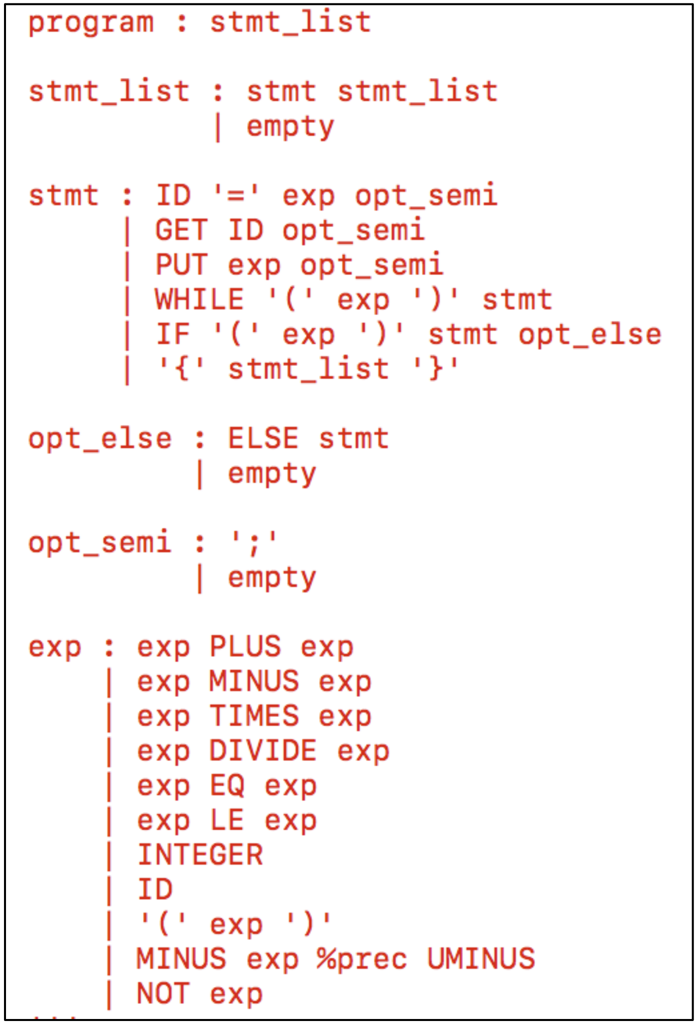
A Basic Compiler

- At a fundamental level compilers can be understood as processors that match AST patterns of the source language and translate them into patterns in the target language.
- Here we will look at a basic compiler that translates Cuppa1 programs into exp1bytecode.

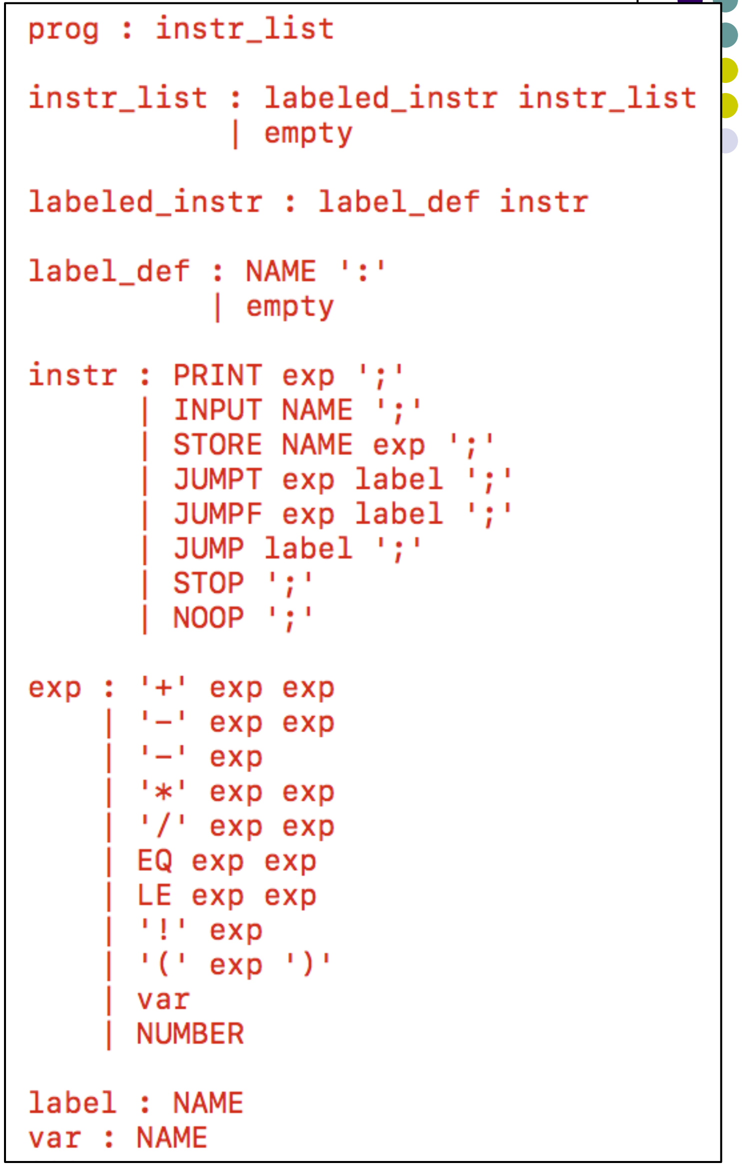
Chap 6

A Basic Compiler



```
program : stmt_list

stmt_list : stmt stmt_list
          | empty

stmt : ID '=' exp opt_semi
     | GET ID opt_semi
     | PUT exp opt_semi
     | WHILE '(' exp ')' stmt
     | IF '(' exp ')' stmt opt_else
     | '{' stmt_list '}'

opt_else : ELSE stmt
         | empty

opt_semi : ';'
         | empty

exp : exp PLUS exp
    | exp MINUS exp
    | exp TIMES exp
    | exp DIVIDE exp
    | exp EQ exp
    | exp LE exp
    | INTEGER
    | ID
    | '(' exp ')'
    | MINUS exp %prec UMINUS
    | NOT exp
```

Cuppa1



```
prog : instr_list

instr_list : labeled_instr instr_list
           | empty

labeled_instr : label_def instr

label_def : NAME ':'
          | empty

instr : PRINT exp ';'
      | INPUT NAME ';'
      | STORE NAME exp ';'
      | JUMPT exp label ';'
      | JUMPF exp label ';'
      | JUMP label ';'
      | STOP ';'
      | NOOP ';'

exp : '+' exp exp
    | '-' exp exp
    | '-' exp
    | '*' exp exp
    | '/' exp exp
    | EQ exp exp
    | LE exp exp
    | '!' exp
    | '(' exp ')'
    | var
    | NUMBER

label : NAME
var : NAME
```

Exp1bytecode

A Basic Compiler

- Consider for example the AST pattern for the assignment statement in Cuppa1,
  - ('assign', name, exp)
- We could easily envision translating this AST pattern into a pattern in Exp1bytecode as follows,
  - store <name> <exp>;
- where <name> and <exp> are the appropriate translations of the variable name and the assignment expression from Cuppa1 into Exp1bytecode.

