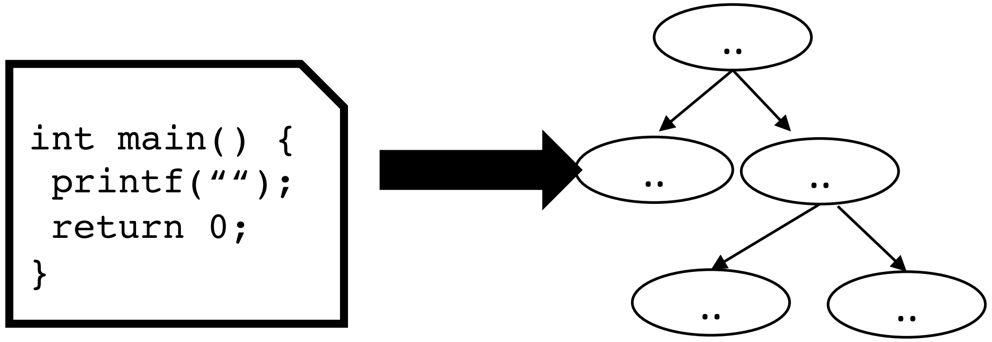
CSE110A: Compilers

May 3, 2023

Topics:

- *One bonus lecture on parsing!*
- *Parser generators*



```
int main() {
 printf(““);
 return 0;
}
```

Announcements

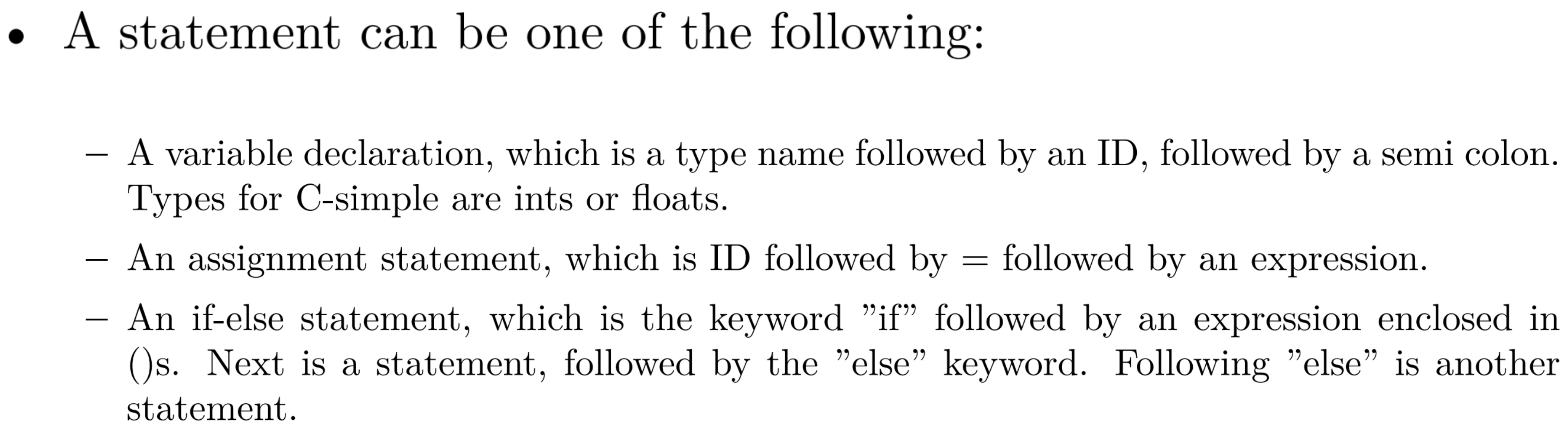
- HW 2!
  - due on Thursday at Midnight
  - Some of office hours left, but be careful because mine fill up quickly
- We are working on grading HW 1
- Midterm will be given on May 8 (Monday)
  - Taken during class
  - Study material is homeworks, slides, and book readings
  - 3 pages of notes (front and back, handwritten or typed)

Announcements

- No Quiz today, work on homework!

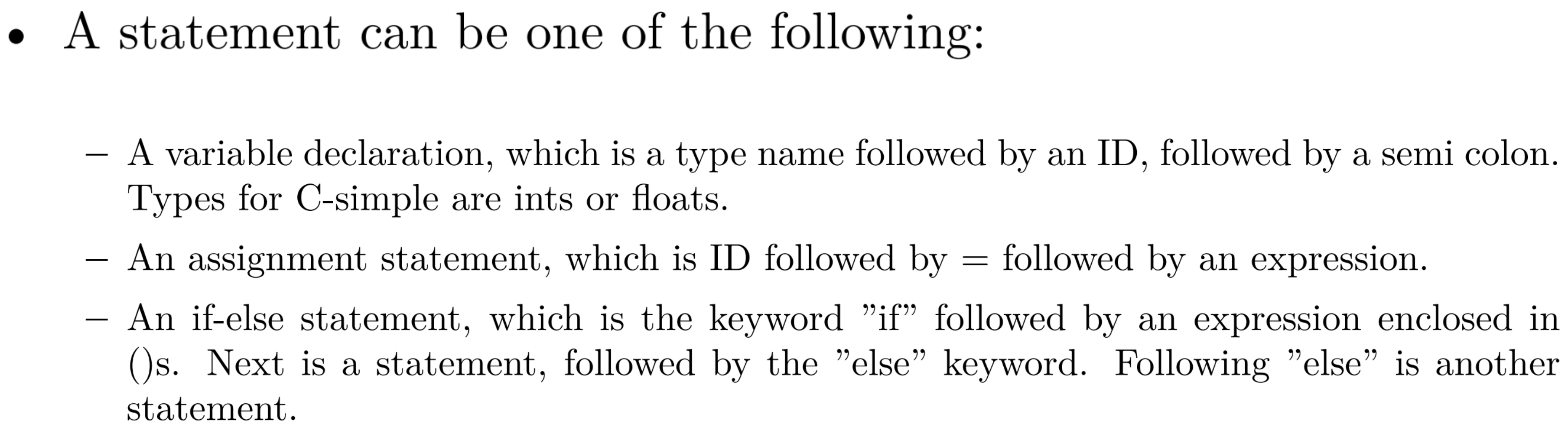
Homework 2 clarifications

- Tip for starting on statement rules



- A statement can be one of the following:
  - A variable declaration, which is a type name followed by an ID, followed by a semi colon. Types for C-simple are ints or floats.
  - An assignment statement, which is ID followed by = followed by an expression.
  - An if-else statement, which is the keyword ”if” followed by an expression enclosed in ()s. Next is a statement, followed by the ”else” keyword. Following ”else” is another statement.

Simply translate the English:



- A statement can be one of the following:
  - A variable declaration, which is a type name followed by an ID, followed by a semi colon. Types for C-simple are ints or floats.
  - An assignment statement, which is ID followed by = followed by an expression.
  - An if-else statement, which is the keyword ”if” followed by an expression enclosed in ()s. Next is a statement, followed by the ”else” keyword. Following ”else” is another statement.

Simply translate the English:

```
Statement ::= variable_declaration
           |   assignment_statement
           |   if_else_statement
```

- A statement can be one of the following:
  - A variable declaration, which is a type name followed by an ID, followed by a semi colon. Types for C-simple are ints or floats.
  - An assignment statement, which is ID followed by = followed by an expression.
  - An if-else statement, which is the keyword ”if” followed by an expression enclosed in ()s. Next is a statement, followed by the ”else” keyword. Following ”else” is another statement.

Simply translate the English:

```
Statement ::= variable_declaration       variable_declaration ::= TYPE ID SEMI
           |   assignment_statement
           |   if_else_statement
```

- A statement can be one of the following:
  - A variable declaration, which is a type name followed by an ID, followed by a semi colon. Types for C-simple are ints or floats.
  - An assignment statement, which is ID followed by = followed by an expression.
  - An if-else statement, which is the keyword ”if” followed by an expression enclosed in ()s. Next is a statement, followed by the ”else” keyword. Following ”else” is another statement.

