CS4613 Lecture 2

David Bremner

January 10, 2024

Simplified Calculator Parser

If we choose an S-expression based surface syntax, we can simplify our parser.

p. 35

parse

```
(define (parse s)
  (local
      [(define (sx n) (list-ref (s-exp->list s) n))
       (define (px n) (parse (sx n)))
       (define (? pat) (s-exp-match? pat s))]
    (cond
      [(? `NUMBER) (num (s-exp->number s))]
      [(? `(+ ANY ANY)) (plus (px 1) (px 2))]
      [(? `(* ANY ANY)) (times (px 1) (px 2))]
      [else (error 'parse (to-string s))])))
```

Testing the new parser

parse

```
(test (parse `{* {+ 1 2} {+ 3 4}})
        (times
         (plus (num 1) (num 2))
         (plus (num 3) (num 4))))
(test/exn (parse `{1 + 2}) "")
```

- How could the negative test be improved?

Connecting the parser and evaluator

p. 36

run

```
(run : (S-Exp -> Number))
(define (run s)
  (calc (parse s)))

(test (run `{+ 1 {+ 2 3}}) 6)
(test (run `{* {+ 2 3} {+ 5 6}}) 55)
```

- more convenient to write tests (and read them)
- more layers to update for new features

Design Choices

Simple conditional

`(if test-ex then-ex else-ex)`

- `test-ex` is evaluated first, and if 'true' (whatever that means!) `then-ex` is evaluated, otherwise `else-ex` p. 38
- Building block for e.g. short circuit evaluation, `cond`

What is truth?

- Trade off between convenience/conciseness and (bad) surprises.
- Defining a small set of "falsy" values is a reasonable option.
- With only numbers, we will define `if0`, with 0 as `true`

A SImPl Plan for a new feature

Extend datatype add constructor

p. 38

Extend Evaluator new case for `type-case`

Extend Parser (if any)

```
(define-type Exp
  [num (n : Number)]
  [plus (left : Exp) (right : Exp)]
  [times (left : Exp) (right : Exp)]
  [cnd (test : Exp) (then : Exp) (else : Exp)])
```

A SImPl Plan for a new feature

Extend datatype add constructor

Extend Evaluator new case for type-case p. 41

Extend Parser (if any)

```
(define (calc e)
  (type-case Exp e
    [(num n) n]
    [(plus l r) (+ (calc l) (calc r))]
    [(times l r) (* (calc l) (calc r))]
    [(cnd c t e) (if (zero? (calc c))
                     (calc t)
                     (calc e))]))
```

Updated parser

if0

```
(define (parse s)
  (local
      [(define (sx n) (list-ref (s-exp->list s) n))
       (define (px n) (parse (sx n)))
       (define (? pat) (s-exp-match? pat s))]
    (cond
      [(? `NUMBER) (num (s-exp->number s))]
      [(? `(+ ANY ANY)) (plus (px 1) (px 2))]
      [(? `(* ANY ANY)) (times (px 1) (px 2))]
      [(? `(if0 ANY ANY ANY))                    ;; NEW
       (cnd (px 1) (px 2) (px 3))]
      [else (error 'parse (to-string s))])))
```

Motivation for value type: adding Boolean

p. 41

```
(define (calc e)
  (type-case Exp e
    [(num n) n]
    [(bool b) b]
    ⋮))
```

- In a statically typed language like plait a function returns one type.
- Interpreters are often implemented in statically typed languages.

Motivation for value type: adding Boolean

p. 41

```
(define (calc e)
  (type-case Exp e
    [(num n) n]
    [(bool b) b]
    ⋮))
```

- In a statically typed language like plait a function returns one type.
- Interpreters are often implemented in statically typed languages.

1. Also in most other, but not all statically typed languages.
2. Why do you think statically typed languages are a common choice for “infrastructure”?

