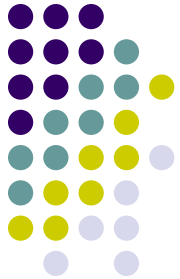
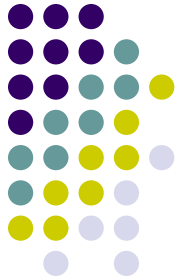
Compiling Scoped Code



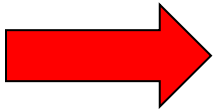
- Compiling scoped code raises a set of issues because most low-level languages do not support scoping.
- Also, our low-level programming languages do not support declarations, you simply start using a variable name, therefore, all declarations need to be resolved in the source language processor.
- Declarations in the source language usually just become assignment statements in our low-level target language.

Compiling Scoped Code



- As long as all variables of the source program are declared globally there is no problem:

```
// computes the factorial of x
declare x = 3;
declare y = 1;
while (1 <= x) {
      y = y * x;
      x = x - 1;
}
put y;
```



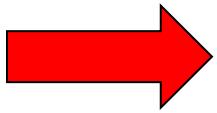
```
      store R$x 3;
      store R$y 1;
L0:
      jumpF (<= 1 R$x) L1;
      store R$y (* R$y R$x);
      store R$x (- R$x 1);
      jump L0;
L1:
      noop
      print R$y;
      stop;
```

Here R$ is just a variable name prefix…it could be any string…it will become significant when considering scoping.

Compiling Scoped Code

- Now consider the following program:

```
declare x = 1;
{
    declare x = 2;
    put x;
}
put x;
```



```
store R$x 1;
store R$x 2;
print R$x;
print R$x;
stop;
```

Expected output: 2 1

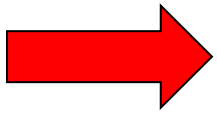
Actual output: 2 2

If we are not careful in our translation our target programs will be incorrect.

Compiling Scoped Code

- Now consider the following program:

```
declare x = 1;
{
    declare x = 2;
    put x;
}
put x;
```



```
store R$x 1;
store R$$x 2;
print R$$x;
print R$x;
stop;
```

Expected output: 2 1

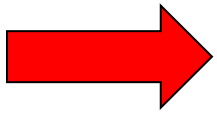
Actual output: 2 1

In the target language we simulate scoping by adding a distinct variable prefix for each scope: R$ - global scope, R$$ first nested scope, R$$$ second nested scope, etc.

Compiling Scoped Code

- Now consider the following program:

```
declare x = 1;
{
    declare x = 2;
    put x;
}
{
    declare x = 3;
    put x;
}
put x;
```



```
store R$x 1;
store R$$x 2;
print R$$x;
store R$$x 3;
print R$$x;
print R$x;
stop;
```