A Basic Compiler

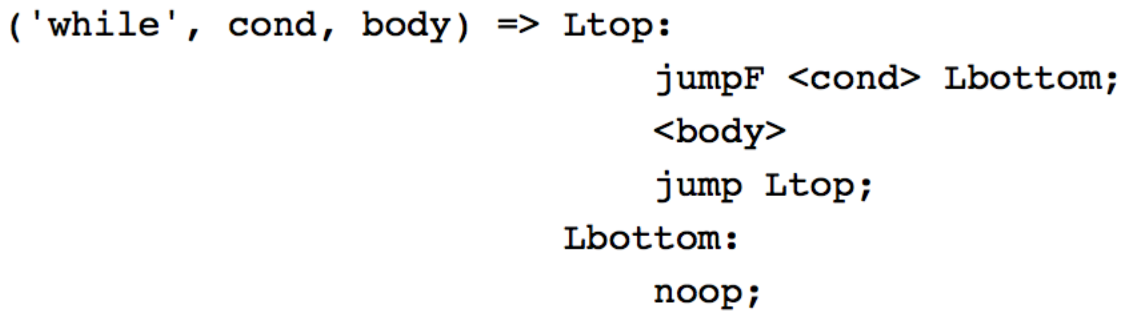
- In our case it is not that difficult to come up with pattern translations for all the non-structured statements and expressions in Cuppa1.
- For all the non-structured statements we have the pattern translations,
  - ('assign', name, exp) => store <name> <exp>;
  - ('put', exp) => print <exp>;
  - ('get', name) => input <name>;

A Basic Compiler

- And for the expressions we have,
  - ('+', c1, c2) => ('+' <c1> <c2>)
  - ('-', c1, c2) => ('-' <c1> <c2>)
  - ('*', c1, c2) => ('*' <c1> <c2>)
  - ('/', c1, c2) => ('/' <c1> <c2>)
  - ('==', c1, c2) => ('==' <c1> <c2>)
  - ('<=', c1, c2) => ('<=' <c1> <c2>)
  - ('id', name) => <name>
  - ('integer', value) => <value>
  - ('uminus', value) => - <value>
  - ('not', value) => ! <value>

A Basic Compiler

- One way to translate the AST pattern for the while loop into a code pattern in Exp1bytecode is,



```
('while', cond, body) => Ltop:
                              jumpF <cond> Lbottom;
                              <body>
                              jump Ltop;
                         Lbottom:
                              noop;
```

- Note that we have to “simulate” the behavior of the Cuppa1 “while” loop with jump statements in Exp1bytecode.

A Basic Compiler

- We can do something similar with if-then statements,

```
('if', cond, then_stmt, ('nil',)) =>     jumpF <cond> Lbottom;
                                         <then_stmt>
                                     Lbottom:
                                         noop;
```

- Finally, adding the else-statement to the if-then statement we have,

```
('if', cond, then_stmt, else_stmt) =>     jumpF <cond> Lelse;
                                          <then_stmt>
                                          jump Lbottom;
                                      Lelse:
                                          <else_stmt>
                                      Lbottom:
                                         noop;
```

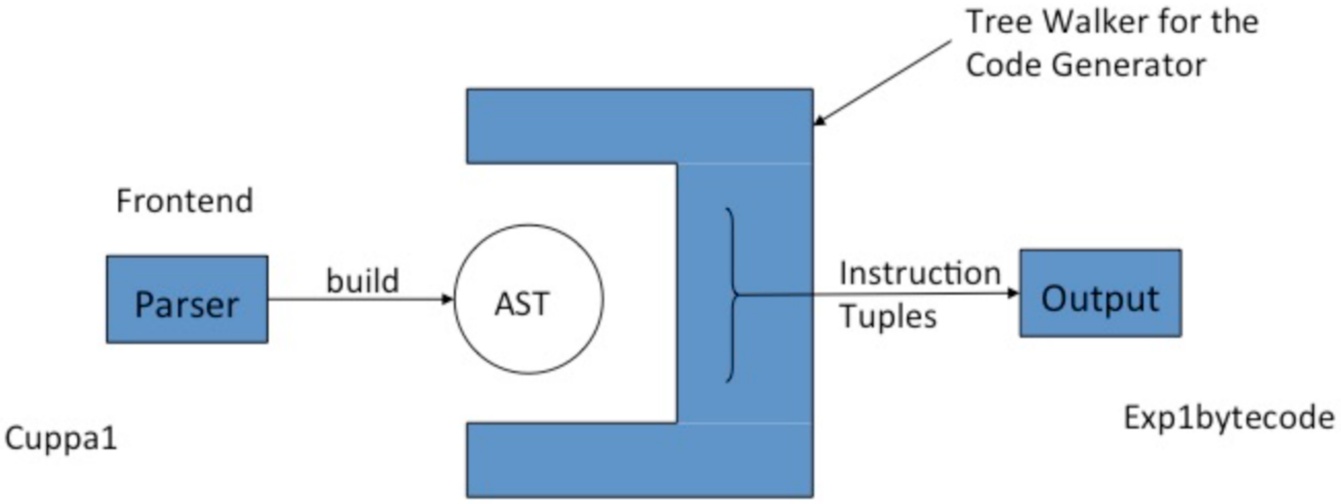
A Basic Compiler

- One thing to keep in mind is the notion of *target pattern compositionality*.
- By that we mean that any target language patterns generated from the same class of AST patterns should be able to be composed,
  - Any one of the Exp1bytecode patterns due to statements in Cuppa1 should be able to be composed with any other Exp1bytecode pattern due to a statement without ever generating incorrect target code.
  - The same thing is true for Exp1bycode patterns generated from Cuppa1 expressions

See the ‘noop’ instructions at the end of the target patterns for ‘while’, ’if-then’, and ‘if-then-else’.

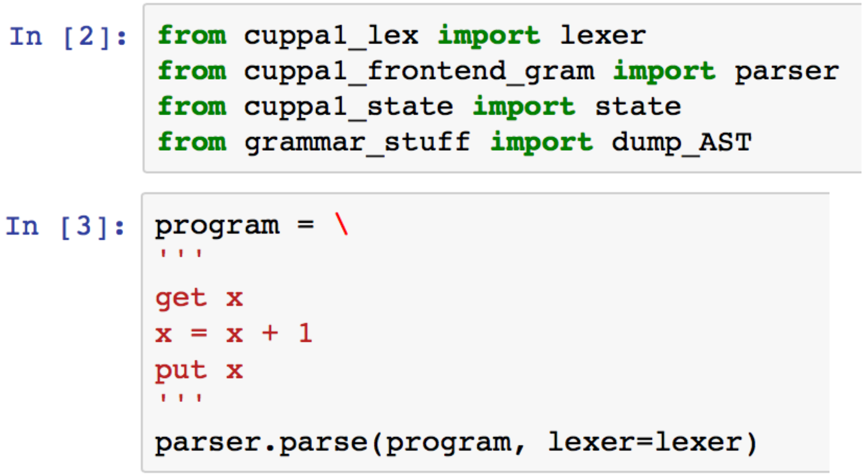
A Basic Compiler Architecture

- Our basic compiler consists of:
  - The Cuppa1 frontend
  - A code generation tree walker



