Parsing and Interpretation

September 22, 2011

CS 42: Principles and Practice of Computer Science

Yes/No Recursive Descent in the Abstract

For each nonterminal in your grammar

```
A -> X ; Y ! | T D & H | B
```

there is a function whose form follows the grammar:

```
match-A():
   if ( ??? ):
      match-X()
      check-for-and-consume ( ; )
      match-Y()
      check-for-and-consume ( ! )
   else if ( ??? ):
      match-T()
      match-D()
      check-for-and-consume ( & )
      match-H()
   else if ( ??? ):
      consume B()
   else
      parse error
```

where the tests `???` are efficient.

Making Efficient Decisions

If our grammar has the rule

```
B -> a W W | b Y
```

then what should our tests be?

```
match-B:
   if ( ??? ):
      consume ( a )
      match-W()
      match-W()
  else if ( ??? ):
      consume ( b )
      match-Y()
   else
      abort "Parse error"
```

Making Decisions

If our grammar has the rules

```
B -> W W | Y
W -> e W | f
Y -> g Y | h
```

then what should our tests be?

```
match-B:
    if ( ??? ):
        match-W()
        match-W()
    else if ( ??? ):
        match-Y()
    else
        report "Parse error"
```

Subtleties: Delaying Decisions ("Left Factoring")

If two choices start the same way, it's hard to make an immediate decision. Solution: delay the decision!

```
;; S -> P + S | P

match-S():
  if ( ??? ) :
     match-P()
     check-for-and-consume ( + )
     match-S()

  else:
     match-P()
```

```
;; S -> P + S | P

match-S():
   match-P()

   if ( ??? ):
      consume ( + )
      match-S()
```

Subtleties: Building Trees

Each function rather than succeeding or failing, returns abstract syntax.

```
;; S -> P + S | P

parse-S():
   p = parse-P()

   if ( ??? ):
      consume ( + )
      s = parse-S()
      return <addition tree with p and s as subtrees>

   else:
      return p
```

Subtleties: Consuming Input Functionally

Each function takes a list of tokens, uses some of them to construct the AST, and returns any leftover tokens.

```
;; S -> P + S | P

parse-S(tokens):
   p, after-p = parse-P(tokens)

   if ( ??? ):
      check that after-p starts with +
      s, after-s = parse-S( (rest after-p) )
      return <addition tree with p and s as subtrees>,
             after-s

   else:
      return p, after-p
```

Subtleties: Racket

```
;; S -> P + S | P
;; Input: a list of tokens
;; Output: the ast, and the list of leftover tokens.
(define (parse-s tokens)
   (let* ([pair    (parse-p tokens)]
          [p       (first pair)]
          [after-p (second pair)])
     (if (equal? (first after-p) #\+)
         (let* ([pair2   (parse-s (rest after-p))]
                [s       (first pair2)]
                [after-s (second pair2)]
                [ast     (list 'add p s)])
            (list ast after-s))
         (list p after-p))))
```

Grammar for the Homework

```
T -> def V L | L
L -> # E | E
E -> P + E | P - E | P
P -> K * P | K / P | K
K -> U ^ I | U
U -> ( L ) | - U | Q
Q -> D ~ D V* | D V* | V V*
        where V* means 0 or more V's

I -> <any integer> | - <any integer>
D -> any *positive* number
V -> any variable name (a string token)
```

How do we know whether the top-level input is a definition or not?
How do we make the right decision for `parse-u`?

Recursive Descent Weaknesses

It's sometimes impossible to make an efficient decision.
The most common cause is "Left Recursion."

```
;; S -> S + P | P

parse-S():
   if ( ??? ):
      s = parse-S()
      consume ( + )
      p = parse-P()
      return <addition tree with s and p as subtrees>

   else:
      p = parse-P()
      return p
```

Handling Left Recursion

Most common solution:

```
S -> P | P + P | P + P + P | ...
```

Use a looping function that gathers as many `+ P`'s as it can.
Turn this list into a left-associative tree.

An evaluator for additions

Recursive Tree Traversal!

```
;; e.g., (eval '(add (add 3 4) 5)) ===> 12

(define (eval T)
  (cond [(number? T)  T]
        [(equal (first T) 'add)
           (+ (eval (second T)) (eval (third T)))]
        [else
           (error "bad input")]))
```

What if *we wanted to allow additions* *and* subtractions?

Variables?

What about `'(add (add x y) 5))`?

Variables?

What about `'(add (add x y) 5))`?

Solution: an *environment* (lookup table) that associates variables with their values.

```
(define (eval env T)
 (cond [(number? T)  T]
       [(symbol? T)
          (lookup env T)]
       [(equal (first T) 'add)
          (+ (eval env (second T))
             (eval env (third T)))]
       [else
          (error "bad input")]))
```

```
(define global-env ...)

(define (eval T)
   (cond [(number? T)  T]
         [(symbol? T)
            (lookup global-env T)]
         [(equal (first T) 'add)
            (+ (eval (second T))
               (eval (third T)))]
         [else
            (error "bad input")]))
```

In the code given for the homework, **unicalc-db** is our global environment.

The functional approach has advantages, but you'd need to rewrite **normalize**, **add**, etc., to take the database-to-use as an extra argument, rather than referring directly to the global **unicalc-db**.

Defining Variables

Suppose we want to introduce a brand-new global variable `x` defined to be `7` (or change the value of an existing variable `x` to be 7)?

Defining Variables

Suppose we want to introduce a brand-new global variable `x` defined to be `7` (or change the value of an existing variable `x` to be 7)?

Functional Approach:

1. Define `env'` by adding `x` $\mapsto$ 7 to the existing `env`.
2. From now on, use `env'`.

Imperative Approach:

1. Modify the global environment so that `x` is now 7.

Functions and Local Variables

Global variables exist forever. Local variables appear and disappear.

How might we evaluate `'(apply-function f 7)` where `f` takes a parameter `x` and returns the value of `'(add x 1)`?

Functions and Local Variables

Global variables exist forever. Local variables appear and disappear.

How might we evaluate `'(apply-function f 7)` where `f` takes a parameter `x` and returns the value of `'(add x 1)`?

Functional Approach:

1. Look up the parameters and body of `f`.
2. Define `env'` by adding `x` $\mapsto$ 7 to the existing `env`
3. Recursively evaluate `'(add x 1)` using `env'`.
4. Ignore `env'` after we're done; we don't need it any more. Go back to using `env` which never changed.

Imperative Approach:

1. Look up the parameters and body of `f`.
2. Save the old global value of `x` (if any)
3. Set the parameter `x` to be `7` in the global environment.
4. Recursively evaluate `'(add x 1)`, save the result.
5. Restore the old global value of `x` (if any)
6. Return the result.

Warning: this basically works, but gives you "dynamic scope," which can lead to weird behavior.

You really want "closures," as discussed in the Programming Languages class (CS 131).