An Optimizing Compiler

- The big difference between interpreters and compilers is that compilers have the ability to think about how to translate a source program into target code in the most effective way.
- Usually that means trying to translate the program in such a way that it executes as fast as possible on the target machine.
- This usually implies either one or both of the following tasks:
  - Rewrite the AST so that it represents a more efficient program – Tree Rewriting
  - Reorganize the generated instructions so that they represent the most efficient target program possible
- This is referred to as *Optimization*.
- There are many optimization techniques available to compilers in addition to the two mentioned above:
  - Register allocation, loop optimization, common subexpression elimination, dead code elimination, etc
Reading

- Chap 6
An Optimizing Compiler

In our optimizing compiler we study:

- Tree rewriting in the context of constant folding, and
- Target code optimization in the context of peephole optimization.
Tree Rewriting

- So far our applications only have looked at the AST as an immutable data structure, e.g.
  - The Cuppa1 interpreter used it as an abstract representation of the original program
  - PrettyPrinter used it to regenerate programs
- But there are many cases where we actually want to transform the AST
  - Consider constant folding
Constant Folding

- Constant folding is an optimization that tries to find arithmetic operations in the source program that can be performed at compile time rather than runtime.
Constant Folding

In constant folding we look at the operations in arithmetic expressions and if the operands are constants then we perform the operation and replace the AST with a result node.

\[ x = 10 + 5 \]

\[ x = 15 \]
Constant Folding

- One way to view constant folding is as an AST rewriting.
- Here the AST for the expression 10 + 5 is replaced by an AST node for the constant 15.
- In order to accomplish this, we need to walk the AST for a program and look for patterns that allow us to rewrite the tree.
- This is very similar to code generation tree walker where we walked the tree and looked for AST patterns that we could translate into Exp1bytecode.
- The big difference being that in the constant folder we will be *returning the rewritten tree from the tree walker* rather than bytecode as in the code generator.
Constant Folding Walker

```python
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' : plus_exp,
    'MINUS' : minus_exp,
    'MUL' : mult_exp,
    'DIV' : div_exp,
    'EQ' : eq_exp,
    'LE' : le_exp,
}
```

cuppa1_fold.py
```python
def plus_exp(node):
    (PLUS, c1, c2) = node
    newc1 = walk(c1)
    newc2 = walk(c2)

    # if the children are constants -- fold!
    if newc1[0] == 'INTEGER' and newc2[0] == 'INTEGER':
        return ('INTEGER', newc1[1] + newc2[1])
    else:
        return ('PLUS', newc1, newc2)
```

```python
>>> from cuppa1_state import state
>>> from cuppa1_fold import walk
>>> from dumpast import dumpast
>>> ast = ('PLUS', ('INTEGER', 1), ('INTEGER', 2))
>>> dumpast(ast)
(PLUS
  |(INTEGER 1)
  |(INTEGER 2))
>>> new_ast = walk(ast)
>>> dumpast(new_ast)
(INTEGER 3)
```
```
def eq_exp(node):
    (EQ, c1, c2) = node
    newc1 = walk(c1)
    newc2 = walk(c2)

    # if the children are constants -- fold!
    if newc1[0] == 'INTEGER' and newc2[0] == 'INTEGER':
        return ('INTEGER', 1 if newc1[1] == newc2[1] else 0)
    else:
        return ('EQ', newc1, newc2)

def stmt_lst(node):
    (STMTLIST, lst) = node
    newlst = []
    for stmt in lst:
        newlst.append(walk(stmt))
    return ('STMTLIST', newlst)
```
Constant Folding

Let's try our walker on our assignment statement example to see if it does what we claim it does,

```python
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_state import state
>>> from cuppa1_fold import walk
>>> ast = ('ASSIGN', ('ID', 'x'), ('PLUS', ('INTEGER', 10), ('INTEGER', 5)))
>>> from dumpast import dumpast
>>> dumpast(ast)

(ASSIGN
 | (ID x)
 | (PLUS
 |  | (INTEGER 10)
 |  | (INTEGER 5)))

>>> new_ast = walk(ast)
>>> dumpast(new_ast)

(ASSIGN
 | (ID x)
 | (INTEGER 15))
>>> ```
We insert our constant folding tree rewriting phase into our Cuppa1 compiler as a tree walker.
Peephole Code Optimization

- A peephole optimizer improves the generated code by reorganizing the generated instructions.
- If you recall the code generator for our Cuppa1 compiler translates Cuppa1 AST patterns into Exp1bytecode patterns and simply composes the generated bytecode patterns into a list of instructions.
- That can lead to very silly looking code.
Peephole Code Optimization

Consider:

```plaintext
get x;
y = 1;
while (1 <= x)
{
    y = y * x;
    x = x - 1;
}
put y;
```