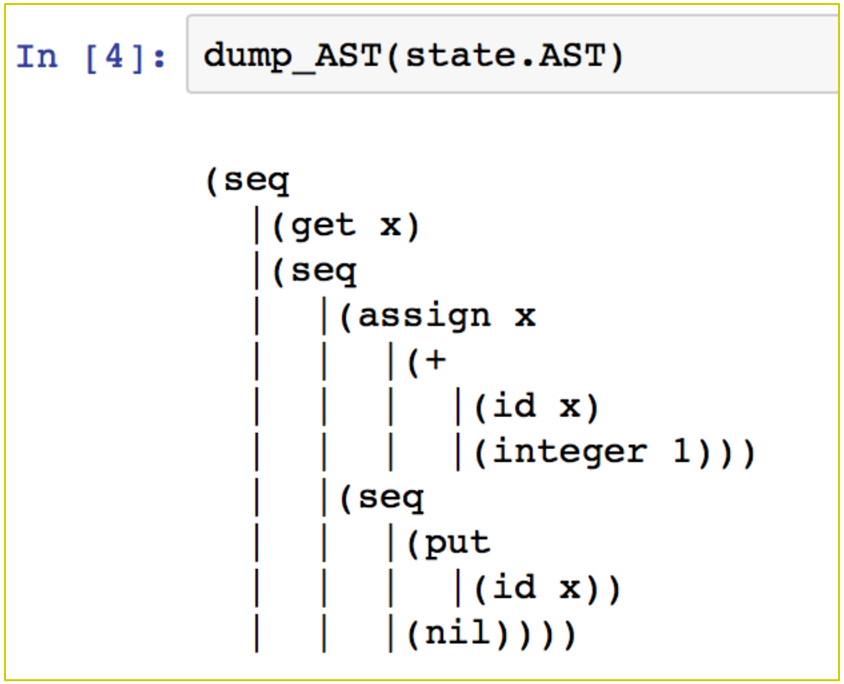
Frontend Pattern Translation

- Recall that the Cuppa1 frontend generates an AST for a source program,



```
In [2]: from cuppa1_lex import lexer
        from cuppa1_frontend_gram import parser
        from cuppa1_state import state
        from grammar_stuff import dump_AST
```

```
In [3]: program = \
        '''
        get x
        x = x + 1
        put x
        '''
        parser.parse(program, lexer=lexer)
```



```
In [4]: dump_AST(state.AST)


        (seq
           |(get x)
           |(seq
           |   |(assign x
           |   |   |(+
           |   |   |   |(id x)
           |   |   |   |(integer 1)))
           |   |(seq
           |   |   |(put
           |   |   |   |(id x))
           |   |   |(nil))))
```

We can easily apply our pattern translations to generate Exp1bytecode:

```
('get', name) => input <name>;
('assign', name, exp) => store <name> <exp>;
('put', exp) => print <exp>;
```

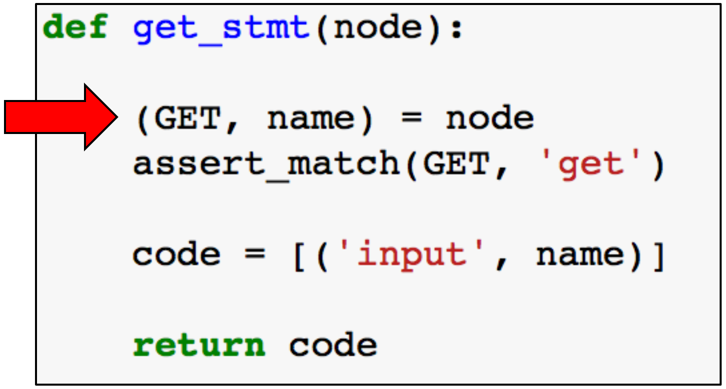
Codegen Tree Walker

- The code generator for our compiler is a tree walker that walks the Cuppa1 AST and for each AST pattern that appears in a pattern translation rule it will generate the corresponding target code.
- Cuppa1 statement patterns will generate Exp1bytecode instructions on a *list* and Cuppa1 expression patterns will generated Exp1bytecode expressions returned as *strings*.
    - The reason for this will become clear later when we look at optimizations in this compiler.

cuppa1_cc_codegen.py

Codegen

- Recall the pattern translation,
  - ('get', name) => input <name>;
- The codegen tree walker has a function for that,



```
def get_stmt(node):

    (GET, name) = node
    assert_match(GET, 'get')

    code = [('input', name)]

    return code
```

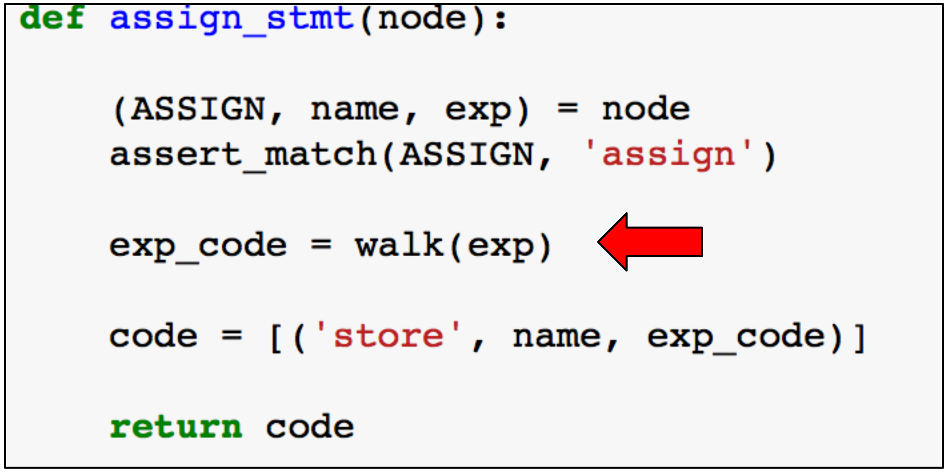
Even though the translation rule for the get statement demands that we also generate the semicolon as part of the translation,
we delay this until we generate the actual machine instructions.

We use Python’s ability to do pattern matching on tuples!

cuppa1_cc_codegen.py

Codegen

- Recall the pattern translation,
  - ('assign', name, exp) => store <name> <exp>;
- The codegen tree walker has a function for that,



```
def assign_stmt(node):

    (ASSIGN, name, exp) = node
    assert_match(ASSIGN, 'assign')

    exp_code = walk(exp)

    code = [('store', name, exp_code)]

    return code
```

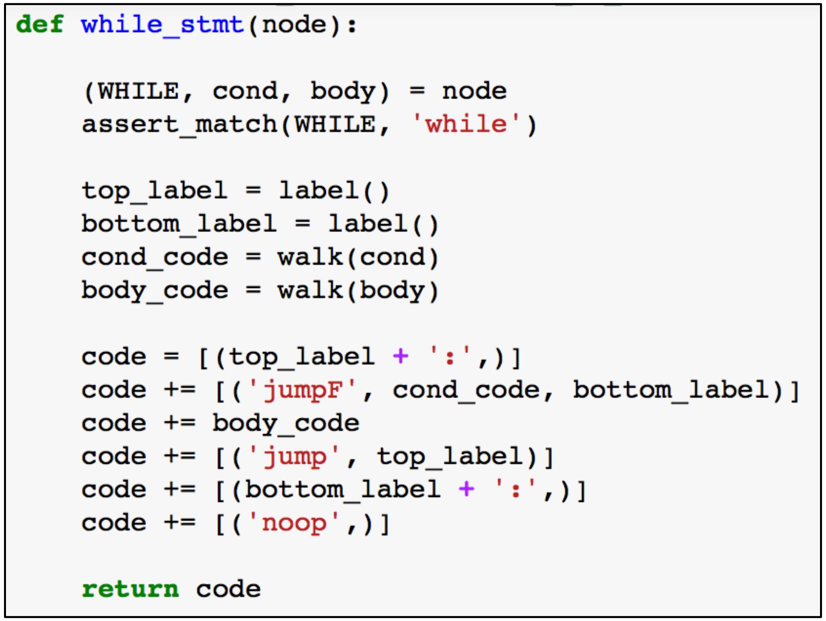
cuppa1_cc_codegen.py

Codegen

- Recall the pattern translation,

```
('while', cond, body) => Ltop:
                              jumpF <cond> Lbottom;
                              <body>
                              jump Ltop;
                          Lbottom:
                              noop;
```

