Interpreters and “eval”

Monday, November 13, 2023
Logistics

• HW 6 in
  • due: Sun 11/12 11:59 pm EST

• HW 7 out
  • due: Sun 11/19 11:59 pm EST
    • Really due: Wed 11/22 11:59 pm EST
  • (no hw over Thanksgiving)
Syntax vs Semantics (Spoken Language)

**Syntax**
- Specifies: **valid language constructs**
  - E.g., `sentence = (subject) noun + verb + (object) noun`
  - “the ball threw the child”
  - Syntactically: **valid**!
  - Semantically: ???

**Semantics**
- Specifies: “**meaning**” of language (constructs)
Syntax vs Semantics (Programming Language)

Syntax
- Specifies: valid language constructs
  - E.g., valid Racket program: s-expressions
  - Valid python program: follows python grammar (including whitespace!)

Semantics
- Specifies: “meaning” of language (constructs)
Syntax vs Semantics (Programming Language)

**Syntax**
- Specifies: valid language constructs
  - E.g., valid **Racket** program: s-expressions
  - Valid **python** program: follows python grammar (including whitespace!)

**Semantics**
- Specifies: “meaning” of language (constructs)

---

**Q:** What is the “meaning” of a program?

**A:** The result of “running” it!

... but how does a program “run”?
Programs run on CPUs

Q: What is the “meaning” of a program?
A: The result of “running” it!

... but how does a program “run”?

Programmers don’t write machine code!

“low level”

“result”
Running Programs: `eval`

```haskell
;; eval : Program -> Result
;; “runs” a given “program”, producing a “result”
```

More generally:
An **interpreter**, i.e., an “eval” function, turns a “program” into a “result”

(But programs are usually not directly interpreted either)

More commonly, a **high-level** program is first **compiled** to a **lower-level** language (and then **interpreted**)

**Q:** What is the “meaning” of a program?
**A:** The result of “running” it!

... but how does a program “run”?
### From Lecture 1

#### "high" level (easier for humans to understand)

<table>
<thead>
<tr>
<th>English</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification langs</td>
<td>Types? pre/post cond?</td>
</tr>
<tr>
<td>Markup (html, markdown)</td>
<td>tags</td>
</tr>
<tr>
<td>Database (SQL)</td>
<td>queries</td>
</tr>
<tr>
<td>Logic Program (Prolog)</td>
<td>relations</td>
</tr>
<tr>
<td>Lazy lang (Haskell, R)</td>
<td>Delayed computation</td>
</tr>
<tr>
<td>Functional lang (Racket)</td>
<td>Expressions (no stmts)</td>
</tr>
<tr>
<td>JavaScript, Python</td>
<td>“eval”</td>
</tr>
<tr>
<td>C# / Java</td>
<td>GC (no alloc, ptrs)</td>
</tr>
<tr>
<td>C++</td>
<td>Classes, objects</td>
</tr>
<tr>
<td>C</td>
<td>Scoped vars, fns</td>
</tr>
<tr>
<td>Assembly Language</td>
<td>Named instructions</td>
</tr>
<tr>
<td>Machine code</td>
<td>Binary</td>
</tr>
</tbody>
</table>

More commonly, a **high-level** program is first **compiled** to a **lower-level** language (and then **interpreted**) (runs on cpu).

---

**NOTE:** This hierarchy is *approximate*.
More commonly, a high-level program is first compiled to a lower-level language (and then interpreted) (runs on cpu)

“high” level (easier for humans to understand)

specification languages
- Markup (html, markdown)
- Database (SQL)
- Logic Program (Prolog)
- Lazy lang (Haskell, R)

functional lang (Racket)
- JavaScript, Python
- C# / Java
- C++
- C
- Assembly Language
- Machine code

Common target languages:
- bytecode (e.g., JS, Java)
- assembly
- machine code

A virtual machine is just a bytecode interpreter
(A (hardware) CPU is just a machine code interpreter)
Semantics

• Specifies: meaning of language constructs
• So: to “run” a program, we need to construct the constructs first
Compiler, step 1 = parser

- surface language
- abstract syntax tree (AST)
- output

- input
- e.g., string of chars (no language constructs)
- e.g., tree (of language constructs)

A compiler actually has many steps (take a compilers course!)
This is a program!

surface language

parser

abstract syntax tree (AST)

input

These must have representations (data definitions!)

