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.")
- Parsing is only a small part of the big picture

Historical Context

- One reason why parsing has been studied so much has to do with the limited computing power of machines during 1960s-1970s.
- Didn't have the memory to store complex models of entire programs
- A lot of interest in "single-pass" translation
- So algorithms focused on that.

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
e['key'] = 0.5 * d
```

- By "describe"--a precise specification
- By "precise"--rigorous like math

Syntax Specification

- Example: Syntax for "assignment"

```
location = expression
```

- That is extremely high-level (vague)
  - What is a "location"?
  - What is an "expression"?
- Ultimately, it must (eventually) map to tokens

Grammar Specification

- Syntax often specified as a Context Free Grammar

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

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

- Notation known as BNF (Backus Naur Form)
- Specifies a collection of choices (| = "or")

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 (a token).
- "nonterminal" - A symbol that can be expanded into other symbols (grammar rules)
- Think about reductions. A nonterminal is something that can be reduced to lower-level primitives. A terminal can't be reduced further.

Grammar Specification

- A BNF specifies <u>substitutions</u>

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

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

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

Grammar Specification

- A BNF specifies <u>substitutions</u>

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

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

Can replace by any of these sequences

"|" means choice (or)

- 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

Recursive Substitution

- Substitutions are recursive

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

- Can self-expand as needed (off to infinity...)

```
expr
expr + expr
expr + expr + expr
expr + expr + expr * expr
expr + expr + expr * expr - expr
```

Problem: Ambiguity

- Consider:

```
expr                            # Expand
expr + expr                     # Expand
expr + expr + expr              # Expand (which one?)
```

- Was it the left expression?

```
expr + expr    --->  (expr + expr) + expr
```

- Or the right expression?

```
expr + expr    --->  expr + (expr + expr)
```

- Why you might care: the order in which you do things affects the final structure/result

Associativity

- There are "order of evaluation" rules from math

```
1 + 2 + 3 + 4 + 5
```

- Left associativity (left-to-right)

```
(((1 + 2) + 3) + 4) + 5
```

- Right associativity (right-to-left)

```
1 + (2 + (3 + (4 + 5)))
```

- Does it matter? Yes.

Associativity

- You might get different answers for some ops

```
1 - 3 - 4
```

- Left associativity (left-to-right)

```
(1 - 3) - 4  --> -6
```

- Right associativity (right-to-left)

```
1 - (3 - 4)  --> 2
```

- Q: Can order be expressed in a grammar?

Associativity

- Expression grammar with left associativity

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

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

- Idea: The recursive expansion of expressions is only allowed on the left-hand side.

Problem: Precedence

- Consider:

