Good Programming

To work program: Design a program -> Write the program in a programming language -> Interpret or compile the program -> Run it and see the results.

**To Work Program**: The elements for **Systematic Program Designs** are Problem Analysis and Data Definitions, Contract (Signature), Purpose (Effect) statement, Header, Functional Examples, Function Definition, and Testing.

**Write the program in a programming language:** Programming Language consists of Peculiar Syntax, some behavior associated with each syntax (semantics), Libraries, and idioms (specific language usage)

**To interpret or compile the program:** The interpreter takes a program and produces a result/Compiler takes a program and produces a program.

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

**Designing a good program:** Contact, purpose, tests, header, and body

Test-Driven Development: Write a test case before writing programs, Write the simplest code to pass, and if your code is not sufficient, write more tests and repeat.

Tests are a great source of documents: precise and executable and always in sync. **Benefit**: Keep simple design, incremental progress, and protect

Racket Basics

(operator operand operand) / (define (function-name param1 …) body) (function-name args1 … ) / Booleans and relations e.g) (and (> 4 3) (<= 10 100)) / (define (function-name param1 …) **cond** [ce1 body1] … [else body])) / Only one basic operation on symbols: symbol=? / **Type Definition**: (define-type type-id [variant\_id1 (field\_id11 contract\_expr11) [variant\_idm (field\_idm1 contract\_exprm1) ]) ./ **Type Deconstruction:** From a given instance of a specific type, get required values or do a specific task for the instance / (type-case type-id expr [variant\_id1(field\_id11 …) expr1] [variant\_idm (field\_idm1 ...) exprm]) / **Lists:** cons, list, append, first, rest, map, foldl, foldr, filter, empty? / Recursion of Lists (define (my-length lst) (cond [(empty? lst) 0] [else (+ 1 (my-length (rest lst)))] ))

Modeling Languages

**Syntax:** dependent on programming language / **Library:** good for the programmer, but not core of the language / **Idioms:** Common practice, can be dangerous, developer-oriented / **Semantics:** Denotational (constructing mathematical objects or functions.), Operational (execution-oriented), Axiomatic (set of logical assertions) / Concrete Syntax (‘expression’): programming language we are using / Abstract Syntax Tree: AST abstractly reflects the syntax of the code using tree format / **Parser**: a component in an interpreter or compiler. Identifies what kinds of program code it is examining and Converts concrete syntax (what we type) into abstract syntax. / **BNF**: a specification of the concrete syntax of the language and a default abstract syntax / <expr>: Non-terminal, Meta-variable / ::=: "Can be written as" / |: "one more choice" (a production) / <...>: literal syntax / Terminal / ?: optional / +: an item exists 1 or more

**Interpreter:** [contract] interp: AE -> number / [Purpose] consumes an AE and compute the corresponding number. / Type deconstruction will be used

Language: Freedom vs. Restriction, Expressiveness vs. Limited but Safe, Fast to write vs. maintainable on a large scale / With strings, you can have multiple strings that have the same contents, but are different objects (they do not compare as eq?). With symbols, two symbols that have the same contents are guaranteed to be the same object

**텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명**a = terminal, A,B = nonterminal,   
alpha, β, γ = string of terminals and/or non-terminals

**Chomsky hierarchy**

**Type3: regular expression / Type2: structure of most programming languages / Type 1: The strings α and β may be empty, but γ must be nonempty / Type 0: Type-0 grammars include all formal grammars.**

**Syntax vs. Semantics**: Semantics: Meaning conveyed by words, phrases, and sentences. It involves understanding the relationships between words and the concepts they represent.

Syntax : How words are structured and combined to form meaningful sentences. It defines the order, arrangement, and relationships between words. **Why racket?:** Benefits of functional language -> deterministic, intuitive, can be represented using formula   
terpreting Process: The interpreter directly executes code on runtime(translates it to machine code) using the following strategy: Parses the code into abstract syntax, Compiling Process: The compiler translate the code before the runtime to a compiled code(it can also run a lot of optimization) so you dont need to compile on each runtime.

