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;
```

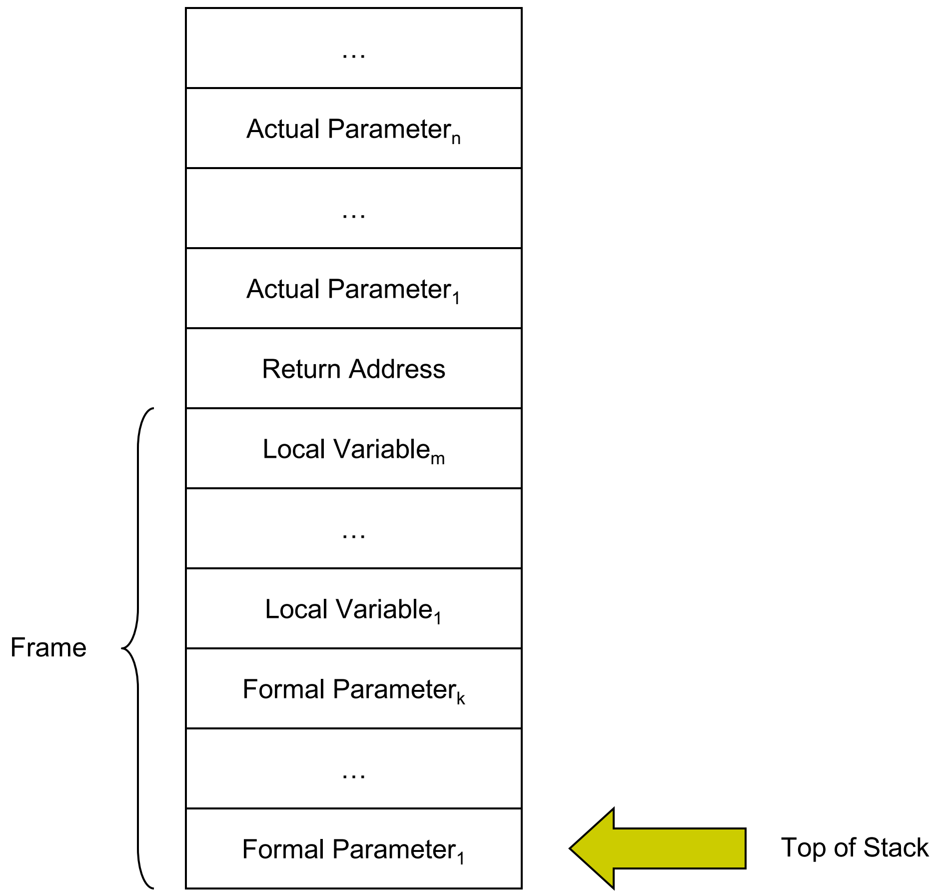
```
store t$0 1 ;
store t$1 2 ;
print t$1 ;
store t$2 3 ;
print t$2 ;
print t$0 ;
stop ;
```

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:

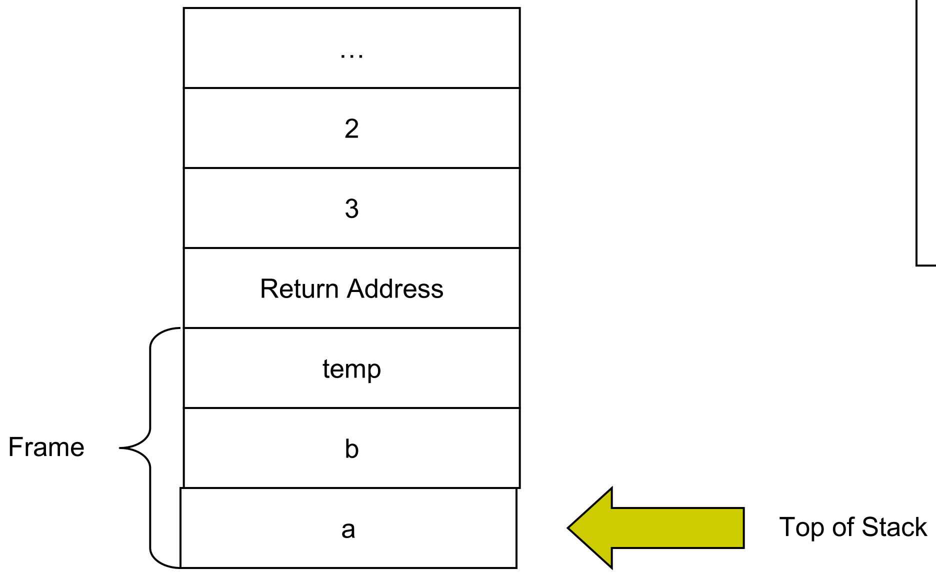


Compiling Functions

- 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:

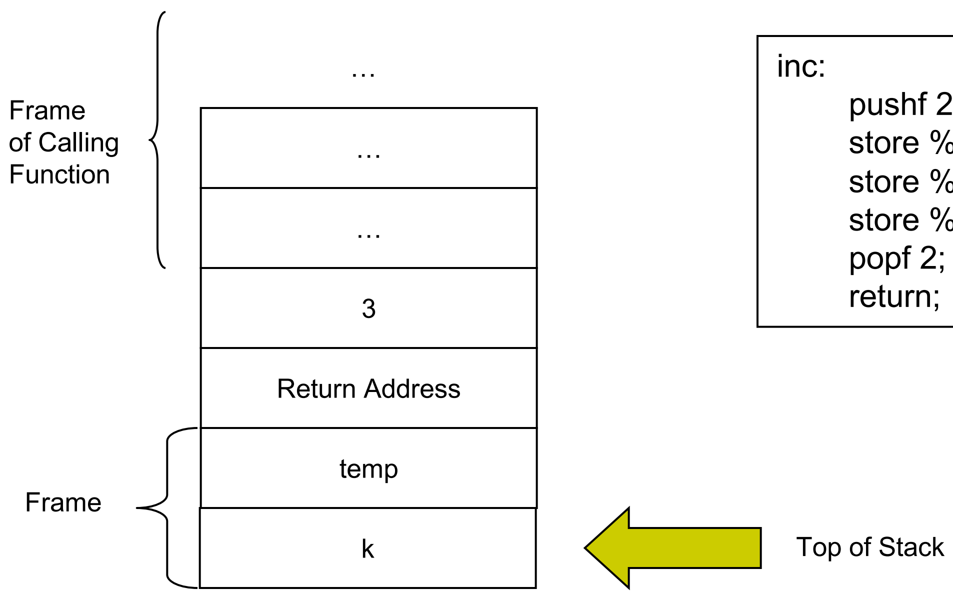
```
// a program with nested functions that makes
// use of static scoping and generates a sequence
// of numbers according to the step variable.

