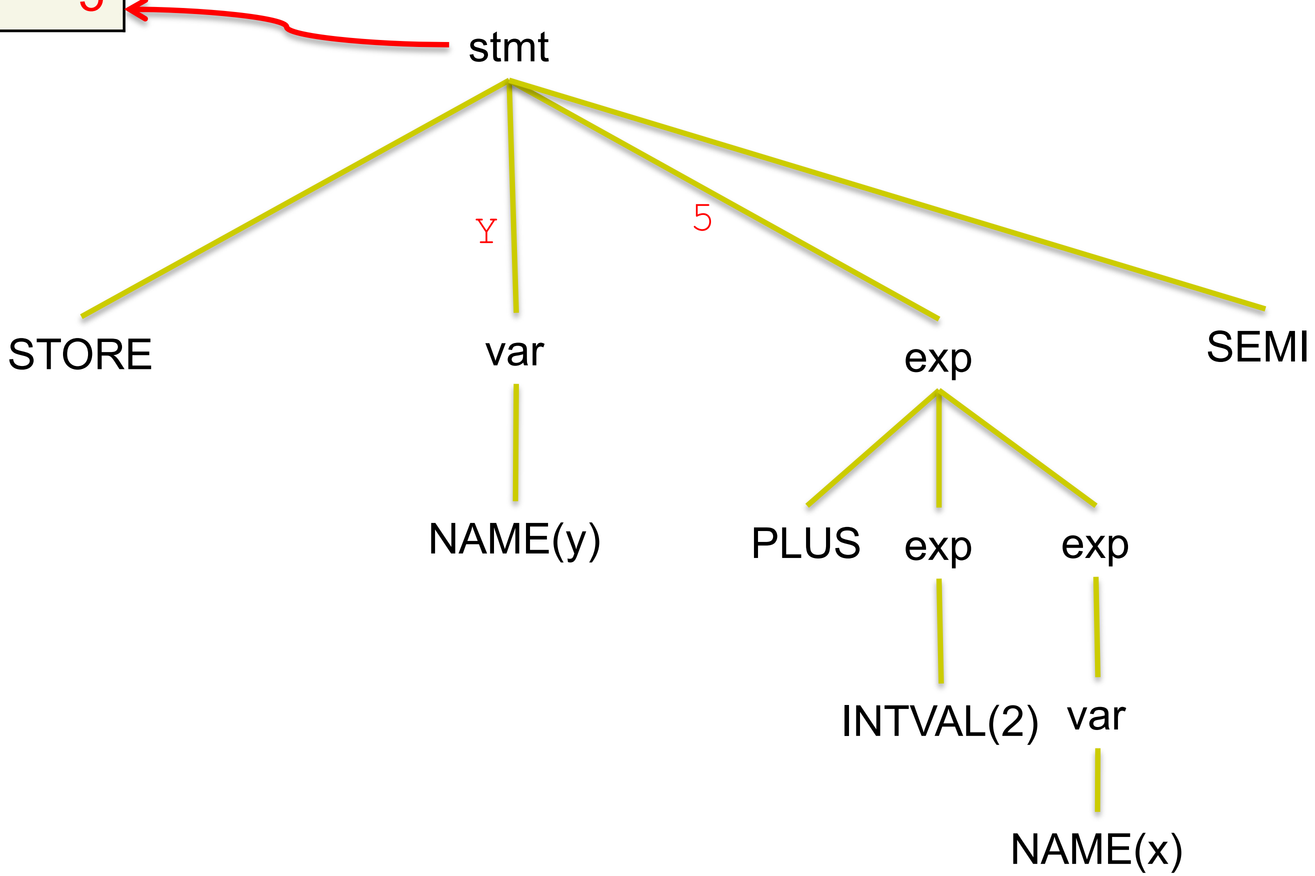
Symbol Table	
x	3
y	???