Simply translate the English:

```
Statement ::= variable_declaration       variable_declaration ::= TYPE ID SEMI
           |   assignment_statement
           |   if_else_statement
```

- A statement can be one of the following:
  - A variable declaration, which is a type name followed by an ID, followed by a semi colon. Types for C-simple are ints or floats.
  - An assignment statement, which is ID followed by = followed by an expression.
  - An if-else statement, which is the keyword ”if” followed by an expression enclosed in ()s. Next is a statement, followed by the ”else” keyword. Following ”else” is another statement.

Simply translate the English:

```
Statement ::= variable_declaration
          |   assignment_statement
          |   if_else_statement
```

```
variable_declaration ::= type ID SEMI
```

```
type ::= FLOAT
     |   INT
```

Homework 2 clarifications

- Statement precedence

- Do we need to encode statement precedence? Or associativity?

Homework 2 clarifications

```
Statement_list ::= Statement_list Statement
               |   Statement
```

Which one do we want?

```
Statement_list ::= Statement Statement_list
               |   Statement
```

Homework 2 clarifications

```
Statement_list ::= Statement_list Statement
                |   Statement
```

We don’t want left recursion for top-down parsing

```
Statement_list ::= Statement Statement_list
                |   Statement
```

We might want left recursion for left associativity

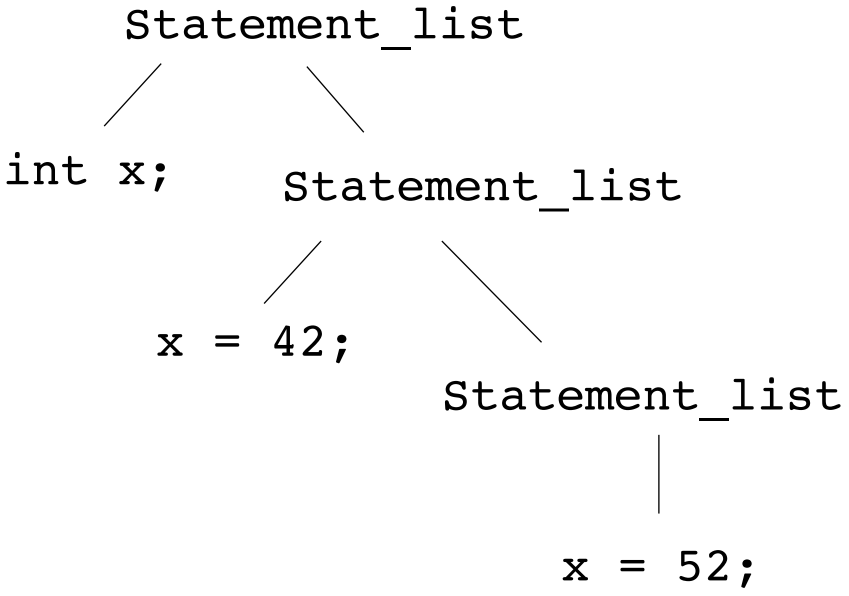
```
int x; x = 42; x = 52;
```

think about this program. We want to evaluate it left to right.

Homework 2 clarifications

```
Statement_list ::= Statement Statement_list
                |   Statement
```

```
int x; x = 42; x = 52;
```



Statement_list

int x;

Statement_list

x = 42;

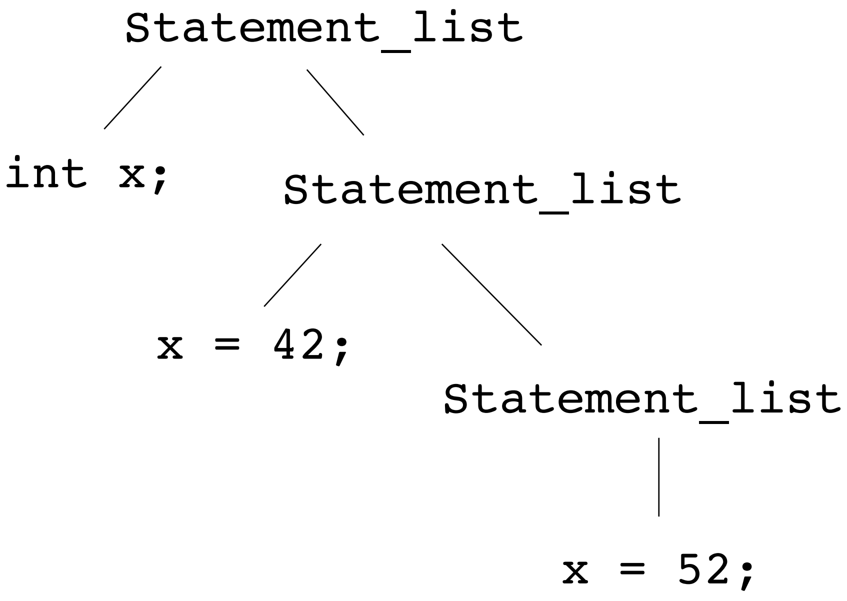
Statement_list

x = 52;

Homework 2 clarifications

```
Statement_list ::= Statement Statement_list
                 |   Statement
```

```
int x; x = 42; x = 52;
```



there is no evaluation associated with a statement list. The evaluation should occur at the statement

Thus we can use the right recursive form with no issue. We also don’t have to worry about statement precedence

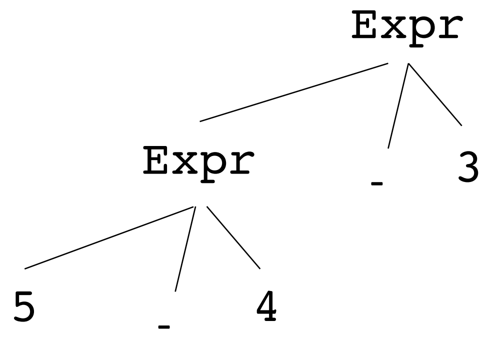
Homework 2 clarifications

- Left associativity and left recursion expressions

Simple grammar for minus expressions

```
Expr ::= Expr MINUS NUM
      |   NUM
```

5 - 4 - 3

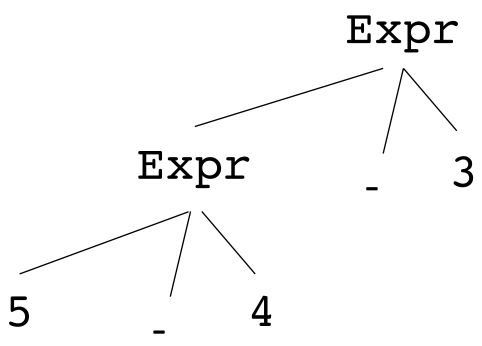


Left recursive grammar makes this parse tree. It encodes associativity

Simple grammar for minus expressions

```
Expr ::= Expr MINUS NUM
      |    NUM
```

```
5 - 4 - 3
```



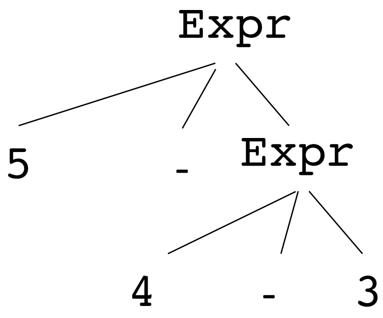
Left recursive grammar makes this parse tree. It encodes associativity.

But left recursion won't work for top-down parsers!

What if we do it right recursive

```
Expr ::= NUM MINUS Expr
      |    NUM
```

```
5 - 4 - 3
```

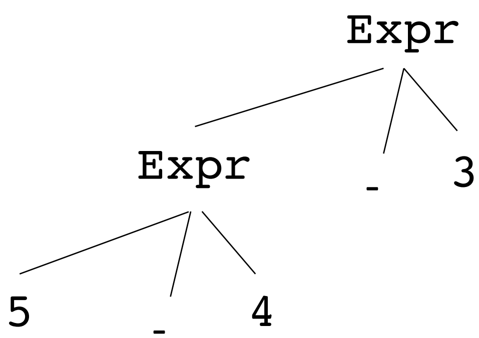


We can use this grammar in a top-down parser, but it doesn't encode associativity

Simple grammar for minus expressions

```
Expr ::= Expr MINUS NUM
       |   NUM
```

5 - 4 - 3



Left recursive grammar makes this parse tree. It encodes associativity.

