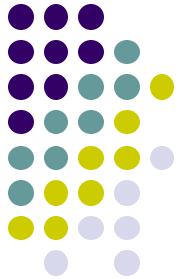
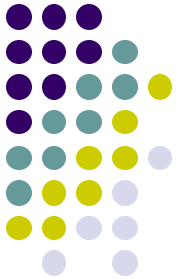
A Basic Compiler



- 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.

Reading

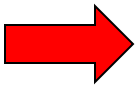
- Chap 6



A Basic Compiler

```
stmt_list : (stmt)*

stmt : ID = exp ;?
     | get ID ;?
     | put exp ;?
     | while \( exp \) stmt
     | if \( exp \) stmt (else stmt)?
     | \{ stmt_list \}

exp : exp_low
exp_low : exp_med ((== | =<) exp_med)*
exp_med : exp_high ((+ | -) exp_high)*
exp_high : primary ((\* | /) primary)*

primary : INTEGER
        | ID
        | \( exp \)
        | - primary
        | not primary

ID : <any valid variable name>
INTEGER : <any valid integer number>
```

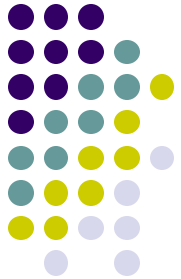
Cuppa1



```
instr_list : (labeled_instr)*

labeled_instr : label_def instr
              | instr

label_def : label \:

instr : print exp ;
      | store var exp ;
      | input var ;
      | jumpt exp label ;
      | jumpf exp label ;
      | jump label ;
      | stop ;
      | noop ;

exp : + exp exp
    | - exp exp?
    | \* exp exp
    | / exp exp
    | ! exp
    | == exp exp
    | =< exp exp
    | \( exp \)
    | var
    | num

label : <any valid label name>
var : <any valid variable name>
num : <any valid integer number>
```

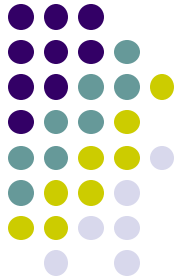
Exp1bytecode

A Basic Compiler



- Consider for example the AST pattern for the assignment statement in Cuppa1,
  - (‘ASSIGN’, (‘ID’, 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.

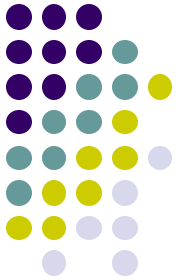
A Basic Compiler



- 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', ('ID', name), exp)   ⇒   store <name> <exp>;
('PUT', exp)                    ⇒   print <exp>;
('GET', ('ID', name))           ⇒   input <name>;
```

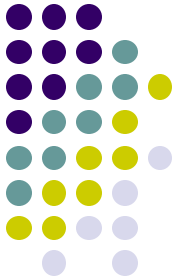
A Basic Compiler



- And for the expressions we have,

```
('PLUS', c1, c2)     ⇒   + <c1> <c2>
('MINUS', c1, c2)    ⇒   - <c1> <c2>
('MUL', c1, c2)      ⇒   * <c1> <c2>
('DIV', c1, c2)      ⇒   / <c1> <c2>
('EQ', c1, c2)       ⇒   == <c1> <c2>
('LE', c1, c2)       ⇒   =< <c1> <c2>
('ID', name)         ⇒   <name>
('INTEGER', value)   ⇒   <value>
('UMINUS', value)    ⇒   - <value>
('NOT', value)       ⇒   ! <value>
```

A Basic Compiler

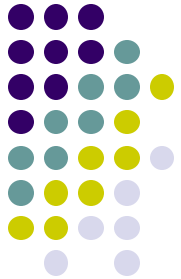


- We have to “simulate” the behavior of the Cuppa1 “while” loop with jump statements in Exp1bytecode.
- One way to translate the AST pattern for the while loop into a code pattern in Exp1bytecode is,

```
('WHILE', cond, body)  ⇒   L1:
                               jumpf <cond> L2;
                               <body>
                               jump L1;
                           L2:
                               noop;
```

Note: labels cannot appear by themselves, so we have to put a noop instruction here in order to make this a legal pattern.

A Basic Compiler



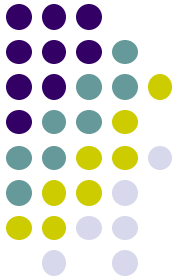
- We can do something similar with if-then statements,