Action: propagate



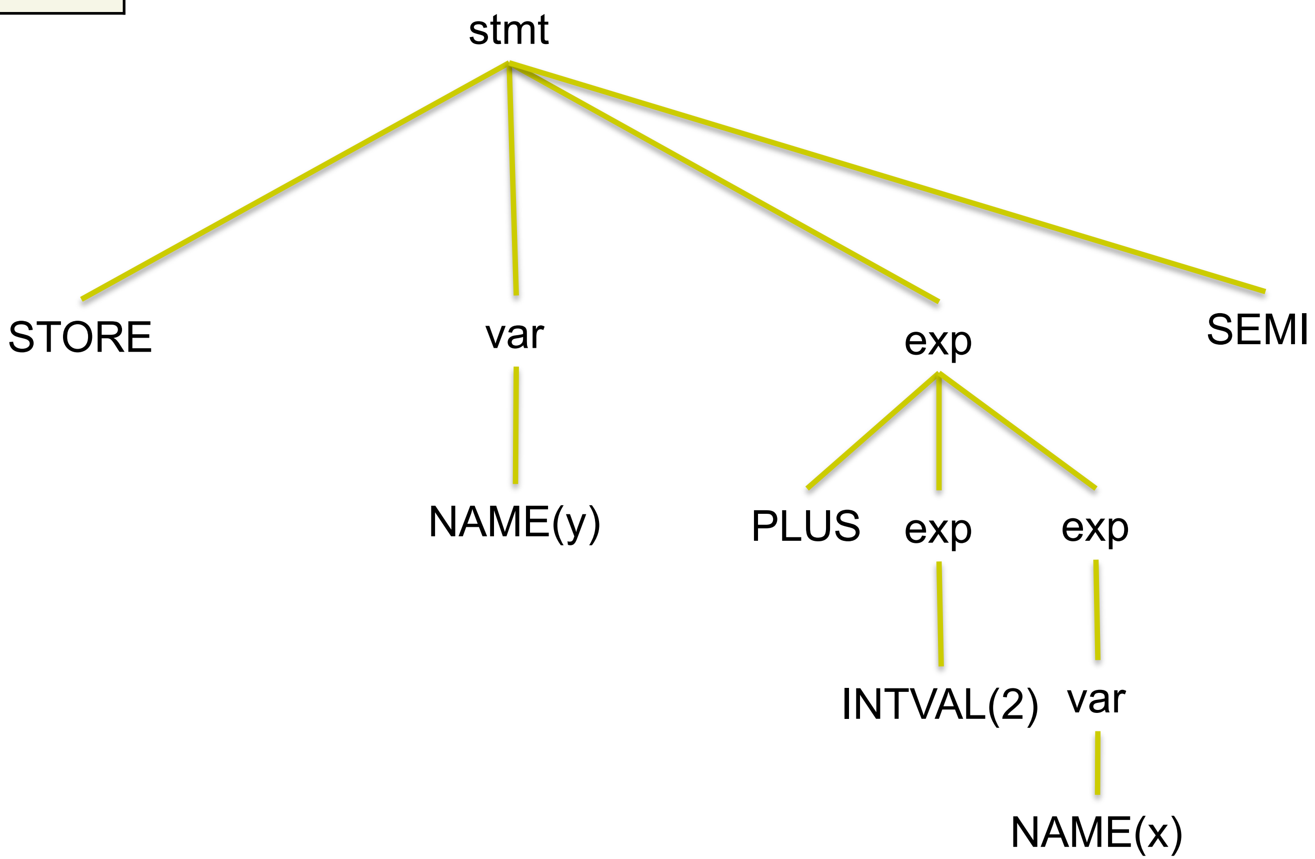
Symbol Table	
x	3
y	5

Action: write to symtab



Symbol Table	
x	3
y	5

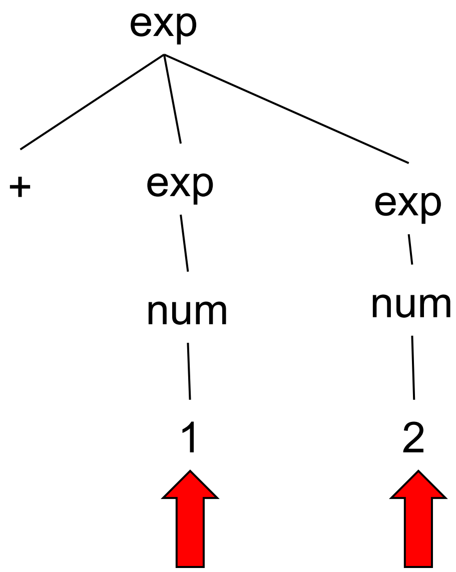
```
Action: done
```



Interpretation

- Consider the Exp1 expression: + 1 2

```
exp          :          + exp exp
             |          - exp exp
             |          \( exp \)
             |          var
             |          num
             ;
```

