Compiling Programs into our Bytecode

- Our goal is to compile Cuppa3 programs into Exp2Bytecode
- The big difference between the two languages is that Cuppa3 is a statically scoped language (supports nested scopes and statically scoped functions) and Exp2Bytecode has no notion of scope (all variables are global variables)
- We saw that in order to make recursion work in Exp2Bytecode we resorted to allocating function local variables in a frame on the runtime stack.

Compiling Global Code

- In terms of global code, nothing has changed from our strategy we developed when we compiled Cuppa2 programs into bytecode:
    - Every program variable that appears in the Cuppa3 program is compiled into a unique global variable in the bytecode

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

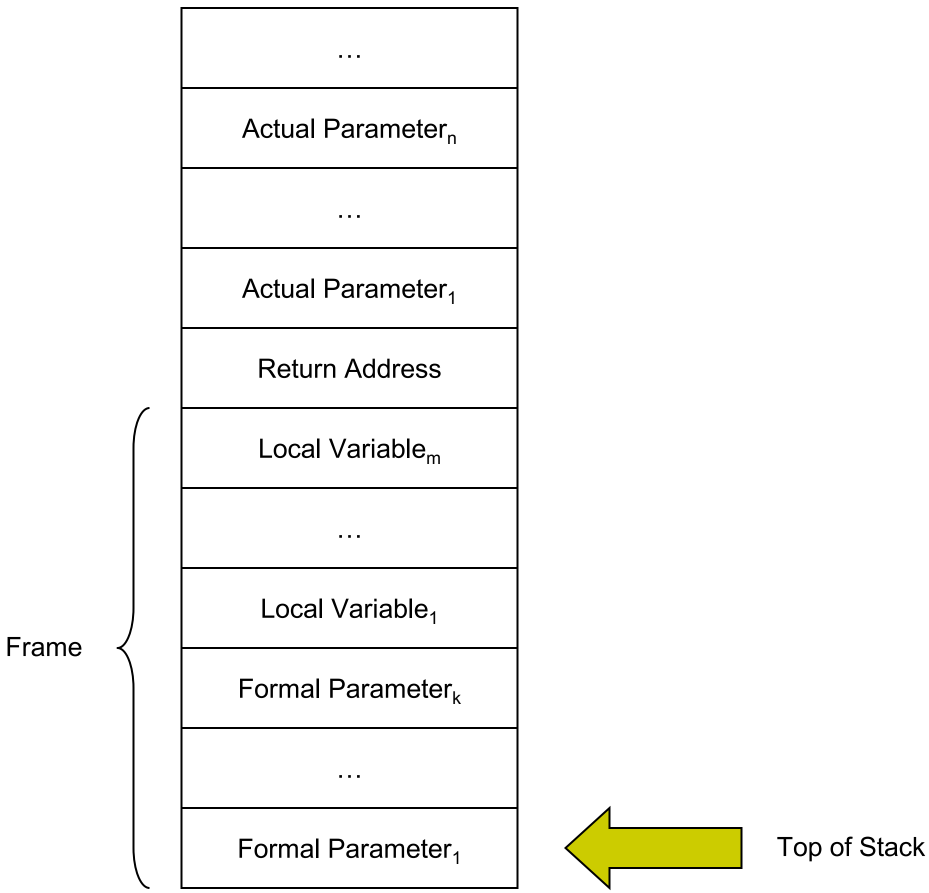
```
store t$0 1 ;
store t$1 2 ;
print t$1 ;
store t$2 3 ;
print t$2 ;
print t$0 ;
stop ;
```

Compiling Functions

- For functions all local variables are stored on the stack
- The actual parameters are pushed on the stack in reverse order, and this is done before the function frame is created.
- Also, during a function call, the return address is pushed onto the stack before the stack frame is created

Compiling Functions

- Here is what the stack looks like during a function call:

