Part 3

Lexing

The Input Problem

- At some point we need to read source code

```
/* Print the first ten factorials */
var n int = 1;
var value int = 1;

while n < 10 {
    value = value * n;
    print value ;
    n = n + 1;
}
```

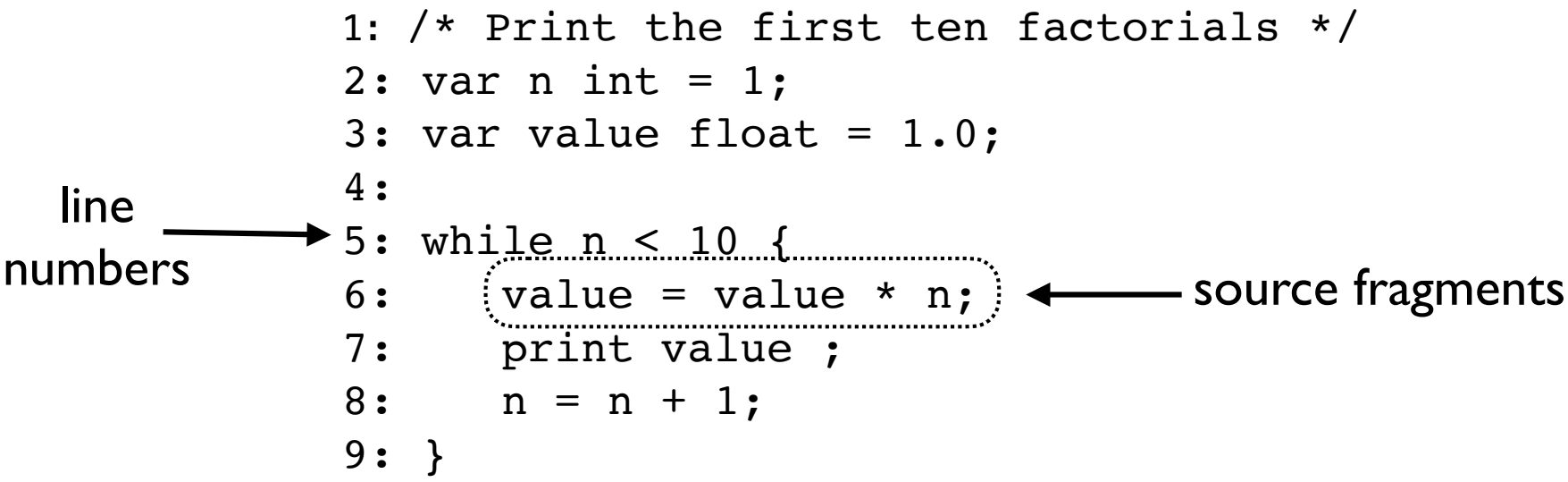
- This is the "parsing" problem.
- Goal: source → model

The "Source" Problem

- Books on compilers often ignore the issue of the source code itself
- Source code is "just text"
- There's NOTHING interesting about text
- Yawn!
- This is wrong!

Source Code

- Source code is what programmers work with!



- Source code is basis of error reporting

```
    value = value * n;
                  ^^^
6: Type error in binary operator. float * int
```

- It's a fairly critical usability concern

Source Abstractions

- It's more complicated than it seems

```
1: /* Print the first ten factorials */
2: var n int = 1;
3: var value float = 1.0;
4:
5: while n < 10 {
6:     value = value * n;
7:     print value ;
8:     n = n + 1;
9: }
```

model

```
BinOp('*',
      Variable('value'),
      Variable('n'))
```

```
.lineno = 6
.start = 103
.end = 112
```

- A mapping between the input and the model
- It's more of a database problem

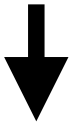
The Lexing Problem

- Source code is processed at the lowest level through a technique known as "lexing"
- Or tokenizing

Lexing in a Nutshell

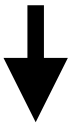
- Convert characters into "tokens"

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



Characters (from source file)

```
[ 'b','a','r',' ','=',' ','4','0',' ','+','2','0','*', ...]
```



Tokens

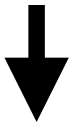
```
[ ('NAME', 'bar'), ('ASSIGN', '='), ('INTEGER', '40'),
  ('PLUS', '+'), ('INTEGER', '20'), ('TIMES', '*'), ... ]
```

- Essentially it's pattern matching

More on Tokens

- Tokens carry types, values, and locations

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



```
Token(
    type='INTEGER',
    value='20',
    lineno=5,
    index=11
)
```

- Represents "what" and "where"

How to implement

- Perform a linear text scan

- Processing the text as characters

```
text = "bar = 40 + 20 * (2 + 3) / 37.5"
n = 0
while n < len(text):
    ...
    yield token
    ...
    n += 1
```

- Produce tokens as you go

Simple Tokens

- Many tokens are single literal characters

```
literals = {
    '+': 'PLUS',
    '-': 'MINUS',
    '*': 'TIMES',
    '/': 'DIVIDE',
    '<': 'LT',
    '>': 'GT',
    '(': 'LPAREN',
    ')': 'RPAREN',
    '{': 'LBRACE',
    '}': 'RBRACE',
    ';': 'SEMI',
    ',': 'COMMA',
    ...
}
```

```
while n < len(text):
    ...
    if text[n] in literals:
        tok = Token(
                literals[text[n]],
                text[n])
        n += 1
    ...
```