- The codegen tree walker has a function for that,



```python
def while_stmt(node):

    (WHILE, cond, body) = node
    assert_match(WHILE, 'while')

    top_label = label()
    bottom_label = label()
    cond_code = walk(cond)
    body_code = walk(body)

    code = [(top_label + ':',)]
    code += [('jumpF', cond_code, bottom_label)]
    code += body_code
    code += [('jump', top_label)]
    code += [(bottom_label + ':',)]
    code += [('noop',)]

    return code
```

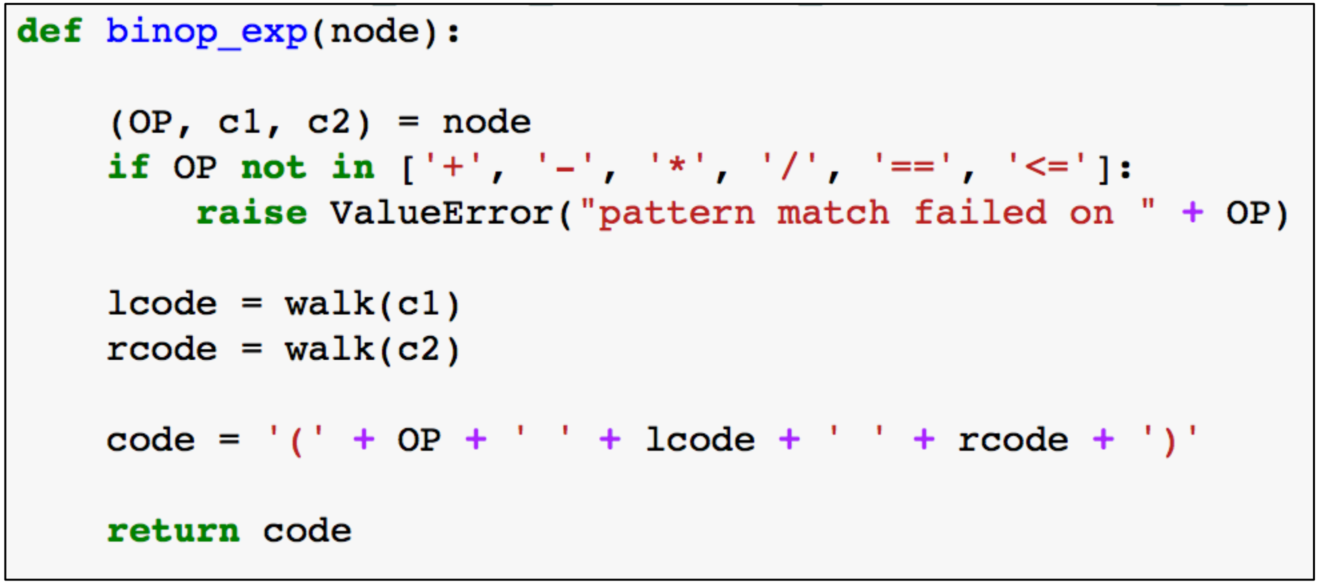
cuppa1_cc_codegen.py

Codegen

- Recall the pattern translation for binops,

```
('+', c1, c2) => ('+' <c1> <c2>)
('-', c1, c2) => ('-' <c1> <c2>)
('*', c1, c2) => ('*' <c1> <c2>)
('/', c1, c2) => ('/' <c1> <c2>)
('==', c1, c2) => ('==' <c1> <c2>)
('<=', c1, c2) => ('<=' <c1> <c2>)
```

- The codegen tree walker has a function for that,

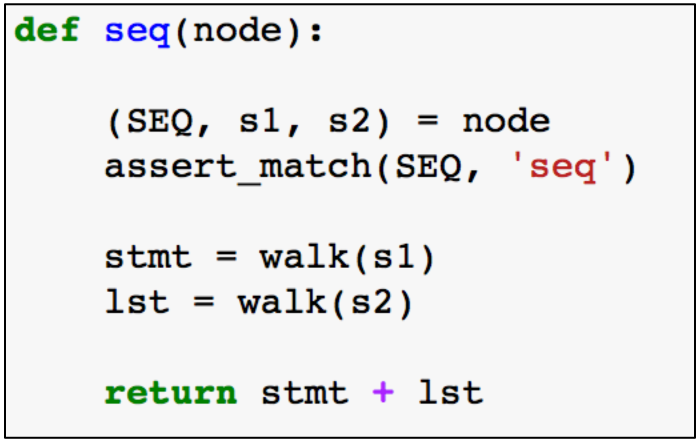


```python
def binop_exp(node):

    (OP, c1, c2) = node
    if OP not in ['+', '-', '*', '/', '==', '<=']:
        raise ValueError("pattern match failed on " + OP)

    lcode = walk(c1)
    rcode = walk(c2)

    code = '(' + OP + ' ' + lcode + ' ' + rcode + ')'

    return code
```

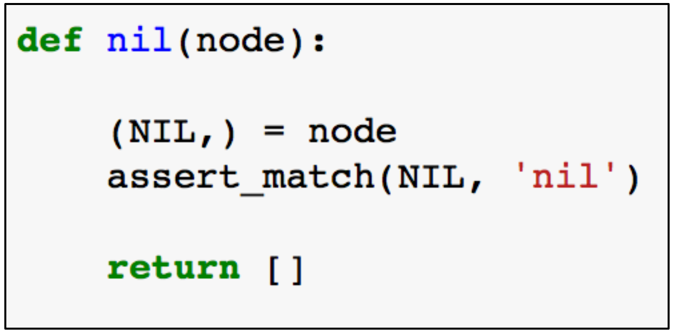
cuppa1_cc_codegen.py

Codegen

- What remains to be looked at is how the tree walker deals with Seq nodes since they act as the glue between the statements in the AST we saw above.
- Related to this is how the walker deals with Nil nodes in a statement sequence since Seq sequences are Nil terminated.

