2013W1-lecture2

September 16, 2013

Contents

1 Question of the Day **1**
1.1 Solution 2

2 DONE Logistics **3**
2.1 **DONE** Prereq letters 3
2.2 **DONE** Programming Assignment and Quiz dates/times . . . 3
2.3 **DONE** How Do I Get Steve's {Awesome, Primitive, Whacked-Out} Text Editor? 3
2.4 Meet your TAs! 4

3 DONE Finishing Last Lecture :) **4**

4 Concrete Syntax, Parsing, and Abstract Syntax **4**
4.1 Extending the Parser to "Non-Deterministic Values" 6

5 What have we learned today? **8**

1 Question of the Day

You have a collection of associations: names and values. (For example, the variables in a function.) Given a name, you want to find and/or set its value.

Brainstorm *fast* ways to do it.

What are some of the fastest your neighborhood can come up with?

1.1 Solution

An unordered list of key/value pairs would be a pretty slow way to do it, but that's what we'll do for a while. Why? Because it won't really be "unordered". We'll use the *order* to describe something else: *scope*, where and when an identifier (like a variable name) is visible.

Other faster ways:

- Ordered list?
- Binary search tree
- Self-balancing binary search tree
- Splay trees (whatever they are)
- Hash Table (not even close to the fastest way)

Now, what if all the keys were just indexes of an array? Lookup would be blazingly fast. That's effectively what compilers do with variable lookup: statically replace the lookup with the address of the value!

Try compiling this code in C to assembly and look for the *names* of the variables. Are they there?

```
int i;
int n;
int sum = 0;

/* When I execute this inside the block, it doesn't ask me for a
   number; it just uses 47 for some reason.  Well, OK! */
scanf("Enter a positive integer: %d", &n);

for (i = 1; i <= n; i++)
  sum += i;

printf("The sum from 1 to %d is: %d.\n", n, sum);
```

Below is the loop in `main`. Check out the line before `L2:`, where we add `1` to `i`. `i` itself (the name) is gone. `i`'s value is stored 28 bytes after the address stored in `%esp`.

`%esp` is the "stack pointer". It basically holds the address where the current function's local variables live.

Can you find `n` and `sum`?

```
    ...
        movl    $1, 28(%esp)
        jmp     L2
L3:
        movl    28(%esp), %eax
        addl    %eax, 24(%esp)
        addl    $1, 28(%esp)
L2:
        movl    20(%esp), %eax
        cmpl    %eax, 28(%esp)
        jle     L3
    ...
```

2 DONE Logistics

2.1 DONE Prereq letters

2.2 DONE Programming Assignment and Quiz dates/times

- PAs: due Fridays at 8PM right now (next one: 13 Sep @ 8PM!)
- Quizzes: start of class Wed's right now. (next one: Wednesday!)
  - Thoughts on how to allow those not attending to participate? (For pedagogical reasons, we *must* give the correct answers immediately after taking the quiz.)
- CAs: due Tuesdays at 8PM right now

2.3 DONE How Do I Get Steve's {Awesome, Primitive, Whacked-Out} Text Editor?

I use an editor called emacs. It does everything and the kitchen sink. Demonstration: M-x tetris.

Is emacs worth learning? Depends on whether you have tremendous patience and free time.

Inside emacs, Org Mode and its Babel module let me do all the nifty stuff you've seen in class. (I've made a few minor customizations; if you have trouble getting what I get, post to Piazza. As soon as I have time, I want to extend Babel to allow Racket or find where someone has already done so.)

2.4 Meet your TAs!

At the end of class.

3 DONE Finishing Last Lecture :)

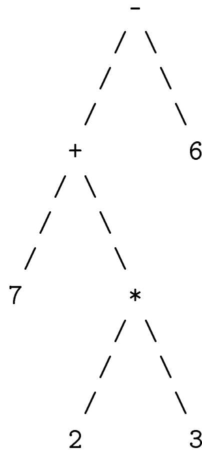
file:2013W1-lecture1.org

4 Concrete Syntax, Parsing, and Abstract Syntax

What is `7 + 2 * 3 - 6`?

How do you know?

Let's draw a version that's **better** for performing the implied computation.



```
        -
       / \
      /   \
     /     \
    +       6
   / \
  /   \
 /     \
7       *
       / \
      /   \
     /     \
    2       3
```

`7 + 2 * 3 - 6` is *contrete syntax*, the text the programmer (or some other program!) creates.

Our drawing is *abstract syntax*: an internal representation of the program—usually in tree form: an *abstract syntax tree*—designed to be easy to for the next stage to use.

In this case, **we're** the next stage. If an interpreter to run our program were the next stage, a drawing would **not** be a good abstract syntax. Instead, we use a recursive datatype like the `ArithC` the book introduced, but it's the **same idea** as the tree:

```
(define-type ArithC
```

