Compiler Principles, PS6

1. Static vs Dynamic Scope

Among the most important issues that we face when designing a compiler for a language is what decisions can the compiler make about a program. If a language uses a policy that allows the compiler to decide an issue, then we say that the language uses a static policy or that the issue can be decided at compile time. On the other hand, a policy that only allows a decision to be made when we execute the program is said to be a dynamic policy or to require a decision at run time.

One issue on which we shall concentrate is the scope of declarations. The scope of a declaration of x is the region of the program in which uses of x refer to this declaration.

One of the basic reasons for scoping is to keep variables in different parts of the program distinct from one another. Since there are only a small number of short variable names, and programmers share habits about the naming of variables (e.g., i for an array index), in any program of moderate size the same variable name will be used in multiple different scopes.

The question of how to match various variable occurrences to the appropriate binding sites is generally answered in one of two ways: static scoping and dynamic scoping.

```
1. (define fact
2.   (lambda (n)
3.     (if (zero? n) 1 (* n (fact (- n 1))))))
4.
5. (let ((* +))
6.   (fact 5))
```

Static Scoping:

With static scope, a variable always refers to its top-level environment. This is a property of the program text and unrelated to the runtime call stack. Because matching a variable to its binding only requires analysis of the program text, this type of scoping is sometimes also called lexical scoping. Static scope is standard in modern functional languages such as ML because it allows the programmer to reason as if variable bindings are carried out by substitution.

Static scoping also makes it much easier to make modular code and reason about it, since its binding structure can be understood in isolation. In contrast, dynamic scope forces the programmer to anticipate all possible dynamic contexts in which the module's code may be invoked.

In other words, a language uses static scope or lexical scope if it is possible to determine the scope of a declaration by looking only at the program. Otherwise, the language uses dynamic scope.

```
1.  (define fact
2.   (lambda (n)
3.     (if (zero? n) 1 (* n (fact (- n 1))))))
4.
5.  (let ((* +))
6.   (fact 5))
7.  120
```

Dynamic Scoping:

With dynamic scope, each identifier has a global stack of bindings. Introducing a local variable with name x pushes a binding onto the global x stack (which may have been empty), which is popped off when the control flow leaves the scope. Evaluating x in any context always yields the top binding. In other words, a global identifier refers to the identifier associated with the most recent environment. Note that this cannot be done at compile time because the binding stack only exists at runtime, which is why this type of scoping is called dynamic scoping.

```
1.  (define fact
2.   (lambda (n)
3.     (if (zero? n) 1 (* n (fact (- n 1))))))
4.
5.  (let ((* +))
6.   (fact 5))
7.  16
```

Deep binding - Dynamic scoping is fairly easy to implement. To find an identifier's value, the program can traverse the runtime stack, checking each activation record (each function's stack frame) for a value for the identifier. This is known as deep binding.

Shallow binding - An alternate strategy that is usually more efficient is to maintain a stack of bindings for each identifier; the stack is modified whenever the variable is bound or unbound, and a variable's value is simply that of the top binding on the stack. This is called shallow binding.

Note that both of these strategies assume a last-in-first-out (LIFO) ordering to bindings for any one variable; in practice all bindings are so ordered.

2. Lexical Addressing:

Chars -> SCANNER -> tokens -> READER -> sexps -> PARSER -> pes ->
-> SEMANTIC ANALYSIS -> pes -> CODE_GEN ->

Lexical addressing is a part of the semantic analysis; We distinguish 2 types of variables:
* Free vars – variables not bounded by any encompassing lambda.
* Param vars – variables that are bounded by the current lambda. we count the index of the parameter.
* Bounded vars Ma Mi – variables bounded by some encompassing lambda. In this case we denote Ma and Mi as the DeBruijn numbers for this variable. The Major (Ma) number tells us in which environment (lambda) the var was defined, the Minor (Mi) number tells us which variable in that environment (lambda) is referred.
After we replace all bounded variables with their lexical address, we no longer really need to save the name of the variable.

Example:
```
(lambda (x y)
(lambda (a b)
    (x a b f)))
```

in the body, x => (0,0), a => param 0 , b => param 1 , f => free.