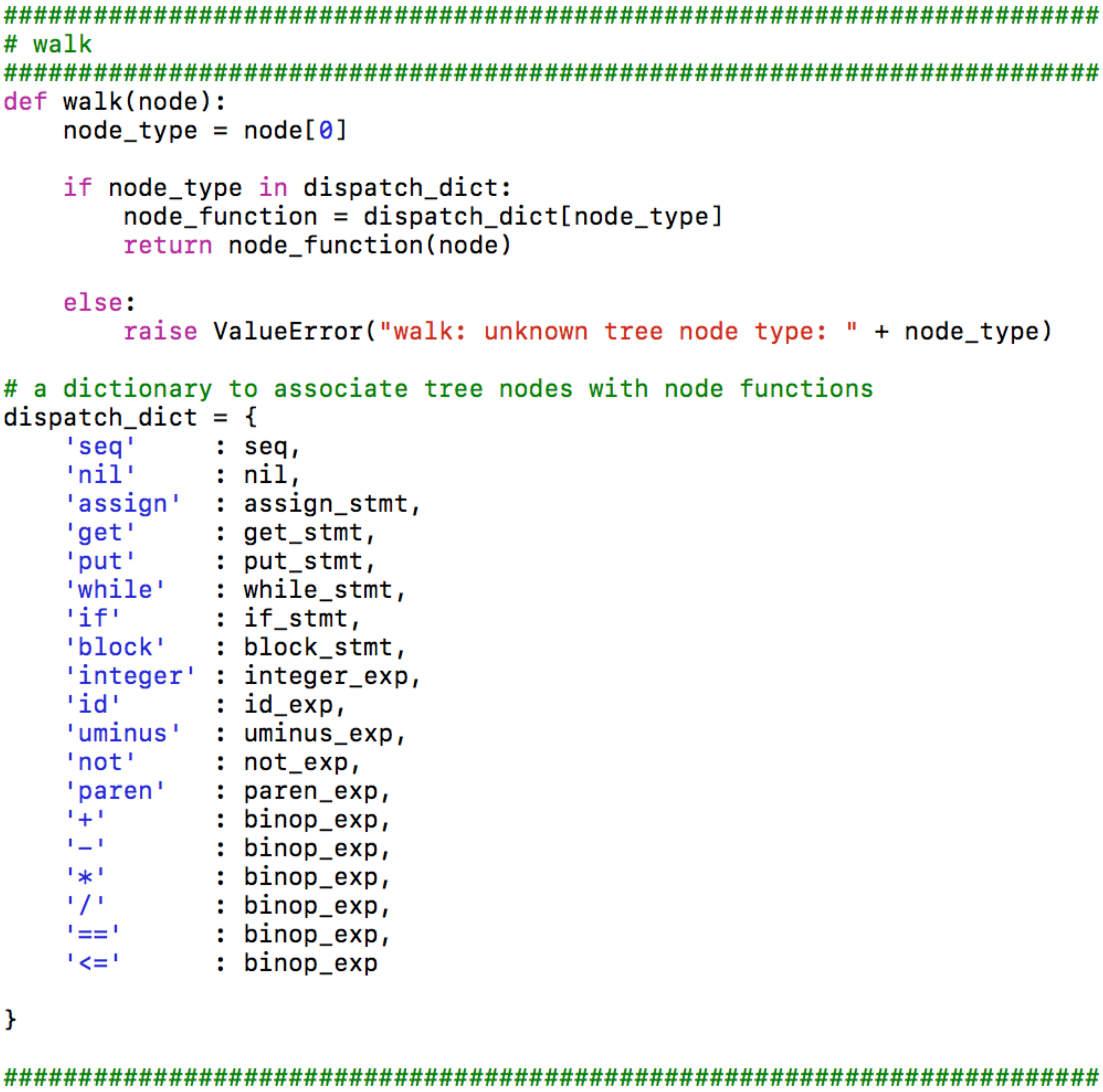


```python
def seq(node):

    (SEQ, s1, s2) = node
    assert_match(SEQ, 'seq')

    stmt = walk(s1)
    lst = walk(s2)

    return stmt + lst
```



```python
def nil(node):

    (NIL,) = node
    assert_match(NIL, 'nil')

    return []
```

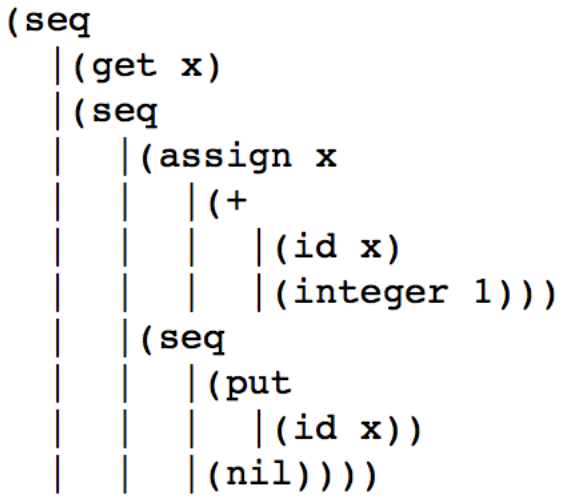
Codegen



```
#########################################################################
# walk
#########################################################################
def walk(node):
    node_type = node[0]

    if node_type in dispatch_dict:
        node_function = dispatch_dict[node_type]
        return node_function(node)

    else:
        raise ValueError("walk: unknown tree node type: " + node_type)

# a dictionary to associate tree nodes with node functions
dispatch_dict = {
    'seq'      : seq,
    'nil'      : nil,
    'assign'   : assign_stmt,
    'get'      : get_stmt,
    'put'      : put_stmt,
    'while'    : while_stmt,
    'if'       : if_stmt,
    'block'    : block_stmt,
    'integer'  : integer_exp,
    'id'       : id_exp,
    'uminus'   : uminus_exp,
    'not'      : not_exp,
    'paren'    : paren_exp,
    '+'        : binop_exp,
    '-'        : binop_exp,
    '*'        : binop_exp,
    '/'        : binop_exp,
    '=='       : binop_exp,
    '<='       : binop_exp

}

#########################################################################
```

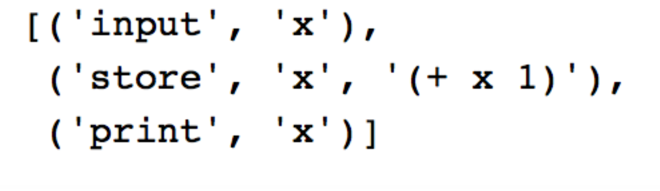
Running Codegen

Consider our AST:



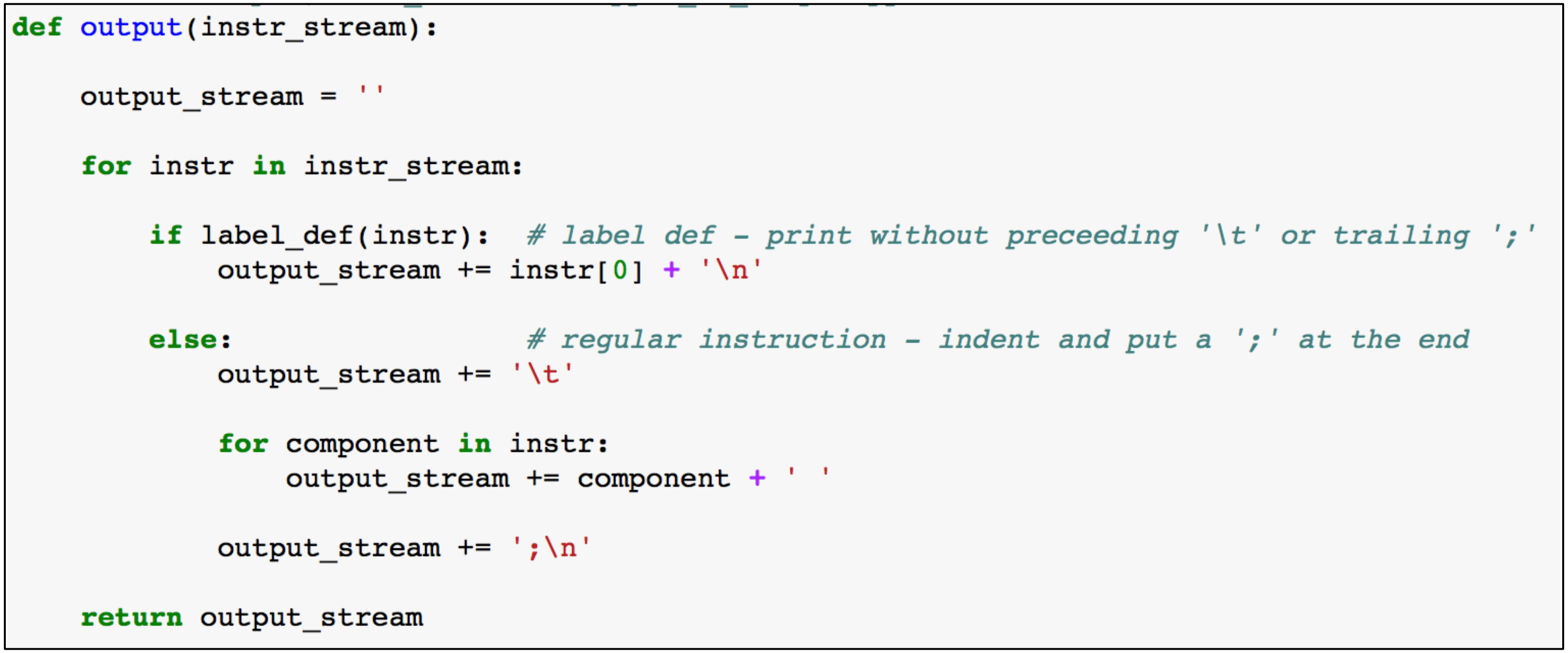
```
(seq
  |(get x)
  |(seq
  |   |(assign x
  |   |   |(+
  |   |   |   |(id x)
  |   |   |   |(integer 1)))
  |   |(seq
  |   |   |(put
  |   |   |   |(id x))
  |   |   |(nil))))
```