But left recursion won’t work for top-down parsers!

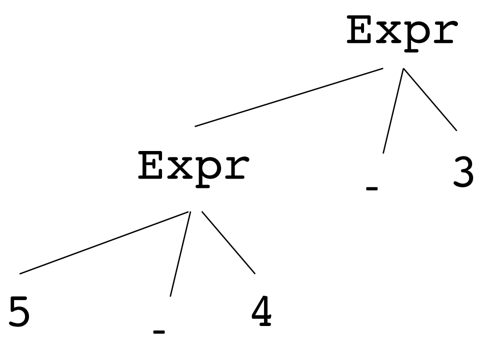
What if we follow the recipe

```
Expr  ::= NUM Expr2
Expr2 ::= MINUS NUM Expr2
        |   “”
```

Simple grammar for minus expressions

```
Expr ::= Expr MINUS NUM
      |    NUM
```

5 – 4 – 3

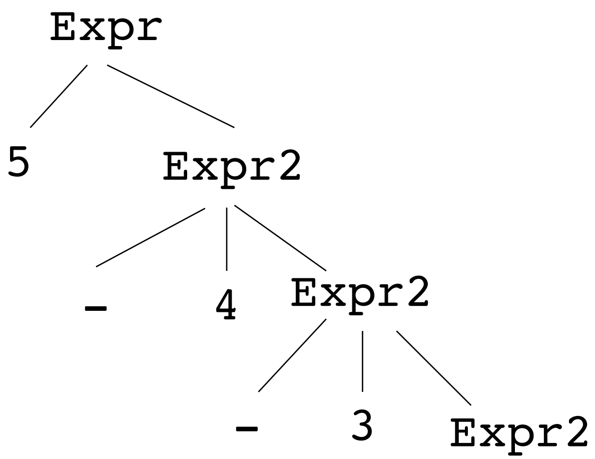


Left recursive grammar makes this parse tree. It encodes associativity.

But left recursion won't work for top-down parsers!

What if we follow the recipe

```
Expr   ::= NUM Expr2
Expr2  ::= MINUS NUM Expr2
        |    “”
```

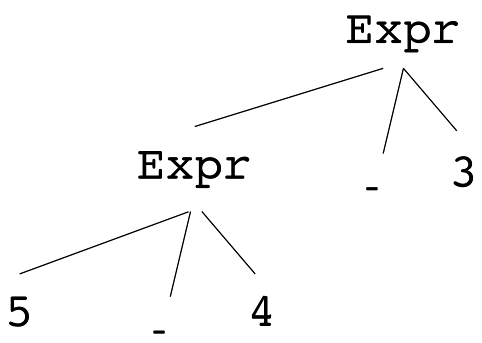


How about this one?

Simple grammar for minus expressions

```
Expr ::= Expr MINUS NUM
       |    NUM
```

5 - 4 - 3

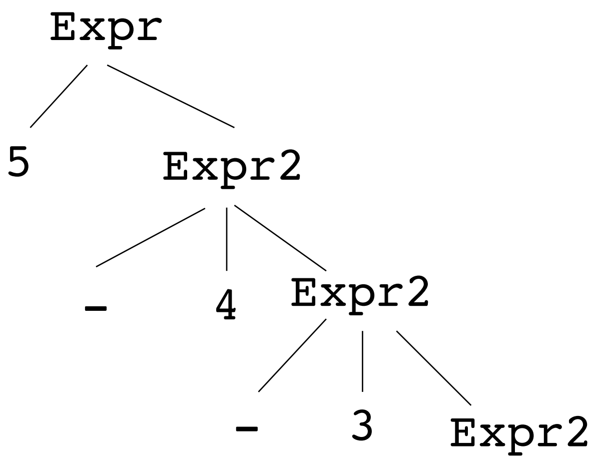


Left recursive grammar makes this parse tree. It encodes associativity.

But left recursion won't work for top-down parsers!

What if we follow the recipe

```
Expr  ::= NUM Expr2
Expr2 ::= MINUS NUM Expr2
        |    “”
```



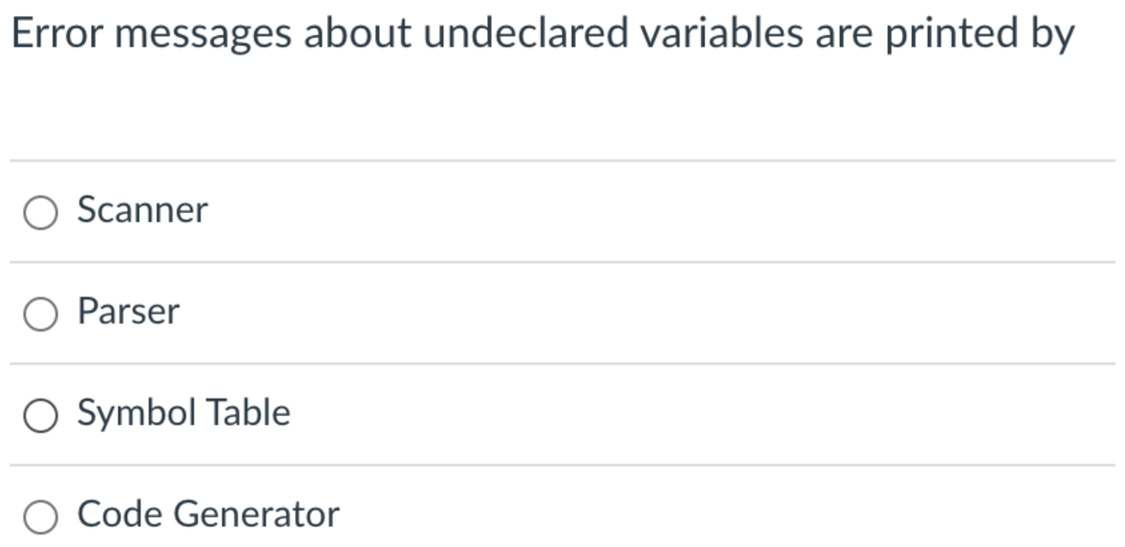
How about this one?

It isn't really clear...

We will talk about it more in the next module; you should encode associativity in your original grammar (1.1) and use the recipe for eliminating left recursion for the rest.

