Enhancing Python Syntax: Increment, Decrement, Do- While, Until, Unless & Switch-Case Statements

By: Zak Ahmed, Pedro Oliveira, Piranaven Selvathayabaran

Department of Computing and Software, McMaster University

April 17th 2019

Problem

Python is one of the most beloved programming languages. It contains a plethora of syntax that enables its users to achieve their goals.

However, it does not contain a handful of syntax that we believe would enrich the programming language and provide developers with more flexibility. These language constructs are **Increment, Decrement, Until, Unless, Do-While Loops, & Switch-Case Statements.** These enhancements, although not necessarily pythonic, would undoubtedly strengthen a developers toolkit.

Process

In order to alter Python, we took advantage of the CPython compiler (V 3.7) which executes the following steps[2]:

1. Parse source code into a parse tree (Parser/pgen.c)
2. Transform parse tree into an Abstract Syntax Tree (Python/ast.c)
3. Transform AST into a Control Flow Graph (Python/compile.c)
4. Emit bytecode based on the Control Flow Graph (Python/compile.c)

Several other files needed to be altered to achieve our results. Most notably:

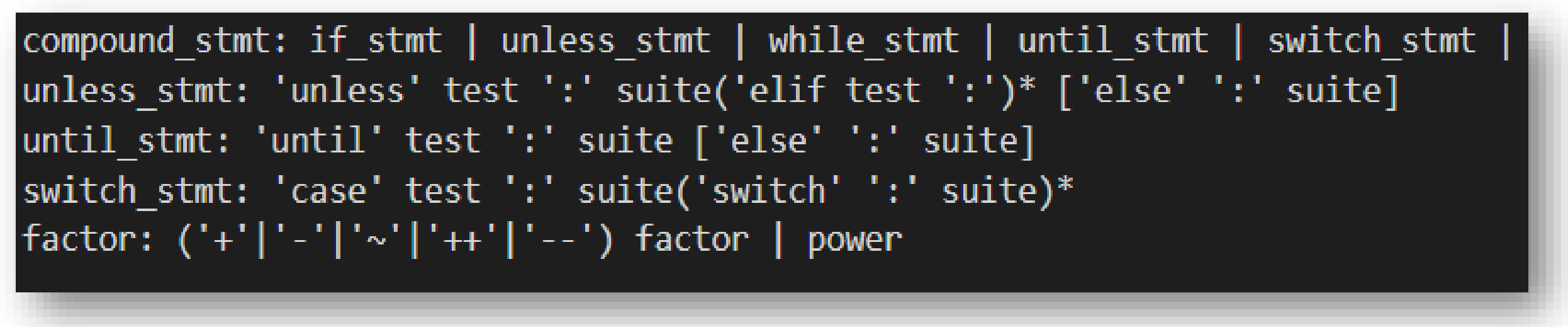
- Python.asdl → Used for parsing into AST
- Grammar → Contains grammar rules
- ceval.c → executes byte-code
- symtable.c → Generates a symbol table from AST

Following these changes, running a configured Makefile regenerated the remaining necessary source code such as graminit.c and node.h. Finally, we used the “dis” module to verify that the generated bytecode was as expected.

