CPSC 311: Definition of Programming Languages[1]

[1] Taught by Prof. Ronald Garcia

Arpit Kumar

Fall 2023/24

These are my class notes, further research and a possibly random collection of facts from this course. The sidenotes try to summarize the wonderful historical facts that Ron showers us with.

Lecture 1: Introduction!

Some nice links:
https://staff.um.edu.mt/afra1/seminar/little-languages.pdf
https://www.students.cs.ubc.ca/ cs-311/current/

Lecture 2: Programming with Arbitrary-Arity Trees

Review from CPSC 110

Ron's shortcuts:

In class you saw that after you have used the rest or the steps of the design recipe to build up your understanding of your problem, you can unleash that potential energy on your function templates, especially if you make use of Racket's keyboard shortcuts. Today I leaned heavily on:

- C-c C-o : the sexpression following the insertion point is put in place of its containing sexpression
- M-C-k : delete forward one S-expression

More shortcuts: https://docs.racket-lang.org/drracket/Keyboard$_S$*hortcuts.html*

Lecture 3: Programming with PLAI Data Types and Abstract Functions

We took a brief interlude to (re-)introduce abstract list functions, namely map and foldr. The map function takes a list and produces a same-length list whose contents are the result of applying some function to each element. This is a super-common operation.

foldr is a quite general way to operate on lists, though it is best used for simple-ish cases where it is easy to understand what the code is doing (you can write very hairy code using foldr: please don't!) . I explained foldr in a "declarative" fashion: in principle, (foldr for-cons for-empty lox) takes the list lox and replaces every instance of empty with for-empty, and every instance of cons with

for-cons, and produces the result of evaluating that. I use the prefix for- here to imply "substitute this for xxx". https://docs.racket-lang.org/reference/pairs.html

Lecture 4: The Gory Details of Parsing

We also talked about some concepts of real-world parsing:

- scanning/lexing turns lists of characters into lists of tokens. It loses whitespace information (how many spaces, how many line breaks, etc.)
- parsing turns a list of tokens into a concrete syntax tree (aka parse tree) It is not lossy: you can recover the string of tokens from the tree.
- tree abstraction turns a concrete syntax tree into an abstract syntax tree.
- It is lossy: it loses grouping token information (how many parentheses did you wrap around that??? i.e. "5 * (6 + 7)" is treated the same as" 5 * ((((6 + 7))))";

The final result is...a data structure in a data definition! That's where we want to be to write interpreters and other program-analyzing functions. However, to get there in this class, we will use a simpler toolchain than the one above, that depends on our using Racket-like syntax for our languages:

- reading turns a string of characters into a symbolic expression (s-expression), a kind of tree
- parsing turns an s-expression representing a program into an abstract syntax tree, or signals an error if the s-expression does not represent a valid program.

This second sense of parsing (parsing á la 311) is a little different than the general one, but we will still use the term because in a more general sense it's the same thing: turn some representation of programs into a tree that we can work with. We will cover this topic next class.

Parsing Gone Wrong: For an example of parsing gone wrong, you can see this post-mortem of the recent UK flight hiccup:

https://jameshaydon.github.io/nats-fail/

Lecture 5: Parsing á la 311: Symbolic Expressions and Frenz

A naive way of designing a parser.

```
;; AEFS -> AE
;; produce an AE value corresponding to the given AE s-expression
;; Effect: signals an error if the given s-expression does not represent an ae

;; AEFS (AE-focused s-expression) is one of:
;; - Number
;; - `{+ ,AEFS ,AEFS}
;; - `{- ,AEFS ,AEFS}
;; - <any other s-expression>
;; interp.  a symbolic expression, but with a focus on those that
;; represent AE expressions.

(define (parse sexp)
  (cond [(number? sexp) (num sexp)]
        [(and (list? sexp)
              (= (length sexp) 3)
              (symbol=? (first sexp) '+))
         (add (parse (second sexp))
              (parse (third sexp)))]
        [(and (list? sexp)
              (= (length sexp) 3)
              (symbol=? (first sexp) '-))
         (sub (parse (second sexp))
              (parse (third sexp)))]
        [else
         (error 'parse "bad AE")]))
```

Symbols, S-expressions and more

Symbols are a very powerful design idea that McCarthy came up with, which allowed what you "symbolic AI" to use data definitions.

```
'hello ;; symbol
(quote hello) ;; or this
(symbol=? 'hello 'hi) ;; #f
(string->symbol "ma man!") ;; gives '|ma man!|
(define G1 (gensym))
> 'g262412
(symbol=? G1 'g262412) ;; #f
;; gensyms are never equal to any other symbol
```

gensym returns an uninterned symbol

Lisp is short for "list processinglanguage"

Early on, every data definition in Lisp was just cons and symbols

```
> (rest (cons 5 6))
. . rest: contract violation
  expected: (and/c list? (not/c empty?))
  given: '(5 . 6)
```

```
> (car (cons 5 6))
5
> (cdr (cons 5 6))
6
> (car (cdr (cons 5 (cons 6 (cons 7 empty)))))
6
> (cadr (cons 5 (cons 6 (cons 7 empty))))
6
> (cons? (cons 5 6))

;; quasiquotes - like quotations, but sometime not quote
> (quasiquote (5 . (6 . (7 . empty))))
'(5 6 7 . empty)
> (quasiquote (5 . (6 . (7 . (unquote empty)))))
'(5 6 7)
> (define L (list 8 9 10))
> (quasiquote (5 . (6 . (7 . (unquote L)))))
'(5 6 7 8 9 10)
> `(5 . 6 . 7 . ,L) ;; allows to eval expr inside quotes
```