```
1 + 2 * 3 + 4
```

- Is this to be expanded as follows?

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

- No, assuming the rules of math class
- It should be this (order of evaluation)

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

- Q: Can this also be encoded in the grammar?

Precedence

- Expression grammar with precedence levels

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

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

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

- Idea: Layering from low->high precedence

Precedence

- Expression grammar with precedence levels

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

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

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

$\boxed{\texttt{term}} + \boxed{\texttt{term}} - \boxed{\texttt{term}}$

- Idea: Layering from low->high precedence

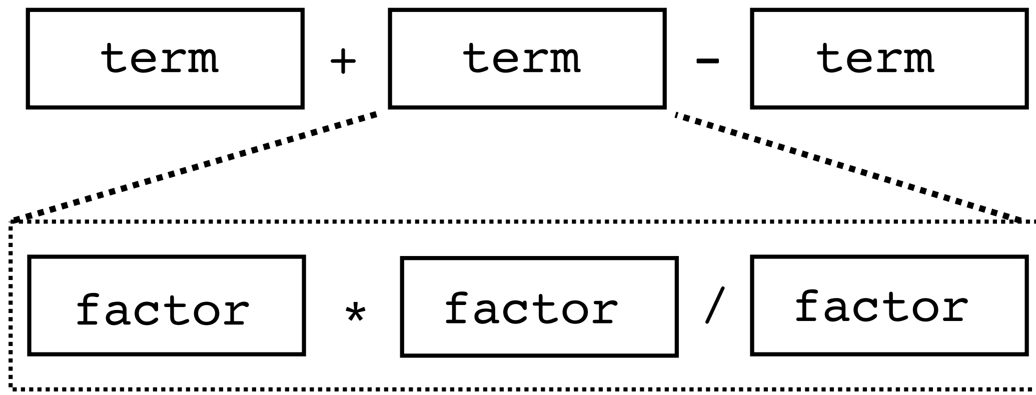
Precedence

- Expression grammar with precedence levels

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

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

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



- Idea: Layering from low->high precedence

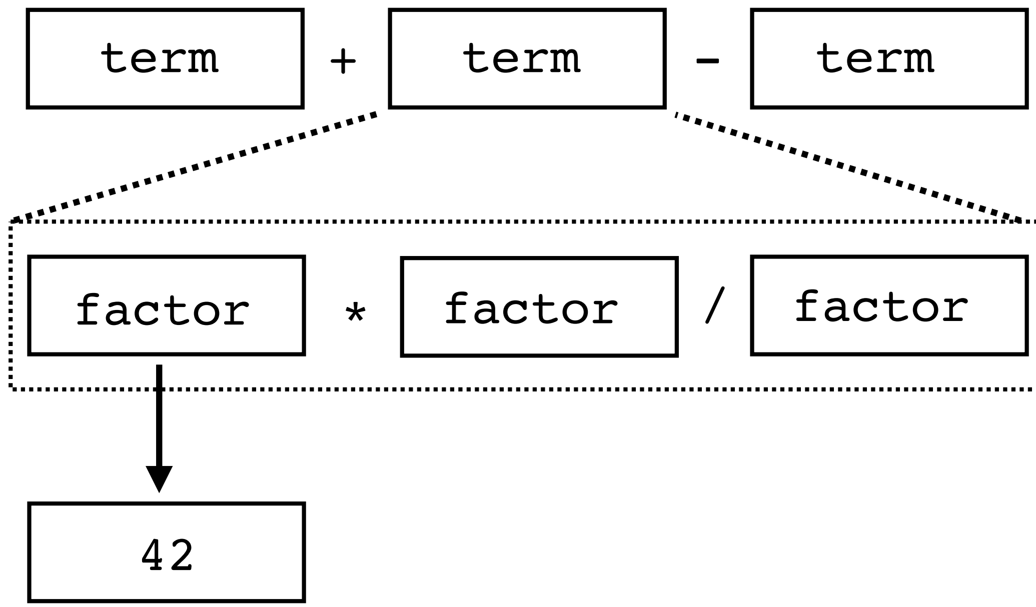
Precedence

- Expression grammar with precedence levels

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

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

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



- Idea: Layering from low->high 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 in the grammar
- Need a separate rule for each precedence layer

Notational Simplification

- What is *actually* being expressed by this rule?

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

- Repetition (of terms).
- Alternative notation: EBNF

```
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 ("/")
- First-match eliminates ambiguity

Alternative: PEGs

- Example:

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

- Rules specifies a first-match strategy

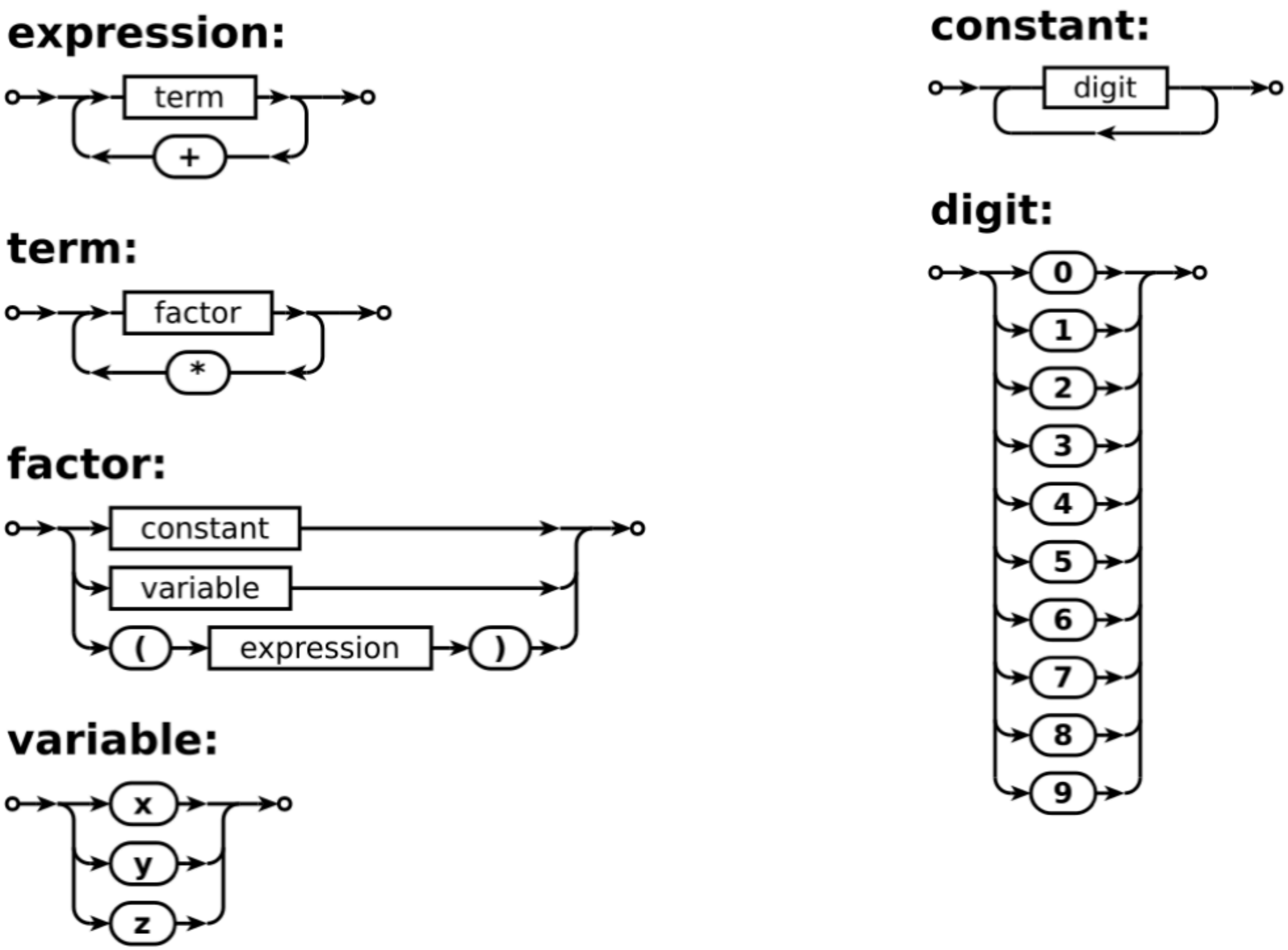
```
1. Try to parse e1.  If success, done.
2. Else, try to parse e2. If success, done.
3. Else, try to parse e3. If success, done.
4. Else, parse error.
```

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

Alternative: PEGs

- PEGs are much 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.

Syntax Diagrams



Parsing Algorithms

- So far, mainly focused on specification
- How does it translate into an algorithm?
- This is a big topic: However, most parsing algorithms are based on some core ideas
- Will illustrate at a conceptual level

Parsing Explained

- Problem: match input text against a grammar

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

- Example: Does it match the assignment rule?

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

- How would you determine a match?

Parsing Algorithms

"Why did the parser cross the road?"

Parsing Algorithms

"Why did the parser cross the road?"

"To get to the other side."

- This a surprisingly accurate description of parsing ("getting to the other side").
- Let's elaborate further...

Parsing Algorithms

- In the beginning, you know nothing...

Grammar:

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

Tokens:

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

Parsing Algorithms

- In the beginning, you know nothing...

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

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

- The goal: move <u>both</u> markers to the other side

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

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

- Think of it as a game

Parsing Algorithms

- In the beginning, you know nothing...

Grammar:
```
assignment : NAME '=' expr ';'
```
Tokens:
```
a = 2 * 3 + 4;
```

- The goal: move both markers to the other side

Grammar:
```
assignment : NAME '=' expr ';'
```
Tokens:
```
a = 2 * 3 + 4;
```

- In this game, you can only eat tokens

Parsing Algorithms

- In the beginning, you know nothing...