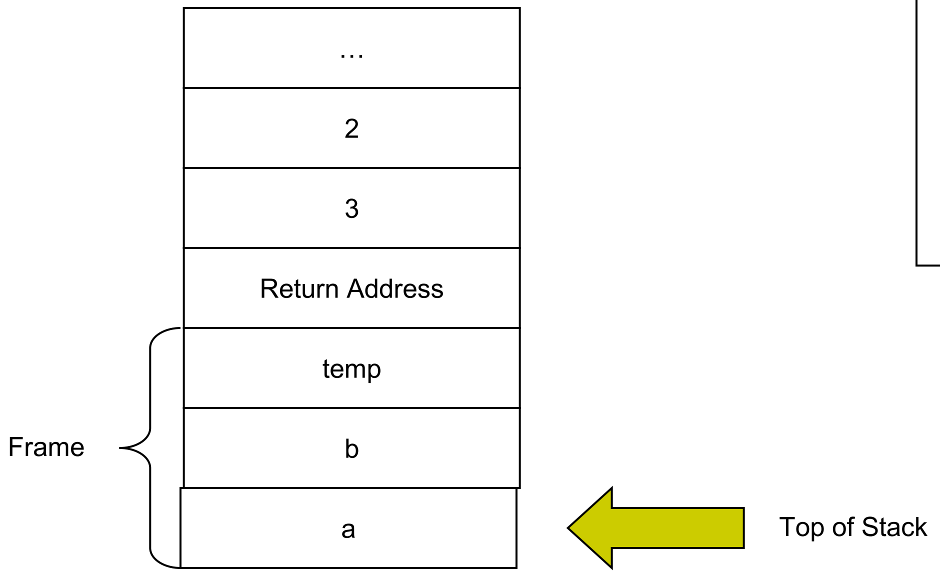


Compiling Functions

- Consider the call add(3,2) to the function defined as

```
declare add(a,b) {
      declare temp = a+b;
      return temp;
 }
```

```
add:
      pushf 3;
      store %tsx[0] %tsx[-4];                  # init a
      store %tsx[-1] %tsx[-5];                 # init b
      store %tsx[-2] (+ %tsx[0] %tsx[-1]); # store temp
      store %rvx %tsx[-2];
      popf 3;
      return;
```



Compiling Functions

- Now consider the following function:

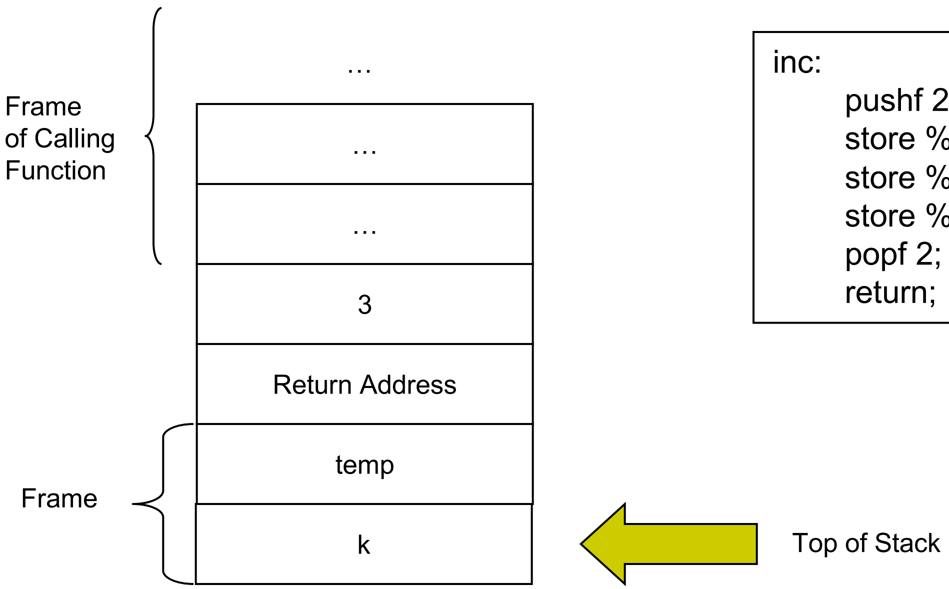
```
// a program with nested functions that makes
// use of static scoping and generates a sequence
// of numbers according to the step variable.

declare seq(n) {
    declare step = 2;
    declare inc(k) return k+step;
    declare i = 1;

    // generate the sequence
    while(i<=n) {
      put(i);
      i = inc(i)
    }
}

// main program
seq(10);
```

Nested function Declarations!