Lecture 6: Parsing 2: Pattern Matching using match

A neater way of implementing a parser: pattern matching!

```
;; Match-based template for AE
#;
(define (fn-for-aefs sexp)
  (match sexp
    [`,n #:when (number? n) (... n)]
    [`(+ ,sexp1 ,sexp2) (... (fn-for-aefs sexp1)
                             (fn-for-aefs sexp2))]
    [`(- ,sexp1 ,sexp2) (... (fn-for-aefs sexp1)
                             (fn-for-aefs sexp2))]
    [else (... sexp)]))

(define (parse sexp)
(match sexp
    [`,n #:when (number? n) (num n)]
    [`(+ ,sexp1 ,sexp2) (add (parse sexp1)
                             (parse sexp2))]
    [`(- ,sexp1 ,sexp2) (sub (parse sexp1)
                             (parse sexp2))]
    [else (error "bad AE:" sexp)]))
```

Lecture 7: Stepping AE and WAE Expressions

```
;; if the first position can step, then step it
;; if the first position can't step, but the second can, then step that
;; if neither position can step, but both positions are numbers, do addition

'{+ {- 6 3} {- 7 {+ 3 1}}}

'{+ 3 {- 7 {+ 3 1}}}
'{+ 3 {- 7 4}}
'{+ 3 3}
'6
```

Word "compiler" attributed to Rear Admiral Grace Hopper - Cobal (3rd oldest language) also played a part in this

Frances Allen - first woman to win the Turing award, she worked at IBM, we can thank her for making our compilers go brrr - "A Catalogue of Optimizing Compilers"

What is the WAE to success?

We now define a new language **WAE** which is really just the **AE** language but with "with"!

```
'x  ;; x occurs free in this WAE expression
;; ERROR x is unbound

'{+ x y}
;; ERROR x is unbound

'{with {y 7} x} ;; x occurs free in this WAE expression
'{with {y x} y} ;; x occurs free in this WAE expression
'{with {x 7} {with {y x} z}} ;; x occurs bound, z occurs free, y is bound
;; but does not occur
```

*How to step **WAE** programs?*

```
;(define WAES3 '{with {x {+ 5 5}} {+ x x}})
'{with {x {+ 5 5}} {+ x x}}
'{with {x 10} {+ x x}}
'{+ 10 10} ;; replace all "free" occurrences of x with the bound value
'20

;; "free" instance means that there isn't a surrounding "with"
```

Lecture 8: Free Identifier Instances

```
    (define-type WAE
        [num (n number?)]
        [add (lhs WAE?) (rhs WAE?)]
        [sub (lhs WAE?) (rhs WAE?)]
        [with (id symbol?) (named-expr WAE?) (body WAE?)]
        [id (name symbol?)])
;; interp.  program in the WAE language, corresponding to the following
;; Backus-Naur Form (BNF) specification
;;   <WAE> ::= <num>
;;           | { + <WAE> <WAE> }
;;           | { - <WAE> <WAE> }
;;           | { with {<id> <WAE>} <WAE>}
;;           | <id>

;; Every AE program is a WAE program
(define AE1 (num 4))
(define AE2 (add AE1 (num 5)))
(define AE3 (sub (num 6) (num 3)))

;; WAE can associate identifiers with expressions
(define WAES1 '{with {x 4} x})
(define WAE1 (with 'x (num 4) (id 'x)))
(define WAES2 '{with {x 4} {+ x x}})
(define WAE2 (with 'x (num 4) (add (id 'x) (id 'x))))
```

Accumulators

Accumulators are used to store some conetext when recursively traversing through a data structure. For eg. it could be used to store our current position in a list and make decision based on the position value.

The following function returns a list of all the free-instance identifiers from a WAE expression.

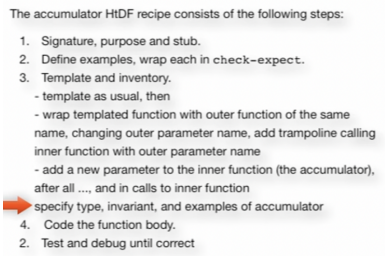


Figure 1: The HtDF accumulator recipe

```
;; WAE -> (listof WaeID)
;; produce a list of the free identifier instances in the given expressions
;; (NOTE: since we want "instances" they need not be unique!)
(define (free-instance-ids wae0)
;; Accumulator: binding-instances is (listof WaeID)
;; Invariant: all identifiers in wae0 with binding instances around wae
(local [(define (fn-for-wae wae binding-instances)
           (type-case WAE wae
```

```
            [num (n) empty]
            [add (l r) (append (fn-for-wae l binding-instances)
                               (fn-for-wae r binding-instances))]
            [sub (l r) (append (fn-for-wae l binding-instances)
                               (fn-for-wae r binding-instances))]
            [with (id named body)
                    (append (fn-for-wae named binding-instances)
                            (fn-for-wae body (cons id binding-instances)))]
            [id (x) (if (not (member x binding-instances))
                        (list x)
                        empty)]))]
    (fn-for-wae wae0 empty)))
```

Lecture 9: Substitution, Naïve and Capture-Avoiding

```
;; subst: WAE symbol WAE -> WAE
;; substitutes second argument with third argument in the
;; first argument. as per rules of substituition, the resulting
;; expression contains no free instances of the second argument

(define (subst expr sub-id val)
    (type-case WAE expr
        [num (n) expr]
        [add (l r) (add (subst l sub-id val)
                        (subst r sub-id val))]
        [sub (l r) (sub (subst l sub-id val)
                        (subst l sub-id val))]
        [with (bound-id named-expr bound-body)
            (if (symbol=? bound-id sub-id)) ;; means a nested `with` with same id
                (with bound-id
                      (subst named-expr sub-id val)
                      bound-body)
                (with bound-id
                      (subst named-expr sub-id val)
                      (subst bound-body sub-id val))]
        [id (v) (symbol=? v sub-id) val v]))
```

