Part 4

Parsing

The Parsing Problem

- Recognize syntactically correct input

```
b = 40 + 20*(2+3)        # YES!
c = 40 + * 20            # NO!
d = 40 + + 20            # ???
```

- Need to transform this input into the structural representation of the program
- Tokens -> Data model (AST)

Disclaimer

- Parsing theory is a huge topic
- It's often what comes to mind when people think of writing a compiler ("oh, I must figure out how to parse this input.")
- Often heavily focused on tools
- Parsing is only a small part of the big picture

Our Focus

- Understanding how to specify syntax
- Develop an intuition for how parsing works
- Write our own parser (by hand)

Syntax Specification

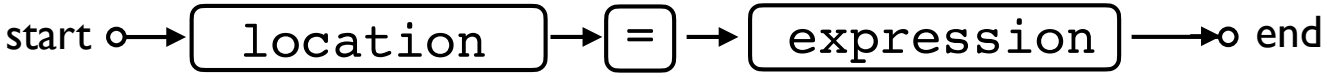
- How do you describe syntax?
- Example: Describe Python "assignment"

```
a = 0
b = 2 + 3
c.name = 2 + 3 * 4
d[1] = (2 + 3) * 4
```

- By "describe"--a precise specification
- By "precise"--something that can be coded

Syntax Diagram

assignment:



- You follow the arrows
- Read it out loud...*assignment is a location followed, an '=', followed by an expression*
- A bit vague perhaps, but technically true
- It can be turned into code...

Converting to Code

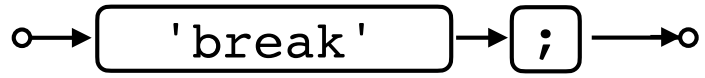
assignment:


```
def parse_assignment(tokens):
    loc = parse_location(tokens)
    tokens.expect('=')
    expr = parse_expression(tokens)
    return Assignment(loc, expr)
```

- Code follows the flow of the diagram
- Left-to-right
- Result is an element of the model

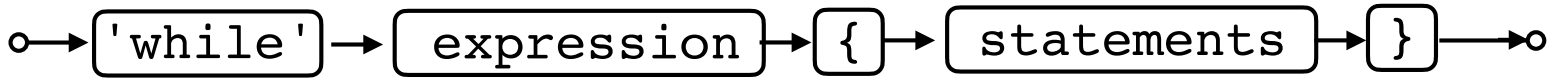
More Examples:

break




```
def parse_break(tokens):
    tokens.expect('BREAK')
    tokens.expect(';')
    return BreakStatement()
```

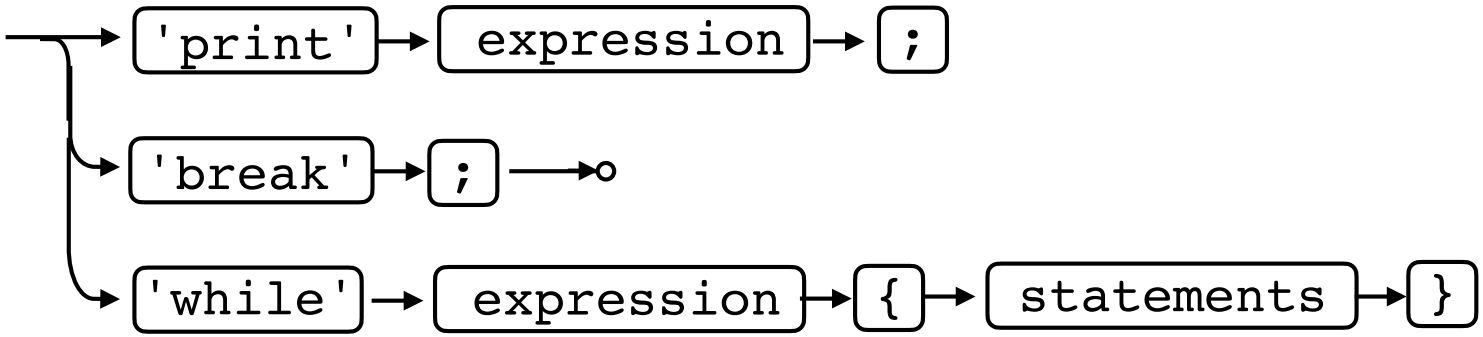
while




```
def parse_while(tokens):
    tokens.expect('WHILE')
    expr = parse_expression(tokens)
    tokens.expect('{')
    statements = parse_statements(tokens)
    tokens.expect('}')
    return WhileStatement(expr, statements)
```