Our interpreter handles this correctly! Try it.

Compiling Functions

- To see the problem with nested function declarations for compilation, let's take a look at the compiled declare inc(k) return k+step; function




```
inc:
      pushf 2;
      store %tsx[0] %tsx[-3] ;                   # init k
      store %tsx[-1] (+ %tsx[0] %tsx[???] );     # inc value into temp
      store %rvx %tsx[-1];
      popf 2;
      return;
```

Note: 'step' is inaccessible from the nested function, 'step' is in the frame of the calling function.

Compiling Functions

- Compiling inc as a global function presents no problems as long as the function is statically scoped.

```
declare step = 2;
declare inc(k) return k+step;

declare seq(n) {
    declare i = 1;

    // generate the sequence
    while(i<=n) {
       put(i);
       i = inc(i)
    }
}

// main program
seq(10);
```

```
inc:
        pushf 2;
        store %tsx[0] %tsx[-3];
        store %tsx[-1] (+ %tsx[0] step$0);
        store %rvx %tsx[-1];
        popf 2;
        return;
```

Conclusion: we will disallow nested function declarations in our compiler.

Compiling Expressions with Functions

- Compiling expressions that contain function calls presents a problem
  - Expressions are represented as terms
  - BUT function calls are statements in our bytecode
  - That means function calls cannot appear in expressions of the bytecode
- Solution: convert the evaluation of expressions into *three-address code* statements.

Three-Address Code

- Three-address code is an intermediate representation
- The name refers to the fact that in a single statement we access *at most* three variables, constants, or functions.
- Each statement in three-address code has the general form of:

  $$x = y \text{ op } z$$

  where x, y and z are variables, constants or temporary variables generated by the compiler and op represents any operator, e.g. an arithmetic operator.

Source: Wikipedia

Three-Address Code

- Expressions containing more than one fundamental operation, such as:

  w = x + y * z

  are not representable in three-address code.
- Instead, they are decomposed into an equivalent series of three-address code statements, such as:

  t1 = y * z
  w = x + t1

Compiling Expressions with Functions

- Consider the expression term:

  3*2+6

- We turn this into three-address code statements by doing only one operation at a time and store the result in a *temporary variable*:

  T$1 = 3*2
  T$2 = T$1+6

Compiling Expressions with Functions

- That is exactly what the compiler will do:

```
put 3*2+4;
```

```
store t$0 (* 3 2) ;
store t$1 (+ t$0 4) ;
print t$1 ;
stop ;
```

Compiling Expressions with Functions

- Now compiling expressions with functions is straightforward
  - Calling a function is just another operation whose result will be stored in a temp
- Consider: 3*2+inc(5)
- We can rewrite the expression term as the following three-address code statements:

```
T$1 = 3*2
T$2 = inc(5)
T$3 = T$1+T$2
```