SExpr

(define (count-sl sl sl sy)
  (cond
    [(empty? sl) 0]
    [else
     (+ (count (first sl) sy) (count-sl (rest sl sy)))]))

output

These must have representations (data definitions!)
A SimpleSexpr (Sexpr) is one of:

- Number
- (list '+ Sexpr Sexpr)
- (list '-' Sexpr Sexpr)
An AST is one of:

- (num Number)
- (add AST AST)
- (sub AST AST)

(struct num [val])
(struct add [lft rgt])
(struct sub [lft rgt])
In-class Coding 11/8: parser

`; parse: SimpleSexpr -> AST
`; Converts a (simple) S-expression to a language AST

`; A SimpleSexpr (Sexpr) is a:
`; - Number
`; - (list '+ Sexpr Sexpr)
`; - (list '- Sexpr Sexpr)

`; An AST is one of:
`; - (num Number)
`; - (add AST AST)
`; - (sub AST AST)
;(struct num [val])
;(struct add [lft rgt])
;(struct sub [lft rgt])`
In-class Coding 11/8: parser

```scheme
(define (parse s)
  (match s
    [(? number?)        ...]
    [(`+ ,x ,y)
      ... (parse x) ... (parse y) ...]
    [(`- ,x ,y)
      ... (parse x) ... (parse y) ...]]))
```

```scheme
;; parse: SimpleSexpr -> AST
;; Converts a (simple) S-expression to a language AST

;; A SimpleSexpr (Sexpr) is a:
;; - Number
;; - (list ‘+ Sexpr Sexpr)
;; - (list ‘- Sexpr Sexpr)

;; An AST is one of:
;; - (num Number)
;; - (add AST AST)
;; - (sub AST AST)
```

(struct num [val])
(struct add [lft rgt])
(struct sub [lft rgt])
In-class Coding 11/8: parser

```scheme
;; parse: SimpleSexpr -> AST
;; Converts a (simple) S-expression to a language AST

;; A SimpleSexpr (Sexpr) is a:
;; - Number
;; - (list '+ Sexpr Sexpr)
;; - (list '-' Sexpr Sexpr)

(define (parse s)
  (match s
    [(? number?) (num s)]
    [(`+ ,x ,y)
      ... (parse x) ... (parse y) ... ]
    [(`- ,x ,y)
      ... (parse x) ... (parse y) ... ]))

;; An AST is one of:
;; - (num Number)
;; - (add AST AST)
;; - (sub AST AST)
(struct num [val])
(struct add [lft rgt])
(struct sub [lft rgt])
```
In-class Coding 11/8: parser

;; parse: SimpleSexpr -> AST
;; Converts a (simple) S-expression to a language AST

;; A SimpleSexpr (Sexpr) is a:
;; - Number
;; - (list '+ SimpleSexpr SimpleSexpr)
;; - (list '- SimpleSexpr SimpleSexpr)

(define (parse s)
  (match s
    [(? number?) (num s)]
    [('+ ,x ,y)
     (add (parse x) (parse y))]
    [('- ,x ,y)
     ... (parse x) ... (parse y) ... ])))

;; An AST is one of:
;; - (num Number)
;; - (add AST AST)
;; - (sub AST AST)
  (struct num [val])
  (struct add [lft rgt])
  (struct sub [lft rgt])
In-class Coding 11/8: parser

;;; parse: SimpleSexpr -> AST
;;; Converts a (simple) S-expression to a language AST

;;; A SimpleSexpr (Sexpr) is a:
;;; - Number
;;; - (list '+' Sexpr Sexpr)
;;; - (list '-' Sexpr Sexpr)

;;; An AST is one of:
;;; - (num Number)
;;; - (add AST AST)
;;; - (sub AST AST)

(define (parse s)
  (match s
    [((? number?) (num s))]
    [(`(+ ,x ,y)
      (add (parse x) (parse y)))]
    [(`(- ,x ,y)
      (sub (parse x) (parse y)))]))
In-class Coding 11/8 #2: run

;;; run: AST -> Result
;;; computes the result of given program AST

;;; A Result is one of:
;;; ;;
;;; ;; ????
;;; ;;

(define (run p)
  (cond
    [(num? p) ... (num-val p) ... ]
    [(add? p) ... (run (add-lft p)) ... 
      ... (run (add-rgt p)) ... ]
    [(sub? p) ... (run (sub-lft p)) ... 
      ... (run (sub-rgt p)) ... ]))

;;; An AST is one of:
;;; ;; - (num Number)
;;; ;; - (add AST AST)
;;; ;; - (sub AST AST)
(define (run p)
  (cond
    [(num? p) p]
    [(add? p) ... (run (add-lft p)) ... 
      ... (run (add-rgt p)) ... ]
    [(sub? p) ... (run (sub-lft p)) ... 
      ... (run (sub-rgt p)) ... ]))
In-class Coding 11/8 #2: run