CPython Modifications & Example Code Additions

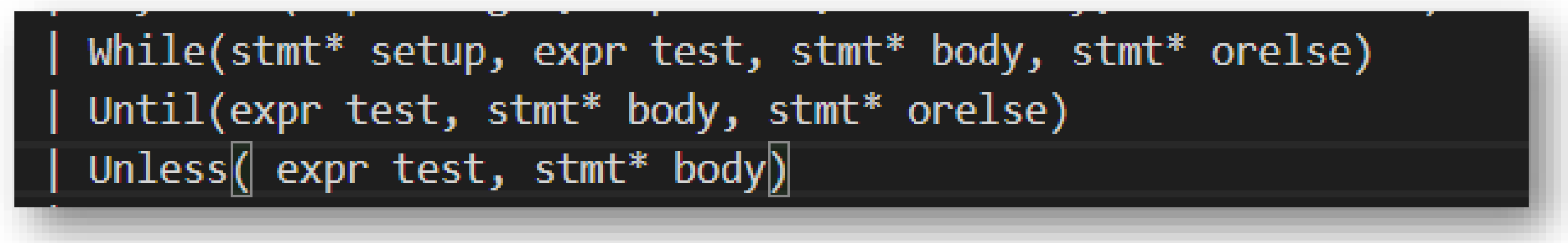
GRAMMAR

All CPython modifications started with some grammar rule changes:



```
compound_stmt: if_stmt | unless_stmt | while_stmt | until_stmt | switch_stmt |
unless_stmt: 'unless' test ':' suite('elif test ':')* ['else' ':' suite]
until_stmt: 'until' test ':' suite ['else' ':' suite]