Branching

statement:



- Multiple choices involve prediction/lookahead

```
def parse_statement(tokens):
    if tokens.peek('PRINT'):
        return parse_print_statement(tokens)
    elif tokens.peek('BREAK'):
        return parse_break_statement(tokens)
    elif tokens.peek('WHILE'):
        return parse_statements(tokens)
    ...
```

Project

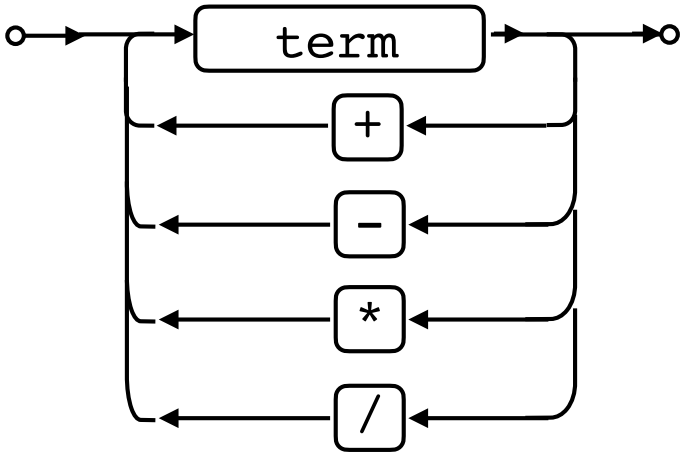
Find the file `wabbit/parse.py`

Follow instructions inside.

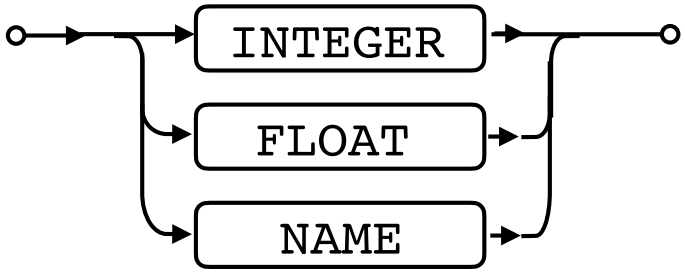
Goal: Build program models from source

Expression Parsing

expression:



term:



examples:

```
term
term + term
term + term * term
```

examples:

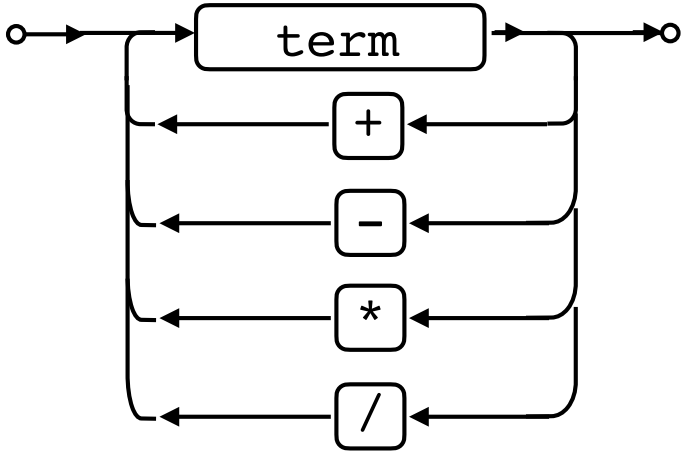
```
3
3.4
x
```

Expression Parsing

code:

```
def parse_expression(tokens):
    term = parse_term(tokens)
    while True:
        op = tokens.accept('+','-','*','/')
        if not op:
            return term
        term = BinOp(op,
                     term,
                     parse_term(tokens))
```

expression:



- Loops in syntax diagram -> loop in the code
- Requires lookahead/optional matching

Associativity

- Operators associate left-to-right usually

```
a + b + c + d  #  (((a + b) + c) + d)
```