Quiz

Quiz



Error messages about undeclared variables are printed by

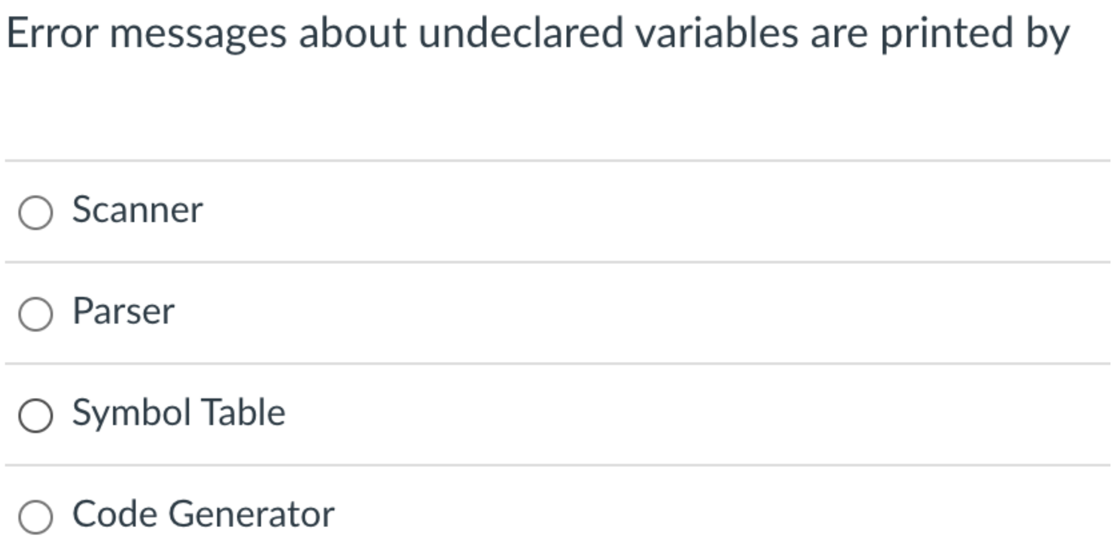
○ Scanner

○ Parser

○ Symbol Table

○ Code Generator

Quiz



Error messages about undeclared variables are printed by

○ Scanner

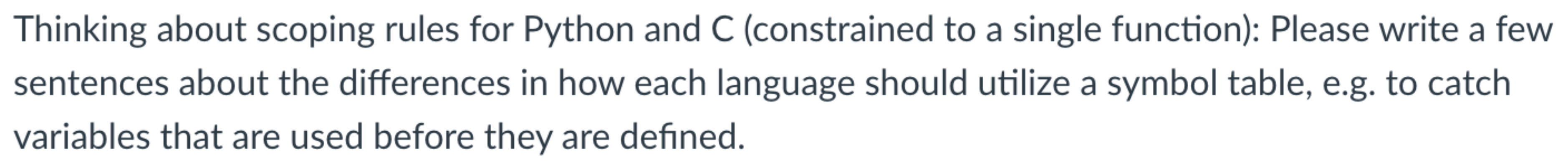
○ Parser

○ Symbol Table

○ Code Generator

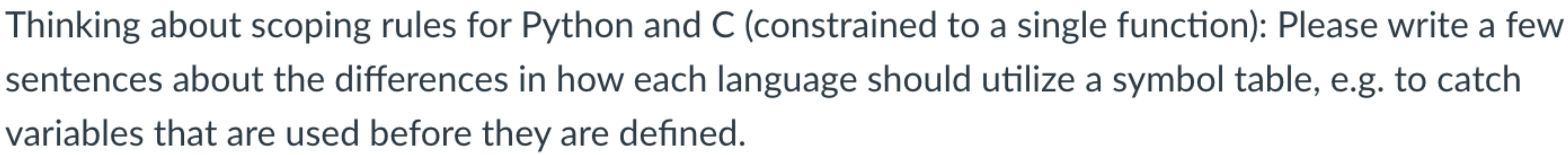
```
int x;
{
  int y;
  x++;
  y++;
}
y++;
```

Quiz



Thinking about scoping rules for Python and C (constrained to a single function): Please write a few sentences about the differences in how each language should utilize a symbol table, e.g. to catch variables that are used before they are defined.

Quiz



Thinking about scoping rules for Python and C (constrained to a single function): Please write a few sentences about the differences in how each language should utilize a symbol table, e.g. to catch variables that are used before they are defined.

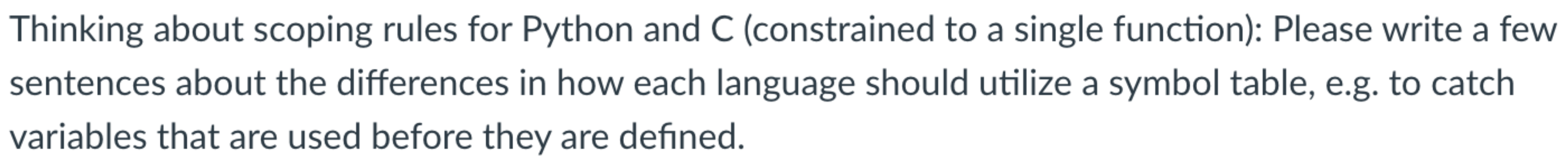
```
if (1):
    x = 5
print(x)
```

is this allowed?

```
int main() {
  if (1) {
    int x = 5;
  }
  printf("%d\n",x);
}
```

is this allowed?

Quiz



Thinking about scoping rules for Python and C (constrained to a single function): Please write a few sentences about the differences in how each language should utilize a symbol table, e.g. to catch variables that are used before they are defined.

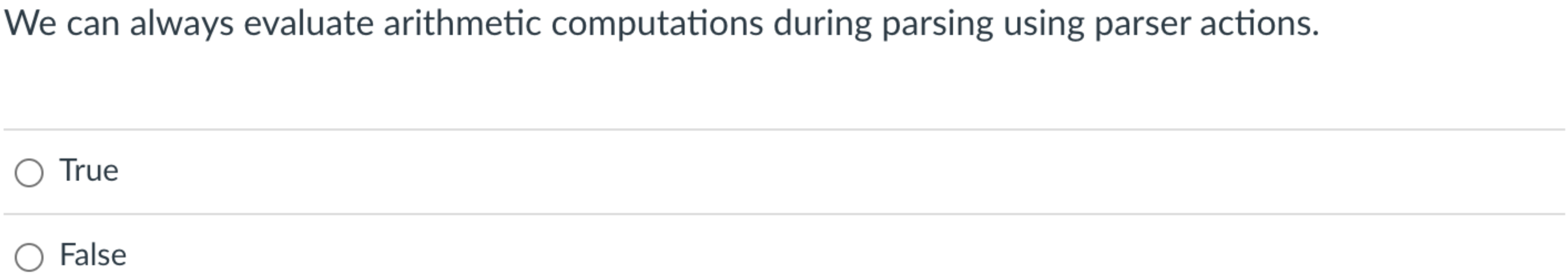
```
if (1):
    x = 5
print(x)
```

is this allowed? yes

```
int main() {
  if (1) {
    int x = 5;
  }
  printf("%d\n",x);
}
```

is this allowed? no

Quiz

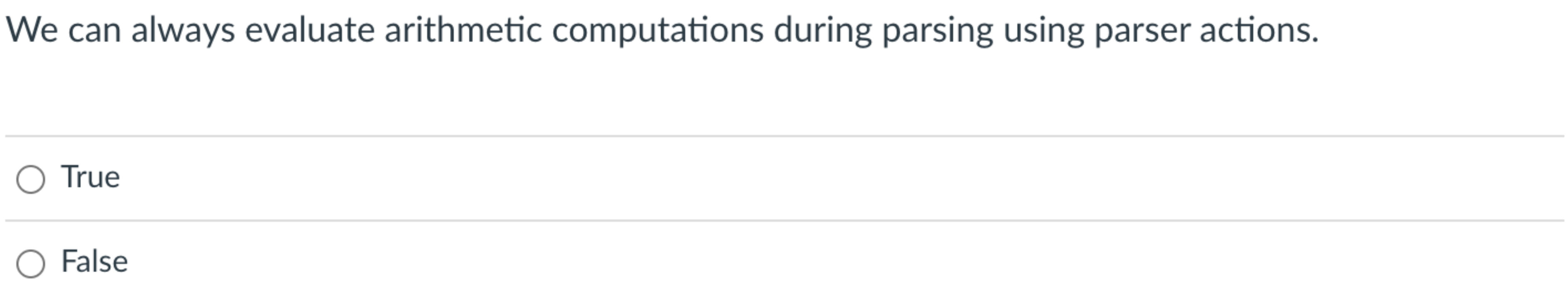


We can always evaluate arithmetic computations during parsing using parser actions.

○ True

○ False

Quiz



We can always evaluate arithmetic computations during parsing using parser actions.

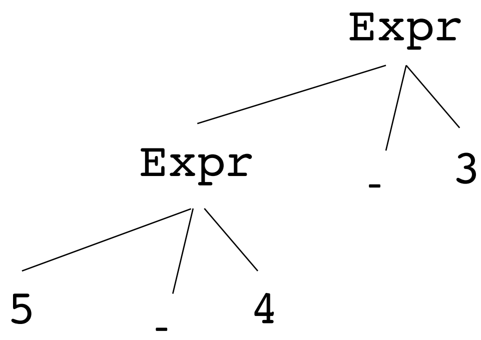
○ True

○ False

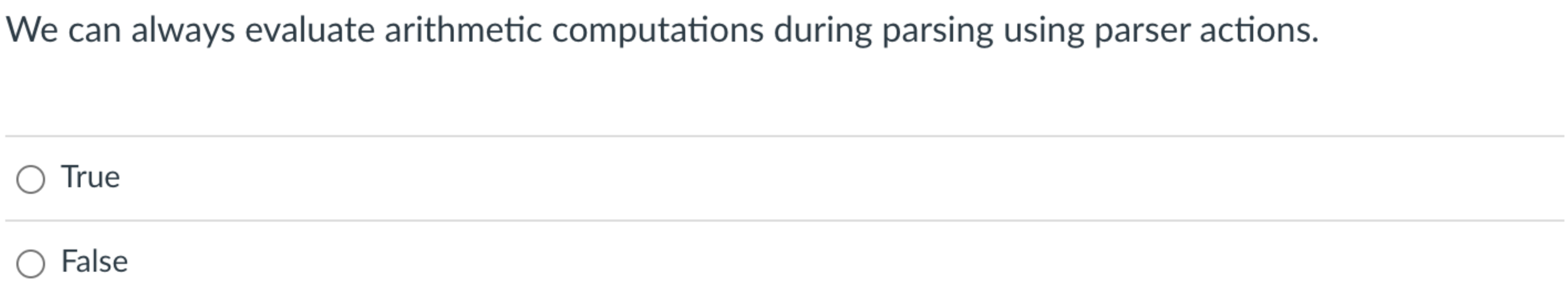
Simple grammar for minus expressions

```
Expr ::= Expr MINUS NUM
      |    NUM
```