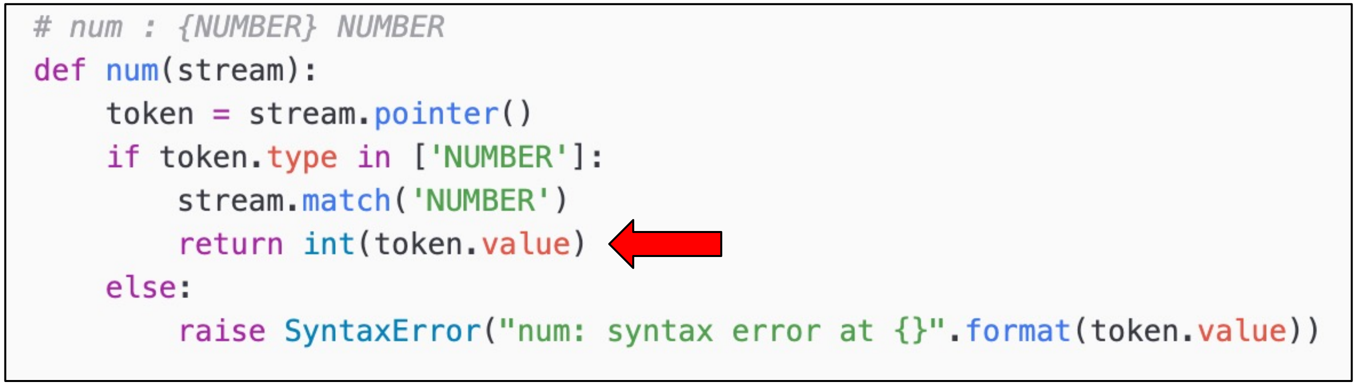


Interpretation means, computing the value of the root node.

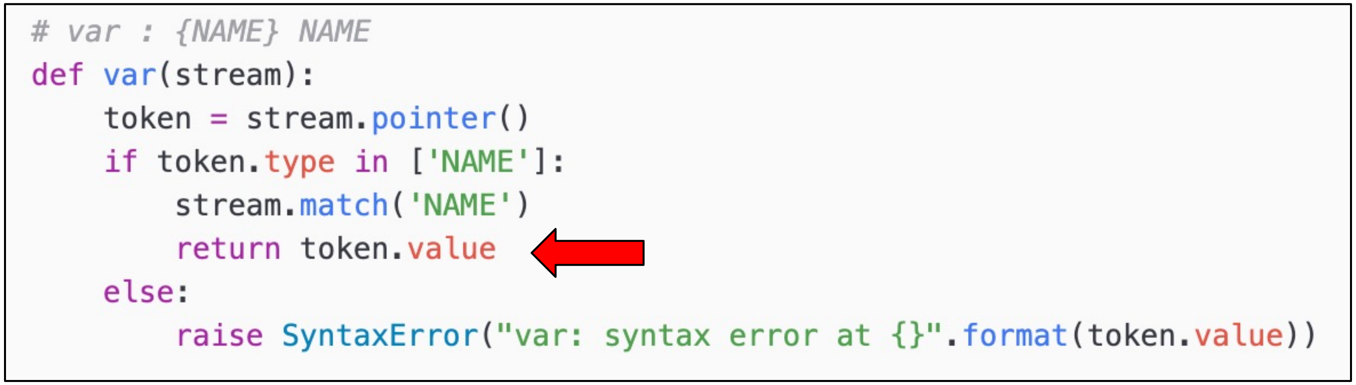
We have to start at the leaves of the tree, that is where the primitive values are and proceed upwards…

What is the value at the root node?