- This isn't directly expressed in the syntax diagram, but it comes out of the coding

```
def parse_expression(tokens):
    term = parse_term(tokens)
    while True:
        op = tokens.accept('+','-','*','/')
        if not op:
            return term
        term = BinOp(op,
                     term,
                     parse_term(tokens))
```

Left side is repeatedly replaced by a BinOp

The Horror: Precedence

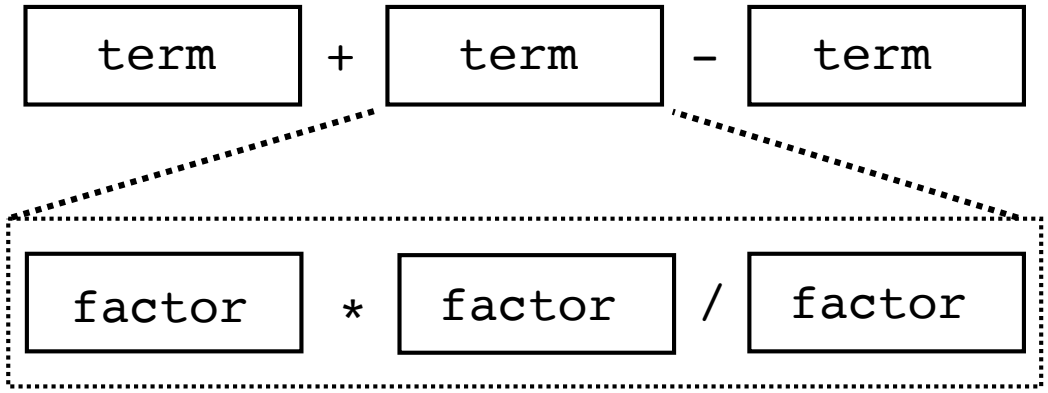
- Operator precedence in expressions

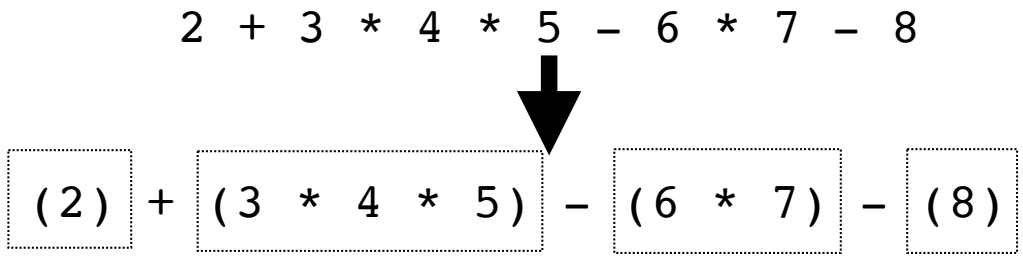
```
a + b * c       #  a + (b * c)
a * b + c       #  (a * b) + c
```

- Learned in 4th grade math class
- Implicit--you just learn that * goes first
- It's not captured in the earlier syntax diagram

Precedence Structure

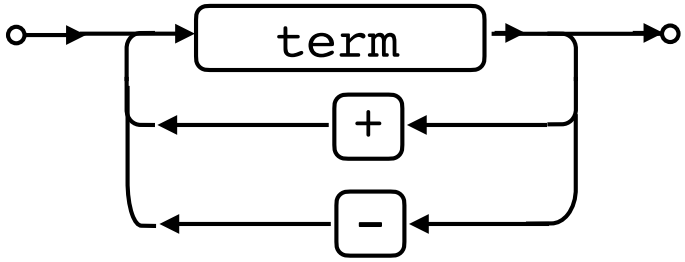
- Operator precedence is organized into levels