;; run: AST -> Result
;; computes the result of given program AST

;; An AST is one of:
;; - (num Number)
;; - (add AST AST)
;; - (sub AST AST)

;; A Result is a:
;; - Number

(define (run p)
    (cond
        [(num? p) p]
        [(add? p) (+ (run (add-lft p))
                     (run (add-rgt p)))]
        [(sub? p) ... (run (sub-lft p)) ...]
        [(sub? p) ... (run (sub-rgt p)) ... ]))
In-class Coding 11/8 #2: run

;; run: AST -> Result
;; computes the result of given program AST

;; An AST is one of:
;; - (num Number)
;; - (add AST AST)
;; - (sub AST AST)

;; A Result is a:
;; - Number

(define (run p)
  (cond
    [(num? p) p]
    [(add? p) (+ (run (add-lft p))
                 (run (add-rgt p)))]
    [(sub? p) ... (run (sub-lft p)) ...
     ... (run (sub-rgt p)) ... ]))
In-class Coding 11/8 #2: run

```scheme
(define (run p)
  (cond
    [(num? p) p]
    [(add? p) (+ (run (add-lft p))
                  (run (add-rgt p)))]
    [(sub? p) (- (run (sub-lft p))
                (run (sub-rgt p)))]
  ))
```

;; run: AST -> Result
;; computes the result of given program AST

;; An AST is one of:
;; - (num Number)
;; - (add AST AST)
;; - (sub AST AST)

;; A Result is a:
;; - Number
The “CS450 LANG” Programming Language

```racket
;; parse : Sexpr -> AST
(define (parse ssexpr) ... )

;; run: AST -> Result
(define (run ast) ... )

(define (eval450 p) (compose run parse))

(define-macro (module expr ...) (eval450 expr ...) )

#lang s-exp "cs450-lang.rkt"

(+ 1 2) ; => 3

A program! written in “CS450 LANG”!
```
“CS450 Lang” Demo

• See cs450f23/inclass-lecture19 github repository
The "CS450 LANG + STRINGS" PL

;; A Ssexpr is a:
;; - Number
;; - (list '+ Ssexpr Ssexpr)
;; - (list '-' Ssexpr Ssexpr)

;; An AST is one of:
;; - (num Number)
;; - (add AST AST)
;; - (sub AST AST)
(struct num [val])

;; A Ssexpr is a:
;; - Number
;; - String
;; - (list '+ Ssexpr Ssexpr)
;; - (list '-' Ssexpr Ssexpr)

;; An AST is one of:
;; - (num Number)
;; - (str String)
;; - (add AST AST)
;; - (sub AST AST)
(struct num [val])
(struct str [val])
(struct add [lft rgt])
(struct sub [lft rgt])
;; parse: SimpleSexpr -> AST
;; Converts a (simple) S-expression to language AST

(define (parse s)
  (match s
    [(? number?) (num s)]
    [(? string?) (str s)]
    [(`+ ,x ,y) (add (parse x) (parse y))]
    [(`- ,x ,y) (sub (parse x) (parse y))]))

;; An AST is one of:
;; - (num Number)
;; - (str String)
;; - (add AST AST)
;; - (sub AST AST)
(struct num [val])
(struct str [val])
(struct add [lft rgt])
(struct sub [lft rgt])

;; A Ssexpr is a:
;; - Number
;; - String
;; - (list ‘+ Ssexpr Ssexpr)
;; - (list ‘- Ssexpr Ssexpr)
Running “CS450 Lang + Strings” Programs

;; run: AST -> Result
;; computes the result of given program AST

;; A Result is a:
;; - Number
;; - String

Let’s look at other languages!

;; An AST is one of:
;; - (num Number)
;; - (str String)
;; - (add AST AST)
;; - (sub AST AST)

What should happen when two strings are added???

e.g., What is the “meaning” of (+ “hello” “world!”)

(define (run p)
  (cond
    [(num? n) n]
    [(str? s) s]
    [(add? p) (???? (run (add-lft p)) (run (add-rgt p)))]
    [(sub? p) (???? (run (sub-lft p)) (run (sub-rgt p)))]))
JavaScript Semantics Exploration: “plus”
Introducing: The “CS450js” Programming Lang!