Compiling Expressions with Functions

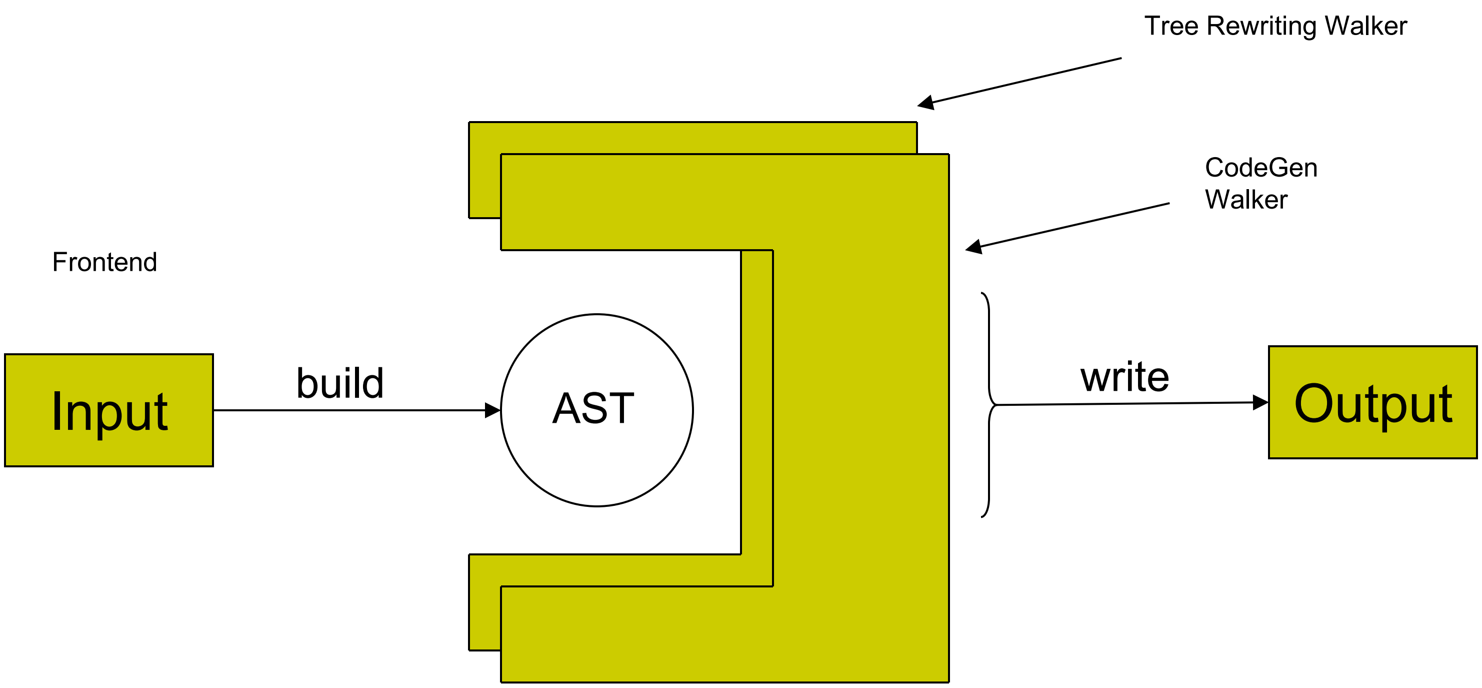
- As compiled code:

```
declare inc(k) return k+1;

put 3*2+inc(5);
```

```
            jump L32 ;
#
# Start of function inc
#
inc:
            pushf 2 ;
            store %tsx[0] %tsx[-3] ;
            store %tsx[-1] (+ %tsx[0] 1) ;
            store %rvx %tsx[-1] ;
            popf 2 ;
            return ;
#
# End of function inc
#
L32:
            noop ;
            store t$0 (* 3 2) ;
            pushv 5 ;
            call inc ;
            popv ;
            store t$1 %rvx ;
            store t$2 (+ t$0 t$1) ;
            print t$2 ;
            stop ;
```

Compiler: Cuppa3 → exp2bytecode

- The compiler has three phases:
  - frontend,
  - semantic analysis/tree rewriting,
  - code generation.
- The symbol table has the same structure as in the interpreter to enforce the semantics of Cuppa3
  - But the symbol table also has structures that support the generation of target code.



Putting it all together

- We use the symbol table to associate source variable names with target names
  - If source variable is a function local variable then the target name will be a stack frame location
- We use the tree rewriting phase to lower the abstraction level of the AST:
  - Insert target names
  - Generate three-address code
- The lowered AST is already in a format that the codegen phase can understand
  - Overall codegen structure similar to Cuppa2 compiler
  - However, lots of details with regards to tracking three address-code result locations and stack manipulation
  - That said, most of the changes from the Cuppa2 to the Cuppa3 compiler are in the function declaration/code generation part and the expression handling part.

