2013W1-lecture3

September 16, 2013

Contents

1 Question of the Day

What language is your {Racket, Java, C} {compiler, interpreter} written in?

1.1 Solution

Well, mine are:

- The Racket interpreter provided by the PLT group, which includes source code directories like this one (just chose some tasty-looking innards): http://git.racket-lang.org/plt/tree/HEAD:/racket/src/racket/src

- The Java reference implementation which will, at least soon, be the OpenJDK implementation, which includes source directories like this one (just chose the parser): http://hg.openjdk.java.net/jdk8/jdk8/langtools/file/3f274927ec18/src

- The gcc C compiler, which includes source directories like this one (again, just chose the parser): http://gcc.gnu.org/viewcvs/gcc/trunk/gcc/c/

So, what languages are they implemented in? Well, multiple, in general, but mainly in the parts we're looking at:

Language compiled/interpreted	Main implementation language
Racket	C
Java	Java
C	C

What?!

Well, once you've implemented *one* compiler for your language, you can always write the next compiler in... your language. It's rather a natural choice. Indeed, for Java, for example, it leads to the nifty statement from the Hitchhiker's Guide to javac:

```
All the tests should pass, all the time, on all platforms.
```

There's also some interesting problems with this. Interested? Read: Reflections on Trusting Trust, Turing Address by Ken Thompson.

No learning goals from this one, really. So hopefully I got through it quickly!

2 Logistics

- **Join Piazza!** Next time, I might just post a list of names of people who haven't yet!
- First quiz next time at the start of class: covers everything up to and including the reading for next class.

3 Finishing Last Lecture :)

file:2013W1-lecture2.org

4 Interpreters, Expressions/ASTs, and Values

Last time we modified PLAI's parser for the arithmetic language to handle "non-deterministic" values. This time, let's extend the given interpreter to interpret our new core abstract syntax.

Reminder: we used to support numbers, addition, and multiplication. Now, we're going to support non-deterministic numbers and addition and multiplication over these. We have to decide what these things mean to proceed.

Let's look at the parts of our interpreter:

```
#lang plai-typed

; 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!

; <expr> ::= <addition>
;          | <multiplication>
;          | <number>
;
; <addition> ::= (+ <expr> <expr>)
;
; <multiplication> ::= (* <expr> <expr>)
;
; <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))]
    [(and (s-exp-list? s) (= (length (s-exp->list s)) 3))
     (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")]))

; Test the "standard" cases:
(test (parse ’1) (numC 1))
(test (parse ’(+ 1 2)) (plusC (numC 1) (numC 2)))
(test (parse ’(* 1 2)) (multC (numC 1) (numC 2)))

; Test one "complex" case, just to be safe.
(test (parse ’(+ (* 1 2) (+ 3 4))) (plusC (multC (numC 1) (numC 2))
                                          (plusC (numC 3) (numC 4))))

; Test various erroneous programs:
(test/exn (parse ’(1 2 3)) "")
(test/exn (parse ’()) "")
(test/exn (parse ’(+)) "")
(test/exn (parse ’(+ 1)) "")
(test/exn (parse ’(+ 1 2 3)) "")
(test/exn (parse ’(*)) "")
(test/exn (parse ’(* 1)) "")
(test/exn (parse ’(* 1 2 3)) "")
(test/exn (parse ’(- 1 2)) "")
(test/exn (parse ’(- 1 2)) "")

; Test some especially odd inputs that are hard to even construct.
(test (parse (number->s-exp 1)) (numC 1))
(test/exn (parse (string->s-exp "hello")) "")
(test/exn (parse (symbol->s-exp ’hello)) "")

; interpret (evaluate) the given well-formed arithmetic expression AST
(define (interp [ast : ArithC]) : number
  (type-case ArithC ast
    [numC (n) n]
    [plusC (l r) (+ (interp l) (interp r))]
    [multC (l r) (* (interp l) (interp r))]))

; run the given (concrete syntax) program to produce its value
(define (run [program : s-expression]) : number
  (interp (parse program)))

; Test the "standard" cases:
(test (run '1) 1)
(test (run '(+ 1 2)) (+ 1 2))
(test (run '(* 1 2)) (* 1 2))

; Test one "complex" case, just to be safe.
(test (run '(+ (* 1 2) (+ 3 4))) (+ (* 1 2) (+ 3 4)))
```

We have:

- EBNF/concrete syntax, instances of which are our programs
- Parser
- Abstract syntax, instances of which are, at least for now, our expressions
- Interpreter
- and.. what? What does the interpreter produce?

```
 SOLUTION
The interpreter produces VALUES.  Its job is EVALUATION.  We don't
have our own type for values yet.  Instead, we use numbers.
```

So, what do we need to change here? Does the parser still consume concrete syntax (s-expressions) and produce abstract syntax (ArithC's)? Does the interpreter still consume abstract syntax and produce values (numbers)?

We'll also need to decide on the *semantics* of our operations. For example, what should `(+ (?  1 2) (?  3 4))` evaluate to? That may seem obvious to you, but here's a few options:

- `(?  4 6)`
- `(?  4 5 5 6)`
- `(?  4 5 6)`

- `(?  6 5 4)`

(Do we care about order? Do we care about repeats?)

Let's write in English what we want our semantics of "non-deterministic numbers" to be:

```
SOLUTION
```

Good. Now, let's write out the pieces we'll need to build:

```
SOLUTION
For tests: something to normalize the form of a non-deterministic
number value so we can see (? 4 5 6) and (? 6 5 4) as equal.
```

5 What have we learned today?

- From the QotD
  - Steve likes to ask QotDs.
- Interpreters and Values
  - Define the terms: value and evaluate.
  - Describe the job of an interpreter in terms of expressions, values, and evaluation.
  - Explain with an example how a single concrete syntax can have multiple different semantics.
  - Modify an existing interpreter to support a small change to one or more of the semantics, abstract syntax, and values that it operates on.