;;; A 450jsExpr is one of:
;;; - Number
;;; - String
;;; - (list ‘+ 450jsExpr 450jsExpr)
;;; - (list ‘- 450jsExpr 450jsExpr)

;;; A 450jsAST is one of:
;;; - (num Number)
;;; - (str String)
;;; - (add 450jsAST 450jsAST)
;;; - (sub 450jsAST 450jsAST)
  (struct num [val])
  (struct str [val])
  (struct add [lft rgt])
  (struct sub [lft rgt])

;;; A 450jsResult is one of:
;;; - Number
;;; - String
Parsing: “CS450js” Programs

;;; parse450js: 450jsExpr -> 450jsAST
;;; Converts a CS450js Lang surface program to its AST

;;; A 450jsExpr is one of:
;;; - Number
;;; - String
;;; - (list ‘+ 450jsExpr 450jsExpr)
;;; - (list ‘- 450jsExpr 450jsExpr)

(define (parse450js s)
  (match s
    [(? number?) (num s)]
    [(? string?) (str s)]
    [('+ ,x ,y) (add (parse450js x) (parse450js y))]
    [( '-', ,x ,y) (sub (parse450js x) (parse450js y))])))

;;; A 450jsAST is one of:
;;; - (num Number)
;;; - (str String)
;;; - (add 450jsAST 450jsAST)
;;; - (sub 450jsAST 450jsAST)
Running: “CS450js” Programs

;; run450js: 450jsAST -> 450jsResult
;; computes the result of running a CS450js program AST

;; A 450jsResult is either:
;; - Number
;; - String

;; A 450jsAST is one of:
;; - (num Number)
;; - (str String)
;; - (add 450jsAST 450jsAST)
;; - (sub 450jsAST 450jsAST)

(define (run p)
  (cond
    [(num? n) n]
    [(str? s) s]
    [(add? p) (???? (run (add-lft p)) (run (add-rgt p)))]
    [(sub? p) (???? (run (sub-lft p)) (run (sub-rgt p)))]))
Running: “CS450js” Programs

;;; run450js: 450jsAST -> 450jsResult
;;; computes the result of running a CS450js program AST

;;; A 450jsResult is either:
;;; - Number
;;; - String

;;; A 450jsAST is one of:
;;; - (num Number)
;;; - (str String)
;;; - (add 450jsAST 450jsAST)
;;; - (sub 450jsAST 450jsAST)

(define (run p)
  (cond
    [(num? n) n]
    [(str? s) s]
    [(add? p) (450js+ (run (add-lft p)) (run (add-rgt p)))]
    [(sub? p) (450js- (run (sub-lft p)) (run (sub-rgt p)))]))
Running: “CS450js” Programs: “plus”

```
;; 450js+: 450jsResult 450jsResult -> 450jsResult
;; “adds” two cs450js result values together
;; (follows js semantics)

;; A 450jsResult is either:
;; - Number
;; - String
```

```
(define (450js+ x y)
  (cond
    [(number? x) ... ]
    [(string? x) ... ]))
```

```
(define (450js+ x y)
  (cond
    [(number? y) ... ]
    [(string? y) ... ]))
```
Two-Argument Templates

• Sometimes ... a fn must process two arguments simultaneously
• This template should combine templates of both args
  • (This is only possible if the data defs are simple enough)

```scheme
(define (450js+ x y)
  (cond
    [(and (number? x) (number? y)) ... ]
    [(and (number? x) (string? y)) ... ]
    [(and (string? x) (number? y)) ... ]
    [(and (string? x) (string? y)) ... ]
  )
)
```

;; 450js+: 450jsResult 450jsResult -> 450jsResult
;; “adds” two cs450js result values together (following js semantics)

(2-argument) TEMPLATE (see why this is typically not recommended?)
;; 450js+: 450jsResult 450jsResult -> 450jsResult
;; “adds” two cs450js result values together (following js semantics)

(define (450js+ x y)
  (cond
   [(and (number? x) (number? y)) ... ]
   [(and (number? x) (string? y)) ... ]
   [(and (string? x) (number? y)) ... ]
   [(and (string? x) (string? y)) ... ]
))
(define (450js+ x y)
  (cond
   [(and (number? x) (number? y)) (+ x y)]
   [(and (number? x) (string? y)) ... ]
   [(and (string? x) (number? y)) ... ]
   [(and (string? x) (string? y)) ... ]
  )
)

;; 450js+: 450jsResult 450jsResult -> 450jsResult
;; “adds” two cs450js result values together (following js semantics)
;; 450js+: 450jsResult 450jsResult -> 450jsResult
;; “adds” two cs450js result values together (following js semantics)