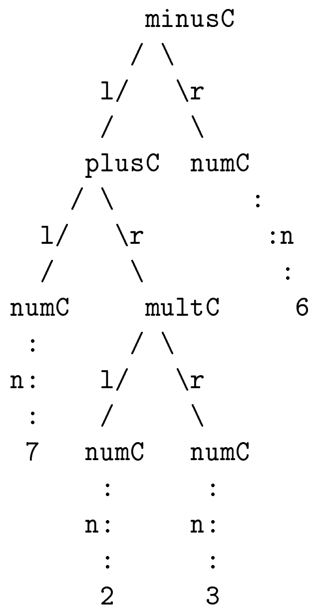
```
  [numC (n : number)]
  [plusC (l : ArithC) (r : ArithC)]
  [multC (l : ArithC) (r : ArithC)]
  [minusC (l : ArithC) (r : ArithC)]) ; added to express the '-'

;; Wait a minute.  CAN we parse 7 + 2 * 3 - 6 into this datatype?
;; It's not in the "s-expression" format like read consumes.
(minusC (plusC (numC 7) (multC (numC 2) (numC 3))) (numC 6))
```

```
  SOLUTION
 #+BEGIN_latex
 \begin{verbatim}
 ;; Can you see the tree structure?
 (minusC (plusC (numC 7)
                (multC (numC 2)
                       (numC 3)))
         (numC 6))
```

+END$_{l}atex$

Can we draw a tree representing an ArithC value?



Parsers turn concrete syntax into abstract syntax. They can even let us match up **different** concrete syntaxes with the same abstract syntax.

Interpreters or compilers (and other processes we'll learn about later) act on that abstract syntax.

And.. that's about it for parsers!

4.1 Extending the Parser to "Non-Deterministic Values"

If we have extra time, let's extend the book's parser and AST to allow a new construct: "non-deterministic values". A non-deterministic value is one of a set of provided numbers, but we don't know which one for sure. We might write them like this in infix notation:

```
[2, 3, 5] + [8, 10]
```

Or like this in Racket-like notation:

```
(+ (? 2 3 5) (? 8 10))
```

Is there anything else we should decide before proceeding?

- order of operations (BEDMAS??)
- What does ? do? What's its semantics? (Is it exactly one of 2 3 5? Is it some subset?)
- Can we still write `(+ 2 3)`? Sure. `(+ (?  2) (?  3))`
- Can we write `(+ (?  )  3)`? No, we decided.

```
; What's our syntax?  We'll use EBNF---Extended Backus-Naur Form---to
; describe it.

; In our version of EBNF, anything in <...> is expanded according to a
; rule.  The left side of the rule tells what we're expanding.  The
; right side tells what it can expand into.  '::=' separates the left
; and right.  '|' separates options on the right.  Anything else is
; just literally what the user would type, except our little comment
; about Racket numbers :)

; John Backus received the Turing Award in large part for developing
; this language for specifying the syntax of programming languages!

;; Original version:
```

```
; <expr> ::= <addition>
;          | <multiplication>
;          | <number>
;
; <addition> ::= (+ <expr> <expr>)
;
; <multiplication> ::= (* <expr> <expr>)
;
; <number> ::= a valid Racket number


;; Our new version:
; <expr> ::= <addition>
;          | <multiplication>
;          | <nondetnum>
;
; <addition> ::= (+ <expr> <expr>)
;
; <multiplication> ::= (* <expr> <expr>)
;
; <nondetnum> ::= (? <non-empty-list-of-numbers>)
;
; <non-empty-list-of-numbers> ::= <number>
;                               | <number> <non-empty-list-of-numbers>
;
; <number> ::= a valid Racket number

(define-type ArithC
  [numC (n : number)]
  [plusC (l : ArithC) (r : ArithC)]
  [multC (l : ArithC) (r : ArithC)])

; Consumes concrete syntax and produces an ArithC AST.
; (Reports an error for invalid syntax.)
(define (parse [s : s-expression]) : ArithC
  (cond
    [(s-exp-number? s) (numC (s-exp->number s))]
    [(s-exp-list? s)
     (let ([sl (s-exp->list s)])
       (case (s-exp->symbol (first sl))
```

```
      [(+) (plusC (parse (second sl)) (parse (third sl)))]
      [(*) (multC (parse (second sl)) (parse (third sl)))]
      [else (error 'parse "invalid list input")]))]
  [else (error 'parse "invalid input")]))
```

5 What have we learned today?

- From the QotD
  - Define the term "environment": a program's mapping from identifiers (names) to values. (Note that later on we'll look at how different program analyses can make use of environments that map identifiers to other quantities.)
  - Compare and contrast static and dynamic solutions to a problem. (In this case, statically and dynamically mapping identifiers to values.)
  - (Begin on:) Discriminate among concepts that impact the semantics of a programming language and those that do not. (The form we use to implement a program's environment does not (necessarily) have any semantic effect. Use of hash tables, unordered association lists, and static replacement of identifiers with addresses–used appropriately–all produce programs with the same meaning.)
  - Believe that you can quickly begin to assess a new programming language with the tools you learn in this course. (Not really an assessable goal, but I hope you'll start feeling that way as we do examples like these!)
- Syntax and Parsers
  - Define the terms "concrete syntax", "abstract syntax", "abstract syntax tree", and "parser".
  - Translate among: (1) concrete syntaxes that you've studied explicitly or reasonably natural new concrete syntaxes, (2) a provided abstract syntax, and (3) sketched tree representations.
  - Given a syntax, determine what purpose it might be best suited to, and justify your answer.
  - Design and understand simple parsers for given concrete/abstract syntaxes.

- EBNF
  - Roughly define the term EBNF (Extended Backus Naur Form). (A standard language for describing syntaxes.)
  - Given the EBNF for a language:
    * Determine whether some small snippets of code are legal syntax.
    * Write some small snippets of code.
  - Given a desired modification to a language and an EBNF, modify the EBNF to describe the new language.