

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 

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

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

`<expr>`

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

`<expr> < <expr>` `<expr> <= <expr>`

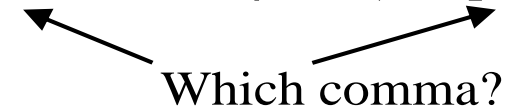
`<expr> > <expr>` `<expr> >= <expr>`

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

```
1 def relexpr():
2     expr()
3     if token.category in [LESSTHAN, LESSEQUAL, EQUAL, NOTEQUAL,
4                           GREATERTHAN, GREATEREQUAL]:
5         advance()
6         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>,)
```

```
1 def printstmt():
2     advance()
3     consume(LEFTPAREN)
4     if token.category != RIGHTPAREN:          # any arguments?
5         relexpr()
6         while token.category == COMMA:
7             advance()
8             if token.category == RIGHTPAREN: # determine which comma
9                 break                        # break if comma at end
10            relexpr()
11    consume(RIGHTPAREN)
```

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

```
1 def expr():  
2     term()  
3     while token.category == PLUS or token.category == MINUS:  
4         advance()  
5         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'
```