```
input x ;
store y 1 ;
L13:      jumpF (<= 1 x) L14 ;
store y (* y x) ;
store x (- x 1) ;
jump L13 ;
L14:      noop ;
print y ;
stop ;
```
Peephole Code Optimization

There is a rule for that:

```
L:
  noop
  <other instruction>
=>
L:
  <other instruction>
```

```
input x;
store y 1;
L13:
  jumpF (<= 1 x) L14;
  store y (* y x);
  store x (~ x 1);
  jump L13;
L14:
  noop;
  print y;
  stop;
```
Peephole Code Optimization

Consider:

```
get x
r = x - 2*(x/2)
if (not r)
  if (x <= 10)
    put x
...
```

```
input x;
store r (- x (* 2 (/ x 2))) ;
jumpF !r L15 ;
jumpF (<= x 10) L16 ;
print x;

L16:
  noop ;
L15:
  noop ;
  stop ;
```

Even Sillier!
Peephole Code Optimization

There is a rule for that:

```
L1:
    noop

L2:
    <other instruction>
=>

L2: -- with L1 backpatched to L2
    <other instruction>
```

```
input x;
  store r (- x (* 2 (/ x 2))) ;
  jumpF !r L15 ;
  jumpF (<= x 10) L16 ;
  print x ;
L16:
    noop ;
L15:
    noop ;
    stop ;
```
Peephole Code Optimization

- One way to think of a peephole optimizer is as a window (the peephole) which we slide across the generated instructions *repeatedly* and apply *rewrite rules* like the ones we developed above to the code within the window.
- The peephole optimizer terminates once no longer any code is being rewritten.
- The repeated nature of the process is necessary because applying one rewrite rule to the instruction list can expose opportunities to apply other rewrite rules.
- So, we need to keep sliding the window across the instructions until no further rewrites are possible.
Peephole Code Optimization

Rewrite Rules
P1 => P1'
P2 => P2'
P3 => P3'
...
Peephole Code Optimization

Rewrite Rules:

```python
# rewrite rule:
# *L:
#    noop
#    <some other instr>
# =>
# *L:
#    <some other instr>
if pattern_fits(3, ix, instr_stream) \ 
    and label_def(curr_instr) \ 
    and relative_instr(1, ix, instr_stream)[0] == 'noop' \ 
    and not label_def(relative_instr(2, ix, instr_stream)):
    # delete noop
    instr_stream.pop(ix+1)
    change = True
```

```python
# rewrite rule:
# *L1:
#    noop
# L2:
# =>
# *L2: -- with L1 backpatched to L2 in instr_stream
elif pattern_fits(3, ix, instr_stream) \ 
    and label_def(curr_instr) \ 
    and relative_instr(1, ix, instr_stream)[0] == 'noop' \ 
    and label_def(relative_instr(2, ix, instr_stream)):
    label1 = get_label_from_def(curr_instr)
    label2 = get_label_from_def(relative_instr(2, ix, instr_stream))
    backpatch_label(label1, label2, instr_stream)
    instr_stream.pop(ix)
    instr_stream.pop(ix)
    change = True
```
Peephole Code Optimization

def peephole_opt(instr_stream):
    ix = 0
    change = False
    while(True):
        curr_instr = instr_stream[ix]

        ### compute some useful predicates on the current instruction
        is_first_instr = ix == 0
        is_last_inst = ix+1 == len(instr_stream)
        has_label = True if not is_first_instr and label_def(instr_stream[ix-1]) else False

        <** rewrite rules here **>

        ### advance ix
        if is_last_instr and not change:
            break

        elif is_last_instr:
            ix = 0
            change = False

        else:
            ix += 1
We insert our peephole optimizer between the code generator and the output phase.
from argparse import ArgumentParser
from cuppa1_fe import parse
from cuppa1_codegen import walk as codegen
from cuppa1_fold import walk as fold
from cuppa1_output import output
from cuppa1_output import peephole_opt

def cc(input_stream, opt = False):
    try:
        ast = parse(input_stream)
        if opt:
            ast = fold(ast)  # constant fold optimizer
        instr_stream = codegen(ast) + [['stop',]]
        if opt:
            peephole_opt(instr_stream)  # peephole optimizer
        bytecode = output(instr_stream)
        return bytecode
    except Exception as e:
        print('error: ' + str(e))
        return None
Testing the Compiler

```
$ cat even.txt
get x
r = x - 2*(x/2) // integer division!
if (not r)
  if (x <= 10)
    put x

$ python3 cuppa1_cc.py -0 -o even.bc even.txt
$ cat even.bc
  input x ;
  store r - x * 2 / x 2 ;
  jump !r L1 ;
  jumpf <= x 10 L1 ;
  print x ;
L1:
  stop ;
$```