switch_stmt: 'case' test ':' suite('switch' ':' suite)*
factor: ('+'|'-'|'~'|'++'|'--') factor | power
```

Similar modifications were made in Python.asdl, which defines the AST nodes

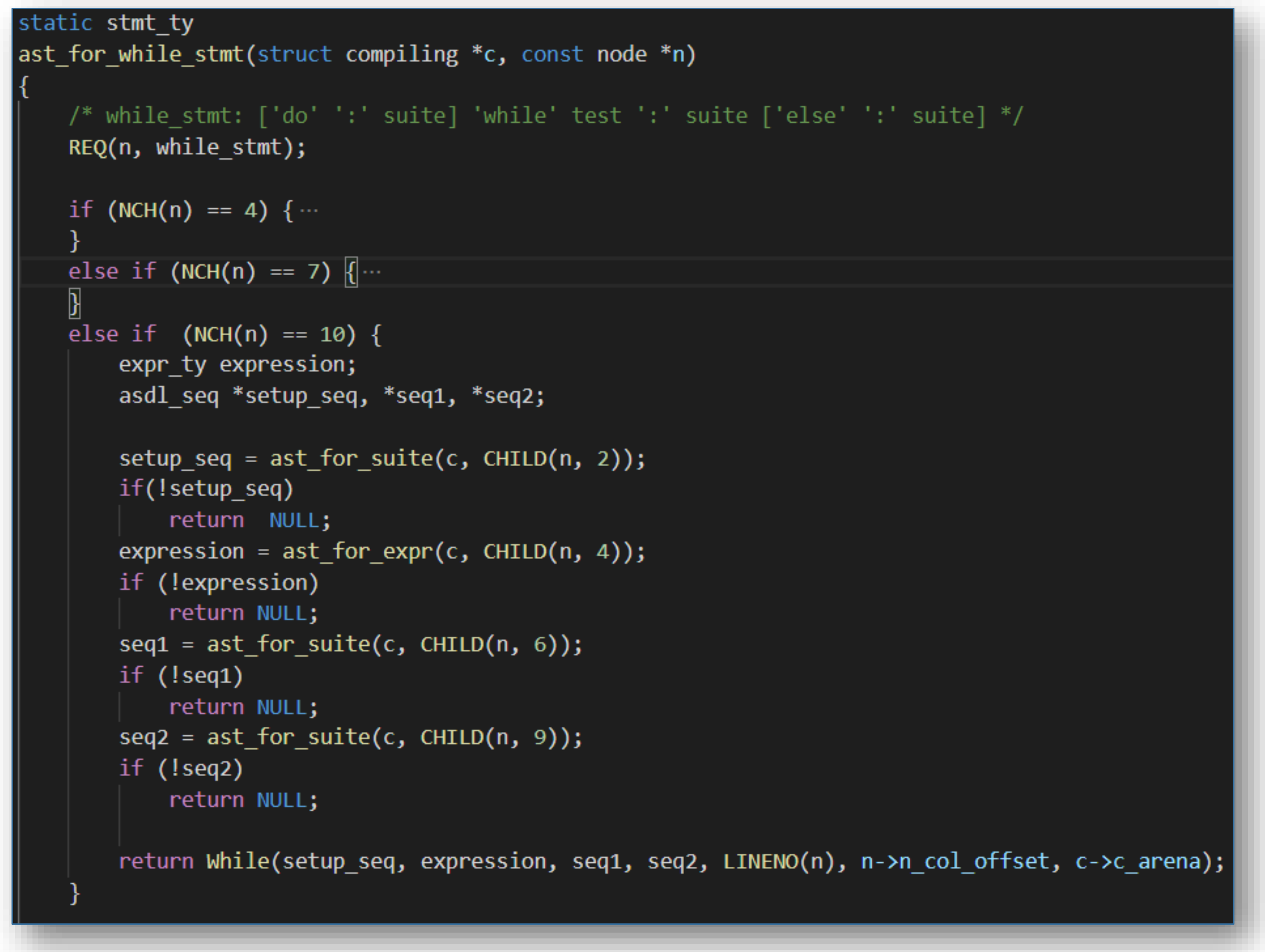


```
| While(stmt* setup, expr test, stmt* body, stmt* orelse)
| Until(expr test, stmt* body, stmt* orelse)
| Unless( expr test, stmt* body)
```

