

CS4613 Lecture 1

David Bremner

September 3, 2025

Syllabus

- ▶ <https://www.cs.unb.ca/~bremner/teaching/cs4613/printable/>
- ▶ <https://www.cs.unb.ca/~bremner/teaching/cs4613/evaluation/>

Online resources

- ▶ <https://www.cs.unb.ca/~bremner/teaching/cs4613/>



- ▶ D2L will only be used for handing in online **SMoL tutorials**
- ▶ Programming assignments will be handed via a custom **handin server**
- ▶ Marked programming assignments will be returned via the same server.

Getting started

Install racket

<https://download.racket-lang.org>

Customize [https://www.cs.unb.ca/~bremner/
teaching/cs4613/racket/setup](https://www.cs.unb.ca/~bremner/teaching/cs4613/racket/setup)

Documentation <https://docs.racket-lang.org>

SMoL: Standard Model of Languages

SMoL Core language containing features used to build many common languages.

p. 14

tutor <https://www.cs.unb.ca/~bremner/teaching/cs4613/smol-tutorials>

reference `raco doc smol` or <https://www.cs.unb.ca/~bremner/teaching/cs4613/docs>

```
1 #lang smol/fun
  (defvar x 10)
  (defun (f y) (+ x y))
  (f 3)
```

Plait: statically typed racket

reference `raco doc plait` or <https://www.cs.unb.ca/~bremner/teaching/cs4613/docs>
tutorials <https://www.cs.unb.ca/~bremner/teaching/cs4613/tutorials>
tour <https://www.cs.unb.ca/~bremner/teaching/cs4613/racket/plait-demo.rkt/>

Interpreters

- ▶ An **interpreter** maps programs to values (+ side effects).
- ▶ A **compiler** translates programs to other programs, typically lower level.
- ▶ Most modern languages use a mix of the two evaluation strategies

Substitution

- ▶ The simplest model of evaluation is **substitution**
- ▶ Consider the following SMoL program

```
smol2 (defun (f x) (+ x 1))  
stacker (f 3)
```

- ▶ We can evaluate it by **substituting** the argument in the function body

```
(f 3)  
→ (+ x 1) [3/x]  
→ (+ 3 1)  
→ 4
```


Substitution continued

Building on the previous example

```
smol3 (defun (f x) (+ x 1))  
      (defun (g z)  
        (f (+ z 4)))  
stacker (g 5)
```

We can evaluate in the same way:

p. 19

```
(g 5)  
→ (f (+ z 4)) [5/z]  
→ (f (+ 5 4)) → (f 9)  
→ (+ x 1) [9/x]  
→ (+ 9 1) → 10
```

Design choices 1: lazy vs. eager

p. 20

Eager

→ (f (+ 5 4))
→ (f 9)
→ (+ x 1) [9/x]
→ (+ 9 1)

Lazy

(f (+ 5 4))
→ (+ x 1) [(+ 5 4)/x]
→ (+ (+ 5 4) 1)

Design choices 2: sequential versus parallel

p. 21

Sequential

$(+ (f\ 3) (f\ 4))$
→ $(+ (+\ x\ 1) [3/x] (f\ 3))$
→ $(+ (+\ x\ 1) [3/x] (+\ x\ 1) [4/x])$

Parallel

$(+ (f\ 3) (f\ 4))$
→ $(+ (+\ x\ 1) [3/x] (+\ x\ 1) [4/x])$

Surface Syntax: Arithmetic Expressions



Consider a grammar (EBNF) for arithmetic with addition and multiplication

p. 47

grammar

```
ae: fac "+" ae  
   | fac
```

```
fac: atom "*" fac  
    | atom
```

```
atom: NUMBER | "(" ae ")"
```

Concrete syntax



driver

```
(parse-string "1 + 2 * 3")
(parse-string "1 * 2 + 3")
(parse-string "(1 + 2) * (3 + 4)")

:
'(ae
  (fac
    (atom "(" (ae (fac (atom 1)) "+" (ae (fac (atom
      2)))))) ")")
    "*"
    (fac (atom "(" (ae (fac (atom 3)) "+" (ae (fac
      (atom 4)))))) ")"))))
```

Abstract Syntax

- ▶ `define-type` provides **Algebraic Data Types** for `plait`
- ▶ We use them as **programs encoding programs**

p. 25

`exp`

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

Parsing S-Expressions

p. 35

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

└ SImPl: Standard Implementation Plan

└ Parsing S-Expressions

```

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

```

1. In a sense this is a *compiler*: it translates one representation of a program to another
2. There is one case per grammar rule here, because the output from the brag parser has the same structure for each rule
3. See the text for a more direct way of parsing s-expressions; here we rely on `s-exp-match?` to replace those tests.
4. The local functions are used just to reduce boilerplate (and fit the parser on the page). `'?`' looks exotic, but it just an identifier for Racket

Testing our parser

p. 36

parse

```
(test
  (parse-s-exp
    `(ae (fac (atom 1)) "+"
          (ae (fac (atom 2)) "*" (fac (atom 3))))))
  (plus (num 1)
        (times (num 2) (num 3))))

(test
  (parse-s-exp
    `(ae (fac (atom 1)) "*" (fac (atom 2))) "+"
          (ae (fac (atom 3))))))
  (plus (times (num 1) (num 2))
        (num 3)))
```

└─ SImPl: Standard Implementation Plan

└─ Testing our parser

1. test is going to be very important in this course
2. test uses `equal?` for equality testing

```
(test
  (parse-s-exp
    '(ae (fac (atom 1)) "+"
      (ae (fac (atom 2)) "*" (fac (atom 3)))))
    (plus (num 1)
      (times (num 2) (num 3))))
  (test
    (parse-s-exp
      '(ae (fac (atom 1)) "*" (fac (atom 2))) "+")
      (ae (fac (atom 3)))))
    (plus (times (num 1) (num 2))
      (num 3)))
```

Recursive Evaluation

The important part

In this course we want to focus on the **back end** of interpreters: processing (abstract) representations of programs.

p. 28

calc

```
(define (calc e)
  (type-case Exp e
    [(num n) n]
    [(plus l r) (+ (calc l) (calc r))]
    [(times l r) (* (calc l) (calc r))]))
```

Testing our evaluator

```
calc (test (calc (num 1)) 1)
(test (calc (num 2.3)) 2.3)
(test (calc (plus (num 1) (num 2))) 3)
(test (calc (plus (plus (num 1) (num 2))
                  (num 3))))
6)
```