```
('IF', cond, then_stmt, ('NIL',))  ⇒      jumpf <cond> L;
                                          <then_stmt>
                                       L:
                                          noop;
```

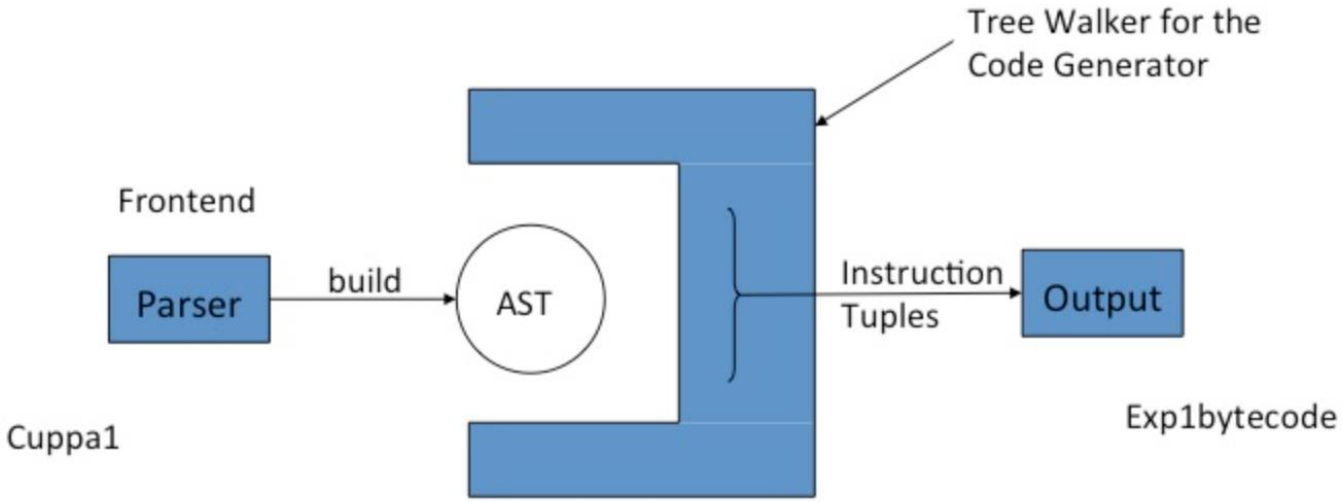
- Finally, adding the else-statement to the if-then statement we have,

```
('IF', cond, then_stmt, else_stmt)  ⇒     jumpf <cond> L1;
                                          <then_stmt>
                                          jump L2;
                                       L1:
                                          <else_stmt>
                                       L2:
                                           noop;
```

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,

```
$ python3
>>> from cuppa1_fe import parse
>>> from dumpast import dumpast
>>> ast = parse("get x; x = x + 1; put x;")
>>> dumpast(ast)

(STMTLIST
  |[
  |  |(GET
  |  |  |(ID x))
  |  |(ASSIGN
  |  |  |(ID x)
  |  |  |(PLUS
  |  |  |  |(ID x)
  |  |  |  |(INTEGER 1)))
  |  |(PUT
  |  |  |(ID x))])
>>>
```

We can easily apply our pattern translations to generate Exp1bytecode:

```
('ASSIGN', ('ID', name), exp)    ⇒    store <name> <exp>;
('PUT', exp)                     ⇒    print <exp>;
('GET', ('ID', name))            ⇒    input <name>;
```

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*
  - Cuppa1 expression patterns will generate Exp1bytecode expressions returned as *strings*.

cuppa1_codegen.py

Codegen

- Recall the pattern translation,
  - (‘GET’, (‘ID’, name)) => input <name>;
- The codegen tree walker has a function for that,

```
def get_stmt(node):

    (GET, (ID, name)) = node

    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.