Putting it all together

- The relevant code:
  - Cuppa3_cc_symtab.py
  - cuppa3_cc_tree_rewrite.py
  - cuppa3_cc_codegen.py

Putting it all together: three-address code generation

- Let’s use this program to follow the translation process through the compiler

In [29]:
```
from cuppa3_lex import lexer
from cuppa3_cc_frontend import parser
from cuppa3_cc_tree_rewrite import walk as rewrite
from cuppa3_cc_codegen import walk as codegen
from cuppa3_cc_output import output
from cuppa3_cc_state import state
from grammar_stuff import dump_AST

from cuppa3_cc import cc
from exp2bytecode_interp import interp as run
```

In [49]:
```
program = \
'''
declare x = (3 + 2) * 4
put x
'''
```

In [50]:
```
run(cc(program))
```

```
> 20
```

Putting it all together: three-address code generation

In [51]:
```
state.initialize()
parser.parse(program,lexer=lexer)
```

In [52]:
```
dump_AST(state.AST)
```

```
(seq
   |(declare x
   |   |(*
   |   |   |(paren
   |   |   |   |(+
   |   |   |   |   |(integer 3)
   |   |   |   |   |(integer 2)))
   |   |   |(integer 4)))
   |(seq
   |   |(put
   |   |   |(id x))
   |   |(nil)))
```

- The AST right after the front end.

Putting it all together: three-address code generation

```
In [53]: state.AST = rewrite(state.AST)
         dump_AST(state.AST)
```

```
(seq
  |(assign t$2
  |   |(* t$1
  |   |   |(+ t$0
  |   |   |   |(integer 3)
  |   |   |   |(integer 2))
  |   |   |(integer 4)))
  |(seq
  |   |(put
  |   |   |(id t$2))
  |   |(nil)))
```

- The AST right after tree rewriting.
- Notice the additional variable name in the expression AST – third address!

Putting it all together: three-address code generation

- The code generated from the lowered AST
- Notice the statements (arrows) due to three-address code generation

In [54]:
```
output_stream = output(codegen(state.AST))
print(output_stream)
```

```
store t$0 (+ 3 2) ;
store t$1 (* t$0 4) ;
store t$2 t$1 ;
print t$2 ;
```

In [55]:
```
run(output_stream)
```

```
> 20
```

Putting it all together: Function local code

- Let's look at this function and trace the translation process with respect to function local variables

In [56]:
```
program = \
'''
declare double_sum(a,b)
{
    return (a+b)*2;
}
'''
```

Putting it all together: Function local code

```
In [57]: state.initialize()
         parser.parse(program,lexer=lexer)
```

```
In [58]: dump_AST(state.AST)
```

```
(seq
  |(fundecl double_sum
  |   |(seq
  |   |   |(id a)
  |   |   |(seq
  |   |   |   |(id b)
  |   |   |   |(nil)))
  |   |(block
  |   |   |(seq
  |   |   |   |(return
  |   |   |   |   |(*
  |   |   |   |   |   |(paren
  |   |   |   |   |   |   |(+
  |   |   |   |   |   |   |   |(id a)
  |   |   |   |   |   |   |   |(id b)))
  |   |   |   |   |   |(integer 2)))
  |   |   |   |(nil))))
  |(nil))
```

- The AST right after the front end

Putting it all together: Function local code

- The AST after rewriting
- Both function local variables and three address code generation are represented!

```
In [59]: state.AST = rewrite(state.AST)
         dump_AST(state.AST)
```

```
(seq
  |(fundef double_sum
  |   |(seq %tsx[0]
  |   |   |(seq %tsx[-1]
  |   |   |   |(nil)))
  |   |(block
  |   |   |(seq
  |   |   |   |(return
  |   |   |   |   |(* %tsx[-3]
  |   |   |   |   |   |(+ %tsx[-2]
  |   |   |   |   |   |   |(id %tsx[0])
  |   |   |   |   |   |   |(id %tsx[-1]))
  |   |   |   |   |   |(integer 2)))
  |   |   |   |(nil))) 4)
  |(nil))
```