AST MODIFICATIONS

Below is an example of the kinds of modifications we had to make to allow for 4 different options in a do-while-else loop. Various combinations would allow for various child node configurations which we had to account for, namely:

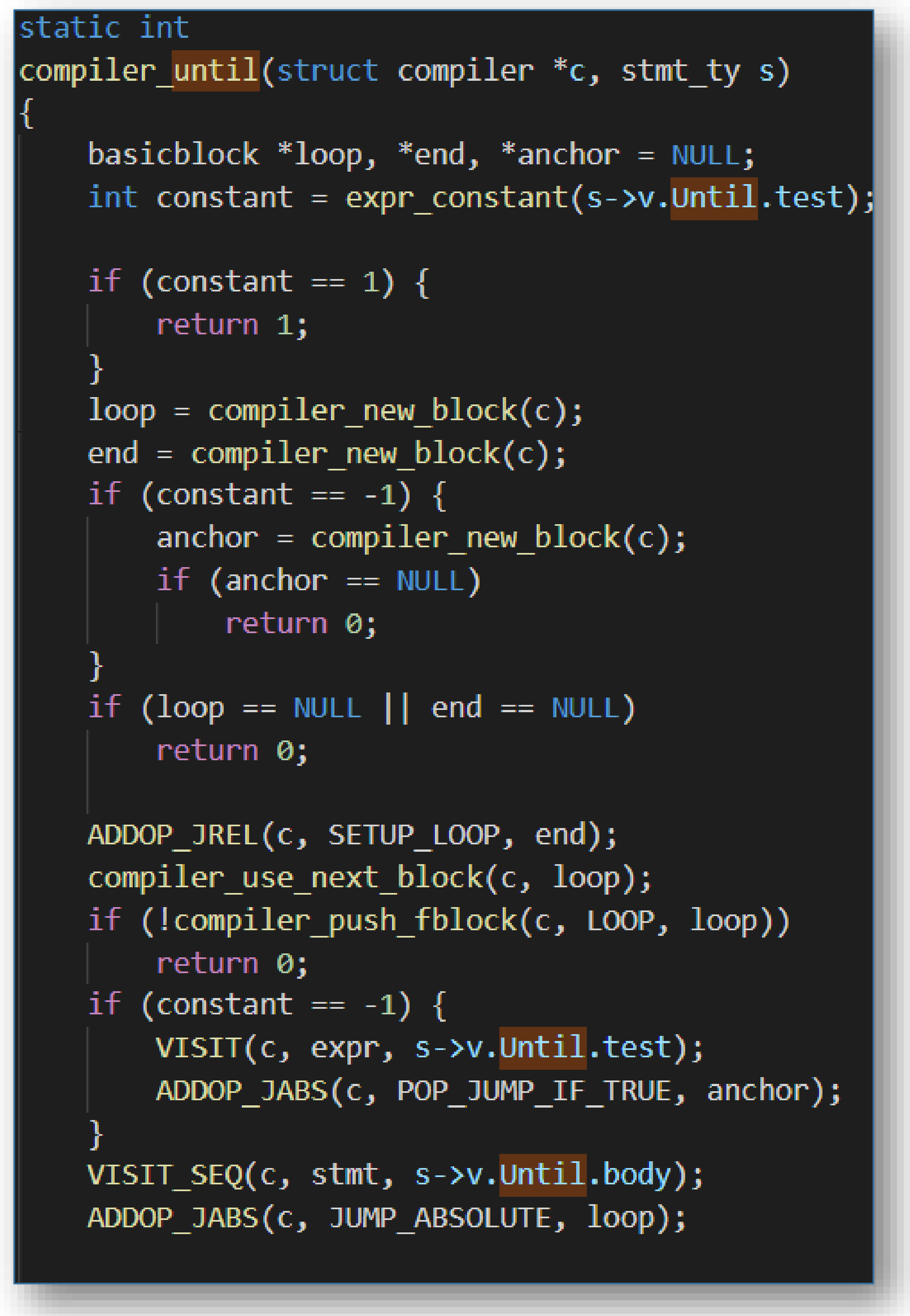
- While: 4 child nodes
- Do While: 7 child nodes
- While, Else: 7 child nodes
- Do While Else: 10 child nodes



```
static stmt_ty
ast_for_while_stmt(struct compiling *c, const node *n)
{
    /* while_stmt: ['do' ':' suite] 'while' test ':' suite ['else' ':' suite] */
    REQ(n, while_stmt);

    if (NCH(n) == 4) {…
    }
    else if (NCH(n) == 7) {…
    }
    else if  (NCH(n) == 10) {
        expr_ty expression;
        asdl_seq *setup_seq, *seq1, *seq2;

        setup_seq = ast_for_suite(c, CHILD(n, 2));
        if(!setup_seq)
            return  NULL;
        expression = ast_for_expr(c, CHILD(n, 4));
        if (!expression)
            return NULL;
        seq1 = ast_for_suite(c, CHILD(n, 6));
        if (!seq1)
            return NULL;
        seq2 = ast_for_suite(c, CHILD(n, 9));
        if (!seq2)
            return NULL;

        return While(setup_seq, expression, seq1, seq2, LINENO(n), n->n_col_offset, c->c_arena);
    }
```