Note: We use Python’s ability to do pattern matching on tuples!
Note: We have the <name> = name identity translation.

cuppa1_codegen.py

Codegen

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

```
def assign_stmt(node):

    (ASSIGN, (ID, name), exp) = node

    exp_code = walk(exp)
    code = [('store', name, exp_code)]
    return code
```

Note: We have the translation <exp> = walk(exp).

cuppa1_codegen.py

Codegen

- Recall the pattern translation,

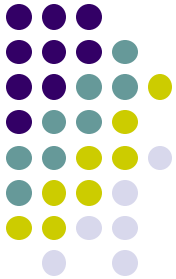
```
('WHILE', cond, body)  ⇒   L1:
                               jumpf <cond> L2;
                               <body>
                               jump L1;
                           L2:
                               noop;
```

- The codegen tree walker has a function for that,

```
def while_stmt(node):

    (WHILE, cond, body) = node

    top_label = label()
    bottom_label = label()
    cond_code = walk(cond)
    body_code = walk(body)
    code = []

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

    return code
```


Note:
<cond> = walk(cond)
<body> = walk(body)

cuppa1_codegen.py



Codegen

- Recall the pattern translation for binops,

```
('PLUS', c1, c2)     ⇒   + <c1> <c2>
('MINUS', c1, c2)    ⇒   - <c1> <c2>
('MUL', c1, c2)      ⇒   * <c1> <c2>
('DIV', c1, c2)      ⇒   / <c1> <c2>
('EQ', c1, c2)       ⇒   == <c1> <c2>
('LE', c1, c2)       ⇒   =< <c1> <c2>
```

- The codegen tree walker has a function for that,
  - <c1> = walk(c1)
  - <c2> = walk(c2)

```
def binop_exp(node):

    (OP, c1, c2) = node

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

    if OP == 'PLUS':
        OPSYM = '+'
    elif OP == 'MINUS':
        OPSYM = '-'
    elif OP == 'MUL':
        OPSYM = '*'
    elif OP == 'DIV':
        OPSYM = '/'
    elif OP == 'EQ':
        OPSYM = '=='
    elif OP == 'LE':
        OPSYM = '=<'
    else:
        raise ValueError('unknown operator: ' + OP)

    code = OPSYM + ' ' + lcode + ' ' + rcode
    return code
```

cuppa1_codegen.py

Codegen

- What remains to be looked at is how the tree walker deals with statement lists.
- And how the walker deals with Nil nodes in a statement.

```
def stmtlst(node):

    (STMTLIST, lst) = node

    outlst = []
    for stmt in lst:
        outlst += walk(stmt)
    return outlst
```

```
def nil(node):

    (NIL,) = node

    return []
```

Codegen

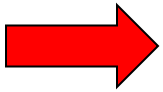
cuppa1_codegen.py

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

    if node_type in dispatch:
        node_function = dispatch[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 = {
    'STMTLIST'  : stmtlst,
    '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,
    'PLUS'      : binop_exp,
    'MINUS'     : binop_exp,
    'MUL'       : binop_exp,
    'DIV'       : binop_exp,
    'EQ'        : binop_exp,
    'LE'        : binop_exp,
}
```

Running Codegen

Consider our AST:

```
(STMTLIST
  |[
  |  |(GET
  |  |  |(ID x))
  |  |(ASSIGN
  |  |  |(ID x)
  |  |  |(PLUS
  |  |  |  |(ID x)
  |  |  |  |(INTEGER 1)))
  |  |(PUT
  |  |  |(ID x))])
```



Generated instruction list:

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

Running Codegen

```
Python 3.8.12 (default, Sep 10 2021, 00:16:05)
[GCC 7.5.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from cuppa1_fe import parse
>>> from cuppa1_codegen import walk
>>> from dumpast import dumpast
>>> import pprint
>>> pp = pprint.PrettyPrinter()
>>> ast = parse("get x; x = x+1; put x")
>>> dumpast(ast)

(STMTLIST
  |[
  |  |(GET
  |  |  |(ID x))
  |  |(ASSIGN
  |  |  |(ID x)
  |  |  |(PLUS
  |  |  |  |(ID x)
  |  |  |  |(INTEGER 1)))
  |  |(PUT
  |  |  |(ID x))])
>>> code = walk(ast)
>>> pp.pprint(code)
[('input', 'x'), ('store', 'x', '+ x 1'), ('print', 'x')]
>>>
```

Note: everything is a string in the instruction tuple list making code generation very easy.

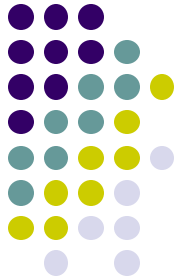
Formatting the Output

```
def output(instr_stream):

    output_stream = ''

    for instr in instr_stream:

        if label_def(instr):  # label def - without preceding '\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_type = instr_tuple[0]