Compilers are faster (execution time is less, since the code is already converted to machine code) , uses less memory,

Interpreter is faster for finding errors (since with compiler, you need to compile the file each time you make changes), easier to edit interpreters itself, more friendly, especially for beginners

Compilers are faster (execution time is less, since the code is already converted to machine code) , uses less memory,

Interpreter is faster for finding errors (since with compiler, you need to compile the file each time you make changes), easier to edit interpreters itself, more friendly, especially for beginners

Substitution

Before implementing an WAE interpreter, we need to think how to deal with identifiers in the interpreter / **Identifier**: to avoid redundancy / Name it and identify the value of an expression / reusing its name in place of large computation / Similar to a variable, but the value can’t be changed / works like a constant / **with keyword:** use 'with' keyword to define an identifier for an arithmetic expression / **Free identifier:** error; An identifier not contained in the scope of any binding instance of its name is said to be free. / **Binding Instance**: the instance of the identifier that gives it its value e.g) {with {**x** 5} {+ x 5}} / **Scope**: the region of program text in which instances of the identifier refer to the value bound by the binding instance / **Bound Instance:** an identifier is bound when it is contained within the scope of a binding instance at that time / **Substitution in our code**: To substitute identifier i in e with the expression v, replace all bound instances of i and replace all free instances of name i in e with v

**; [contract] subst: WAE(e) symbol(i) number(v) -> WAE**

Syntax Error and Semantic Error / **Syntactic Sugar and Desugaring:** New expression implemented using defined expressions (at the parser level)

Syntactic sugar is a syntax within a programming language designed to read or express code more easily. It can write code shorter and more naturally.

Desugaring is the process of returning the syntatic sugar back to its original state, and in short, it is the process of converting it into a basic language. This allows a compiler or interpreter to process code easily.

Introduction to Functions

<FunDef> ::= {deffun {<id> <id>} <F1WAE>} ;; for function definition

<F1WAE> ::= <num> | {+ <F1WAE> <F1WAE>} | {- <F1WAE> <F1WAE>} | {with {<id> <F1WAE>} <F1WAE>} | <id> | {<id> <F1WAE>} ;; for function call

**(define-type FunDef** [fundef (fun-name symbol?) (arg-name symbol?) (body F1WAE?)])

**(define-type F1WAE** [num (n number?)] [add (lhs F1WAE?) (rhs F1WAE?)] [sub (lhs F1WAE?) (rhs F1WAE?)] [with (name symbol?) (named-expr F1WAE?) (body F1WAE?)] [id (name symbol?)] [app (ftn symbol?) (arg F1WAE?)]) ;; application syntax

**; parse : sexp -> F1WAE**

**(define (parse sexp)** (match sexp [(? number?) (num sexp)] [(list '+ l r) (add (parse l) (parse r))] [(list '- l r) (sub (parse l) (parse r))] [(list 'with (list i v) e) (with i (parse v) (parse e))] [(? symbol?) (id sexp)] [(list f a) (app f (parse a))][else (error 'parse "bad syntax: ~a" sexp)]))

; parse-fd: sexp -> FunDef

**(define (parse-fd sexp)** (match sexp [(list 'deffun (list f x) b) (fundef f x (parse b))]))

; lookup-fundef: symbol list-of-FunDef -> FunDef

**(define (lookup-fundef name fundefs)** (cond [(empty? fundefs) (error 'lookup-fundef "unknown function")] [else (if (symbol=? name (fundef-fun-name (first fundefs))) (first fundefs) (lookup-fundef name (rest fundefs)))]))