Interpretation & the Parser

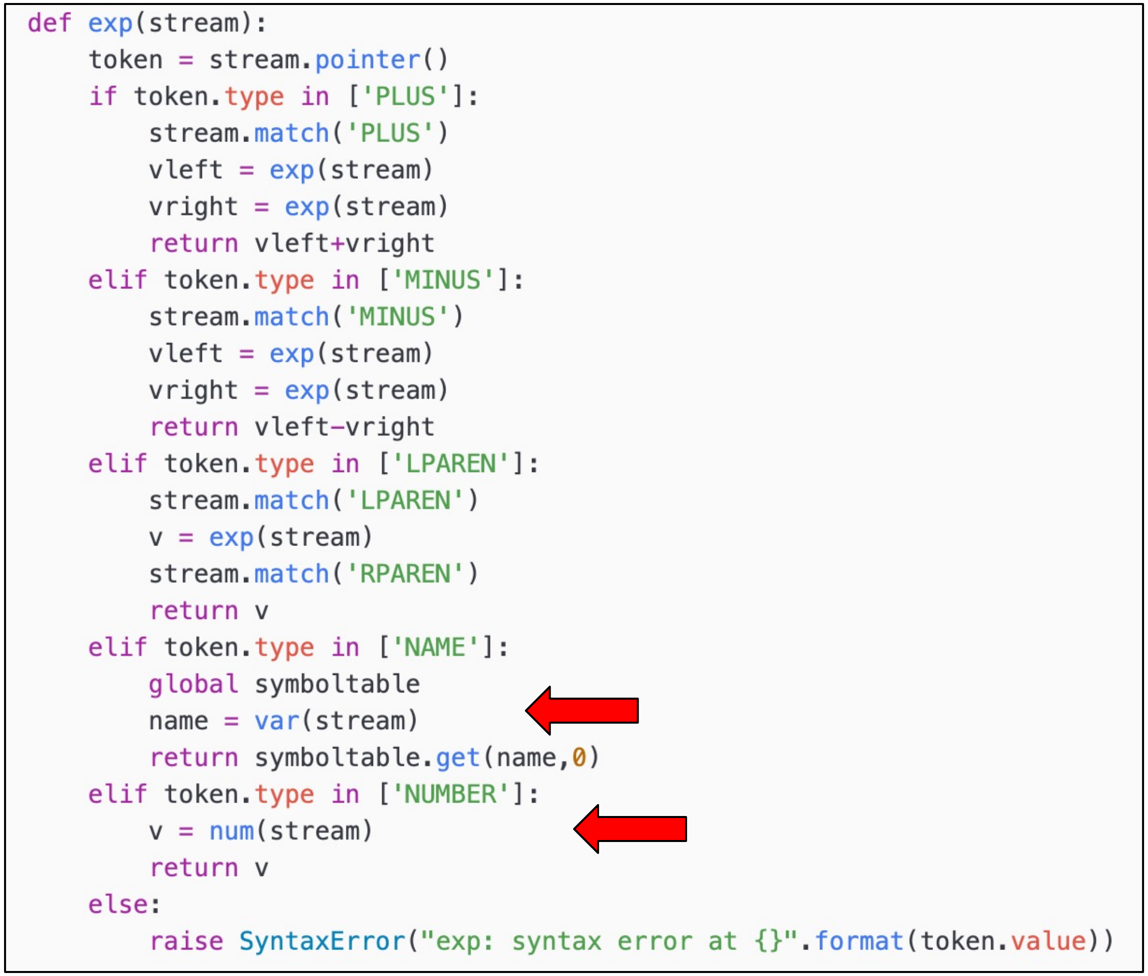


```
# num : {NUMBER} NUMBER
def num(stream):
    token = stream.pointer()
    if token.type in ['NUMBER']:
        stream.match('NUMBER')
        return int(token.value)
    else:
        raise SyntaxError("num: syntax error at {}".format(token.value))
```



```
# var : {NAME} NAME
def var(stream):
    token = stream.pointer()
    if token.type in ['NAME']:
        stream.match('NAME')
        return token.value
    else:
        raise SyntaxError("var: syntax error at {}".format(token.value))
```