This substituition function is really naïve in it's approach. Consider the following call to 'subst'

```
(define EXPR (with (y 9) x))
(define WAE1 (with (x y) (with (y 3) (add x y))))
(subst EXPR x WAE1)
```

The result of this call is

```
(with (y 9) (with (x y) (with (y 3) (add x y))))
```

Notice that in the second 'with', y is longer a free variable and is bound to 9 from the first with. In other words, this free instance has been "captured". In general, we want to ensure that when we are substituting free instances in a WAE expression with other WAE expressions, the free instances in the WAE being substituted in remain free.

This is achieved by **capture-avoiding substitution**. This essentially involves renaming identifiers in a way such that substituted in free-identifiers are not captured and remain free. For example, continuing from the previous example, if we rename the first 'y' to 'g700', we will fix our problem.

```
(with (g700 9) (with (x y) (with (y 3) (add x y))))  ;; now x, y remain free instances
```

The following modification to our 'subst' function does exactly this. [2]

[2] Though, it is important to note that this version may rename identifiers even when it isn't necessary

```
;; WAE symbol WAE -> WAE
;; substitute val for free instances of sub-id in expr, avoiding the capture
;; of any free identifier instances in wae2.
;(define (ca-subst expr sub-id wae2) (num 0)) ; stub
(define (ca-subst wae0 x0 wae1)
    (local [(define (fn-for-wae wae)
            (type-case WAE wae
                [num (n) (num n)]
                [add (l r) (add (fn-for-wae l)
                                (fn-for-wae r))]
                [sub (l r) (sub (fn-for-wae l)
                                (fn-for-wae r))]
                [with (x named body)
                    (let ([g (gensym)])
                    (with g  ;; notice we rename binding instance
                             ;; this ensures no clashes with our substitution
                          (fn-for-wae named)
                          (fn-for-wae (ca-subst body x (id g)))
                [id (x) (if (symbol=? x x0)
                            wae1
                            (id x))]))]]
    (fn-for-wae wae0)))))
```

```
  {with {x {+ 5 5}} {with {y {- x 3}} {+ y y}}}
= {with {x 10} {with {y {- x 3}} {+ y y}}}
= {with {y {- 10 3}} {+ y y}}
= {with {y 7} {+ y y}}
= {+ 7 7}
= 14
```

Couldn't we have also written it this way?

```
  {with {x {+ 5 5}} {with {y {- x 3}} {+ y y}}}
= {with {y {- {+ 5 5} 3}} {+ y y}}
= {+ {- {+ 5 5} 3} {- {+ 5 5} 3}}
= {+ {- 10 3} {- {+ 5 5} 3}}
= {+ {- 10 3} {- 10 3}}
= {+ 7 {- 10 3}}
= {+ 7 7}
= 14
```

Figure 2: Eager vs Lazy reduction

Eager vs Lazy Reductions

Lecture 10: Environments and Environment-Passing Interpreters

In practice substitution is expensive because it walks the entire program and reconstructs it every time an identifier is bound. Substitution provides a clear specification of program behaviour, and is a good implementation for a debugger or automatic stepper, but it's expensive for a real program implementation.

An environment, on the other hand, tends to be cheaper because adding to the environment is constant time and lookup can be pretty cheap too, compared to the lexical nesting depth of a typical program. So an environment is another example of an accumulator in action. Its invariant is what ensures that we produce the same final behaviour as substitution.

```
;; Env is Symbol -> Number
;; Effect: signals an error if a looked up identifier is not bound in the env
;; interp. a table for looking up identifier bindings

;; Env
(define empty-env
    (\lambda (x) (error 'lookup-env "Undefined identifier: ~a" x)))

;; Env Symbol Number -> Env
(define (extend-env env x0 n0)
```

```
    (\lambda (x)
    (if (symbol=? x x0)
        n0
        (env x))))

;; Env Symbol -> Number
;; look up x in env
;; Effect: signal an error if x is not in env
(define (lookup-env env x)
    (env x))
```

This takes us back to the recurring theme: programs are data. Here we are using functions to implement a data structure.

```
;; Environment-passing interpreter

;; WAE -> Number
;; consumes a WAE and computes the corresponding number
;(define (interp/wae wae) 0) ; stub

(define (interp/wae-env wae0)
    ;; env is Env
    ;; Invariant: represents bindings (in inside-out order) of identifiers
    ;;            to values *due to pending substitutions*
    (local [(define (interp/wae-env wae env)
            (type-case WAE wae
                [num (n) n]
                [add (l r) (+ (interp/wae-env l env)
                              (interp/wae-env r env))]
                [sub (l r) (- (interp/wae-env l env)
                              (interp/wae-env r env))]
                [with (x named body)
                    (let ([vnamed (interp/wae-env named env)])
                        (interp/wae-env body
                                        (extend-env env x vnamed)))]
                [id (x) (with-handlers ([exn:fail?
                                        ( (_)
                                            (error "Unbound identifier: ~a" x))])
                        (lookup-env env x))]))]
    (interp/wae-env wae0 empty-env)))
```

In other words, store a map of identifiers and their eagerly evaluated values.