; interp: F1WAE list-of-FuncDef -> number

**(define (interp f1wae fundefs)** (type-case F1WAE f1wae [num (n) n] [add (l r) (+ (interp l fundefs) (interp r fundefs))] [sub (l r)(- (interp l fundefs) (interp r fundefs))] [with (x i b) (interp (subst b x (interp i fundefs)) fundefs)] [id (s) (error 'interp "free identifier")] [app (f a) (local [(define a\_fundef (lookup-fundef f fundefs))] (interp (subst (fundef-body a\_fundef) (fundef-arg-name a\_fundef) (interp a fundefs)) fundefs))]))

; [contract] subst: F1WAE symbol number -> F1WAE

**(define (subst f1wae idtf val)** (type-case F1WAE f1wae [num (n) f1wae] [add (l r) (add (subst l idtf val) (subst r idtf val))] [sub (l r) (sub (subst l idtf val) (subst r idtf val))] [with (i v e) (with i (subst v idtf val) (if (symbol=? i idtf) e (subst e idtf val)))] [id (s) (if (symbol=? s idtf) (num val) f1wae)] [app (f a) (app f (subst a idtf val))]))

Deferring Substitution

Cost of substitution: expensive > using cache for substitution

**(define-type DefrdSub** [mtSub] [aSub (name symbol?) (value number?) (saved DefrdSub?)])

**(define (interp f1wae fundefs)** (type-case F1WAE f1wae [num (n) n] [add (l r) (+ (interp l fundefs) (interp r fundefs))] [sub (l r)(- (interp l fundefs) (interp r fundefs))] [with (x i b) (interp e (aSub i (interp v ds) ds))] [id (s) (lookup s ds)] [app (f a) (local [(define a\_fundef (lookup-fundef f fundefs))] (interp (subst (fundef-body a\_fundef) fundefs (aSub (fundef-arg-name a-fundef) (interp a fundefs ds) **(mtSub)**) ))] ))  
This mtSub makes the implement static scope, if ds -> dynamic scope

Interpreting function body starts with only one substitution / **Scope**: static scope -> the scope of an identifier's binding is a syntactically delimited region / Dynamic scope -> the scope of an identifier's binding is the entire remainder of the execution during which that binding is in effect  
Disadvantages of static scopes: lack of flexibility, difficulty handling recursive functions,

Advantages of Static Scope: Predictable Behavior, Scope Chain Optimization

Disadvantages of dynamic scopes: difficult to predict, difficult to debugging

Advantages of dynamic scope: Flexible scope change.

**Role of look up function:** Looking for the identifier, using new data structure will increase the speed / Static scope and dynamic scope / Complexity: O(n\*m) and O(m) which n is called, m is number of functions

First Class Function

**First-order function**: Functions are not values in languages, Names must be given for use in the remainder of a program. **Higher-order functions:** Functions can return other functions as values. **First-class Functions**: Functions are values with all the rights of other values. Can be supplied as the value of arguments, returned by functions as answer, and stored in data structures

**<FWAE> ::=** <num> |{+ <FWAE> <FWAE>} | {- <FWAE> <FWAE>} | {with {<id> <FWAE>} <FWAE>} | <id> | {<FWAE> <FWAE>} | {fun {<id>} <FWAE>}

**(define-type FWAE** [num (n number?)] [add (lhs FWAE?) (rhs FWAE?)] [sub (lhs FWAE?) (rhs FWAE?)] [with (name symbol?) (named-expr FWAE?) (body FWAE?)] [id (name symbol?)] [fun (param symbol?) (body FWAE?)] [app (ftn FWAE?) (arg FWAE?)])