Generated instruction list:



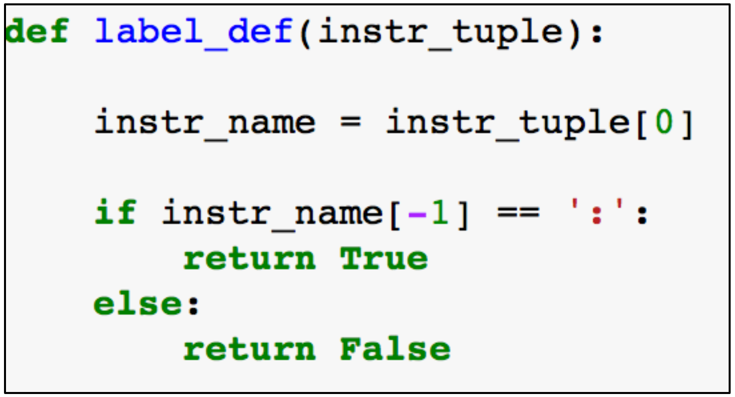
```
[('input', 'x'),
 ('store', 'x', '(+ x 1)'),
 ('print', 'x')]
```

Formatting the Output



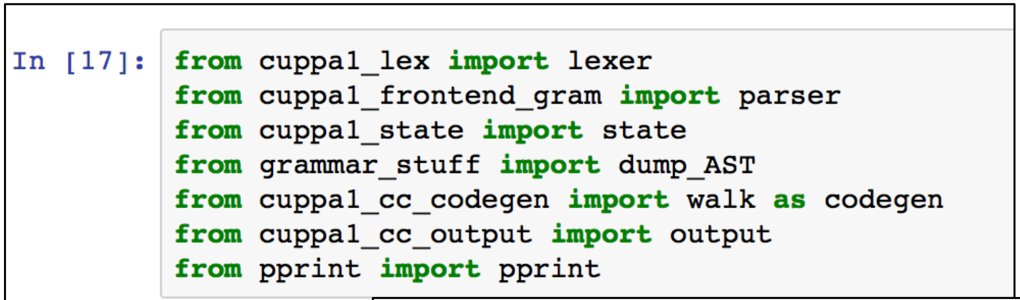
```
def output(instr_stream):

    output_stream = ''

    for instr in instr_stream:

        if label_def(instr):  # label def - print without preceeding '\t' or trailing ';'
            output_stream += instr[0] + '\n'

        else:                 # regular instruction - indent and put a ';' at the end
            output_stream += '\t'

            for component in instr:
                output_stream += component + ' '

            output_stream += ';\n'

    return output_stream
```

Convert the instruction tuple list into a printable target program.

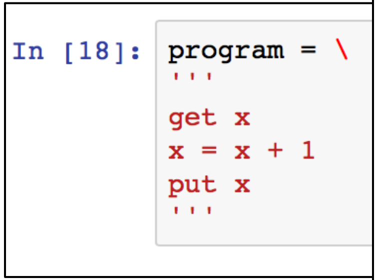


```
def label_def(instr_tuple):

    instr_name = instr_tuple[0]

    if instr_name[-1] == ':':
        return True
    else:
        return False
```

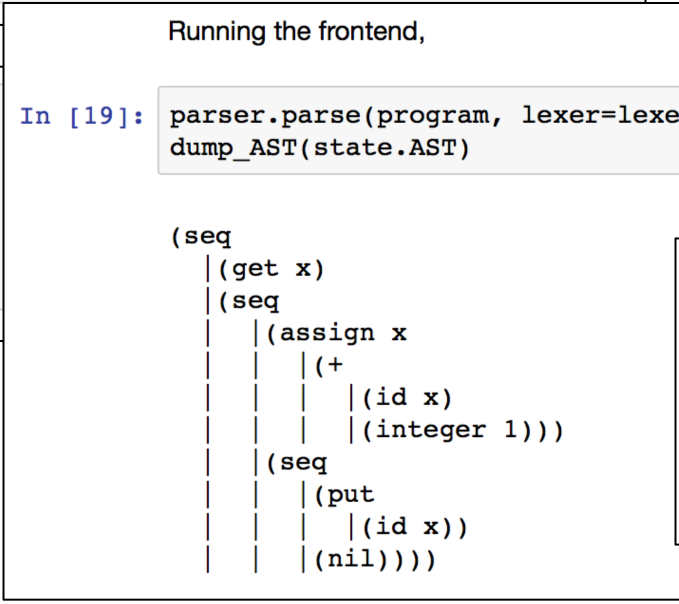
Running the Phases of the Compiler



```
In [17]: from cuppa1_lex import lexer
         from cuppa1_frontend_gram import parser
         from cuppa1_state import state
         from grammar_stuff import dump_AST
         from cuppa1_cc_codegen import walk as codegen
         from cuppa1_cc_output import output
         from pprint import pprint
```



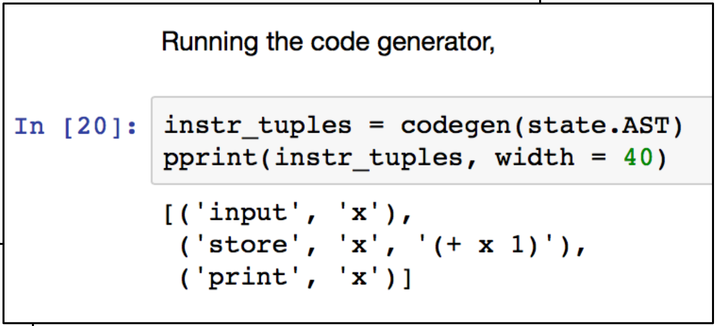
```
In [18]: program = \
         '''
         get x
         x = x + 1
         put x
         '''
```



Running the frontend,

```
In [19]: parser.parse(program, lexer=lexer)
         dump_AST(state.AST)
```