Formal arguments

3-addr code temps

Formal arguments

Putting it all together: Function local code

- Generated code

```
In [60]: output_stream = output(codegen(state.AST))
         print(output_stream)
```

```
        jump L7 ;
#
# Start of function double_sum
#
double_sum:
        pushf 4 ;
        store %tsx[0] %tsx[-5] ;
        store %tsx[-1] %tsx[-6] ;
        store %tsx[-2] (+ %tsx[0] %tsx[-1])
        store %tsx[-3] (* %tsx[-2] 2) ;
        store %rvx %tsx[-3] ;
        popf 4 ;
        return ;
        popf 4 ;
        return ;
#
# End of function double_sum
#
L7:
        noop ;
```

Formal arguments

3-addr code temps

Putting it all together

- The program

```
In [61]: program = \
'''
declare double_sum(a,b)
{
    return (a+b)*2;
}

declare x = double_sum(3,2);
put x;
'''
```

Putting it all together

```
(seq
  |(fundecl double_sum
  |   |(seq
  |   |   |(id a)
  |   |   |(seq
  |   |   |   |(id b)
  |   |   |   |(nil)))
  |   |(block
  |   |   |(seq
  |   |   |   |(return
  |   |   |   |   |(*
  |   |   |   |   |   |(paren
  |   |   |   |   |   |   |(+
  |   |   |   |   |   |   |   |(id a)
  |   |   |   |   |   |   |   |(id b)))
  |   |   |   |   |   |(integer 2)))
  |   |   |   |(nil))))
  |(seq
  |   |(declare x
  |   |   |(callexp double_sum
  |   |   |   |(seq
  |   |   |   |   |(integer 3)
  |   |   |   |   |(seq
  |   |   |   |   |   |(integer 2)
  |   |   |   |   |   |(nil)))))
  |   |(seq
  |   |   |(put
  |   |   |   |(id x))
  |   |   |(nil))))
```

- The front end AST

Putting it all together

```
(seq
  |(fundef double_sum
  |   |(seq %tsx[0]
  |   |   |(seq %tsx[-1]
  |   |   |   |(nil)))
  |   |(block
  |   |   |(seq
  |   |   |   |(return
  |   |   |   |   |(* %tsx[-3]
  |   |   |   |   |   |(+ %tsx[-2]
  |   |   |   |   |   |   |(id %tsx[0])
  |   |   |   |   |   |   |(id %tsx[-1]))
  |   |   |   |   |   |(integer 2)))
  |   |   |   |(nil))) 4)
  |(seq
  |   |(assign t$1
  |   |   |(callexp t$0 double_sum
  |   |   |   |(seq
  |   |   |   |   |(integer 3)
  |   |   |   |   |(seq
  |   |   |   |   |   |(integer 2)
  |   |   |   |   |   |(nil)))))
  |   |(seq
  |   |   |(put
  |   |   |   |(id t$1))
  |   |   |(nil))))
```

- The rewritten AST

Putting it all together

```
        jump L8 ;
#
# Start of function double_sum
#
double_sum:
        pushf 4 ;
        store %tsx[0] %tsx[-5] ;
        store %tsx[-1] %tsx[-6] ;
        store %tsx[-2] (+ %tsx[0] %tsx[-1]) ;
        store %tsx[-3] (* %tsx[-2] 2) ;
        store %rvx %tsx[-3] ;
        popf 4 ;
        return ;
        popf 4 ;
        return ;
#
# End of function double_sum
#
L8:
        noop ;
        pushv 2 ;
        pushv 3 ;
        call double_sum ;
        popv ;
        popv ;
        store t$0 %rvx ;
        store t$1 t$0 ;
        print t$1 ;
```