- Can encode as a table lookup

Exact Sequences

- Some tokens are literal sequences of multiples

```
literals = {
    '<=': 'LE',
    '>=': 'GE',
    '==': 'EQ',
    '!=': 'NE',
    '&&': 'LAND',
    '||': 'LOR',
    ...
}
```

- Also easy to match (table lookup)
- Caveat: Must match longer tokens first

```
'<=':  'LE'
'<':   'LT'
'==':  'EQ'
'=':   'ASSIGN'
```

Ignored Text

- Certain characters are ignored (whitespace)

```
ignored = { ' ', '\t', '\n', '\r' }
```

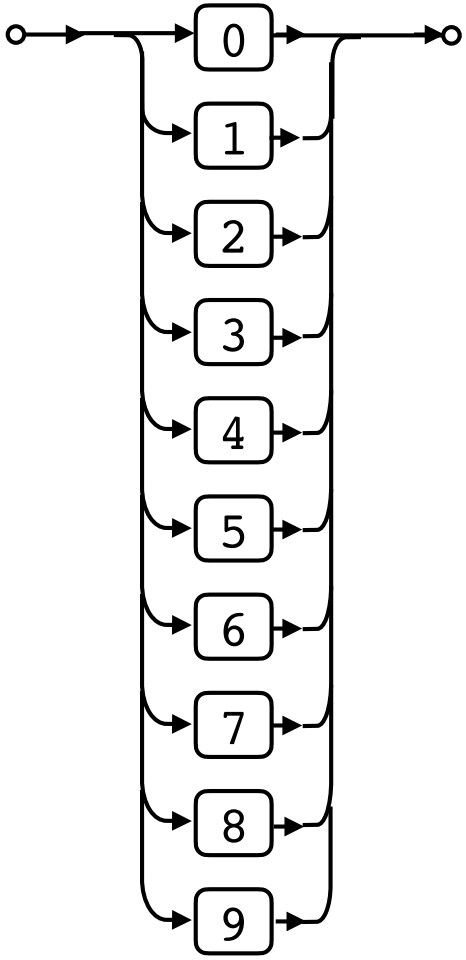
- Represents empty space between tokens
- Comments:

```
// ignored ...  \n
/* ignored ... */
```

- If you see the start of a comment, you read ahead and discard characters until the end of comment is detected.

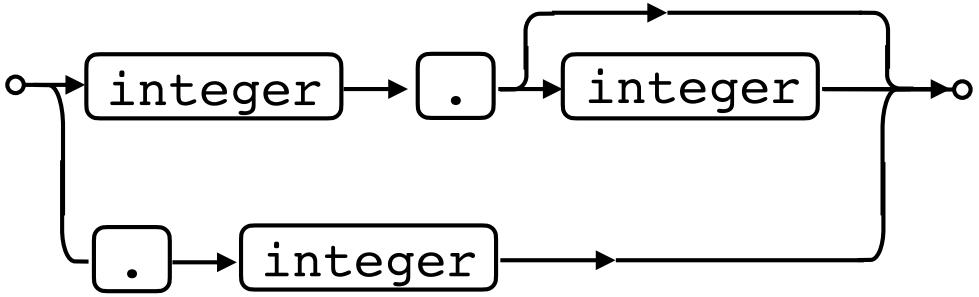
Number Tokens

digit:



integer:

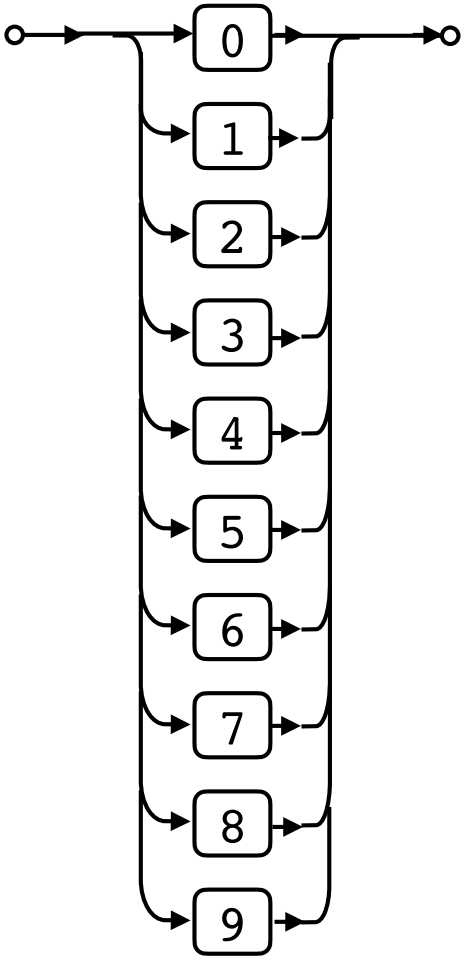
float:



This is a "syntax diagram". You write code that follows the arrows. Also called a "railway diagram".

Numbers

digit:



integer:


```
def isdigit(c):
    return c >= '0' and c <= '9'

while n < len(text):
    ...
    if isdigit(text[n]):
        start = n
        while isdigit(text[n]):
            n += 1
        tok = Token(
                  'NUMBER',
                  text[start:n])
    ...
```

Project

- Read source (`wabbit/source.py`)
- Tokenize source (`wabbit/tokenize.py`)
- Follow instructions inside
- Will group code parts of it