5 - 4 - 3



Quiz

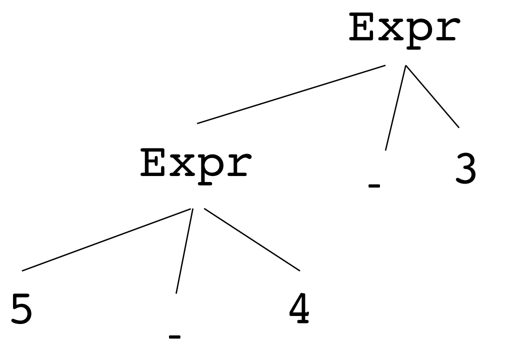


We can always evaluate arithmetic computations during parsing using parser actions.

○ True

○ False

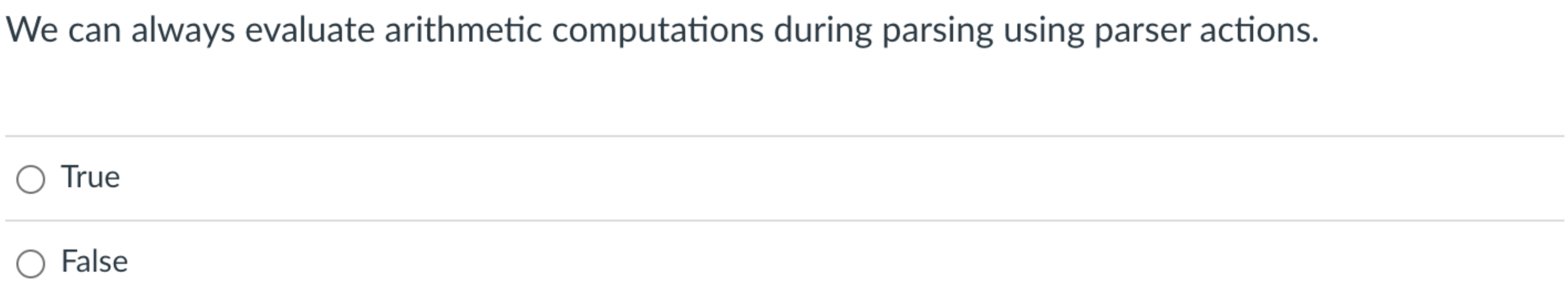
5 - 4 - 3



Simple grammar for minus expressions

```
Expr ::= Expr MINUS NUM {return $1 - $3}
      |    NUM              {return $1}
```

Quiz

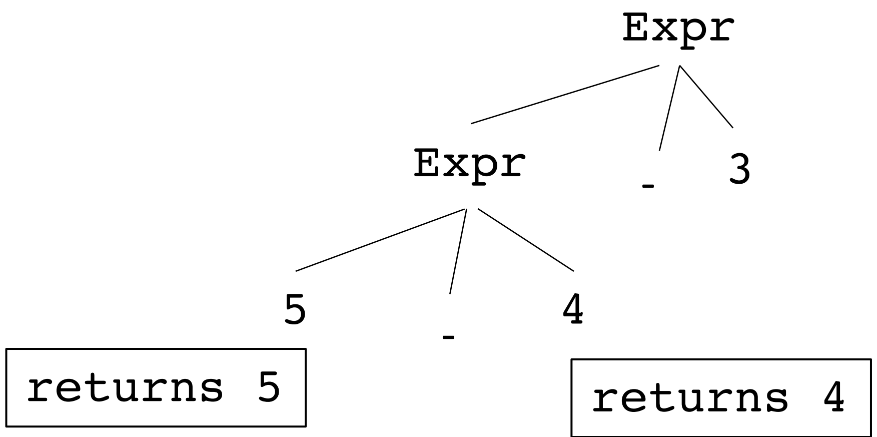


We can always evaluate arithmetic computations during parsing using parser actions.

○ True

○ False

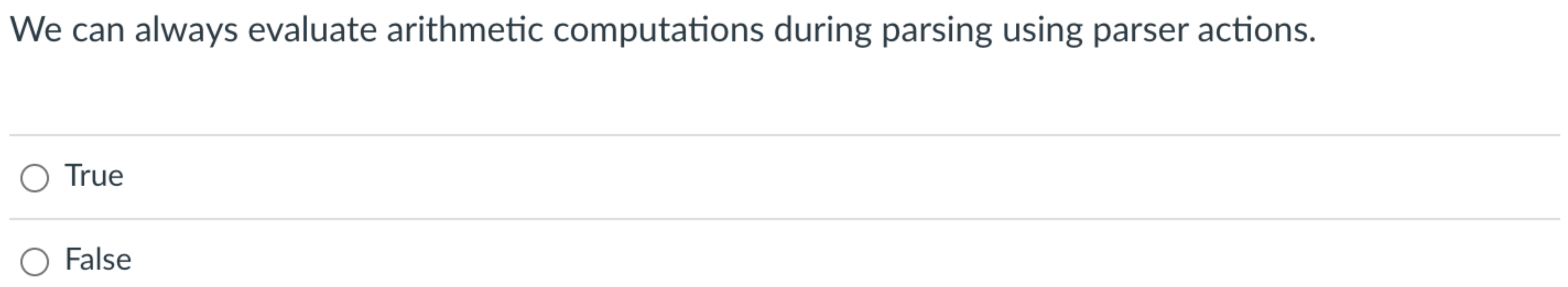
5 - 4 - 3



Simple grammar for minus expressions

```
Expr ::= Expr MINUS NUM {return $1 - $3}
      |   NUM               {return $1}
```

Quiz



We can always evaluate arithmetic computations during parsing using parser actions.