Lecture 11: First-Order Procedures/Functions, and a taste of Dynamic Scope

Stepping F1WAE Programs

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example
'{define-fn {double x} {+ x x}}
'{with {x 5} {double x}}
'{double 5}
'{+ 5 5}
'10
```

- **First-order functions:** Functions are not values in the language. They can only be defined in a designated portion of the program, where they must be given names for use in the remainder of the program. The functions in F1WAE are of this nature, which explains the 1 in the name of the language.
- **Higher-order functions:** Functions can return other functions as values.
- **First-class functions:** Functions are values with all the rights of other values. In particular, they can be supplied as the value of arguments to functions, returned by functions as answers, and stored in data structures.

The environment-based interpreter for the F1WAE language follows.

```
;; F1WAE (listof FunDef) -> Number
;; interpret the expression f1wae in the context of the fundefs
(define (interp-f1wae f1wae0 fundefs)
    ;; Accumulator env is Env
    ;; Invariant: bindings (in inside-out order) of identifiers to values
    ;;            *due to pending substitutions*
    (local [(define (interp-f1wae f1wae env)
            (type-case F1WAE f1wae
                [num (n) n]
                [add (l r) (+ (interp-f1wae l env)
                              (interp-f1wae r env))]
                [sub (l r) (- (interp-f1wae l env)
                              (interp-f1wae r env))]
                [with (x named body)
                    (let ([val (interp-f1wae named env)])
                        (interp-f1wae body (extend-env env x val)))]
```

```
                [id (x) (with-handlers ([exn:fail?
                                         ( (_)
                                             (error "Unbound identifier ~a" x))])
                          (lookup-env env x))]
                [app (fun-name arg)
                    (let ([fundef (lookup-fundef fun-name fundefs)])
                        (let ([val (interp-f1wae arg env)])
                        (interp-f1wae (fundef-body fundef)
                                       (extend-env empty-env ;; NOT env! Making this env gives you
                                       ;; dynamic scoping
                                                  (fundef-arg-name fundef) val))))]
                [if0 (p c a)
                    (if (zero? (interp-f1wae p env))
                        (interp-f1wae c env)
                        (interp-f1wae a env))]))]
    (interp-f1wae f1wae0 empty-env)))
```

Dynamic Scoping: Let's say that function x call function y in its body. In a dynamically scoped language, function y will have access to all the local variables defined in the body of function x (before the call to function y was made).

Lecture 12: First-class anonymous functions, and stepping Dynamic Scope

Stepping F1WAE-DYN

```
'{define-fn {double x} {+ x x}}
'{with {x 5} {double x}}
;;;;
'{bind {x 5} {double x}}
'{bind {x 5} {double 5}} ;; find the innermost binding of x surrounding it
                         ;; and replace x with the bound
                         ;; value
;; procedure call: bind the argument to the call to the formal parameter of the procedure
;;                 in the body of the procedure
'{bind {x 5}
       {bind {x 5}
             {+ x x}}}
'{bind {x 5}
       {bind {x 5}
             {+ 5 x}}}
'{bind {x 5}
```

```
       {bind {x 5}
             {+ 5 5}}}
'{bind {x 5}
       {bind {x 5}
             10}}
;; if the body of a bind is a value
;; hoist it out of the enclosing bind
'{bind {x 5}
        10}
'10
```

A very famous researcher Christoper Strachey (wrote a paper called "Fundamental Concepts in Programming Languages") coined the term "first-class" functions

Stepping F1WAE

F1WAE introduces lambdas, which we call fun.

```
'{with {double {fun {x} {+ x x}}}
       {with {x 5} {double x}}}
'{with {x 5} {{fun {x} {+ x x}} x}}
'{{fun {x} {+ x x}} 5}
'{+ 5 5}
'10

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
'{7 9}
;; ERROR: 7 is not a procedure
```

Cannot redefine native procedure (eg. with + ...) is not allowed

Lecture 13: Self-Reference and Recursion

New language - FFWAE (FWAE with FIX)

Peter Landin showed that Algol-16 just boils down to our FWAE language - years later, Shriram Krishnamurthi did the same to Javascript

Rósza Péter considered the "mother" of recursion schemes

```
    ;; EXAMPLE 1
    '{fix f f}  ;; just replace f with this entire expr itself
    '{fix f f}
    '{fix f f}
    ;; and so on

    ;; EXAMPLE 2
    '{fix a {fix b a}}  ;; step rule: almost like with
                        ;; replace free occurence of a with this entire expr
    '{fix b {fix a {fix b a}}}
    '{fix a {fix b a}}  ;; you don't stick in b here because
                        ;; b is a binding instance here
    '{fix b {fix a {fix b a}}}
    ;; and so on, remember to be METICULOUS
```

```
;; EXAMPLE 3
'{fix f {fun {x} {if0 x 9 {f {- x 1}}}}}
'{fun {x} {if0 x 9 {{fix f {fun {x} {if0 x 9 {f {- x 1}}}}} {- x 1}}}}
;; we are left with a fun ->
;; these are values in FWAE (functions are first-class)
;; and so we are done
```

De-sugar-ing refers to adding features to a programming languages by defininining it with existing features - so it gives syntatic sugar to use, but under the hood the implementation doesn't add anything new.

In our FFWAE language data-definition, we don't have a "with" since its equivalent to saying

```
{with {x e1} {add e1 e2}}
;; equivalent to
{fun {x} {add x e2} e1}
```

@TODO

Any error your interpreter produces should be defined by you, if PLAI throws an error for you, will be considered a segfault

Lecture 14: Y Combinator

@TODO

Lecture 15: Call-by-name and non-strict semantics via stepping and substitution

Lazy reduction! Don't evaluate named expressions early on.