; parse: sexp -> FWAE

; purpose: to convert sexp to FWAE

**(define (parse sexp)** (match sexp [(? number?) (num sexp)] [(list '+ l r)(add (parse l) (parse r))] [(list '- l r) (sub (parse l) (parse r))] [(list 'with (list i v) e)(withi (parse v) (parse e))] [(? symbol?) (id sexp)] [(list 'fun (list p) b) (fun p (parse b))] [(list f a) (app (parse f) (parse a))] [else (error 'parse "bad syntax: ~a" sexp)]))

; interp: FWAE -> FWAE

**(define (interp fwae)** (type-case FWAE fwae [num (n) fwae] [add (l r)(num+ (interp l) (interp r))] [sub (l r) (num- (interp l) (interp r))] [with (i v e) (interp (subst e i (interp v)))] [id (s) (error 'interp "free identifier")] [fun (p b) fwae] [app (f a) (local [(define ftn (interp f))] (interp (subst (fun-body ftn) (fun-param ftn) (interp a))))]))

; num-op: (number number -> number) -> (FWAE FWAE -> FWAE)

**(define (num-op op)** (lambda (x y) (num (op (num-n x) (num-n y))))) **(define num+ (num-op +)) (define num- (num-op -))**

**Lambda Expression**: Code brevity / Remove unnecessary loop / Reuse a function definition / Better performance based on Laziness

Cons: Could be slower / Difficult to track function call stack while debugging / Make code difficult to understand.

**Anonymous function**: a function definition that is not bound to an identifier

e.g)numbers = [1, 2, 3, 4] doubled = list(map(lambda x: x \* 2, numbers)) print(doubled)

It allows you to temporarily use and discard specific functions, and it's concise.

Efficient in terms of memory management.  
**Closure:** Closure is a concept in programming language theory. It's a data structure that includes a function and the environment in which that function was created. The environment contains the variables and their values at the time the function was defined. **In our scenario**, ClosureV has a deferred substitution cache to handle pending variable substitutions when the closure is evaluated.

**(define (subst fwae idtf val)** (type-case FWAE fwae [num (n) fwae] [add (l r) (add (subst l idtf val) (subst r idtf val))] [sub (l r) (sub (subst l idtf val) (subst r idtf val))] [with (i v e) (with i (subst v idtf val) (if (symbol=? i idtf) e (subst e idtf val)))] [id (s) (if (symbol=? s idtf) (num val) fwae)] [app (f a) (app (subst f idtf val) (subst a idtf val))] [fun (id body)(if (equal? idtf id) exp (fun id (subst body idtf val)))] ))

The problem of Scope appears, deferred substitution will solve this situation.

**No more** **with:** <FAE> ::= <num> | {+ <FAE> <FAE>} | {- <FAE> <FAE>} | <id>

| {fun {<id> <FAE>} | {<FAE> <FAE>}

**Substitution cache** with its corresponding value so that we can avoid dynamic scope and we will not forget the pending substitution for the function.

**(define-type FAE-Value** [numV (n number?)] [closureV (param symbol?) (body FAE?) (ds DefrdSub?)])

**(define (parse sexp)** (match sexp [(? number?)(num sexp)] [(list '+ l r) (add (parse l) (parse r))] [(list '- l r) (sub (parse l) (parse r))] [(list 'with (list i v) e)(app (fun i (parse e)) (parse v))] [(? symbol?) (id sexp)] [(list 'fun (list p) b) (fun p (parse b))] [(list f a) (app (parse f) (parse a))] [else (define-type DefrdSub [mtSub] [aSub (name symbol?) (value FAE-Value?) (ds DefrdSub?)])

**; interp: FAE DefrdSub -> FAE-Value**

**(define (interp fae ds)** (type-case FAE fae [num (n) (numV n)] [add (l r) (num+ (interp l ds) (interp r ds))] [sub (l r) (num- (interp l ds) (interp r ds))] [id (s) (lookup s ds)] [fun (p b)(closureV p b ds)] [app (f a) (local [(define f-val (interp f ds)) (define a-val (interp a ds))] (interp (closureV-body f-val) (aSub (closureV-param f-val) a-val (closureV-ds f-val)))) ] ))

**Environment**: rather than substituting immediately, we can defer substitution; avoids the need for source-to-source rewriting and maps nicely to low-level machine representations

**Closure:** a technique for implementing lexically scoped name binding in a language with first-class functions. Operationally, a closure is a record storing a function together with an environment.