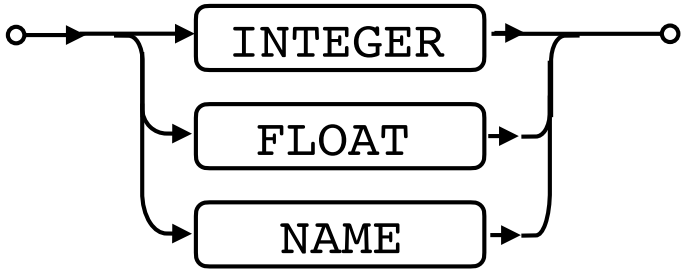


Diagrams w/ Precedence

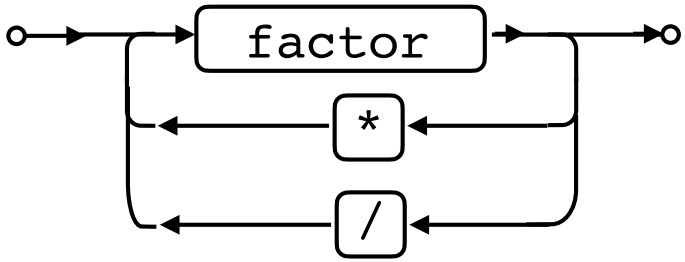
expression:



factor:



term:



Idea: Each level is described by its own diagram. Low precedence links to higher precedence.

Precedence

- There may be many more levels

```
a + b < c + d and e * f > h or i == j
```

```
(a + b < c + d and e * f > h) or (i == j)
```

```
((a + b < c + d) and (e * f > h)) or (i == j)
```

```
(((a + b) < (c + d)) and ((e * f) > h)) or (i == j)
```

- All of this can be encoded via diagrams
- Separate diagram for each level (messy)

Parser Error Handling

- Error handling in parsers is tricky

```
print 3 + 5 x + 10;
```

syntax error

- Can be reported, but then what?!?
- Common strategy: synchronization

```
print 3 + 5 x + 10;
```

syntax error sync character

- Idea: Search for sync character, throwing away all tokens until you find it. Then continue.

Project

Continued work on parser.py

Add support for expressions/operators

Symbolic Grammars

- Syntax is often expressed in symbolic form

```
assignment: location = expr

expr ::= expr + term
       |  expr - term
       |  term

term ::= term * factor
       |  term / factor
       |  factor

factor ::= INT
       |  FLOAT
       |  NAME
```

- Notation known as BNF (Backus Naur Form)

Grammar Specification

- A BNF specifies substitutions

```
assignment: location = expr

expr ::= expr + term
       |  expr - term
       |  term

term ::= term * factor
       |  term / factor
       |  factor

factor ::= INT
       |  FLOAT
       |  NAME
```

- Any name listed on left can be replaced by the symbols on the right (and vice versa).

Analogy

- It's like equational reasoning in algebra class

```
x = y + 10

z = x - 6  ---------->  z = (y + 10) - 6
           substitute x
```

- Think of a BNF as a collection of equations
- You can interchange one side with the other

Commentary

- BNF is a textual description used by tools
- Easier for tools to work with than diagrams
- Will most commonly find grammars specified as some form of BNF, etc.

EBNF

- A more compact BNF representation
- Includes repetition and optional items

```
expr = term { "+"|"-" term }
```

- Notational guide

```
a | b | c          # Alternatives
{ ... }            # Repetition (0 or more)
[ ... ]            # Optional (0 or 1)
```

EBNF Example

- Grammar as a EBNF

```
assignment = NAME '=' expr ';'
expr = term { '+'|'-' term }
term = factor { '*'|'/' factor }
factor = INTEGER | FLOAT | NAME | '(' expr ')'
```

- EBNF is a fairly common standard for grammar specification
- You see it a lot in standards documents
- Mini exercise: Look at Python grammar

Alternative: PEGs

- Parsing Expression Grammar (PEG)

```
assignment <- NAME '=' expr ';'
expr       <- term { '+'/'-' term }
term       <- factor { '*'/'/' factor }
factor     <- NUMBER / NAME / '(' expr ')'
```

- Looks somewhat similar to an EBNF
- Choice ("|") is replaced by first match ("/")

Alternative: PEGs

- Example:

```
rule <- e1 / e2 / e3
```

- Rules specifies a first-match strategy

```
1. Try to parse e1.  If success, done.
2. Else, try to parse e2. If success, done.
3. Else, try to parse e3. If success, done.
4. Else, parse error.
```

- Specification order has significance (rules listed first have higher priority)
- Implies back-tracking (to retry alternatives)

Alternative: PEGs

- PEGs are more modern
- Bryan Ford, "*Parsing Expression Grammars: A Recognition-Based Syntactic Foundation*", POPL 2004 (ACM).
- Not found in most traditional compiler books
- Have seen increased use. Python switched in 3.9.