Defining datatypes

Need to distinguish (unevaluated) expressions from values. p. 42

```
(define-type Exp
  [numE (n : Number)]
  [boolE (b : Boolean)]
  [plusE (left : Exp) (right : Exp)]
  [timesE (left : Exp) (right : Exp)]
  [cndE (test : Exp) (then : Exp) (else : Exp)])
```

One constructor per (evaluated) type

```
(define-type Value
  [numV (the-number : Number)]
  [boolV (the-boolean : Boolean)])
```

Defining datatypes

Need to distinguish (unevaluated) expressions from values.

p. 42

```
(define-type Exp
  [numE (n : Number)]
  [boolE (b : Boolean)]
  [plusE (left : Exp) (right : Exp)]
  [timesE (left : Exp) (right : Exp)]
  [cndE (test : Exp) (then : Exp) (else : Exp)])
```

One constructor per (evaluated) type

```
(define-type Value
  [numV (the-number : Number)]
  [boolV (the-boolean : Boolean)])
```

1. Renaming of `Exp` constructors is optional, nothing would break if we kept the old names. On the other hand, we will see a few places where the distinction is important.

New return type for evaluator

p. 44

```
(calc : (Exp -> Value))
(define (calc e)
  (type-case Exp e
    [(numE n) (numV n)]
    [(boolE b) (boolV b)]
    ⋮))
```

The following has multiple type issues. What are they?

```
[(plusE l r) (+ (calc l) (calc r))]
```

What this problem needs is more indirection

```
(define (num-op op expr1 expr2)
  (local [(define (unwrap v)
            (type-case Value v
              [(numV n) n]
              [else (error 'num-op "NaN")]))]
    (numV (op (unwrap expr1)
              (unwrap expr2)))))
```

Now our arithmetic cases looks like

```
[(plusE l r) (num-op + (calc l) (calc r))]
[(timesE l r) (num-op * (calc l) (calc r))]
```

What this problem needs is more indirection

```
(define (num-op op expr1 expr2)
  (local [(define (unwrap v)
            (type-case Value v
              [(numV n) n]
              [else (error 'num-op "NaN")]))]
    (numV (op (unwrap expr1)
              (unwrap expr2)))))
```

Now our arithmetic cases looks like

```
[(plusE l r) (num-op + (calc l) (calc r))]
[(timesE l r) (num-op * (calc l) (calc r))]
```

1. The book uses a simpler function `add` because there is only one arithmetic operation.

Updating conditional

We saw the question of what to consider as truthy is surprisingly complicated.

p. 45

```
[(cndE c t e) (if (boolean-decision (calc c))
                  (calc t)
                  (calc e))]
```

The book's version is strict:

```
(define (boolean-decision v)
  (type-case Value v
    [(boolV b) b]
    [else (error 'if "not a boolean")]))
```

Updating conditional

We saw the question of what to consider as truthy is surprisingly complicated.

p. 45

```
[(cndE c t e) (if (boolean-decision (calc c))
                  (calc t)
                  (calc e))]
```

The book's version is strict:

```
(define (boolean-decision v)
  (type-case Value v
    [(boolV b) b]
    [else (error 'if "not a boolean")]))
```

1. In functional programming, any time something seems complicated, the usual way to break into more tractable pieces is to define a function. If nothing else, the name of a function acts as documentation

Alternative conditional semantics

```
(define (boolean-decision v)
  (type-case Value v
    [(boolV b) b]
    [(numV n) (not (zero? n))]))
```

- This is convenient, but what should we do when values can be functions?

E-Value-ator

calc

```
(define (calc e)
  (type-case Exp e
    [(numE n) (numV n)]
    [(boolE b) (boolV b)]
    [(plusE l r) (num-op + (calc l) (calc r))]
    [(timesE l r) (num-op * (calc l) (calc r))]
    [(cndE c t e) (if (boolean-decision (calc c))
                      (calc t)
                      (calc e))]))

(test (calc (plusE (numE 3) (numE 4))) (numV 7))
(test (calc (cndE (boolE #t) (numE 0) (numE 1)))
      (numV 0))
```