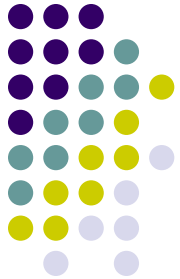
Expected output: 2 3 1

Actual output: 2 3 1

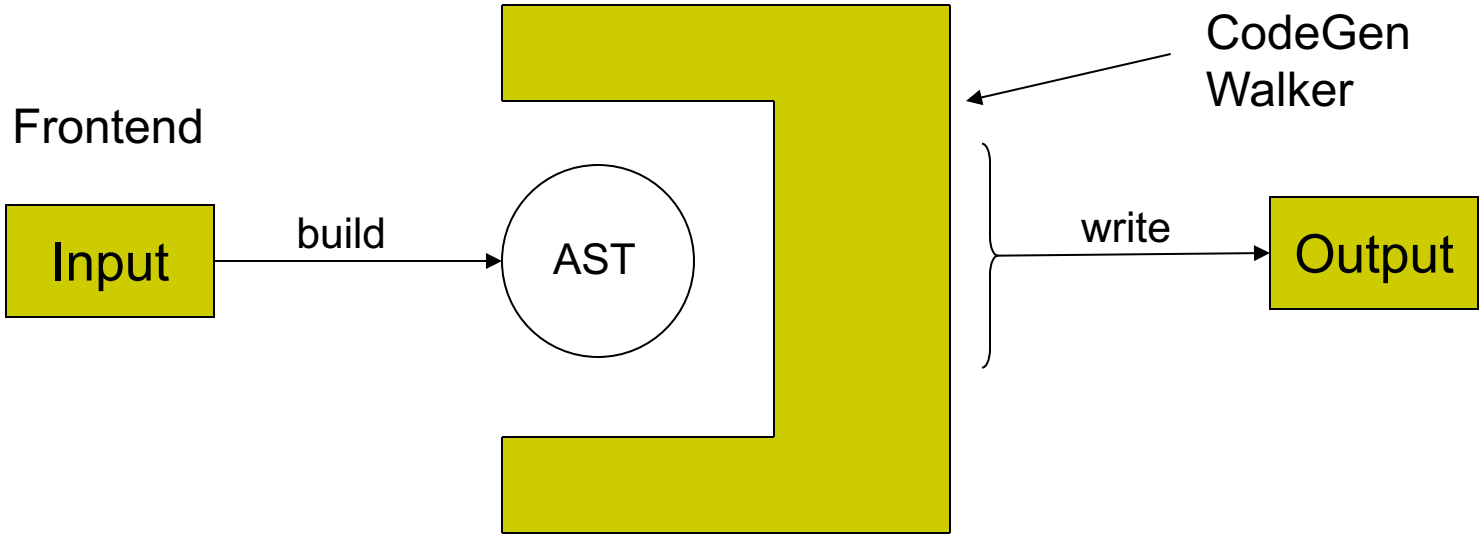
This still works because two nested scopes at the same level can never be active at the same time.

You have to be careful that the variable is properly initialized if you use it for the second scope.

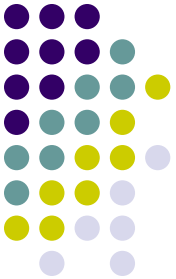
Compiler: Cuppa2 → Exp1bytecode



- The compiler follows the Cuppa1 architecture.
- Uses the same symbol table as the interpreter, but computes the variable prefix instead of storing values

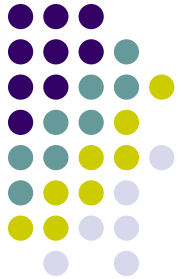


Observations on Compilers



- Compilers do not compute values (as interpreters do)
- Compilers validate the source program, making sure that the intended behavior is correct but do not execute it
- Compilers generate code for the target machine that then execute the intended behavior

Observations on the Symbol Table

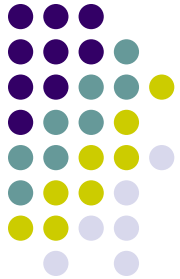


- The fact that compilers do not compute values but validate the source program has an effect on the symbol table:
  - rather than storing variable-value pairs the symbol table act merely as a record holder for variables seen/declared
  - in our case, the symbol table stores the variable-'target name' (scoped name) pairs

Observations on the Symbol Table

```
class SymTab:


    #-------
    def __init__(self):
            …
    #-------
    def push_scope(self):
        # push a new dictionary onto the stack - stack grows to the left
            …
    #-------
    def pop_scope(self):
        # pop the left most dictionary off the stack
            …
    #-------
    def declare_sym(self, sym):
        # declare the symbol in the current scope: dict @ position 0

        # first we need to check whether the symbol was already declared
        # at this scope
        if sym in self.scoped_symtab[CURR_SCOPE]:
            raise ValueError("symbol {} already declared".format(sym))

        # enter the symbol in the current scope
        n_scopes = len(self.scoped_symtab)
        prefix = create_prefix(n_scopes-1)
        scope_dict = self.scoped_symtab[CURR_SCOPE]
        scope_dict[sym] = prefix + sym # value is the prefixed name


    #-------
    def lookup_sym(self, sym):
        # find the first occurence of sym in the symtab stack
        # and return the associated value
            …
    #-------
    def update_sym(self, sym):
        # for the compiler version updating is the same as looking up
        # in order to check if sym is updateable.
        self.lookup_sym(sym)
        return
```

