Introduction to Interpreters

February 11, 2002

CS 131: Programming Languages

Compilers and Interpreters

- Compilers translate code from one language (*source language*) to another (*target language*).
  - Ideally, the translation preserves meaning
  - Target often assembly or machine language, but could be bytecodes, or C, ...
- Interpreters execute code "as given"
- In practice, the boundary is fuzzier.
  - Interpreters rarely deal directly with programs as strings

To Interpret or Compile?

- It is true that...
  - some languages are easier to compile than others.
  - some languages have been historically compiled, and others have been historically interpreted.
  - some languages require an interactive system
- But compilation or interpretation is *not* an intrinsic property of the language!

Wrong, wrong, wrong:

"...it's clear that a well-written program in Java could never run as fast as a well-written program in C or C++. That's because the Java bytecode is interpreted, not compiled. Programs written in C are compiled into binaries which can be executed by a specific computer processor. Programs written in Java require one more step – they must be interpreted by the Java `virtual machine' before running on a particular computer architecture. As a result, a computer running a Java program has to execute more machine-language instructions to do the same amount of work than a computer running an equivalent program written in C."

Simson Garfinkel, Salon Magazine, 1/8/2001

Today's Goal

- A very simple interpreter for a simple functional language
  - Essentially a subset of rex or ML
- Key ideas for today
  - Representing abstract syntax in ML
  - Variable scope and free variables
  - Substitution
  - A first interpreter in ML

Abstract Syntax: notation

```
exp ::= num                          integers
      | bool                         booleans
      | var                          variables
      | (var) => exp                 functions
      | exp exp                      applications
      | exp + exp                    additions
      | exp == exp                   equality-test
      | exp ? exp : exp              conditionals
      | <exp,exp>                    pairs
      | fst exp                      1st projection
      | snd exp                      2nd projection
      | let var be exp in exp        local definitions
```

SML Implementation

```
datatype absyn =
   Num   of int
 | Bool  of bool
 | Var   of string
 | FnVal of string * absyn
 | Apply of absyn * absyn
 | Plus  of absyn * absyn
 | Equal of absyn * absyn
 | Cond  of absyn * absyn * absyn
 | Pair  of absyn*absyn
 | Fst   of absyn
 | Snd   of absyn
 | Let   of string*absyn*absyn
```

Exercise

- Give the SML representation of the expression written informally as:

```
(x) => <snd(x), 1+snd(x)>
```

Binding and Scope

- Most language have a notions of
  - Variable binding (declaration of new variable)
  - Scope of variables (where variables can be referenced)

```
let x be 3 in (x + x)
```

  - Here x is a bound variable
  - The scope of x is the expression x + x

Bound Variables

- Every use of a bound variable refers to a binding

```
let x be 3 in x + x
```

```
let x be 10
   in ((let x be x+1 in x + x) + x)
```

- Nested bindings of same variable called "shadowing"
  - Usual rule: use of variable refers to the variable with the innermost enclosing scope.

Free Variables

- Variables in a term not in the scope of a definition of that variable are said to be "free"

```
let x be 3 in (x + y)
```

(Here x is bound and y is free)

```
x + (let x be 3 in x)
```

(Here x occurs both bound and free)

Free Variables

- Free-ness of variables is relative

```
let x be 3 in (x + y)
```

Here x and y are free in the "body" of the let, but x is *not* free in the entire expression.

Substitution

- Replacing *free* variables with terms
- Written *exp*[*var*←*exp'*]

```
(x + (let x be 3 in x + y))[y←4] =
```

```
(x + (let x be 3 in x + y))[x←4] =
```

Substitution

- Careful: substitution is an operation on abstract syntax (trees)!

```
(x + x)[x←y+1]  =  ?
```

Implementing Substitution

- Define a function

```
subst : absyn*string*absyn -> absyn
```

A First Interpreter

- When should an expression be considered a complete program?
- What are the possible results for these programs?

A First Interpreter

- Define a function

```
eval : absyn -> absyn
```

  using substitution.