Laziness

An expression is evaluated only if its result is needed. Efficient.

Languages like Racket, Java, and C are called eager. An expression is evaluated when it is encountered. We should not evaluate the expression and keep the static scope at the same time.

**<LFAE> :: =** <num> | {+ <LFAE> <LFAE>} | {- <LFAE> <LFAE>} | <id> | {fun {<id>} <LFAE>} | {<LFAE> <LFAE>}

**Short-circuit vs. Laziness:** Stop right after you know the result. vs. Evaluate only when it is needed.

**(define (num-op op x y)** (numV (op (numV-n (strict x)) (numV-n (strict y)))))

(define (num+ x y) (num-op + x y))

(define (num- x y) (num-op - x y))

; strict: LFAE-Value -> LFAE-Value

**(define (strict v)** (type-case LFAE-Value v [exprV (expr ds) (strict (interp expr ds))] [else v]))

**(define-type LFAE-Value** [numV (n number?)] [closureV (param symbol?) (body LFAE?) (ds DefrdSub?)] [exprV (expr LFAE?) (ds DefrdSub?)])

**(define (interp lfae ds)** (type-case LFAE lfae [num (n) (numV n)] [add (l r) (num+ (interp l ds) (interp r ds))] [sub (l r) (num- (interp l ds) (interp r ds))] [id (s) (lookup s ds)] [fun (p b)(closureV p b ds)] [app (f a) (local [(define f-val (interp f ds)) (define a-val (exprV a ds (box #f)))] (interp (closureV-body f-val) (aSub (closureV-param f-val)

a-val (closureV-ds f-val))))]))

**Memoization**: Memoization is a technique in computer programming where previously computed values are stored in memory to prevent redundant calculations when the same computation needs to be repeated, thus speeding up the overall execution.

In the case of laziness, the calculation is delayed until the calculation result is necessary, and if the calculation is not necessary, the calculation is not performed. On the other hand, in the case of memoization, repeated calculation results are stored for reuse, and both can be effective in reducing time.

Box: In Racket, the box is a mutable container for a single value. we can contain a value in the box, and we can update, and extract the value from the box. In our scenario, we used it to store what we needed.

**What is a function?**

In computer programming, a function or subroutine is a sequence of program instructions that performs a specific task, packaged as a unit. This unit can then be used in programs wherever that particular task should be performed.

**What is substitution?**

A substitution is a syntactic transformation on formal expressions. To apply a substitution to an expression means to consistently replace its variable, or placeholder, symbols with other expressions.

The resulting expression is called a substitution instance, or instance for short, of the original expression.

**What is deferred substitution?**

create and use a repository of deferred substitutions. Concretely, here’s the idea. Initially, we have no substitutions to perform, so the repository is empty. Every time we encounter a substitution (in the form of a with or application), we augment the repository with one more entry, recording the identifier’s name and the value (if eager) or expression (if lazy) it should eventually be substituted with.

**What is different from deferred substitution and laziness?**

Deferred substitution is a specific form of laziness that involves caching the result of an expression to avoid recomputation, while laziness is a more general concept of delaying expression evaluation until necessary without necessarily caching results.

**Static and Dynamic Example:**

{with {x 3} { with {f {fun {y} {+x y}}} {with {x 5} {f 4}}}} -> in static: 7 / dynamic: 9

**Recursion**  
<RCFAE> ::= <num> | {+ <RCFAE> <RCFAE>} | {- <RCFAE> <RCFAE>} | {\* <RCFAE> <RCFAE>} | <id> | {fun {<id>} <RCFAE>} | {<RCFAE> <RCFAE>} | {if0 <RCFAE> <RCFAE> <RCFAE>} | {rec {<id> <RCFAE>} <RCFAE>}

{with {fac {fun {n} {if0 n 1 {\* n {fac {- n 1}}}}}} {fac 10}} ⇒ free identifier 'fac'!