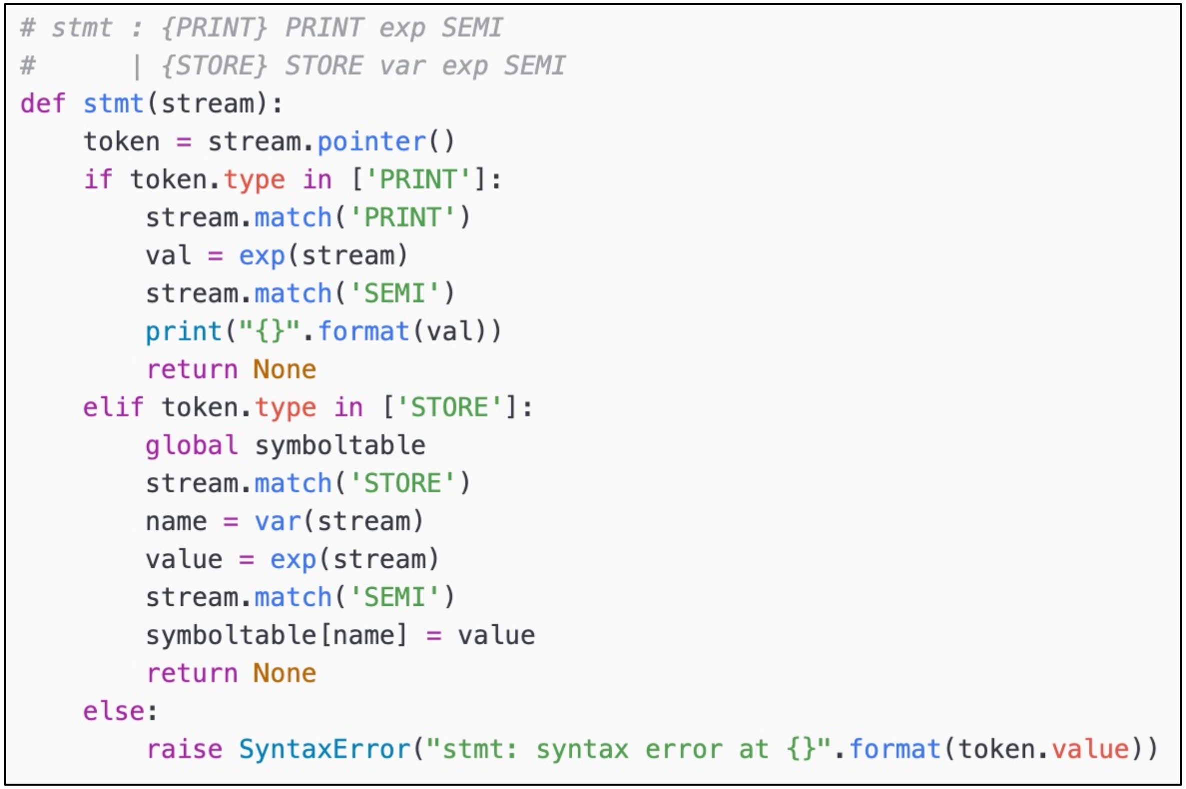
Interpretation & the Parser



```
def exp(stream):
    token = stream.pointer()
    if token.type in ['PLUS']:
        stream.match('PLUS')
        vleft = exp(stream)
        vright = exp(stream)
        return vleft+vright
    elif token.type in ['MINUS']:
        stream.match('MINUS')
        vleft = exp(stream)
        vright = exp(stream)
        return vleft-vright
    elif token.type in ['LPAREN']:
        stream.match('LPAREN')
        v = exp(stream)
        stream.match('RPAREN')
        return v
    elif token.type in ['NAME']:
        global symboltable
        name = var(stream)
        return symboltable.get(name,0)
    elif token.type in ['NUMBER']:
        v = num(stream)
        return v
    else:
        raise SyntaxError("exp: syntax error at {}".format(token.value))
```

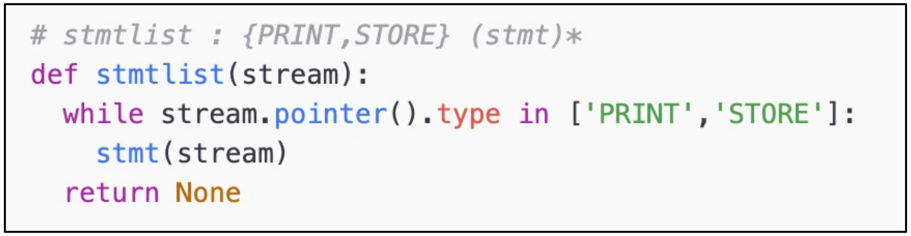
Recursion lets
the values percolate up.