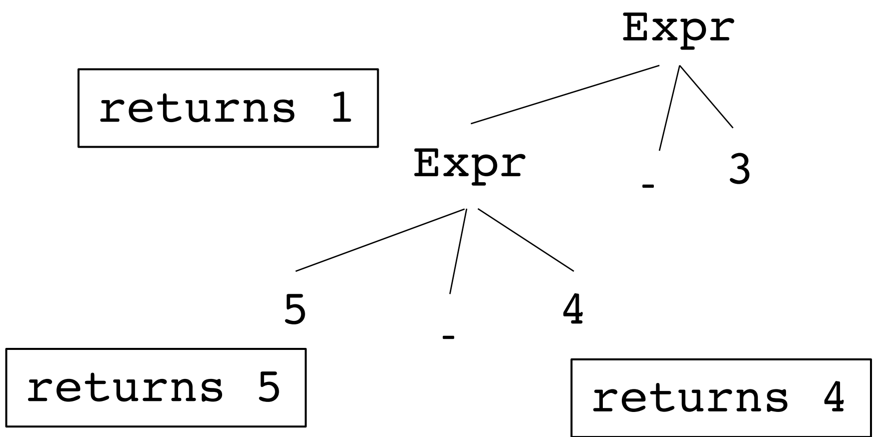
○ True

○ False

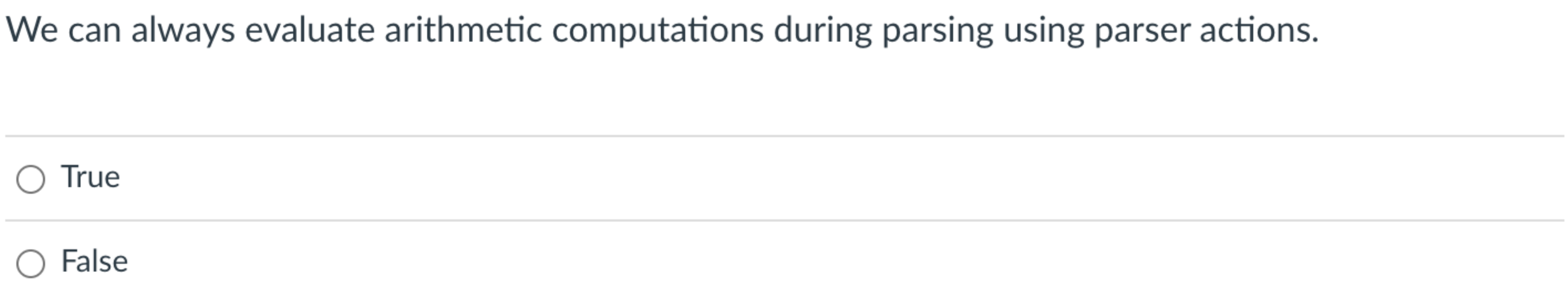
Simple grammar for minus expressions

```
Expr ::= Expr MINUS NUM {return $1 - $3}
      |    NUM             {return $1}
```

5 - 4 - 3



Quiz



We can always evaluate arithmetic computations during parsing using parser actions.

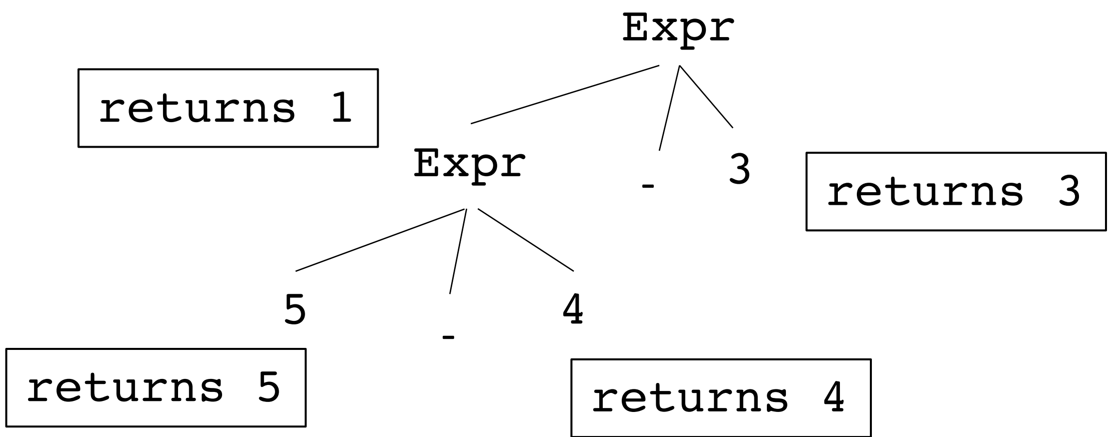
○ True

○ False

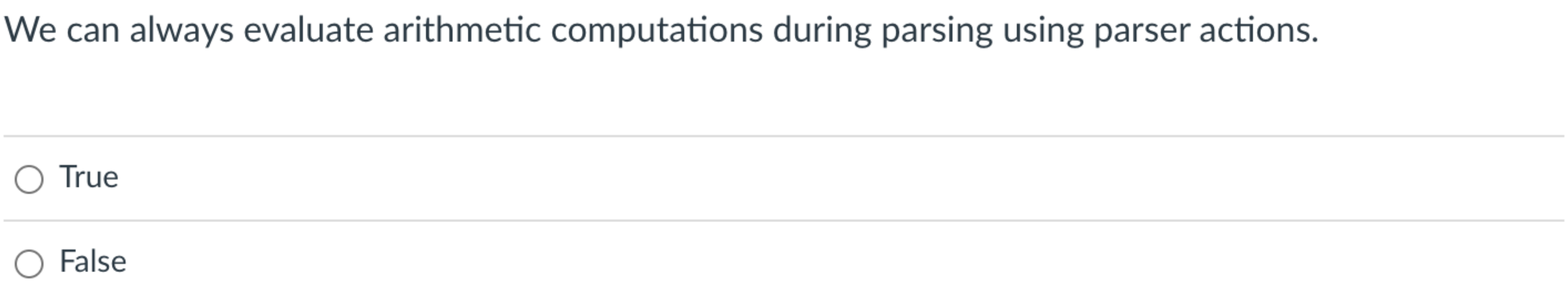
Simple grammar for minus expressions

```
Expr ::= Expr MINUS NUM {return $1 - $3}
      |   NUM             {return $1}
```

5 - 4 - 3



Quiz



We can always evaluate arithmetic computations during parsing using parser actions.

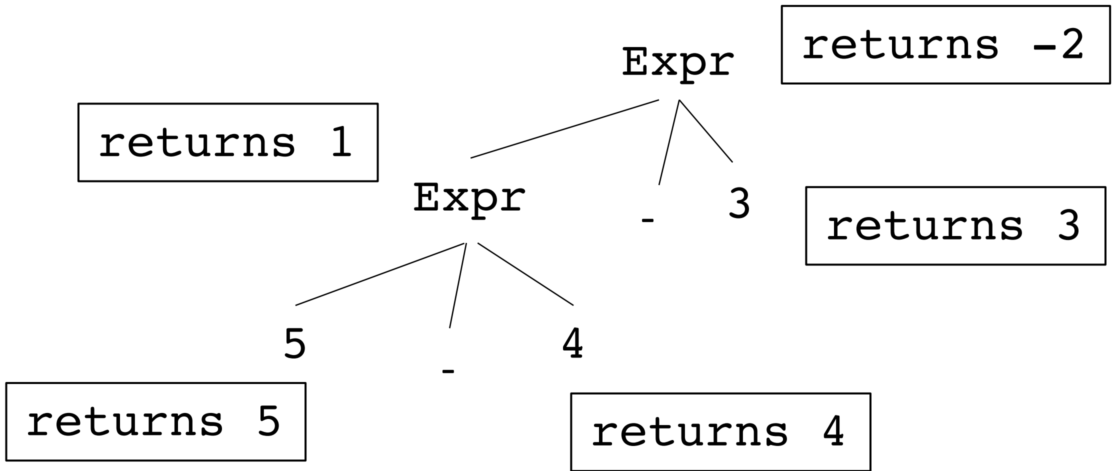
○ True

○ False

Simple grammar for minus expressions

```
Expr ::= Expr MINUS NUM {return $1 - $3}
      |    NUM              {return $1}
```

