CS 131 Programming Languages

October 8, 2009

Defining constructors N and T (non-terminal and terminal):

```
type ('a, 'b) symbol = | N of 'a | T of 'b;;
```

Defining an optional value symbol:

```
type 'a option = | None | Some of 'a;;
```

A constructor of constructors:

```
type ('a, 'b) symbol2 = | X of 'a | Y of 'b;;
type ('a, 'b) symsym = | X1 of ('a, 'b) symbol | X2 of ('a, 'b) symbol2;;  (* X1 (N "x") *)
match n with | X1 (N a) -> Some a | _ -> None;;
```

Homework 2

```
type fragment = nucleotide list;;     (* for convenience *)
type pattern = | Frag of fragment
               | List of pattern list
               | Or of pattern list
               | Junk of int
               | Closure of pattern;;
```

A value of type *pattern* is a pointer to a piece of storage that contains the constructor of one of the types `Frag, List, Or, Junk,` or `Closure` (such as `[Closure, pattern]`). In a match statement, all of the patterns have to match values *of the same type*.

The "==" statement: *compare addresses* (versus "=" which means *compare contents*). Suppose you have a pattern of ((AT)* G*)* that matches a sequence of zero or more AT followed by zero or more G, repeated zero or more times. This matches the empty fragment because of the wildcard. We're writing our function to compute the matcher recursively. To build a matcher for `Closure of pat`:

- `let mp = matcher for pat;;`
- ```
  return self = fun frag accept ->
      a.  | None -> mp frag (fun frag1) -> self frag1 accept
      b.  | x -> x;;
  ```

However, this code will loop endlessly because mp will match the empty string itself. We have to gobble up at least one input symbol before calling ourselves (we have to make sure we've made some progress).

- ```
  return self = fun frag accept ->
      a.  | None -> mp frag (fun frag1) -> if frag == frag1 then None else (self frag1 accept)
      b.  | x -> x;;
  ```

In this case, "==" is much faster, and is correct, since it compares addresses instead of the contents (O(1) vs. O(n)).

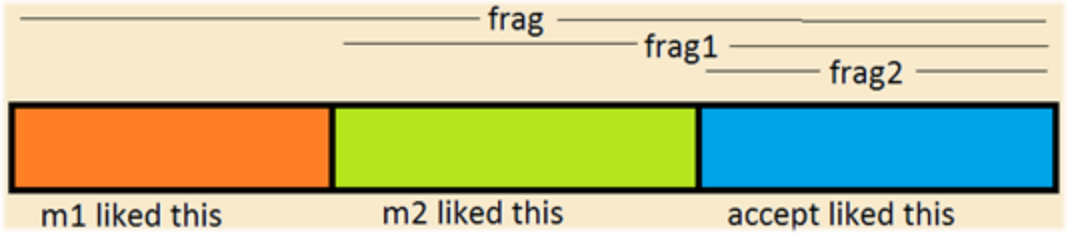
Let's look at append_matchers function from the hint code.

```
let append_matchers m1 m2 frag accept =
       m1 frag (fn frag1 -> m2 frag1 accept);;
```

It glues together existing matchers, forming third matcher. Longer form of the same code:

```
let append_matchers m1 m2 =
       fun frag accept ->
              m1 frag (fun frag1 -> m2 frag accept);;
```

The `(fun frag1 -> m2 frag accept)` is also an acceptor built on-the-fly at the point when *append_matchers* is called. However, it is not finalized until it itself is called – because the compiler doesn't know the form of *frag* and *accept* variables until that function is called.



Back to Previous Lecture (Compilers, Linkers, Assemblers)

An alternative, common approach is to use the *interpreter* that does not compile the code, but rather runs it on-the-fly by implementing its own virtual machine (since there is no such hardware that can run code that's not compiled).

- Advantages of the interpreter
    a. Easier to debug
        i. We can design and write an interpreter that's easier to debug
        ii. Any errors that we run into can be easily translated into something programmer can understand
    b. No need to recompile everything after every change
    c. Portable (code can run on multiple systems given that interpreter is available)
    d. Easier to write interpreter than compiler
    e. Faster translation of the code (easier to translate, lazy translation)
    f. Catches runtime errors more often
    g. Often generates smaller bytecode that can fit into processor's cache – therefore, faster execution
- Disadvantages of the interpreter
    a. Slower at execution time

Java Virtual Machine

- Java source (*.java) files
- *javac* (compiler) generates Java bytecode (*.class) files
- *jre* (interpreter) runs the bytecode
- Also, there is a "turbo booster" in the system.
    - The *jre* doesn't only execute the bytecodes – it also keeps a counter for each method. Inside the *jre* is a bytecode compiler that compiles the most used methods into, let's say, *x86* machine code.
    - From then on, the interpreter will execute the machine code instead of the bytecode.
    - The compiled code is highly specialized – it is highly optimized for the data types that are most commonly passed to the method.

Static and Dynamic Linking

One or more object files (*a.o*, *b.o*, etc) link together into an executable. User then loads the result into RAM to run it. A component of linking is libraries (*libc.a*, etc) but they're simply compressed collections of object files.

This approach has a few problems. Suppose there's a function in C library that reads a line from STDIN and returns the input. But if there is a bug in this function that happened to be located in *libc.a* (or other library) then, in order to fix it, one must re-link to propagate library fixes. Software may have to use old libraries because it was compiled on them, therefore still containing known security issues. Also, multiple copies of the C library may be present in RAM, wasting valuable space.

Most applications don't do static linking anymore, instead using *dynamic linking*. All the dynamic linking occurs as the program starts up. With that approach, the linker takes the file such as *libc.so*, in which only external symbols are saved, and link against it. All of the *libc.so* will be read-only code (can be run without any changes at all) that doesn't care where it is (*relocatable* - there's nothing in the code that cares where the address of the code is).

We can store one copy of the *libc.so* in the RAM, then build a virtual subsystem of our library and map *libc.so* from the RAM into the virtual memory block for application A; then we can proceed by taking care of program B and C the same way, and so on. Therefore, we have only one copy of the library in RAM while each application refers to a different address in order to access it.

To solve the problem of fix propagation, another mechanism is used. The compiled executable not contain any implementation code of the libraries it was built against, but rather the stubs that refer to where the code is (in the library files). For the new versions of the libraries to be used, the running applications (processes) must be killed and started again.

One can dynamically link everything before the program starts, or the application can programmatically link the libraries as it needs them. The term "DLL hell" describes the Windows problem of applications requiring different versions of libraries on the system, therefore conflicting with each other.

Names and Bindings

A binding is an association between the name and the value. Binding time is the time at which the name gets associates with its value. Let's have `int x = 27;` The *value* of the variable `x` is a data type (such as *int*) and gets bound at the run time (*assignment* time) and may change later. The *address* of the variable `&x` gets bound upon the *declaration* of the variable and doesn't change during the run time. The linker decides the address of the variable at the link time. The size of the variable `sizeof x` gets decided by the hardware and compiler designers, initialized at the *ABI* (application binary interface) time. Suppose we take a look at the first byte in x: `int x = 0x01020304; char *p = &x; return *p;` What value will be returned? It depends on the machine's architecture – Big Endian vs. Little Endian. In the Little Endian machine, the bytes are numbered right-to-left. Therefore, `&x` can return the address of the top byte or the bottom byte of `x`.

Reading assignment: chapters 1-6 plus all the ML chapters.