*… so pass {fac 10} as an argument!*

{with {fac {fun {n} {with {facX {fun {facY n} {if0 n 1 {\* n {facY facY {- n 1}}}}}} {facX facX n}}}} {fac 10}} -> but we don’t implement the program with double arguments

Eta reduction - If two functions lead to the same result, they are the same functions.

Rec Template

{with {mk-rec {fun {body-proc} {with {fX {fun {fY} {with {f {fun {x} {{fY fY} x}}} {body-proc f}}}} {fX fX}}}} {with {fac {mk-rec {fun {fac} {fun {n} {if0 n 1 {\* n {fac {- n 1}}}}}}}} {fac 10}}}

{rec {fac {fun {n} {if0 n 1 {\* n {fac {- n 1}}}}} {fac 10}}

(define-type RCFAE [num (n number?)] [add (lhs RCFAE?) (rhs RCFAE?)] [sub (lhs RCFAE?) (rhs RCFAE?)] [id (name symbol?)] [fun (param symbol?) (body RCFAE?)] [app (fun-expr RCFAE?) (arg-expr RCFAE?)] [if0 (test-expr RCFAE?) (then-expr RCFAE?) (else-expr RCFAE?)] [rec (name symbol?) (named-expr RCFAE?) (fst-call RCFAE?)])

(define-type DefrdSub [mtSub] [aSub (name symbol?) (value RCFAE-Value?) (ds DefrdSub?)] [aRecSub (name symbol?) (value-box (box/c RCFAE-Value?)) (ds DefrdSub?)])

(define (lookup name ds) (type-case DefrdSub ds [mtSub () (error ’lookup "free variable")] [aSub (sub-name val rest-ds) (if (symbol=? sub-name name) val (lookup name rest-ds))] *[aRecSub (sub-name val-box rest-ds) (if (symbol=? sub-name name) (unbox val-box) (lookup name rest-ds))]))*

(define (interp rcfae ds) (type-case RCFAE rcfae [num (n) (numV n)] [add (l r) (num+ (interp l ds) (interp r ds))] [sub (l r) (num- (interp l ds) (interp r ds))] [mul (l r) (num\* (interp l ds) (interp r ds))] [id(s) (lookup s ds)] [fun (p b)(closureV p b ds)] [if0 (test-expr then-expr else-expr) (if(numzero? (interp test-expr ds)) (interp then-expr ds) (interp else-expr ds))] [app (f a) (local [(define f-val (strict (interp f ds))) (define a-val (exprV a ds (vector #f)))] (interp (closureV-body f-val) (aSub (closureV-param f-val) a-val (closureV-ds f-val))))] *[rec (bound-id named-expr first-call) (local [(define value-holder (box (numV 628))) (define new-ds (aRecSub bound-id value-holder ds))] (begin (set-box! value-holder (interp named-expr new-ds)) (interp first-call new-ds)))]*

*))*

**Mutable Data Structure**

Mutable -> liable to change

Pure function: A function produces the same results every time for the same arguments.

Box -> data structure that can hold any type of a single value

<BFAE> ::= <num> | {+ <BFAE> <BFAE>} | {- <BFAE> <BFAE>} | <id> | {fun {<id} <BFAE>} | {<BFAE> <BFAE>} | {newbox <BFAE>} | {setbox <BFAE> <BFAE>} | {openbox <BFAE>} | *{seqn <BFAE> <BFAE>} Run two expressions sequentially*

(define-type BFAE-Value [numV (n number?)] [closureV (param symbol?) (body BFAE?)

(ds DefrdSub?)] [boxV (container (box/c BFAE-Value?))])

Abstract Syntax  
[newbox (v BFAE?)] [setbox (bn BFAE?) (v BFAE?)] [openbox (v BFAE?)] [seqn (ex1 BFAE?) (ex2 BFAE?)]