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

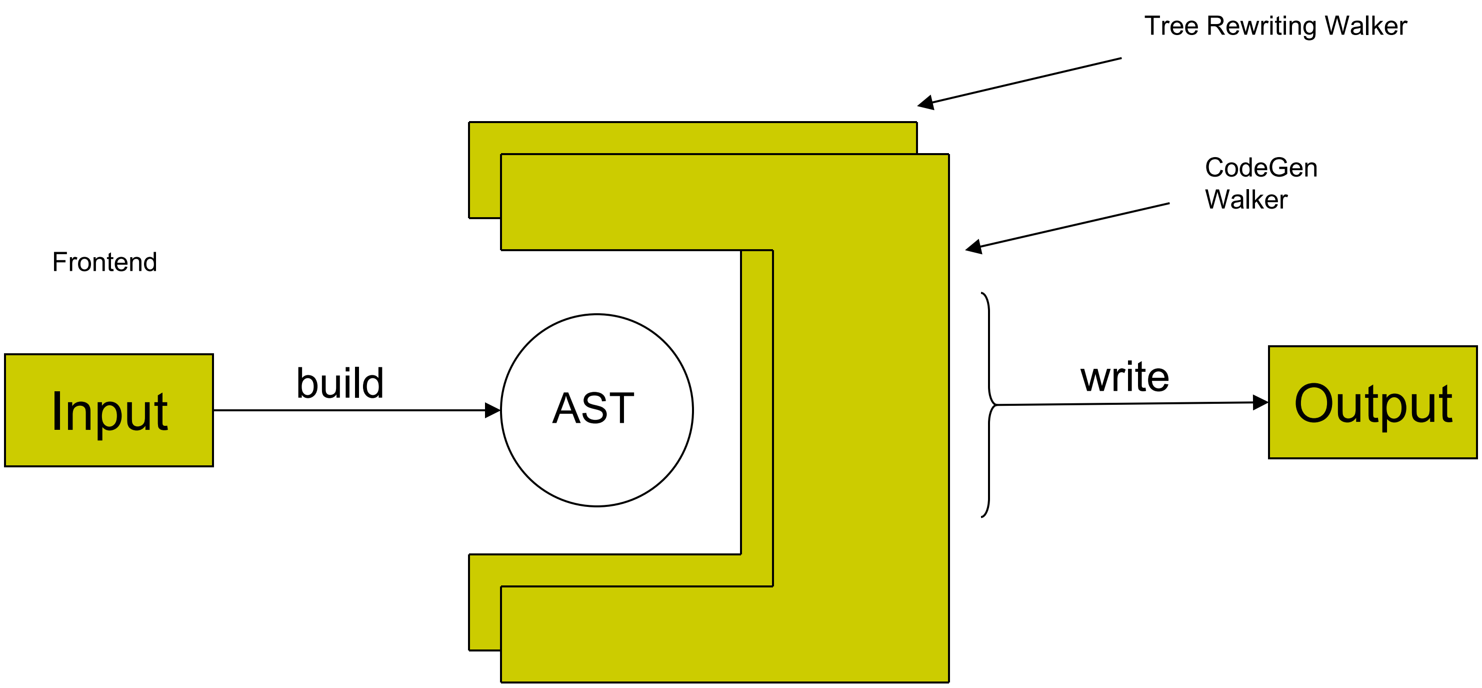
- As compiled code:

```
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 rewrting,
  - 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.



Compiler: Cuppa3 → exp2bytecode

- Let’s look at some code:
  - cuppa3_cc_tree_rewrite.py
  - cuppa3_cc_codegen.py
- Look at Notebook for test suites for Cuppa3 compiler: ‘Cuppa3 CC Tests’