COMPILING TO BYTECODE

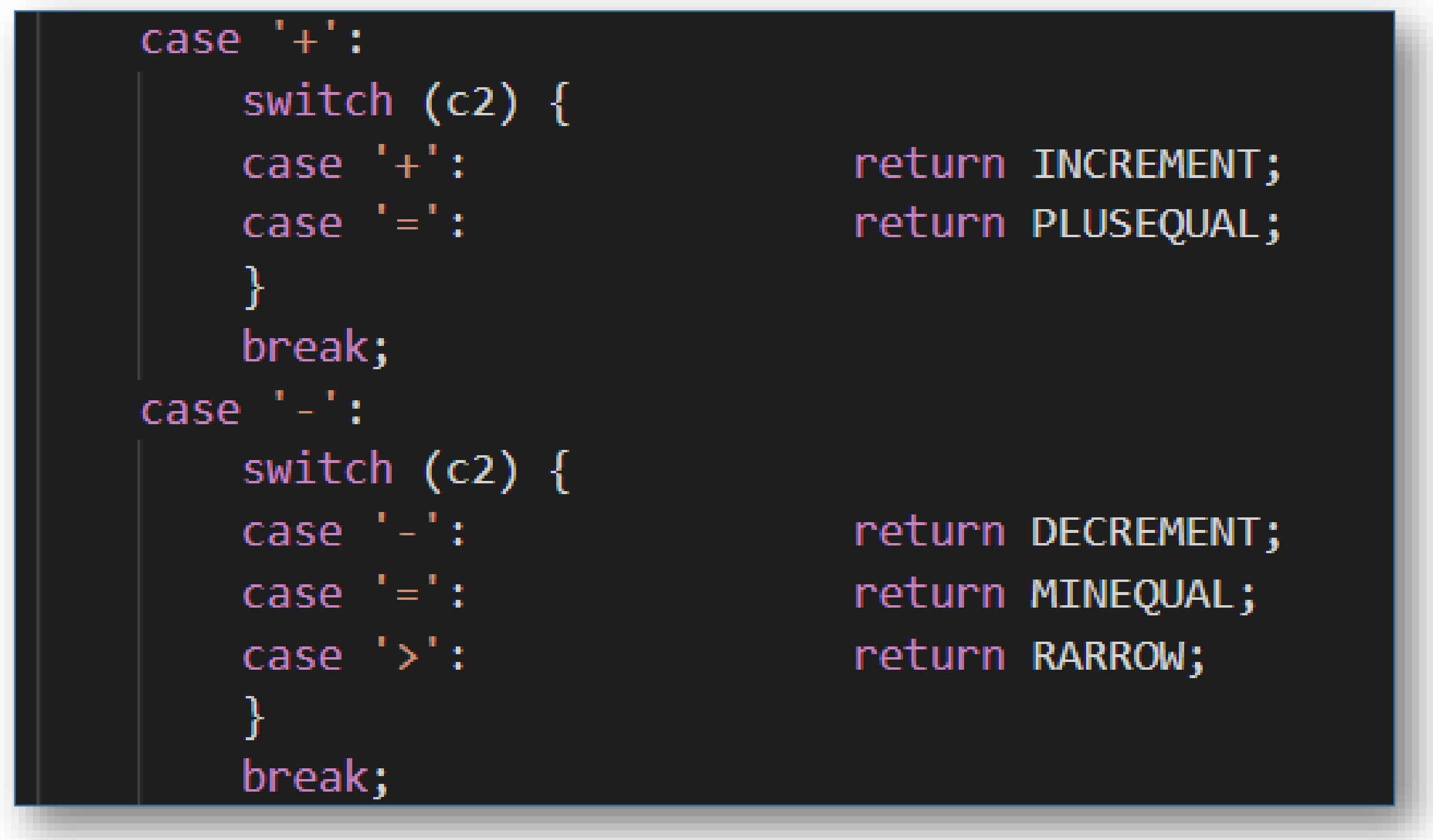
The implementation on the right demonstrates the code needed in order to compile the AST to bytecode. The bytecode is handled via several macros. There are also various helper functions like compiler_use_next_block which handles the offsetting of the next instruction.



```
static int
compiler_until(struct compiler *c, stmt_ty s)
{
    basicblock *loop, *end, *anchor = NULL;
    int constant = expr_constant(s->v.Until.test);

    if (constant == 1) {
        return 1;
    }
    loop = compiler_new_block(c);
    end = compiler_new_block(c);
    if (constant == -1) {
        anchor = compiler_new_block(c);
        if (anchor == NULL)
            return 0;
    }
    if (loop == NULL || end == NULL)
        return 0;

    ADDOP_JREL(c, SETUP_LOOP, end);
    compiler_use_next_block(c, loop);
    if (!compiler_push_fblock(c, LOOP, loop))
        return 0;
    if (constant == -1) {
        VISIT(c, expr, s->v.Until.test);
        ADDOP_JABS(c, POP_JUMP_IF_TRUE, anchor);
    }
    VISIT_SEQ(c, stmt, s->v.Until.body);
    ADDOP_JABS(c, JUMP_ABSOLUTE, loop);
```

PARSE TREE

Parsing tokens like increment and decrement required additional changes to tokenizer.c and token.h. We explicitly defined and handled them since they represented a double token. As seen in the switch case statement below, after first checking for the ‘+’ or ‘-’ symbol, a following check for the same symbol is done such that the appropriate call to increment/decrement is made.