```
;; Example 1: Call-by-value
'{with {x {+ 5 7}}
       {- x x}}
;; evaluate the bound expression...
'{with {x 12}
       {- x x}}
;;.. then perform substitution
'{- 12 12}
0

;; Example 2: Call-by-name evaluation (substitute the entire named expression!)
'{with {x {+ 5 7}}
       {- x x}}
'{- {+ 5 7} {+ 5 7}}
```

```
;; Example 3: substitution blows away the bound expression without ever
;; evaluating it
'{with {x {+ 5 7}}
       {- 9 2}}
'{- 9 2}
'7

;; Example 4: Wait a minute, that means...
'{with {room-warmer {fix f f}}
       {- 9 2}}
'{- 9 2}  ;; NO INFINITE LOOP, sadly the room stays cold :(
'7

;; Call-by-name generalizes to *non-strict* evaluation semantics,
;; wherein in addition to binding expressions to identifiers,
;; constructor expressions do not evaluate their arguments: they are values!
'{pair {fix f f} {fix f f}}  ;; <-- this is a value (a pair of expressions)

;; this program, unlike the previous, *diverges*
'{left {pair {fix f f} {fix f f}}}
'{fix f f} ;; this does evaluate and goes into infinite loop since
;; fix is not a constructor expression

;; As per our previous languages, a legal program can have free identifiers
;; NOOOOO....this is no longer lexical scoping
;; naive substituition doesn't cut it anymore for this language
'{with {y {- 10 x}}
       {with {x 7}
             y}}
'{with {x 7}
    {- 10 x}}
'{- 10 7}
'3

;; YES:
'{with {g 7} ;; rename x -> g
    {- 10 x}}
'{- 10 x}
;; ERROR Unbound x

;; Fix:
;; OPTION 1: Capture avoiding substituition
;; OPTION 2: restrict what counts as a program to programs
;;           that do not have free identifiers ("closed")
```

```
;; PLAI is closed
```

Along the way, we discovered an important property: naïve substitution is not enough! In particular, a program with free identifiers, which should signal an error, might produce a value result if we implement the language using naive substitution. This leads us to two solutions: either ban free identifiers or use capture-avoiding substitution. For now we go with the latter, though languages like Haskell go with the former.

We briefly talked about why someone might allow programs that we *know* could produce an error. There is a robustness/correctness tradeoff to consider, which has affected HTML's supremacy over XHTML, and may have motivated JavaScript's "keep-on-truckin'" behaviour in the face of undefined : better for some code to limp along half-working than to simply crash the moment something goes wrong.

So this explains why javascript is so weird. ew.

Lazy evaluator, cons should not evaluate its value - 2 papers that came out in the same year, haskell is an example of a pure, non-strict programming language

Lecture 16: Non-strict semantics cont'd. Now via environment-passing

Wikipedia definition of currying

> currying is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument.

```
(define (addron a b) (+ a b))
(addron 1 2)

;; Example of currying
(define (addronc a) ( (b) (+ a b)))
((addronc 1) 2)
```

Lecture 17: MIDTERM !!!

WAS BRUTAL, NO TIME

ANYWAYS...

Lecture 18: Mutable Boxes

equal? checks if two entites "look alike" - i.e structural equality, while eq? checks for identity - "is it the same thing", i.e mutation should

David Turner, a very influential PL programmer and the reason for Haskell's existence passed away yesterday

equal? is a very expensive call, can in theory DDOS using equal? - not really used in production code

REPL PLAI doesn't print the void object

affect both these objects, it is equivalent to pointer equality

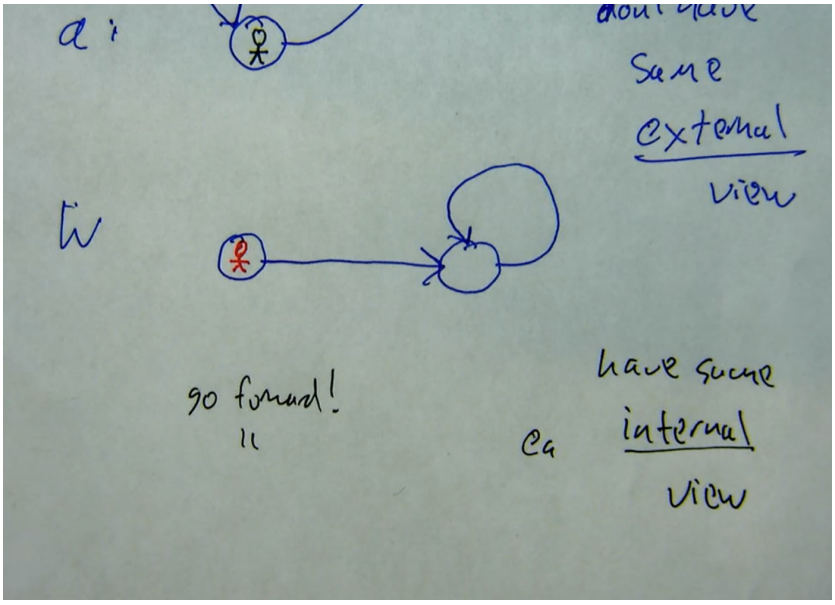


Figure 3: equal? considers the **internal** view to match structure

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Problem 1
'{newbox 9} ;; Program

'{newbox 9} ;; Initial Step of Stepping
'()         ;; the empty store

'{boxV loc0} ;; a box value representing a new location in the store
'((loc0 9))  ;; the store, which binds locations to storable objects (values)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Problem 2
'{openbox {newbox 9}} ;; Program

'{openbox {newbox 9}}
'()

'{openbox {boxV loc0}}
'((loc0 9))

;; openbox evaluates its argument
;; if the result is not a boxV signal an error
;; if it is a boxV, produce the value its location points to in the store
```

```
'9
'((loc0 9))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Problem 3
'{with {x {newbox 9}}      ;; Program
       {seqn {setbox x 12}
             {openbox x}}}

'{with {x {newbox 9}}      ;; Program
       {seqn {setbox x 12}
             {openbox x}}}

'()

'{with {x {boxV loc0}}      ;; Program
       {seqn {setbox x 12}
             {openbox x}}}

'((loc0 9))


'{seqn {setbox {boxV loc0} 12}
       {openbox {boxV loc0}}}
'((loc0 9))

;; seqn evaluates the first position to a value
;; and then throws it away and runs the second position

'{seqn {setbox {boxV loc0} 12}
       {openbox {boxV loc0}}}
'((loc0 9))

;; setbox evaluates its first argument, and expects a boxV back
;; then evaluates its second argument to any value
;; then replaces the value in the store with the second argument's value
;; and produces the previous value from the store

'{seqn 9
       {openbox {boxV loc0}}}
'((loc0 12))

'{openbox {boxV loc0}}
```

```
'((loc0 12))
;; openbox evaluates its argument and expects a boxV back
;; then produces the value in the store

'12
'((loc0 12))
```