5 - 4 - 3



returns -2

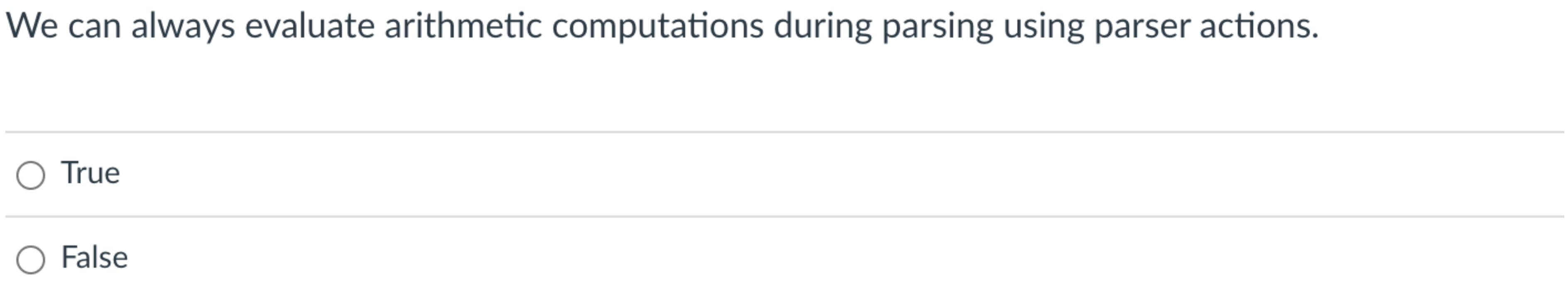
returns 1

returns 3

returns 5

returns 4

Quiz



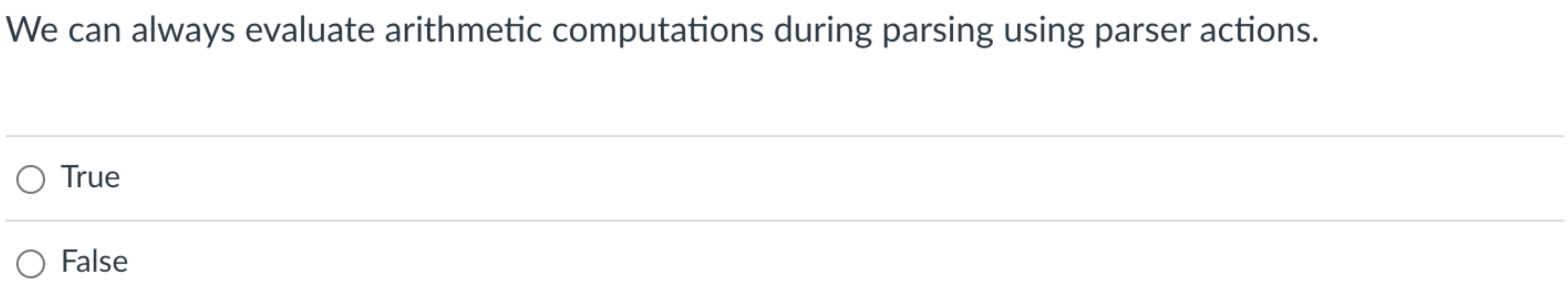
We can always evaluate arithmetic computations during parsing using parser actions.

○ True

○ False

So why can’t we always evaluate arithmetic expressions during parsing?

Quiz



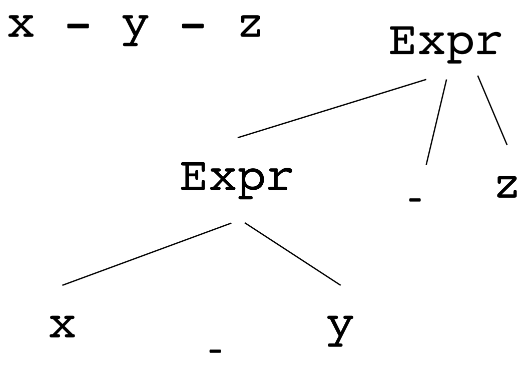
We can always evaluate arithmetic computations during parsing using parser actions.

○ True

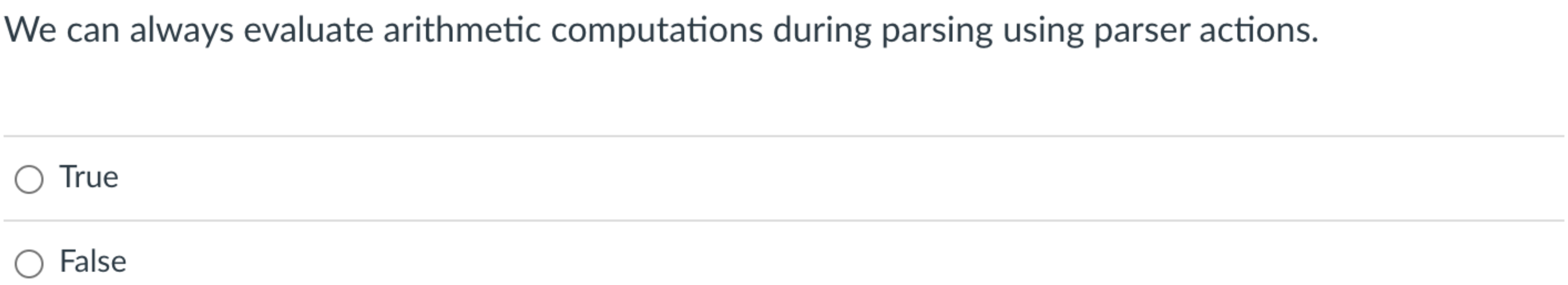
○ False

So why can't we always evaluate arithmetic expressions during parsing?

```
Expr ::= Expr MINUS UNIT {return $1 - $3}
      |   UNIT           {return $1}
UNIT ::= NUM             {return $1}
      |   ID             {return $1}
```



Quiz



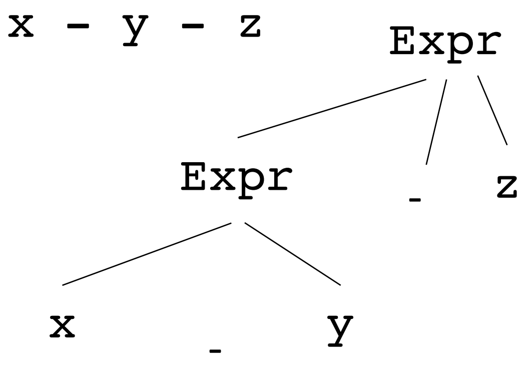
We can always evaluate arithmetic computations during parsing using parser actions.

○ True

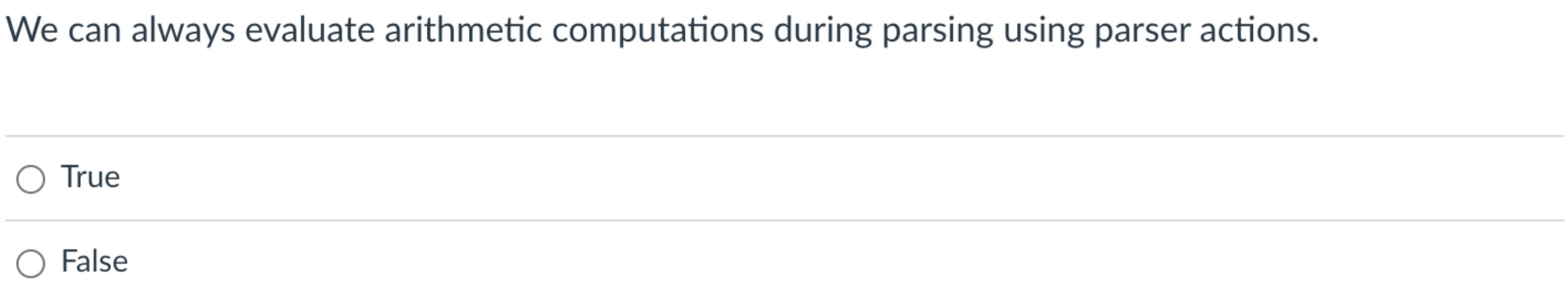
○ False

We cannot evaluate the program unless we know the value of x,y,z. What are some examples when we wouldn't know the values?

```
Expr ::= Expr MINUS UNIT {return $1 - $3}
      |   UNIT            {return $1}
UNIT ::= NUM              {return $1}
      |   ID              {return $1}
```



Quiz



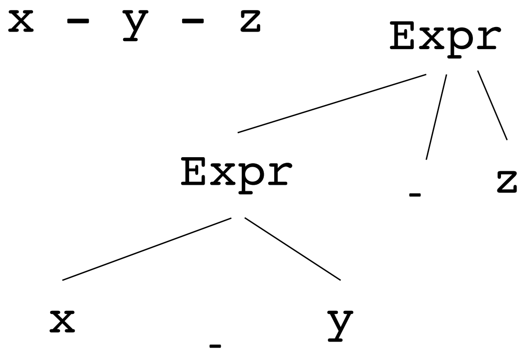
We can always evaluate arithmetic computations during parsing using parser actions.

