Parser and Abstract Syntax Tree Report

Alice Tedeschi, Dawson Finklea

Grammar

```
program ::= {var def | func def | class def }* stmt*

class def ::= class ID ( ID ) : NEWLINE INDENT class body DEDENT

class body ::= pass NEWLINE
| {var def | func def }+

func def ::= def ID ( {typed var {, typed var }*}? ) {-> type}? : NEWLINE INDENT func body

func body ::= {global decl | nonlocal decl | var def | func def }* stmt+

typed var ::= ID : type

type ::= ID | IDSTRING | [ type ]

global decl ::= global ID NEWLINE

nonlocal decl ::= nonlocal ID NEWLINE

var def ::= typed var = literal NEWLINE

stmt ::= simple stmt NEWLINE
| if expr : block {elif expr : block }* {else : block }?
| while expr : bloc}
| for ID in expr : bloc}

simple stmt ::= pass
| return {expr }?
| expr
| { target = }+ expr
block ::= NEWLINE INDENT stmt+ DEDENT

literal ::= None
| True
| False
| INTEGER
| IDSTRING | STRING

# expr ::= e_or0_expr e_if0_expr
# e_if0_expr ::= e_if_expr
# | eps

# e_if_expr ::= if e_if_expr else e_if_expr
# | eps

# e_or0_expr - e_and0_expr e_or_expr

# e_or_expr - or e_and0_expr e_or_expr
# | eps

# e_and0_expr - e_not_expr e_and_expr

# e_and_expr - and e_not_expr e_and_expr
# | eps

# e_not_expr - not e_not_expr
# | cexpr

# cexpr ::= fexpr c_0_expr
# | - cexpr

# c_0_expr ::= . ID parenthesis c_0_expr
# | [ expr ] c_0_expr
# | bin op cexpr c_0_expr
# | eps

parenthesis ::= ( {expr {, expr }*}? )
# | eps

bin_op ::= + | - | * | // | % | == | != | <= | >= | < | > | is

fexpr ::= ID parenthesis
| literal
| [ {expr {, expr }*}? ]
| ( expr )
```

The following productions were not used in our program:

```
target ::= ID
| cexpr target_1

target_1 ::= . ID | [expr]
```

This is a rewritten form of the original reference grammar, refactored to eliminate ambiguity, and left-recursion.

Likely the most difficult part of this project was refactoring the grammar in such a way that an Abstact Syntax Tree could still be written into the parser and adding a hierarchy for some operators.

Another sticking point was the `target` non-terminal. For every assignment our code just matches it with an expression and then if an Assign token occurs checks the type of the expression node: if it is a Identifier Node, a Member Node or an Index Node it generates a Target Node.

Last but not least, as it often happens with lots of coding, the debugging took much longer than we could have anticipated.