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
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
  - Bytecode interpreter used it to execute instructions
  - 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 a 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 Cuppa1 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 \textit{returning the rewritten tree from the tree walker} rather than bytecode as in the code generator.
Constant Folding

Consider:

```python
from grammar_stuff import assert_match, dump_AST
from cuppa1_cc_fold import *

def plus_exp(node):
    (OP, c1, c2) = node
    assert_match(OP, '+')
    ltree = walk(c1)
    rtree = walk(c2)

    # if the children are constants -- fold!
    if ltree[0] == 'integer' and rtree[0] == 'integer':
        return ('integer', ltree[1] + rtree[1])
    else:
        return ('+', ltree, rtree)
```

```
In [47]: plus_node = ('+', ('integer', 10), ('integer', 1))
dump_AST(plus_node)
```

```
(+  
 |(integer 10)  
 |(integer 1))
```

```
In [48]: plus_exp(plus_node)
Out[48]: ('integer', 11)
```
Constant Folding

Consider:

def eq_exp(node):
    (OP, c1, c2) = node
    assert_match(OP, '==')
    ltree = walk(c1)
    rtree = walk(c2)

    # if the children are constants -- fold!
    if ltree[0] == 'integer' and rtree[0] == 'integer':
        return ('integer', 1 if ltree[1] == rtree[1] else 0)
    else:
        return ('==', ltree, rtree)

def assign_stmt(node):
    (ASSIGN, name, exp) = node
    assert_match(ASSIGN, 'assign')
    exp_tree = walk(exp)

    return ('assign', name, exp_tree)
Constant Folding

Consider:

cuppa1_cc_fold.py

```python
# 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,
    '+' : plus_exp,
    '-' : minus_exp,
    '*' : mult_exp,
    '/' : div_exp,
    '==' : eq_exp,
    '<=' : le_exp
}
```
Let's try our walker on our assignment statement example to see if it does what we claim it does,

```python
In [50]:
stmt = ('assign', 'x', ('+', ('integer', 10), ('integer', 5)))
dump_AST(stmt)

(assign x
     (+
      | (integer 10)
      | (integer 5)))

In [51]:
from cuppal_cc_fold import walk as fold

In [52]:
new_stmt = fold(stmt)
dump_AST(new_stmt)

(assign x
     | (integer 15))
```
As an example we insert a 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:

```python
In [53]: from cuppal_examples import fact
In [54]: print(fact)

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

```python
In [55]: bytecode = ccl(fact)
In [56]: print(bytecode)

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;
```

Really Silly!
Peephole Code Optimization

There is a rule for that:

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

```python
In [55]: bytecode = ccl(fact)
In [56]: print(bytecode)
   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;
```

```python
In [57]: new_bytecode = \
   ...:   input x;
   ...:   store y 1;
   ...:   L13:
   ...:     jumpF (<= 1 x) L14;
   ...:     store y (* y x);
   ...:     store x (- x 1);
   ...:     jump L13;
   ...:   L14:
   ...:     print y;
   ...:     stop;
   ...:
```
Peephole Code Optimization

Consider:

```python
In [58]:

print_even = 

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

```python
In [60]:

bytecode = ccl(print_even)

In [61]:

print(bytecode)
```

```
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>

In [60]:
    bytecode = ccl(print_even)

In [61]:
    print(bytecode)

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

    L16:
        noop ;
    L15:
        noop ;
        stop ;

In [62]:
    new_bytecode = 
        ... 
        input x ;
        store r (- x (* 2 (/ x 2))) ;
        jumpF !r L15 ;
        jumpF (<= x 10) L15 ;
        print x ;
    L15:
        ... 
        stop ;
        ...
Peepholes 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
```
Peephole Code Optimization

# apply peephole optimization. The instruction tuple format is:
# (instr_name_str, [param_str1, param_str2, ...])
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_instr = 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
Optimizing Compiler Architecture

- We insert our peephole optimizer between the code generator and the output phase.
Optimizing Compiler

Top-level Driver Function

```python
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_fold import walk as fold
from cuppa1_cc_output import output
from cuppa1_cc_output import peephole_opt

def cc(input_stream, opt=False):
    
    # initialize the state object
    state.initialize()

    # build the AST
    parser.parse(input_stream, lexer=lexer)

    # run the constant fold optimizer
    if opt:
        state.AST = fold(state.AST)

    # generate the list of instruction tuples
    instr_stream = codegen(state.AST) + ["'stop',"]

    # run the peephole optimizer
    if opt:
        peephole_opt(instr_stream)

    # output the instruction stream
    bytecode = output(instr_stream)

    return bytecode
```

cuppa1_cc.py