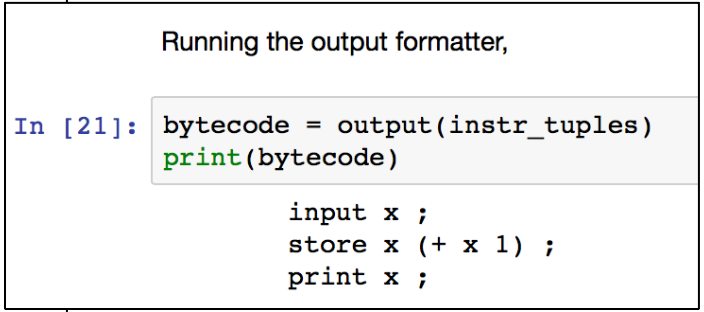
```
(seq
  |(get x)
  |(seq
  |  |(assign x
  |  |  |(+
  |  |  |  |(id x)
  |  |  |  |(integer 1)))
  |  |(seq
  |  |  |(put
  |  |  |  |(id x))
  |  |  |(nil))))
```



Running the code generator,

```
In [20]: instr_tuples = codegen(state.AST)
         pprint(instr_tuples, width = 40)
```

```
[('input', 'x'),
 ('store', 'x', '(+ x 1)'),
 ('print', 'x')]
```

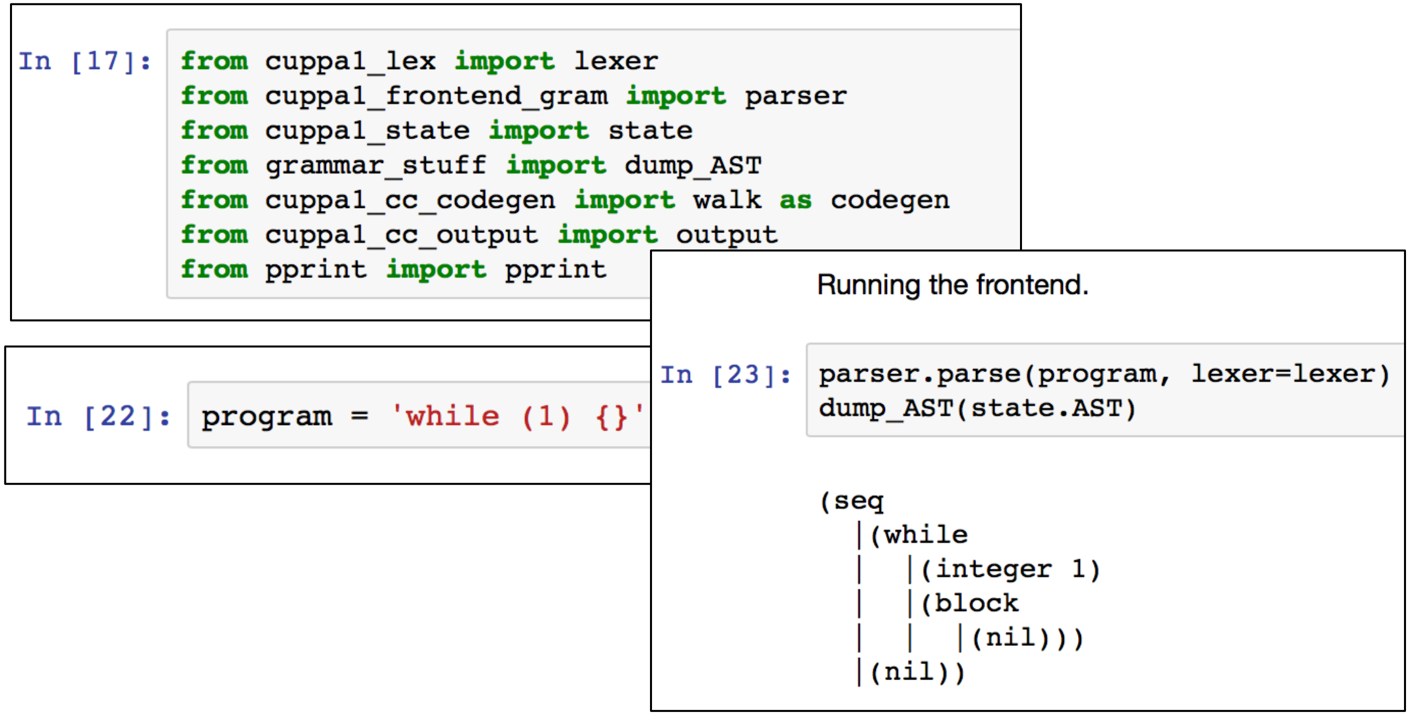


Running the output formatter,

```
In [21]: bytecode = output(instr_tuples)
         print(bytecode)
```

```
          input x ;
          store x (+ x 1) ;
          print x ;
```

Running the Phases of the Compiler

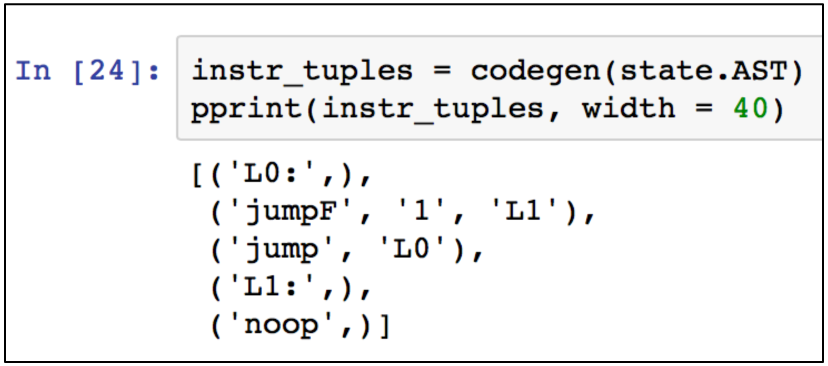


```
In [17]: from cuppa1_lex import lexer
         from cuppa1_frontend_gram import parser
         from cuppa1_state import state
         from grammar_stuff import dump_AST
         from cuppa1_cc_codegen import walk as codegen
         from cuppa1_cc_output import output
         from pprint import pprint
```

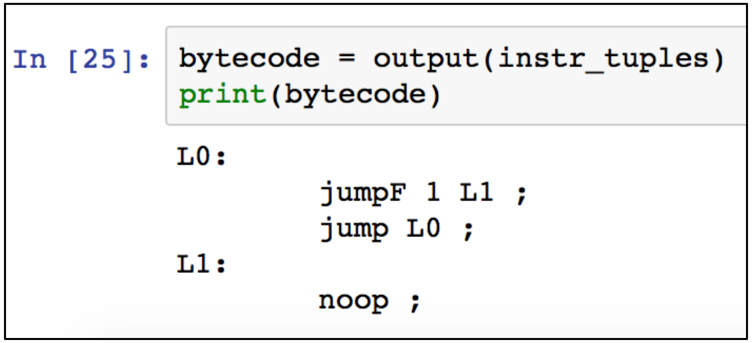
```
In [22]: program = 'while (1) {}'
```

Running the frontend.

```
In [23]: parser.parse(program, lexer=lexer)
         dump_AST(state.AST)

         (seq
           |(while
           |   |(integer 1)
           |   |(block
           |   |   |(nil)))
           |(nil))
```