{with {a {newbox 1}} {with {f {fun {x} {+ x {openbox a}}}} {seqn {setbox a 2} {f 5}}}}  
We need two repositories (caches) One for keeping a memory address value of a box for static scope

(define-type Store [mtSto] [aSto (address integer?) (value BFAE-Value?) (rest Store?)])

(define-type BFAE-Value [numV (n number?)] [closureV (param symbol?) (body BFAE?) (ds DefrdSub?)] [boxV (address integer?)])

(define-type Value\*Store [v\*s (value BFAE-Value?) (store Store?)])

*Binding id → address to value → value*

*Binding id → address to box → address to value → value*

(define (interp bfae ds st) (type-case BFAE bfae [num (n)(v\*s (numV n) st)] [add (l r) (type-case Value\*Store (interp l ds st) [v\*s (l-value l-store) (type-case Value\*Store (interp r ds l-store) [v\*s (r-value r-store) (v\*s (num+ l-value r-value) r-store)])])] [sub (l r) (type-case Value\*Store (interp l ds st) [v\*s (l-value l-store) (type-case Value\*Store (interp r ds l-store) [v\*s (r-value r-store) (v\*s (num- l-value r-value) r-store)])])] [id(s) (v\*s (store-lookup (lookup s ds) st) st)] [fun (p b)(v\*s (closureV p b ds) st)] [app (f a)(type-case Value\*Store (interp f ds st) [v\*s (f-value f-store) (type-case Value\*Store (interp a ds f-store) [v\*s (a-value a-store) (local ([define new-address (malloc a-store)])

(interp (closureV-body f-value) (aSub (closureV-param f-value) new-address (closureV-ds f-value)) (aSto new-address a-value a-store)))])])] [newbox (val) (type-case Value\*Store (interp val ds st) [v\*s (vl st1) (local [(define a (malloc st1))] (v\*s (boxV a)

(aSto a vl st1)))])] [setbox (bx-expr val-expr) (type-case Value\*Store (interp bx-expr ds st) [v\*s (bx-val st2) (type-case Value\*Store (interp val-expr ds st2) [v\*s (val st3)

(v\*s val (aSto (boxV-address bx-val) val st3))])])] [openbox (bx-expr) (type-case Value\*Store (interp bx-expr ds st) [v\*s (bx-val st1) (v\*s (store-lookup (boxV-address bx-val) st1) st1)])] [seqn (a b) ;(interp-two a b ds st (lambda (v1 v2 st1) (v\*s v2 st1)))] (type-case Value\*Store (interp a ds st) [v\*s (a-value a-store) (interp b ds a-store)])]

))

*[add (l r) (type-case Value\*Store (interp l ds st) [v\*s (l-value l-store) (type-case Value\*Store (interp r ds l-store) [v\*s (r-value r-store) (v\*s (num+ l-value r-value) r-store)])])]⇒ [add (l r) (interp-two l r ds st(lambda (v1 v2 st1) (v\*s (num+ v1 v2) st1)))]*

**Variable**

| {setvar <id> <BMFAE>}

(define-type BFAE-Value [numV (n number?)] [closureV (param symbol?) (body BFAE?) (ds DefrdSub?)] [refclosV(param symbol?) (body RBMFAE?) (ds DefrdSub?)] [boxV (address integer?)])

(define-type DefrdSub [mtSub] [aSub(name symbol?) (address integer?) (ds DefrdSub?)])  
(define-type Store [mtSto] [aSto (address integer?) (value BMFAE-Value?) (rest Store?)])