- The rewritten AST

The Symbol Table

- The symbol table has additional functionality to deal with
  - The stack frame
  - Scalars vs functions
  - Function local vs global variables

The Symbol Table

```
class SymTab:

    def __init__(self):
        # global scope dictionary must always be present
        self.scoped_symtab = [{}]
        # keep track of wether we are in a function declaration of not
        self.in_function = False
        # counter used to generate unique global names
        self.temp_cnt = 0
        # counter to compute the frameoffset of function local variables
        self.offset_cnt = 0
```

```
# frame functions

def _make_target_name(self):
    # in functions all function local variables are on the runtime stack
    if self.in_function:
        name = "%tsx[0]" if self.offset_cnt == 0 \
                         else "%tsx[" + str(- self.offset_cnt) + "]"
        self.offset_cnt += 1
    else:
        name = "t$" + str(self.temp_cnt)
        self.temp_cnt += 1
    return name

def get_frame_size(self):
    return self.offset_cnt
```

The Symbol Table

```
# scope manipulation functions

def get_config(self):
    # we make a shallow copy of the symbol table
    return list(self.scoped_symtab)

def set_config(self, c):
    self.scoped_symtab = c

def push_scope(self):
    # push a new dictionary onto the stack - stack grows to the left
    self.scoped_symtab.insert(CURR_SCOPE,{})

def pop_scope(self):
    # pop the left most dictionary off the stack
    if len(self.scoped_symtab) == 1:
        raise ValueError("cannot pop the global scope")
    else:
        self.scoped_symtab.pop(CURR_SCOPE)

def enter_function(self):
    # if we are in a function declaration we are not allowed to start another one
    if self.in_function:
        raise ValueError("Function declarations cannot be nested.")

    self.in_function = True
    self.push_scope()
    self.offset_cnt = 0

def exit_function(self):
    self.in_function = False
    self.pop_scope()
```

The Symbol Table

```
# symbol declaration functions

def declare_scalar(self, sym):
    # declare the scalar in the current scope
    # 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
    self.scoped_symtab[CURR_SCOPE] \
        .update({sym : ('scalar', self._make_target_name())})

def declare_fun(self, sym, init):
    # declare a function in the current scope
    # 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 function in the current scope
    self.scoped_symtab[CURR_SCOPE] \
        .update({sym : ('function', init)})
```

The Symbol Table

```
# msic. functions

def lookup_sym(self, sym):
    # find the first occurence of sym in the symtab stack
    # and return the associated value

    n_scopes = len(self.scoped_symtab)

    for scope in range(n_scopes):
        if sym in self.scoped_symtab[scope]:
            val = self.scoped_symtab[scope].get(sym)
            return val

    # not found
    raise ValueError("{} was not declared".format(sym))

def get_target_name(self, sym):
    (type, name) = self.lookup_sym(sym)
    if type != 'scalar':
        raise ValueError("{} is not a scalar.".format(sym))
    return name
```

Tree Rewriting

- Generate three address code
- Replace original variable names with target names

Tree Rewriting

```
def binop_exp(node):
    # turn expressions into three-address codes

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

    t1 = walk(c1)
    t2 = walk(c2)

    target_name = declare_temp()

    return (OP, target_name, t1, t2)
```

- Insert a temporary variable into expression as the “third address”

```
def uminus_exp(node):

    (UMINUS, exp) = node
    assert_match(UMINUS, 'uminus')

    t = walk(exp)

    target_name = declare_temp()

    return ('uminus', target_name, t)
```

Tree Rewriting

```python
def call_exp(node):

    (CALLEXP, name, actual_args) = node
    assert_match(CALLEXP, 'callexp')

    return handle_call('callexp', name, actual_args)
```