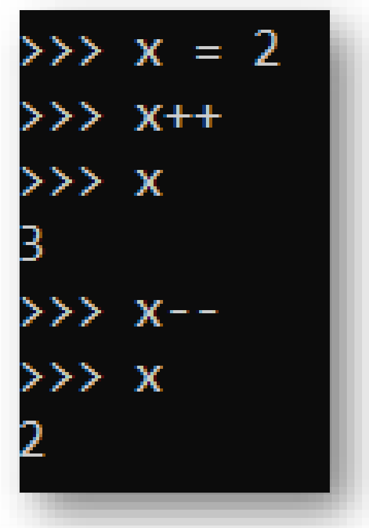
```
    case '+':
        switch (c2) {
        case '+':              return INCREMENT;
        case '=':              return PLUSEQUAL;
        }
        break;
    case '-':
        switch (c2) {
        case '-':              return DECREMENT;
        case '=':              return MINEQUAL;
        case '>':              return RARROW;
        }
        break;
```

Results

OUTPUT FROM TEST CASES

Increment/ Decrement

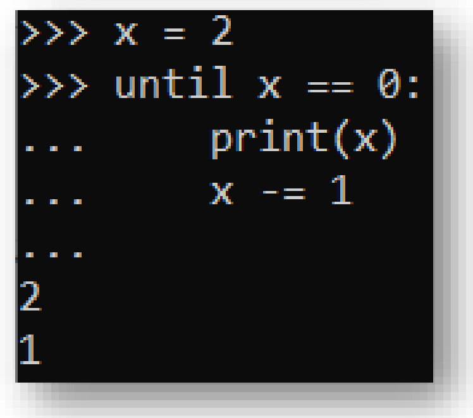
Increment and decrement operators are unary operators that add or subtract one from their operand, respectively.



```
>>> x = 2
>>> x++
>>> x
3
>>> x--
>>> x
2
```

Until

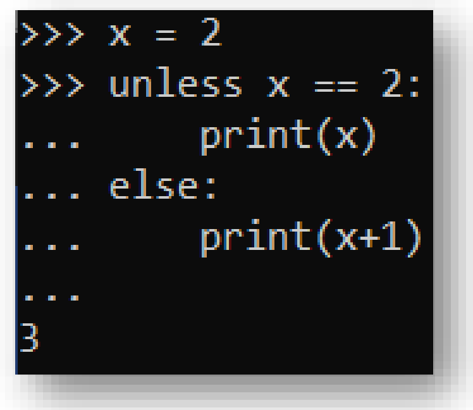
An until loop statement repeatedly executes a target statement as long as a given condition is false.



```
>>> x = 2
>>> until x == 0:
...     print(x)
...     x -= 1
...
2
1
```

Unless

The unless expression is the opposite of the if expression. If the value is false the “else“ expression is executed



```
>>> x = 2
>>> unless x == 2:
...     print(x)
... else:
...     print(x+1)
...
3
```

Conclusion

The most challenging part of this project was the size and complexity of the code. It was very challenging at first to find what we were looking for and understand how everything in the CPython compiler was connected. This was further magnified by the out of date documentation and heavy use of macros. In hindsight, we didn’t have to write too many lines of code and it was truly a situation of quality over quantity. If we were to do this project again, we would definitely try to implement some compiler optimizations like constant folding. Looking forward, we would love to try and implement some of existing PEP’s, most notably PEP 3124 - Overloading, Generic Functions, Interfaces, and Adaptation.

References

[1] Python Software Foundation. (n.d.). 24. Changing CPython's Grammar. Retrieved April 09, 2019, from https://devguide.python.org/grammar/

[2] Python Software Foundation. (n.d.). 25. Design of CPython's Compiler. Retrieved March 26, 2019, from https://devguide.python.org/compiler/

[3.] Python Software Foundation. (n.d.). 10. Full Grammar specification. Retrieved April 6, 2019, from https://docs.python.org/3/reference/grammar.html

[4] Python Software Foundation. (n.d.). PEP 3103 -- A Switch/Case Statement. Retrieved March 22, 2019, from https://www.python.org/dev/peps/pep-3103/