(define (450js+ x y)
  (cond
   [(and (number? x) (number? y)) (+ x y)]
   [(and (number? x) (string? y)) ... ]
   [(and (string? x) (number? y)) ... ]
   [(and (string? x) (string? y)) (string-append x y)]
)
(define (450js+ x y)
  (cond
    [(and (number? x) (number? y)) (+ x y)]
    [(and (number? x) (string? y)) ??? ]
    [(and (string? x) (number? y)) ... ]
    [(and (string? x) (string? y)) (string-append x y)]
)

;;; 450js+: 450jsResult 450jsResult -> 450jsResult
;;; “adds” two cs450js result values together (following js semantics)
;; 450js+: 450jsResult 450jsResult -> 450jsResult
;; "adds" two cs450js result values together (following js semantics)

(define (450js+ x y)
  (cond
   [(and (number? x) (number? y)) (+ x y)]
   [(and (number? x) (string? y)) (string-append (string->js x) y)]
   [(and (string? x) (number? y)) ... ]
   [(and (string? x) (string? y)) (string-append x y)]
   ))
`; 450js+: 450jsResult 450jsResult -> 450jsResult
`; “adds” two cs450js result values together (following js semantics)

```
(define (450js+ x y)
  (cond
   [(and (number? x) (number? y)) (+ x y)]
   [(and (number? x) (string? y)) (string-append (num->str x) y)]
   [(and (string? x) (number? y)) ... ]
   [(and (string? x) (string? y)) (string-append x y)]
   )
```

;;; 450js+: 450jsResult 450jsResult -> 450jsResult
;;; “adds” two cs450js result values together (following js semantics)

(define (450js+ x y)
  (cond
   [(and (number? x) (number? y)) (+ x y)]
   [(and (number? x) (string? y)) (string-append (num->str x) y)]
   [(and (string? x) (number? y)) (string-append x (num->str y))]
   [(and (string? x) (string? y)) (string-append x y)]
   ))
;; 450js+: 450jsResult 450jsResult -> 450jsResult
;; “adds” two cs450js result values together (following is semantics)

(define (450js+ x y)
  (cond
   [(or (string? x) (string? y))
    (string-append (res->str x) (res->str y))
   [else (+ x y)])))
Running: “CS450js” Programs: “minus”

```lisp
(define (450js- x y)
 (cond
   [(and (number? x) (number? y)) (- x y)]
   [(and (number? x) (string? y)) ...]
   [(and (string? x) (number? y)) ...]
   [(and (string? x) (string? y)) ...]))
```
JavaScript Semantics Exploration: “minus”
“Not a Number”

NaN

From Wikipedia, the free encyclopedia

In computing, NaN (/næn/), standing for Not a Number, is a particular value of a numeric data type (often a floating-point number) which is undefined or unrepresentable, such as the result of 0/0. Systematic use of NaNs was introduced by the IEEE 754 floating-point standard in 1985, along with the representation of other non-finite quantities such as infinities.

// mdn web docs

NaN

The NaN global property is a value representing Not-A-Number.
Running: “CS450js” Programs

;; run450js: 450jsAST -> 450jsResult
;; computes the result of running a CS450js program AST

;; A 450jsAST is one of:
;; - (num Number)
;; - (str String)
;; - (add 450jsAST 450jsAST)
;; - (sub 450jsAST 450jsAST)

;; A 450jsResult is either:
;; - Number
;; - String
;; - NaN
(struct nan [[]]
(define NaN (nan))

;; res->str: 450jsResult -> String
(define (res->str x)
  (cond
   [(string? x) x]
   [(number? x) (number->string? x)]
   [(nan? x) "NaN"])))

Don’t forget to update all “Result” functions!
(define (450js- x y)
  (cond
    [(and (number? x) (number? y)) (- x y)]
    [else NaN])))
In-class Coding 11/13: put it all together!

; parse450js: 450jsExpr -> 450jsAST
; Parses “CS450js Lang” program to AST

; run450js: 450jsAST -> 450jsResult
; Computes result of running CS450js AST

; A 450jsExpr is one of:
; - Number
; - String
; - (list '+ 450jsExpr 450jsExpr)
; - (list '-' 450jsExpr 450jsExpr)

; A 450jsResult is one of:
; - Number
; - String
; - NaN

Program in “CS450JS LANG”!

- **Change** require in cs450js-lang.rkt to point to your cs450js-<your last name>.rkt file
- **Write** “CS450JS LANG” code by putting this at top of any file: #lang s-exp “cs450js-lang.rkt”
  - See cs450js-prog1.rkt as an example
No More Quizzes!

but push your in-class work to:
Repo: cs450f23/lecture19-inclass