Interpretation & the Parser



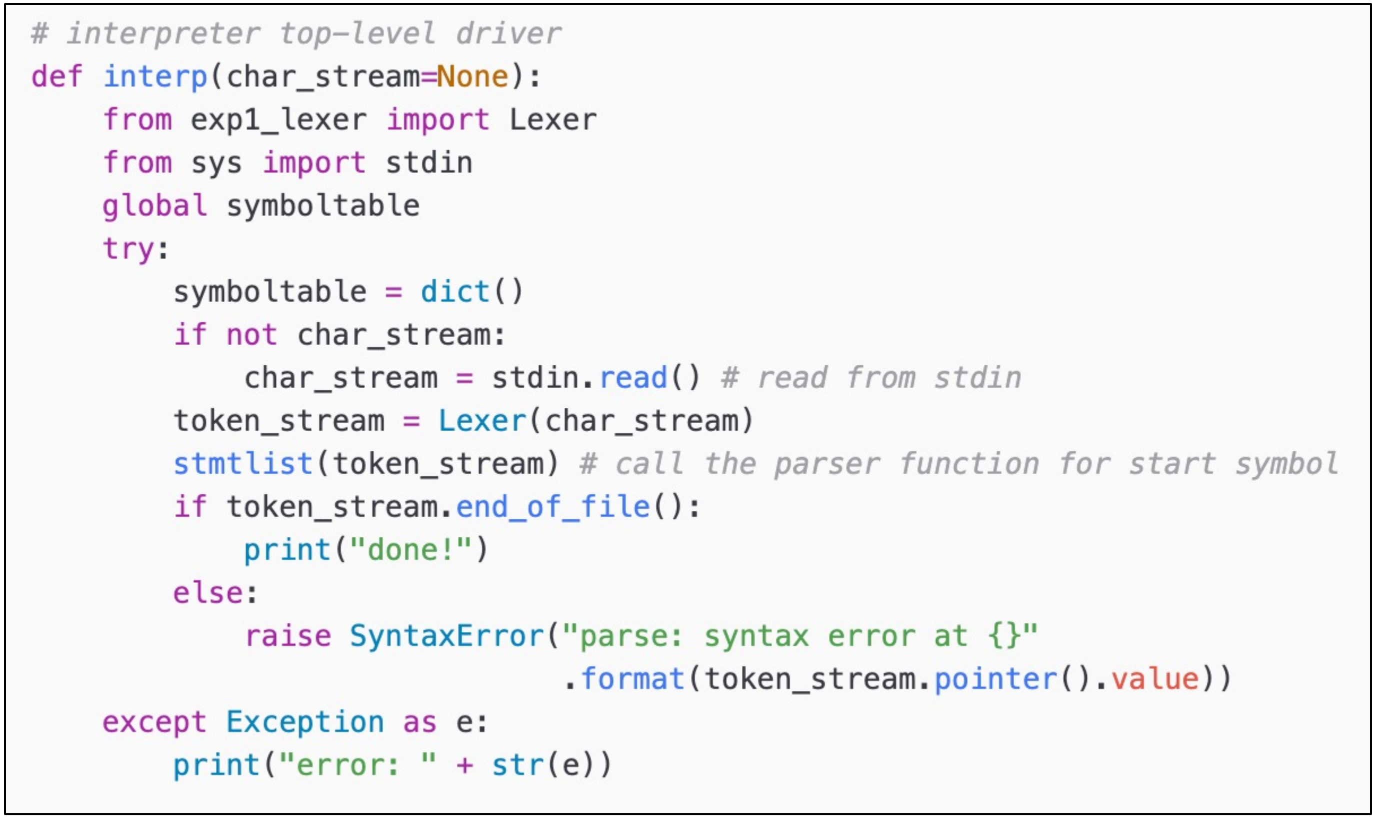
```
# stmt : {PRINT} PRINT exp SEMI
#      | {STORE} STORE var exp SEMI
def stmt(stream):
    token = stream.pointer()
    if token.type in ['PRINT']:
        stream.match('PRINT')
        val = exp(stream)
        stream.match('SEMI')
        print("{}".format(val))
        return None
    elif token.type in ['STORE']:
        global symboltable
        stream.match('STORE')
        name = var(stream)
        value = exp(stream)
        stream.match('SEMI')
        symboltable[name] = value
        return None
    else:
        raise SyntaxError("stmt: syntax error at {}".format(token.value))
```

Recursion lets
the values percolate up.