(define (interp bmfae ds st) (type-case BMFAE bmfae [num (n) (v\*s (numV n) st)] [add (l r) (interp-two l r ds st (lambda (v1 v2 st1) (v\*s (num+ v1 v2) st1)))] [sub (l r) (interp-two l r ds st(lambda (v1 v2 st1) (v\*s (num- v1 v2) st1)))] [id (s) (v\*s (store-lookup (lookup s ds) st) st)] [fun (p b)(v\*s (closureV p b ds) st)] [refun (p b) (v\*s (refclosV p b ds) st)] [app (f a)(type-case Value\*Store (interp f ds st) [v\*s (f-value f-store) (type-case BMFAE-Value f-value [closureV (c-param c-body c-ds) (type-case Value\*Store (interp a ds f-store) [v\*s (a-value a-store) (local ([define new-address (malloc a-store)]) (interp (closureV-body f-value) (aSub (closureV-param f-value) new-address (closureV-ds f-value)) (aSto new-address a-value a-store)))])] [refclosV (rc-param rc-body rc-ds) (local ([define address (lookup (id-name a) ds)]) (interp (closureV-body f-value) (aSub (closureV-param f-value) address (closureV-ds f-value)) (f-store)))] [else (error interp "trying to apply a number")])])]

[newbox (val) (type-case Value\*Store (interp val ds st) [v\*s (vl st1) (local [(define a (malloc st1))] (v\*s (boxV a) (aSto a vl st1)))])] [setbox (bx-expr val-expr) (type-case Value\*Store (interp bx-expr ds st) [v\*s (bx-val st2) (type-case Value\*Store (interp val-expr ds st2) [v\*s (val st3) (v\*s val (aSto (boxV-address bx-val) val st3))])])]

[openbox (bx-expr) (type-case Value\*Store (interp bx-expr ds st) [v\*s (bx-val st1) (v\*s (store-lookup (boxV-address bx-val) st1) st1)])] [seqn (a b) ;(interp-two a b ds st (lambda (v1 v2 st1) (v\*s v2 st1)))] (interp-two a b ds st (lambda (v1 v2 st1) (v\*s v2 st1)))] [setvar (id val-expr) (local [(define a (lookup id ds))] (type-case Value\*Store (interp val-expr ds st) [v\*s (val st) (v\*s val (aSto a val st))]))]))

**Continuation**

Continuation

* Rest of computation to be evaluated from one point.
* Rest of work that has to happen to finish the evaluation of a program
* Abstract representation of the control state of a program.

Continuation Passing Style (CPS)

* Easy to transform your representation of stacks from the actual stack to heap

KCFAE

(define-type KCFAE-Value [numV(n number?)] [closureV(param symbol?) (body KCFAE?) (ds DefrdSub?)] [contV (c procedure?)])

(define-type KCFAE-Value [numV (n number?)] [closureV (p procedure?)] [contV (c procedure?)])  
(define (interp kcfae ds k) (type-case KCFAE kcfae [num (n)(k (numV n))] [add(l r)(interp l ds lambda (lv) interp r ds lambda (rv) k (num+ lv rv))))))] sub(l r)(interp l ds (lambda (lv) (interp r ds (lambda (rv) (k (num- lv rv))))))] [mul(l r)(interp l ds (lambda (lv) (interp r ds (lambda (rv) (k (num\* lv rv))))))] [id (s) (k (lookup s ds))] [fun (p b)(k (closureV (lambda (a-val dyn-k) (interp b (aSub p a-val ds) dyn-k))))] [app (f a) (interp f ds

(lambda (f-val) (interp a ds (lambda (a-val) (type-case KCFAE-Value f-val [closureV (c) (c a-val k)] [contV (c) (c a-val)] [else (error "not an applicable value")])))))]

[if0 (test t f) (interp test ds (lambda (tv) (if(eq? (interp test ds k) (numV 0)) (interp t ds k) (interp f ds k))))] [withcc (cont-var body) (interp body (aSub cont-var (contV (lambda (val) (k val))) ds) k)]))

so that we can access to the continuation at every stage.

Interpreter takes an extra argument k (continuation) a.k.a a receiver.

(interp kcfae ds k) -> k: (lambda (x) x)