15 Constructing a Parser Level 2

Introduction

- `True`, `False`, and `None`
- floating-point numbers
- strings delimited with single quotes (for example, `'hello'`)
- subtraction
- floating-point division
- relational expressions using `==`, `!=`, `<`, `<=`, `>`, and `>=`
- `pass` statement
- `if` statement
- `while` statement

Our New Grammar

```
<program>       → <stmt>* EOF

<stmt>          → <simplestmt> NEWLINE
<stmt>          → <compoundstmt>

<simplestmt>    → <assignmentstmt>
<simplestmt>    → <printstmt>
<simplestmt>    → <passstmt>

<compoundstmt>  → <whilestmt>
<compoundstmt>  → <ifstmt>

<assignmentstmt>→ NAME '=' <relexpr>
<printstmt>     → 'print' '(' [<relexpr> (',' <relexpr> )* [',']] ')'
<passstmt>      → 'pass'
<whilestmt>     → 'while' <relexpr> ':' <codeblock>
<ifstmt>        → 'if' <relexpr> ':' <codeblock> ['else' ':' <codeblock>]

<codeblock>     →  NEWLINE INDENT <stmt>+ DEDENT
<relexpr>       → <expr> [ ('==' | '!=' | '<' | '<=' | '>' | '>=') <expr> ]
<expr>          → <term> (('+' | '-') <term>)*
<term>          → <factor> (('*' | '/') <factor>)*

<factor>        → '+' <factor>
<factor>        → '-' <factor>
<factor>        → UNSIGNEDINT
<factor>        → UNSIGNEDFLOAT
<factor>        → NAME
<factor>        → '(' <relexpr> ')'
<factor>        → STRING
<factor>        → 'True'
<factor>        → 'False'
<factor>        → 'None'
```

Figure 15.1

```
<ifstmt> → 'if' <relexpr> ':' <codeblock> ['else' ':' <codeblock>]
```

check here if `else` is the current token

```
def ifstmt():
   advance()                          # advance past 'if'
   relexpr()
   consume(COLON)
   codeblock()                        # parse code block
   if token.category == ELSE:         # check if there is an else part
      advance()                       # advance past "else"
      consume(COLON)                  # check for and advance past colon
      codeblock()                     # parse the else code block
```

```
<codeblock> → NEWLINE INDENT <stmt>+ DEDENT


<expr>
<expr> == <expr>      <expr> != <expr>
<expr> <  <expr>      <expr> <= <expr>
<expr> >  <expr>      <expr> >= <expr>


<relexpr>       → <expr> [ ('==' | '!=' | '<' | '<=' | '>' | '>=') <expr> ]
```

```
def relexpr():
   expr()
   if token.category in [LESSTHAN, LESSEQUAL, EQUAL, NOTEQUAL,
                         GREATERTHAN, GREATEREQUAL]:
      advance()
      expr()
```

```
<assignmentstmt> → NAME '=' <relexpr>

<printstmt> → 'print' '(' [ <relexpr> (',' <relexpr> )* [','] ] ')'
```

Which comma?

Figure 15.2

```
print()             # effect is to go the next line
print(<relexpr>)    # single argument

# comma separates successive <relexpr> arguments
print(<relexpr>, <relexpr>, ..., <relexpr>)

# comma allowed after last <relexpr>
print(<relexpr>, <relexpr>, ..., <relexpr>,)
```

```
def printstmt():
   advance()
   consume(LEFTPAREN)
   if token.category != RIGHTPAREN:       # any arguments?
      relexpr()
      while token.category == COMMA:
         advance()
         if token.category == RIGHTPAREN: # determine which comma
            break                         # break if comma at end
         relexpr()
   consume(RIGHTPAREN)
```

```
<expr>      → <term> (('+' | '-') <term>)*
<term>      → <factor> (('*' | '/') <factor>)*
```

```
def expr():
   term()
   while token.category == PLUS or token.category == MINUS:
      advance()
      term()
```

```
<factor>        → UNSIGNEDFLOAT
<factor>        → '(' <relexpr> ')'
<factor>        → STRING
<factor>        → 'True'
<factor>        → 'False'
<factor>        → 'None'
```

```
...
elif token.category == UNSIGNEDFLOAT:
      advance()
elif ...
...
```

Shortcomings of our Grammar

```
s = 'hello' / 'bye'
```