○ True

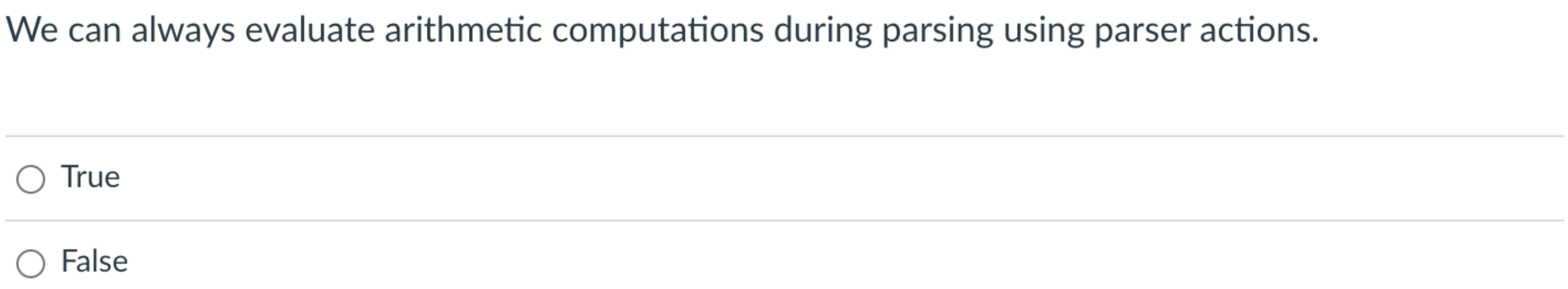
○ False

But we might be able to do some optimizations...

```
Expr ::= Expr MINUS UNIT {return $1 - $3}
      |   UNIT            {return $1}
UNIT ::= NUM             {return $1}
      |   ID              {return $1}
```



Quiz



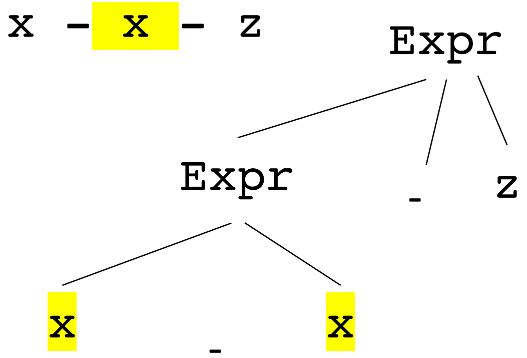
We can always evaluate arithmetic computations during parsing using parser actions.

○ True

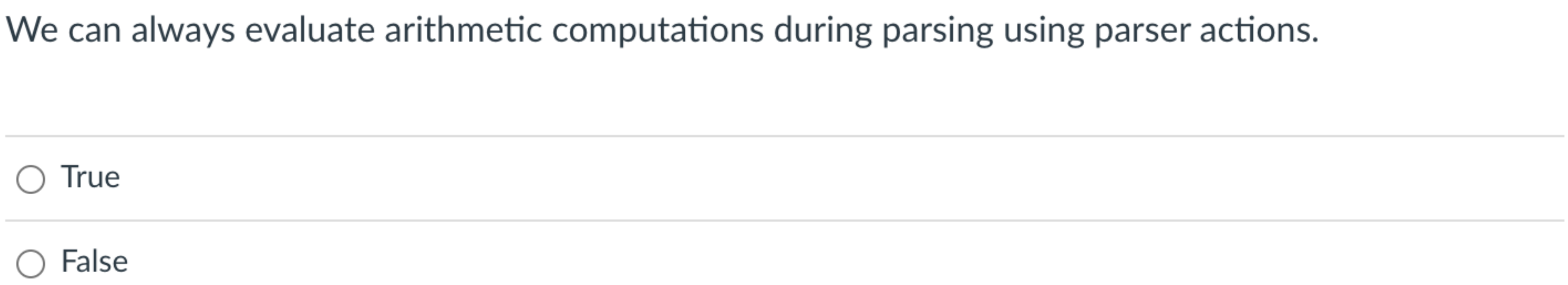
○ False

But we might be able to do some optimizations…

```
Expr ::= Expr MINUS UNIT {return $1 - $3}
      |  UNIT            {return $1}
UNIT ::= NUM             {return $1}
      |  ID              {return $1}
```



Quiz



We can always evaluate arithmetic computations during parsing using parser actions.

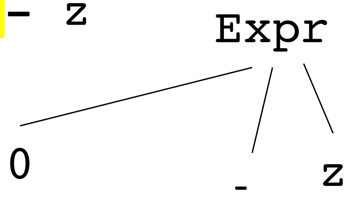
○ True

○ False

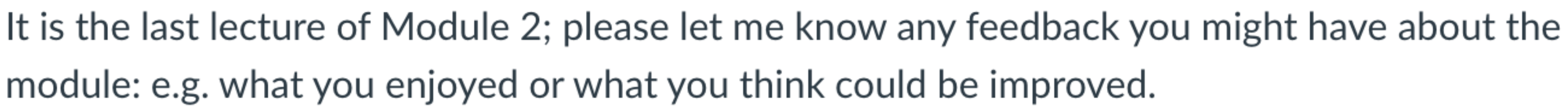
But we might be able to do some optimizations...

```
Expr ::= Expr MINUS UNIT {if $1 == $3 then 0 else ...}
      |   UNIT           {return $1}
UNIT ::= NUM             {return $1}
      |   ID             {return $1}
```

x - x - z



Quiz



It is the last lecture of Module 2; please let me know any feedback you might have about the module: e.g. what you enjoyed or what you think could be improved.

Parser generators

calculator example

These slides follow the calculator example from the PLY documentation

calculator example

```
import ply.lex as lex

tokens = ["NUM", "MULT", "PLUS", "MINUS", "DIV", "LPAR", "RPAR"]

t_NUM = '[0-9]+'
t_MULT = '\*'
t_PLUS = '\+'
t_MINUS = '-'
t_DIV = '/'
t_LPAR = '\('
t_RPAR = ‘\)'

t_ignore = ' '

# Error handling rule
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    exit(1)

lexer = lex.lex()
```

Set up the lexer

calculator example

- *Import the library*

```
import ply.yacc as yacc
```

- Simple rule

```
def p_expr_num(p):
    "expr : NUM"
    p[0] = int(p[1])
```

functions are given prefixed by `p_`

production rules are the doc string

return values are stored in p[0]
children values are in p[1], p[2], etc.

calculator example

- *Try it out*

calculator example

- *Next rule*

```
def p_expr_plus(p):
    "expr : expr PLUS expr"
    p[0] = p[1] + p[3]
```

- Try it again

calculator example

- Set associativity (and precedence)

```
precedence = (
    ('left', 'PLUS'),
)
```

calculator example

- *Next rules*

```
def p_expr_minus(p):
    "expr : expr MINUS expr"
    p[0] = p[1] - p[3]


def p_expr_mult(p):
    "expr : expr MULT expr"
    p[0] = p[1] * p[3]


def p_expr_div(p):
    "expr : expr DIV expr"
    p[0] = p[1] / p[3]
```

```
precedence = [
    ('left', 'PLUS', 'MINUS'),
    ('left', 'MULT', 'DIV'),
]
```

calculator example

- *Last rule for expressions*

```
def p_expr_par(p):
    "expr : LPAREN expr RPAREN"
    p[0] = p[2]
```

calculator example

- *An extra we can easily implement*

```
def p_expr_div(p):
    "expr : expr DIV expr"
    if p[3] == 0:
        print("divide by 0 error:")
        print("cannot divide: " + str(p[1]) + " by 0")
        exit(1)
    p[0] = p[1] / p[3]
```

calculator example

- *Combining rules:*

```
def p_expr_plus(p):
    "expr : expr PLUS expr"
    p[0] = p[1] + p[3]


def p_expr_minus(p):
    "expr : expr MINUS expr"
    p[0] = p[1] - p[3]


def p_expr_mult(p):
    "expr : expr MULT expr"
    p[0] = p[1] * p[3]
```

```
def p_expr_bin(p):
    """
    expr : expr PLUS expr
         | expr MINUS expr
         | expr MULT expr
    """
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    else:
        assert(False)
```

calculator example

- *Even simpler implementation using functions as token values*

calculator example

- Other useful options
  - Error recovery?
  - Error reporting (it is better in our top down parsers because we can say which token we were looking for)

calculator example

- Recovering from errors
    - Be careful! Only do this if your users expect it!

```
def p_error(p):
    if p:
        print("Syntax error at token, ignoring and moving on", p.type)
        # Just discard the token and tell the parser it's okay.
        parser.errok()
    else:
        print("Syntax error at EOF")
```

See you on Friday!

- Finish HW 2

- Starting the next module: intermediate representations