<u>Grammar:</u>

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

<u>Tokens:</u>

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

- The goal: move <u>both</u> markers to the other side

<u>Grammar:</u>

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

<u>Tokens:</u>

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

- Or expand grammar rules

Parsing Illustrated

- Parsing involves stepping through tokens

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

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

- You try to match to the grammar as you go

Parsing Illustrated

- Parsing involves stepping through tokens

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

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

- You try to match to the grammar as you go

- Forward progress if there is a token match

Parsing Illustrated

- Parsing involves stepping through tokens

Grammar:

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

Tokens:

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

- You try to match to the grammar as you go

- Forward progress if there is a token match

Parsing Illustrated

- Parsing involves stepping through tokens

Grammar: `expr : term { '+'|'-' term }`

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

- You try to match to the grammar as you go
- Matching descends into grammar rules

Parsing Illustrated

- Parsing involves stepping through tokens

Grammar:
```
term : factor { '*'|'/' factor }
```

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

- You try to match to the grammar as you go

- Matching descends into grammar rules

Parsing Illustrated

- Parsing involves stepping through tokens

Grammar: `factor : INTEGER | FLOAT`

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

- You try to match to the grammar as you go
- Matching descends into grammar rules

Parsing Illustrated

- Parsing involves stepping through tokens

Grammar: `factor : INTEGER | FLOAT`

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

- You try to match to the grammar as you go
- Matching descends into grammar rules
- Can only make forward progress on tokens

Parsing Illustrated

- Parsing involves stepping through tokens

Grammar: `term : factor { '*'|'/' factor }`

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

- You try to match to the grammar as you go
- Matching descends into grammar rules
- Can only make forward progress on tokens

Parsing Illustrated

- Parsing involves stepping through tokens

Grammar: `term : factor { '*'|'/' factor }`

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

- You try to match to the grammar as you go
- Matching descends into grammar rules
- Can only make forward progress on tokens

Parsing Illustrated

- Parsing involves stepping through tokens

Grammar: `term : factor { '*'|'/' factor }`

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

- You try to match to the grammar as you go
- Matching descends into grammar rules
- Can only make forward progress on tokens

Parsing Illustrated

- Parsing involves stepping through tokens

Grammar: `expr : term { '+'|'-' term }`

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

- You try to match to the grammar as you go
- Matching descends into grammar rules
- Can only make forward progress on tokens

Parsing Illustrated

- Parsing involves stepping through tokens

Grammar: `expr : term { '+'|'-' term }`

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

- You try to match to the grammar as you go
- Matching descends into grammar rules
- Can only make forward progress on tokens

Parsing Illustrated

- Parsing involves stepping through tokens

Grammar: `expr : term { '+'|'-' term }`

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

- You try to match to the grammar as you go
- Matching descends into grammar rules
- Can only make forward progress on tokens

Parsing Illustrated

- Parsing involves stepping through tokens

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

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

- You try to match to the grammar as you go
- Matching descends into grammar rules
- Can only make forward progress on tokens

Parsing Illustrated

- Parsing involves stepping through tokens

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

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

- You try to match to the grammar as you go
- Matching descends into grammar rules
- Can only make forward progress on tokens
- You made it! A successful parse.

Algorithms

- There are MANY different parsing algorithms and strategies, with varying degrees of power and implementation difficulty
- Usually given cryptic names
  - LL(1), LL(k)
  - LR(1), LALR(1), GLR
- Honestly, details aren't that important here

Parsing Strategies

- Top Down: Work with the grammar rules. Make forward progress by looking at what tokens you expect (according to the rules).
- Bottom Up: Work with the tokens. Make progress by matching the tokens seen so far with the grammar rules that they might match.

Writing a Parser

- It is not too hard to write one by hand
- Common algorithm: Recursive Descent

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

Rules become functions

(match left-to-right)

Create object (from model)

```
def parse_assignment():
    name = expect('NAME')
    expect('=')
    expr = parse_expr()
    expect(';')
    return Assignment(name, expr)

def parse_expr():
    ...
```

descend

Project

Find the file `wabbit/parse.py`

Follow instructions inside.

Goal: Build program models from source