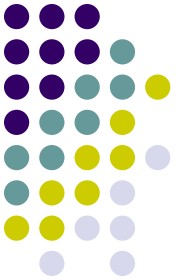
Cuppa2 Compiler



- We reuse the Cuppa2 interpreter frontend but we need to include the new symbol table therefore we need to rename it
  - cuppa2_cc_frontend.py
- Because the compiler is based on the Cuppa1 compiler we can use most of that code generator but need to modify it for:
  - Declarations
  - Get
  - Assignments
  - Variables in expressions
  - cuppa2_cc_codegen.py
- The output phase is the same as the Cuppa1 output phase, but to keep things simple we deleted the peephole optimization
  - to make your compiler more sophisticated you can add this back in ☺
  - cuppa2_cc_output.py

Cuppa2 Codegen

cuppa2_cc_codegen.py



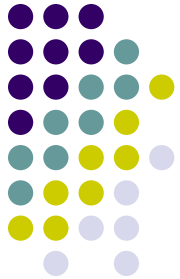
```python
def declare_stmt(node):

    try: # try the declare pattern without initializer
        (DECLARE, name, (NIL,)) = node
        assert_match(DECLARE, 'declare')
        assert_match(NIL, 'nil')

    except ValueError: # try declare with initializer
        (DECLARE, name, init_val) = node
        assert_match(DECLARE, 'declare')

        state.symbol_table.declare_sym(name)
        scoped_name = state.symbol_table.lookup_sym(name)
        value = walk(init_val)
        code = [('store', scoped_name, str(value))]

        return code

    else: # declare pattern matched
        # when no initializer is present we init with the value 0
        state.symbol_table.declare_sym(name)
        scoped_name = state.symbol_table.lookup_sym(name)

        code = [('store', scoped_name, '0')]

        return code
```

```python
def assign_stmt(node):

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

    exp_code = walk(exp)
    scoped_name = state.symbol_table.lookup_sym(name)
    code = [('store', scoped_name, exp_code)]

    return code
```

```python
def get_stmt(node):

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

    scoped_name = state.symbol_table.lookup_sym(name)
    code = [('input', scoped_name)]

    return code
```

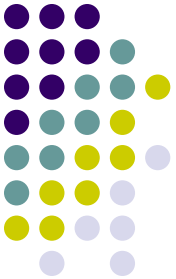
```python
def id_exp(node):

    (ID, name) = node
    assert_match(ID, 'id')

    scoped_name = state.symbol_table.lookup_sym(name)

    return scoped_name
```

Cuppa2 Compiler



- Observation:
  - The difference between the Cuppa1 and Cuppa2 language is the introduction of scope and declarations
  - These are purely high-level language constructs and we see this manifested in that the only thing that really changed in the Cuppa2 compiler compared to the Cuppa1 compiler is how variables are named!
  - That means the Cuppa2 compiler is completely responsible for enforcing scope it cannot pass that through to the underlying abstract machine.

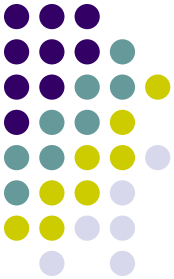
Cuppa2 Driver Function



```
from cuppa2_lex import lexer
from cuppa2_cc_frontend_gram import parser
from cuppa2_cc_state import state
from cuppa2_cc_codegen import walk as codegen
from cuppa2_cc_output import output

def cc(input_stream):

    # initialize the state object
    state.initialize()

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

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

    # output the instruction stream
    bytecode = output(instr_stream)

    return bytecode
```

cuppa2_cc.py

Testing the Compiler



```
In [26]: from cuppa2_examples import scope1, scope2
         from cuppa2_cc import cc
```

```
In [21]: print(cc("declare x = 7; put x"))
```

```
store R$x 7 ;
print R$x ;
stop ;
```

```
In [27]: print(scope1)
```

```
declare x = 1;
{
    declare x = 2;
    put x;
}
{
    declare x = 3;
    put x;
}
put x;
```

```
In [28]: print(cc(scope1))
```

```
store R$x 1 ;
store R$$x 2 ;
print R$$x ;
store R$$x 3 ;
print R$$x ;
print R$x ;
stop ;
```

```
In [29]: print(scope2)
```

```
declare x = 1;
put x;
{
    x = 2;
}
put x;
{
    x = 3;
}
put x;
```

```
In [30]: print(cc(scope2))
```

```
store R$x 1 ;
print R$x ;
store R$x 2 ;
print R$x ;
store R$x 3 ;
print R$x ;
stop ;
```