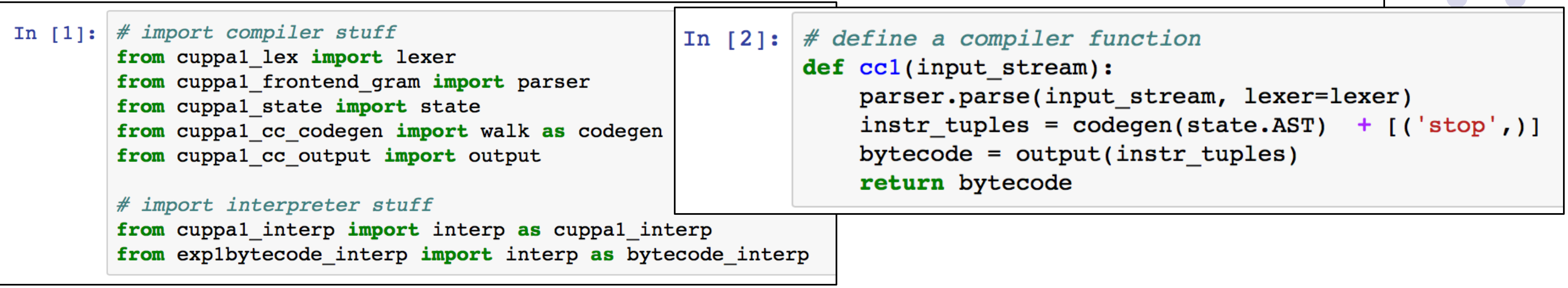


```
In [24]: instr_tuples = codegen(state.AST)
         pprint(instr_tuples, width = 40)

         [('L0:',),
          ('jumpF', '1', 'L1'),
          ('jump', 'L0'),
          ('L1:',),
          ('noop',)]
```



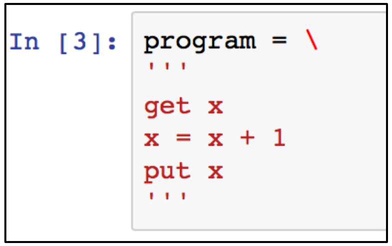
```
In [25]: bytecode = output(instr_tuples)
         print(bytecode)

         L0:
                 jumpF 1 L1 ;
                 jump L0 ;
         L1:
                 noop ;
```

Compilation vs Interpretation



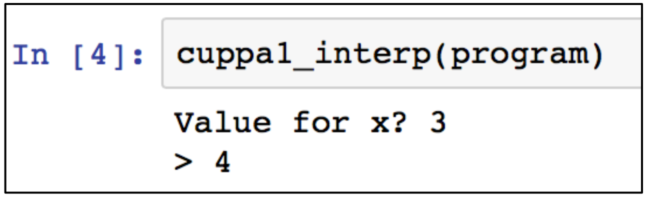
```
In [1]: # import compiler stuff
        from cuppa1_lex import lexer
        from cuppa1_frontend_gram import parser
        from cuppa1_state import state
        from cuppa1_cc_codegen import walk as codegen
        from cuppa1_cc_output import output

        # import interpreter stuff
        from cuppa1_interp import interp as cuppa1_interp
        from exp1bytecode_interp import interp as bytecode_interp
```

```
In [2]: # define a compiler function
        def cc1(input_stream):
            parser.parse(input_stream, lexer=lexer)
            instr_tuples = codegen(state.AST)  + [('stop',)]
            bytecode = output(instr_tuples)
            return bytecode
```



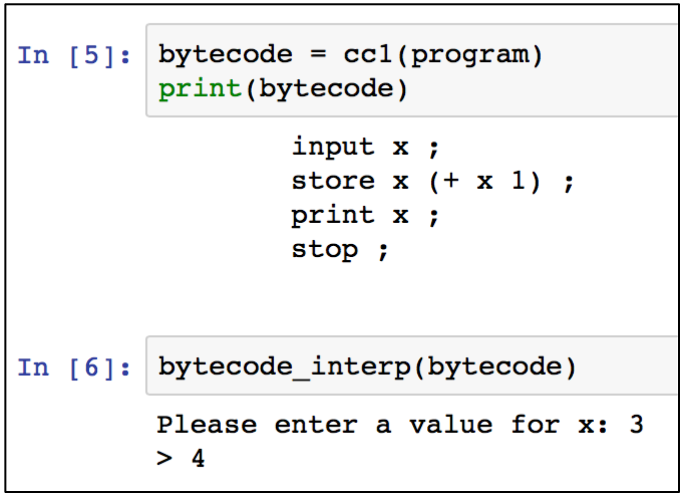
```
In [3]: program = \
        '''
        get x
        x = x + 1
        put x
        '''
```

Interpretation



```
In [4]: cuppa1_interp(program)

        Value for x? 3
        > 4
```

Compilation



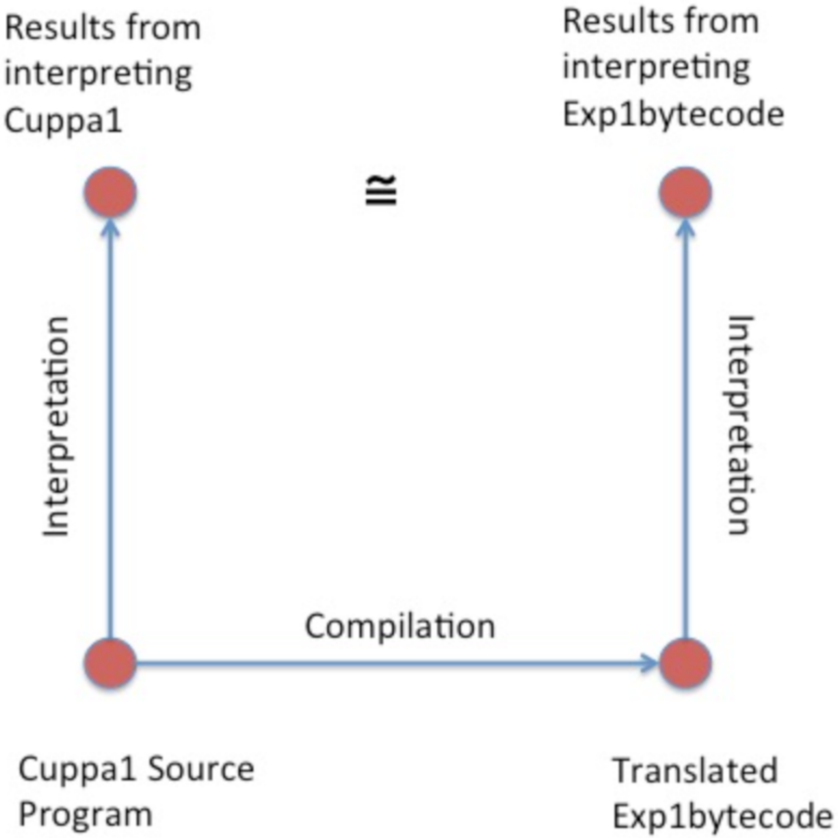
```
In [5]: bytecode = cc1(program)
        print(bytecode)

                    input x ;
                    store x (+ x 1) ;
                    print x ;
                    stop ;


In [6]: bytecode_interp(bytecode)

        Please enter a value for x: 3
        > 4
```

Compiler Correctness

- We now have two ways to execute a Cuppa1 program:
  - We can interpret the program directly with the Cuppa1 interpreter.
  - We can first translate the Cuppa1 program into Exp1bytecode and then execute the bytecode in the abstract bytecode machine.

👍 A compiler is *correct* if the translated program, when executed, gives the same results as the interpreted program.

Compiler Correctness



Results from interpreting Cuppa1

≅

Results from interpreting Exp1bytecode

Interpretation

Interpretation

Compilation

Cuppa1 Source Program

Translated Exp1bytecode

Assignment

- Assignment #6 – see webpage.