Lecture 19: Mutable Variables - Implementing Mutable Boxes

Ok, first up, **threaded accumulators**.

```
;; Run -> Natural
;; Calculate the score induced by this run
(define (run-score! run0)
    ;; Accumulator: factor is Natural
    ;; Invariant: factor is (expt 2 N) where N is the number of 5's in run0
    ;;            before run.
    (local [(define factor (void))
            ;; Run -> Natural
            ;; Effect: mutates factor
            (define (run-score--acc run)
            (cond [(empty? run) 0]
                  [else
                  (let ([value (* (first run) factor)])
                      (begin
                      (set! factor (if (= (first run) 5)
                                       (* factor 2)
                                       factor))
                      (+ value
                          (run-score--acc (rest run)))))]))]
    ;; 1 = (expt 2 0)
    (begin (set! factor 1)
            (run-score--acc run0))))
```

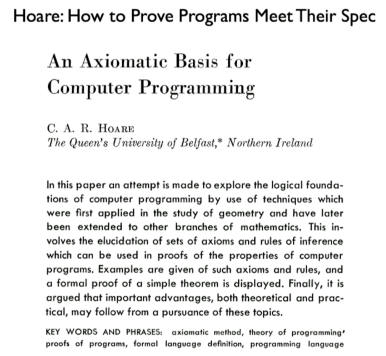
Mutable Variables - "by reference" vs "by value"

```
;; Mutable Variables (Binding "by-value")

'{with {x 5}
       {with {y x}
             {seqn {setvar y 7}
                   x}}}

'{with {x 5}
```

Sir Charles Antony Richard "Tony" Hoare (1934 – NOW)

Figure 4: The inventor of the Quicksort algorithm, inspired by the Shell sort algorithm - the thing that made it possible for him to implement Quicksort was that Algol-60 had recursion. He also invented null pointers!!! He calls it his billion dollar mistake.

Hoare: How to Prove Programs Meet Their Spec



An Axiomatic Basis for Computer Programming

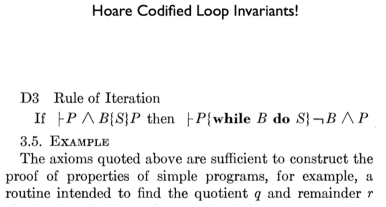
C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

KEY WORDS AND PHRASES: axiomatic method, theory of programming, proofs of programs, formal language definition, programming language

Figure 5: He introduced the idea of reasoning about program with mutation

Hoare Codified Loop Invariants!



D3 Rule of Iteration
If $\vdash P \wedge B\{S\}P$ then $\vdash P\{\textbf{while } B \textbf{ do } S\}\neg B \wedge P$

3.5. Example
The axioms quoted above are sufficient to construct the proof of properties of simple programs, for example, a routine intended to find the quotient q and remainder r

Figure 6: His rule of iteration

```
        {with {y x}
                {seqn {setvar y 7}
                    x}}}
'()

;; substitute a "variable location" for all free references to the bound
;; variable in the body of the with expression
;; so identifiers not denote locations instead of values

'{with {y {varL loc0}}
        {seqn {setvar y 7}
                {varL loc0}}}
'((loc0 5))

;; to step a varL (i.e. a variable location), replace it with the current
;; value assigned to it in the store
'{with {y 5}
        {seqn {setvar y 7}
                {varL loc0}}}
'((loc0 5))


;; replace the current value assigned to the relevant location in the store
;; with the given value, and produce the previously stored value

'{seqn {setvar {vaL loc1} 7}
        {varL loc0}}
'((loc1 5) (loc0 5))

'{seqn 5
        {varL loc0}}
'((loc1 7) (loc0 5))

'{varL loc0}
'((loc1 7) (loc0 5))

'{varL loc0}
'((loc1 7) (loc0 5))

'5
'((loc1 7) (loc0 5))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Mutable Variables (Binding "by-reference")
```

```
'{with {x 5}
       {with {y x}
              {seqn {setvar y 7}
                  x}}}

'{with {x 5}
       {with {y x}
              {seqn {setvar y 7}
                  x}}}
'()

;; substitute a "variable location" for all free references to the bound
;; variable in the body of the with expression

'{with {y {varL loc0}}
       {seqn {setvar y 7}
              {varL loc0}}}
'((loc0 5))

'{seqn {setvar {varL loc0} 7}
       {varL loc0}}
'((loc0 5))

'{seqn 5
       {varL loc0}}
'((loc0 7))

'{varL loc0}
'((loc0 7))

'7
'((loc0 7))
```