- Insert a temporary variable into call expression as the “third address”

```python
def handle_call(call_kind, name, actual_arglist):

    (type, val) = state.symbol_table.lookup_sym(name)

    if type != 'function':
        raise ValueError("{} is not a function".format(name))

    # unpack the funval tuple
    (FUNVAL, formal_arglist) = val

    if len_seq(formal_arglist) != len_seq(actual_arglist):
        raise ValueError("function {} expects {} arguments" \
                         .format(sym, len_seq(formal_arglist)))

    # convert the actual values into three-address codes
    actual_val_args = eval_actual_args(actual_arglist)

    if call_kind == 'callexp':
        return ('callexp', declare_temp(), name, actual_val_args)
    else:
        return ('callstmt', name, actual_val_args)
```

Tree Rewriting

```
# Note: We are walking the body of a function declaration to figure out
# how many local variables there are. We need this information in order
# to compute the frame size of the function. Also, we need to replace
# original function local variables with their stack frame target names.
def fundecl_stmt(node):

    try: # try the fundecl pattern without arglist
        (FUNDECL, name, (NIL,), body) = node
        assert_match(FUNDECL, 'fundecl')
        assert_match(NIL, 'nil')

    except ValueError: # try fundecl with arglist
        (FUNDECL, name, arglist, body) = node
        assert_match(FUNDECL, 'fundecl')

        # we don't need the function body - abbreviated function value
        funval = ('funval', arglist)
        state.symbol_table.declare_fun(name, funval)

        state.symbol_table.enter_function()
        new_arglist = declare_formal_args(arglist)
        t = walk(body)
        state.symbol_table.exit_function()
        frame_size = state.symbol_table.get_frame_size()

        return ('fundef', name, new_arglist, t, frame_size)


    else: # fundecl pattern matched
        # no arglist is present
        # we don't need the function body - abbreviated function value
        funval = ('funval', ('nil',))
        state.symbol_table.declare_fun(name, funval)

        state.symbol_table.enter_function()
        t = walk(body)
        state.symbol_table.exit_function()
        frame_size = state.symbol_table.get_frame_size()

        return ('fundef', name, ('nil',), t, frame_size)
```

- Function declarations get rewritten into function definitions which contain the frame size.

Tree Rewriting

```
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')

        t = walk(init_val)
        state.symbol_table.declare_scalar(name)
        target_name = state.symbol_table.get_target_name(name)

        return ('assign', target_name, t)

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

        return ('assign', target_name, ('integer', 0))
```

- Original variable names are replaced with target names.

```
def assign_stmt(node):

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

    t = walk(exp)
    target_name = state.symbol_table.get_target_name(name)

    return ('assign', target_name, t)
```

```
def id_exp(node):

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

    target_name = state.symbol_table.get_target_name(name)

    return ('id', target_name)
```

Code Generation

- Very similar to the Cuppa2 compiler
- All statement level patterns carry over almost unmodified
- Expressions need to keep track of the “third address” or “target address” of the three address code encoding.

Code Generation

```
def binop_exp(node):

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

    (lcode, lloc) = walk(c1)
    (rcode, rloc) = walk(c2)

    code = lcode + rcode
    code += [('store', temp, '(' + OP + ' ' + lloc + ' ' + rloc + ')')]
    loc = temp

    return (code, loc)
```

- The code generator keeps track of the location where the value of an expression is stored
- Returns a pair for an expression:
  (1) code
  (2) location

```
def call_exp(node):

    (CALLEXP, temp, name, actual_args) = node
    assert_match(CALLEXP, 'callexp')

    code = push_args(actual_args)
    code += [('call', name)]
    code += pop_args(actual_args)
    code += [('store', temp, '%rvx')]
    loc = temp

    return (code, loc)
```

```
def uminus_exp(node):

    (UMINUS, temp, e) = node
    assert_match(UMINUS, 'uminus')

    (code, loc) = walk(e)

    code += [('store', temp, '-' + loc)]
    loc = temp

    return (code, loc)
```

Code Generation

```
def while_stmt(node):

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

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

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

    return code
```

- The while statement is probably the best example to see how the pair returned by generating code for an expression is used.
- Note that we first put the code for the expression into the output stream
- Then we use the location in the conditional jump instruction.