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

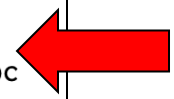
Running the Phases of the Compiler

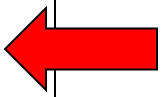


```
Python 3.8.12 (default, Sep 10 2021, 00:16:05)
[GCC 7.5.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from cuppa1_fe import parse
>>> from dumpast import dumpast
>>> from cuppa1_codegen import walk
>>> from cuppa1_output import output
>>> ast = parse("get x; x = x+1; put x")
>>> dumpast(ast)

(STMTLIST
  |[
  |  |(GET
  |  |  |(ID x))
  |  |(ASSIGN
  |  |  |(ID x)
  |  |  |(PLUS
  |  |  |  |(ID x)
  |  |  |  |(INTEGER 1)))
  |  |(PUT
  |  |  |(ID x))])
>>> code = walk(ast)
>>> code
[('input', 'x'), ('store', 'x', '+ x 1'), ('print', 'x')]
>>> output(code)
'\tinput x ;\n\tstore x + x 1 ;\n\tprint x ;\n'
>>> print(output(code))
    input x ;
    store x + x 1 ;
    print x ;
```

Running the Compiler

```
### source program
$ cat loop.txt
x = 3; while (x) {put x; x = x-1}

### compile the program into Exp1bytecode
$ python3 cuppa1_cc_basic.py -o loop.bc loop.txt
$ cat loop.bc
   store x 3 ;
L0:
   jumpf x L1 ;
   print x ;
   store x - x 1 ;
   jump L0 ;
L1:
   noop ;
   stop ;

### run the Exp1bytecode virtual machine on translated code
$ python3 ../../chap04/exp1bytecode/exp1bytecode_interp.py loop.bc
3
2
1

### run the Cuppa1 interpreter on original program
$ python3 ../../chap05/cuppa1/cuppa1_interp.py loop.txt
3
2
1
$
```



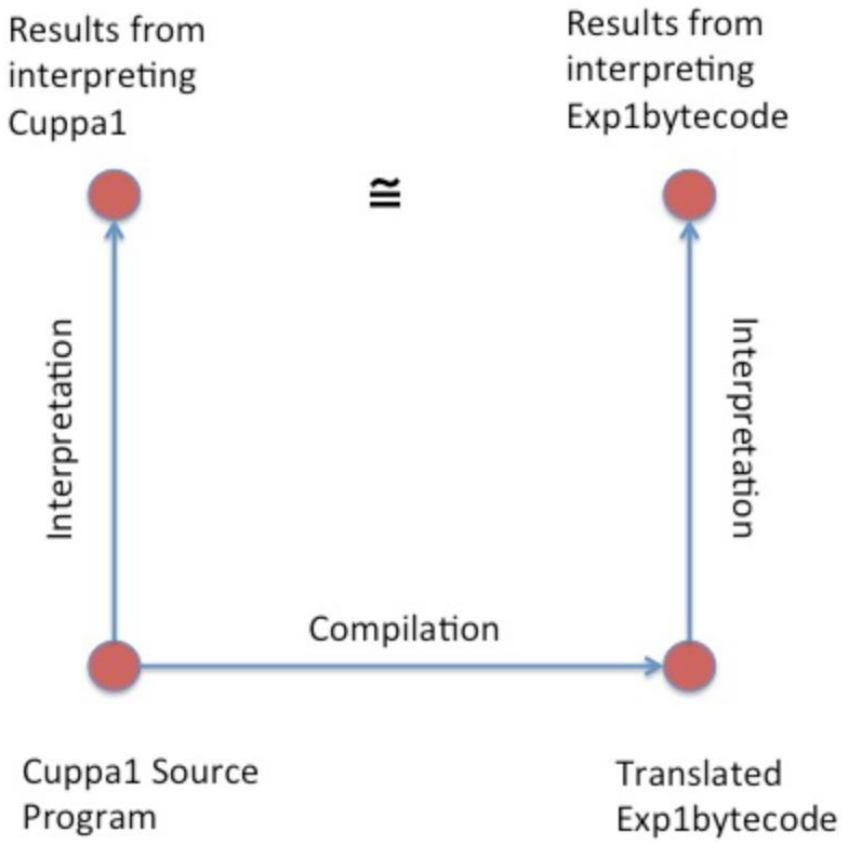


Compiler Correctness

- 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



Assignment

- Assignment #3 – see BrightSpace.