Lecture 20: Implementing Mutable Variables by Following Type Structure

@TODO

Lecture 21: Exceptions

Today we focused on learning about exceptions, using a model of exception handling inspired by the OCaml programming language (https://ocaml.org/). For those not familiar with it, OCaml is a typed eager functional programming language that has many (but not all) of the niceties of Racket, but with static type checking and quite good performance. It is used at places like Facebook (who have also introduced a new surface syntax for OCaml called Reason (https://reasonml.github.io/)).

```
;; Pattern matching-style exception handler:
;; {match/handle <TEL>
;;  [<id> <TEL>]
;;  [{raze <tag> <id>} <TEL>]}

;; Raise an exception:
;; {raze <tag> <TEL>}

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example 1
'{raze oops 5}  ;; does not step, *but* this is NOT a value
                ;; we call this (and values) "canonical forms"


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example 2
'5  ;; is a value, and is also a canonical form


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Example 6

'{match/handle
  {match/handle {+ {raze oops 7} 3}
   [x {+ x 9}]
   [{raze oops x} {+ x 12}]}
  [y {+ -5 y}]
  [{raze doh y} y]}

'{match/handle
  {match/handle {raze oops 7}
   [x {+ x 9}]
   [{raze oops x} {+ x 12}]}
  [y {+ -5 y}]
```

```
  [{raze doh y} y]}

'{match/handle
  {+ 7 12}
  [y {+ -5 y}]
  [{raze doh y} y]}

'{match/handle 19
  [y {+ -5 y}]
  [{raze doh y} y]}

'{+ -5 19}

'14
```

Effects affect the behaviour of every possible operation of your language (if your languge is structured such that operation can have effectful operands). Now, **order matters**. Make sure to use 'let*' to enfore ordering in your interpreter (or nested 'let's).

```
;; Tel -> Canonical
;; produce the canonical form result of interpreting the given TEL
;; (subst based interpreter)
(define (interp/tel-cf tel)
    (type-case TEL tel
    [num (n) (value (num n))]
    [add (l r)
            (type-case Canonical (interp/tel-cf l)  ;; first evaluate left operand
            [value (vl)
                    (type-case Canonical (interp/tel-cf r)
                    [value (vr) (add-value vl vr)]
                    [razed (tag payload) (razed tag payload)])]
            [razed (tag payload) (razed tag payload)])]
    [id (x) (error 'interp "Unbound identifier: ~a" x)]
    [fun (x body) (value (fun x body))]
    [app (rator rand)
            ;; needs fixing
            (let ([vrator (interp/tel-cf rator)]
                [vrand (interp/tel-cf rand)])
            (apply-value vrator vrand))]
    [if0 (p c a)
            ;; needs fixing
            (let ([vp (interp/tel-cf p)])
```

```
        (if (zero?-value vp)
            (interp/tel-cf c)
            (interp/tel-cf a)))]
    [fix (x body) (interp-fix x body)]
    [match/handle (expr vid vbody etag eid ebody)
                  (let ([cexpr (interp/tel-cf expr)])
                  (interp-match/handle cexpr
                                       vid vbody
                                       etag eid ebody))]
    [raze (tag expr)
            (let ([cexpr (interp/tel-cf expr)])
            (interp-raze tag cexpr))]))
```

Lecture 22: Effect Abstractions 1 - Exceptions

Goal of this lecture is to use macros to encapsulate the complexity of addiing effectful features to a language.

Introducing effects means we impose a dependency order on our computations - it might matter if we evaluate left operand before the right or vice-versa for example. Similarly, when we introduced mutation, we had to evaluate the left operand first since it might update the store. Note that we imposed the order of the dependency, it could be that we decide the right operand is evaluated first and introduce a r->l dependency order.

The purpose of this class to understand how interpreters work under the hood, and for that reason, they are not always the most efficient. Treat them as reference interpreters rather than performant ones since they refrain from using underlying features of PLAI and implement everything from scratch (well, mostly).

Per Brinch Hansen

Figure 7: He said "When a programming conept is understood informally, it would seem to be a trivial matter to invent a language notation for it. But in practice, this is hard to do. The main problem is to replace an intuitive, vague idea with a precise, unambiguous definition of its meaning and restrictions.

```
;; Effect Abstraction (Exceptions)

;; (raise/eff t v) - raise an exception with tag t and payload v

;; (match-exn/eff e1
;;  [val (x) e2]
;;  [handle tag (x) e3])
;; -- dispatch on whether e1 produced a value or an exn


;; Generic Interface (expresses dependencies between computations)
;; (return/eff e) - returns the value of e
;; (run/eff e) - run an effectful computation, and turn exceptions into
;;               PLAI errors

;; (let/eff ([x e1]) e2) - bind x to the value of e1 in e2 or
;;                         propagate an exception from e1
```

```
;; Value -> Computation
(define (return/eff e)
  (value e))

;; Computation -> Value
;; Effect: thows an exception if the given computation represents an error
(define (run/eff c)
  (type-case Computation c
    [value (v) v]
    [razed (tag payload) (error 'interp/tel "Uncaught Exception: ~a"
                                (razed tag payload))]))

;; uses MACROS!
(define-syntax let/eff
  (syntax-rules ()  ;; syntax-rules is the weakest macro system, look at syntax-case
    [(_ ([x e1]) e2)
     (type-case Computation e1
       [value (v) (let ([x v]) e2)]  ;; v is automatically gensymmed
                                     ;; C preprocessor doesn't understand
                                     ;; scope, so doesn't gensym
       [razed (tag payload) (razed tag payload)])]))


;; Compose many computations
(define-syntax let/eff*
    (syntax-rules ()
    [(_ () e) e]  ;; needs a base case, this is a recursive macro
    [(_ ([x e1] [x* e1*] ...) e2)
    (let/eff ([x e1])
                (let/eff* ([x* e1*] ...) e2))]))
;; peerful effects needs low level implementation from your end
```