Example:
```
(lambda (x)
  (lambda (y z)
    [ (lambda (x v)
        (f z x))              => f - free, z - (0,1) , x – param-0
      (+ v z x)               => + - free, v – free, z – param 1, x - (0,0)
      v                       => v - free
    ] )
```

Example:
```
(define fact
   (lambda (n)
      (if (zero? n)
          1
          (* n (fact (- n 1))))))
```

n => param 0

zero? , *, - , fact => free

4. Tail-Calls

A **tail call** is a subroutine call which is followed by a return to the calling code. Such a call is said to be in **tail position**. If a subroutine performs a tail call to itself, it is called **tail recursive**. Tail calls can usually be optimized into more efficient code by removing unnecessary call stack manipulation--in effect, turning the subroutine call to a jump to the subroutine in tail position.

```
(define foo
 (lambda(x)
 (if (= x 1)
     (* 5 (hoo x))    ;; not tail call
     (if (= x 2)
        (hoo x)    ;; tail call
        (foo x))))) ;; tail recursive

(define fact
 (lambda(n)
 (if (zero? n)
     1
     (* n (fact (- n 1)))))) ;; not tail position
```

Note: If an application cannot be determined statically to be the last application in the current frame, then it should be compiled with the condition of being in tail position taken as **false**. (Think about what this means for and and or in Scheme)

Tail call optimization is a compiler optimization that reuses the existing stack frame (environment).

Consider:

```
int foo(int b) {
     return bar(b * b);
 }
int bar(int a) {
     printf("bar called with arg %d\n", a);
     return a * a;
```

```
 }
void main(…) {
     printf("hi");
     foo(3);
     printf("bye");
 }
```

What will happen when *foo*(3) is evaluated?

1. create a stack frame (environment) for the call to *foo*, in which b=3. The frame includes the return address to our main code (to the command following *foo*(3)).
2. compute the body of *foo*
3. create a stack frame for *bar*, in which a = 9, including the return address to the next command in the *foo* code.
4. compute the body of *bar* in this frame
5. the last line of code is a return to *foo* with the value 27
6. foo has only 1 more code command - return to the main program with the value 27

A compiler with tail call optimization would create shorter and more efficient code:

1. create a stack frame (environment) for the call to *foo*, in which b=3. The frame includes the return address to our main code (to the line following *foo*(3)).
2. compute the body of *foo*
3. ~~create a stack frame for *bar*, in which a = 9, including the return address to the next command in the *foo* code.~~ Re-adjust the current frame so that a=9
4. compute the body of *bar* in this frame (goto!)
5. ~~5.~~ the last line of code is a return to ~~*foo* with the value 27~~
   ~~foo has only 1 more code command - return to~~ the main program with the value 27

Note: Using CPS without tail call optimization (TCO) will cause not only the explicit continuation to grow during recursion, but also the function stack itself.

Tail recursion is a special kind of tail call optimization. It is somewhat easier to handle, because the size of the frame remains the same. Optimizing a tail-recursive call involves the following steps:

- Compute the new values for the variables in the frame. Overwrite the existing values with the new values.
- Perform a goto to the beginning of the function.

It is easy to demonstrate the tail-recursion optimization in high-level C. Consider the following version of Ackermann's function:

```
int ack(int a, int b) {
  if (a == 0) return b + 1;
  if (b == 0) return ack(a - 1, 1);
  return ack(a - 1, ack(a, b - 1));
}
```

The tail-recursive calls have been marked in **bold**, to help you identify them. The optimized version is as follows:

```
int ack(int a, int b) {
  Lack:
    if (a == 0) return b + 1;
    if (b == 0) { --a; b = 1; goto Lack; }
    b = ack(a, b - 1); /* not in tail position */
    --a;
    goto Lack;
}
```

It is nicer to optimize all tail calls, but for a language with no iterative constructs, the tail-recursion optimization is the bare minimum.

All loops -- for-loops, while-loops, etc, are implemented in Scheme as tail-recursive functions. By performing either the tail-call optimization or the tail-recursion optimization on them, recursive calls are converted into simple iteration, running faster, using up less stack space, and generally taking no more than corresponding loops in C.