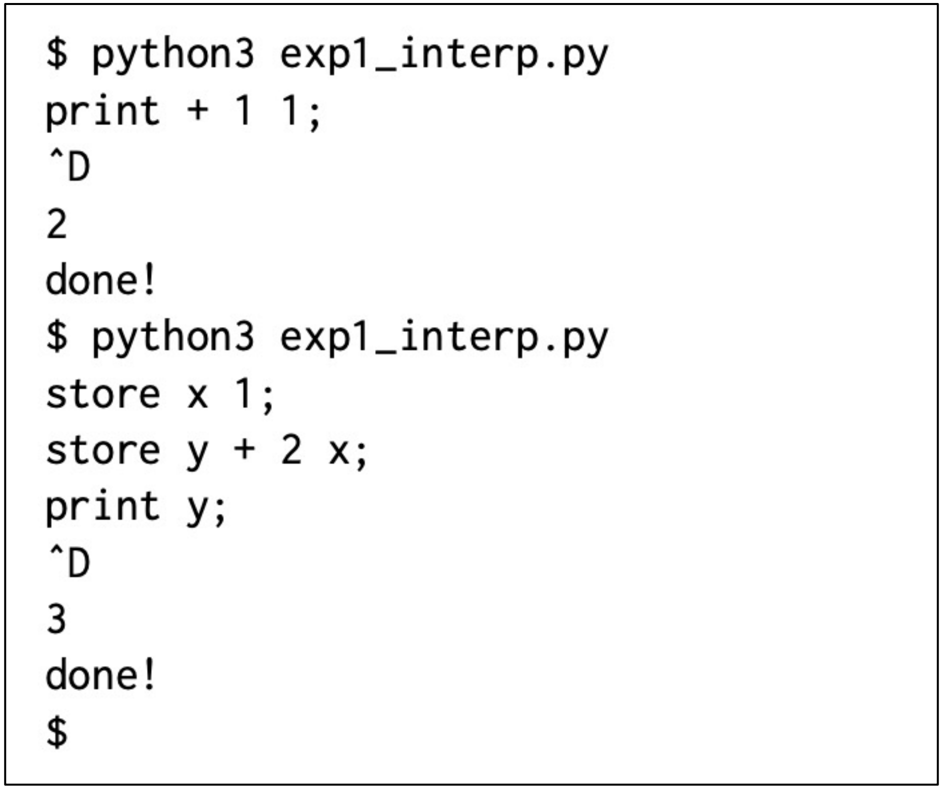
```
# stmtlist : {PRINT,STORE} (stmt)*
def stmtlist(stream):
  while stream.pointer().type in ['PRINT','STORE']:
    stmt(stream)
  return None
```

Interpretation & the Parser



```python
# interpreter top-level driver
def interp(char_stream=None):
    from exp1_lexer import Lexer
    from sys import stdin
    global symboltable
    try:
        symboltable = dict()
        if not char_stream:
            char_stream = stdin.read() # read from stdin
        token_stream = Lexer(char_stream)
        stmtlist(token_stream) # call the parser function for start symbol
        if token_stream.end_of_file():
            print("done!")
        else:
            raise SyntaxError("parse: syntax error at {}"
                              .format(token_stream.pointer().value))
    except Exception as e:
        print("error: " + str(e))
```

Testing the Interpreter



```
$ python3 exp1_interp.py
print + 1 1;
^D
2
done!
$ python3 exp1_interp.py
store x 1;
store y + 2 x;
print y;
^D
3
done!
$
```

Reading

- Chapter 3
- Assignment #2 – please see website