Lecture 23: Effect Abstractions 2 - Threaded Accumulators

Hygienic macros you need to macros to handle effects and not functions since racket is call-by-value and so it run the argument before passing it into the function - no way to perform exception handling non-strict languages dont have this issue - which is why haskell doesn't have a ood macro system

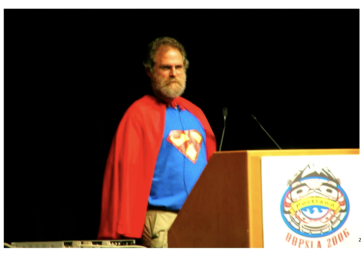
assignment in python is like letrec

@TODO

Eugenio Moggi

Figure 8: Took obscure ideas from an obscure branch of mathematics and applied it thinking about how we describe programming languages - resulted in effect abstractions!

Phil Wadler



Lecture 24: Effect Abstractions 3 - Threaded Accumulators and Layering Multiple Effects

Based on the above post, I am guessing that you are talking about Problem 1 from Assignment 5? If so, then maybe the following can help. We have studied several different effect abstractions, and combinations of them:

Exception/Canonical - when you might produce a value or raze an exception. This is like having an "error code" like when in C or Java someone produces a null value, but here you have no choice but to handle the error case (which is a good thing). ;; Computation is one of: ;; - A ;; - B Where A is your usual value, and B is your "exceptional case"

Threaded Accumulator/State-passing - when you have a threaded accumulator in your code, like in the tree form score-a-lot. We also used threaded accumulators to thread the store in Hannah, and used match-let to separate the return value from the updated accumulator. Computation is Acc -> (list Value Acc) where Value is your normal return value and Acc is the threaded accumulator No Effect (i.e. NoFX) - when you don't actually do anything with effects. But you can still format an interpreter this way so that it's easier to add effects later (as we did in tutorial 7). Computation is Value

Accumulator/Environment - when you have a normal (i.e. non-threaded) accumulator. We used this in tutorial 7 to switch between an environment-passing interpreter (since an environment is just an accumulator) and substitution (using NoFX). Computation is Acc -> Value

Stream/Backtracking - we used this in Lola to implement backtracking Computation is (streamof Value)

Continuation - we used this in MCA to support first-class continuations. It can also help us make our program tail-recursive (by adding this effect everywhere) Computation is Kont -> Value Where Kont is the type of a continuation (which is a procedural representation of an accumulator) and Value is the result.

We also blended accumulators. Boxtel combines the threaded accumulator effect with the exception effect. Tomax combines the threaded accumulator effect with the backtracking effect.

Ok, so that's all the effects we have studied. Now, for your mission, the first thing you need to think is what effect among the above is the program already using. That should help you narrow down what should be considered. You should be able to figure this out just by reading the interpreter, especially the main interp functions, but also the helpers it calls, and how it calls them. Compare that to other interpreters you've seen. Once you know which effect(s) is(are)

in play, you can think about how to abstract those details. First thing you should do is figure out what ;; Computation is ... should be. Nearly everything follows from this type signature. You can use it to guide your design. I don't necessarily recommend trying to systematically derive the effect like I did for state in class. That was meant to be more pedagogical about how state works. It might work for some, but it might be asking too much of yourself.

The book "Beautiful Code" Scheme Programming Language by Kent

R. Kent Dybvig

Figure 10: Creator of Chez Scheme - used on Cisco routers back in the day becasue C was too slow

Lecture 25: Generalized Backtracking via Streams

Prommises in Racket The delay construct is used together with the procedure force to implement lazy evaluation or call by need. (delay <expression>) returns an object called a promise which at some point in the future may be asked (by the force procedure) to evaluate <expression>, and deliver the resulting value. The effect of <expression> returning multiple values is unspecified.

```
(define a-stream
  (letrec ((next
            (lambda (n)
              (cons n (delay (next (+ n 1)))))))
    (next 0)))
(define head car)
(define tail
  (lambda (stream) (force (cdr stream))))

(head (tail (tail a-stream)))
                                        ===>  2
```

Figure 11: Streams

@TODO Until Lecture 30

Lecture 30: Inductive Definitions and Well-Formedness

Invariants - the compiler relies on the invariant that your input is a valid program. If that's not the case, it can really do whatever it likes and that is NOT a compiler error = examples of this are seen on the C compiler github issues page where issues are often shut down with "yeah so? that exactly what it is supposed to do".

"Parse, Don't check!" - Alexsis King. Defining a function to check if WAE is closed and then deciding to call the interpreter works, but again its a form of checking. Instead, it would be nice to find a

Figure 12: The Cousots! Seminal work in static analysis

Princples of Abstract Interpretation - By Patrick Cousot

way of parsing that encapsulates closed-ness. Inductive definitions!
(which Ron describes as BNFs on steroids).