Lecture 5. FuncLang - Functions

February 23, 2021

Overview

- FuncLang: writing programs in functional programming languages
  - lambda expression
  - recursion
  - high-order functions
  - build-in functions (list, pair)
  - control structures
- Syntax
- Semantics
- Implementation

Abstraction in Programming Languages

- Variable in imperative programming languages
  - fixed abstraction – you cannot change computation
  - representing values

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- Function (procedure, method):
  - **parameterization for computation**
  - you can reuse the functionality for different concrete input
  - language features that can define a procedure and call a procedure

Lambda Expression

- Defining anonymous function

```
(
  lambda        //Lambda special function for defining functions
  (x)           //List of formal parameter names of the function
   x            //Body of the function
)
```

- Compare to ALGOL family languages: C, C++, Java ... (syntax):
  - not specify the name of the function
  - formal parameter name only, no types precede or follow
  - no explicit return is needed
- Compare to ALGOL family languages: C, C++, Java ... (semantics): Procedures and methods: proxy of the location of a section of code
  - adjust the environment
  - jump to the location

  Lambda abstraction:
  - generate runtime values
  - each of the runtime values can be used multiple times

Examples: Lambda Expression

```
(
 lambda          //Lambda special form for defining functions
 (x)             //List of formal parameter names of the function
 (+ x 1)         //Body of the function
)
```

(lambda (x y) (+ x y))

Examples: Calling the Lambda function

```
(                       //Begin function call syntax
  (lambda (x) x)        //Operator: function being called
  1                     //Operands: list of actual parameters
)                       //End function call syntax
```

```
(
  (lambda (x y) (+ x y))
  1 1
)
```

`Program`	::=	`DefineDecl`* `Exp?`	*Program*
`DefineDecl`	::=	`(define Identifier Exp)`	*Define*
`Exp`	::=		*Expressions*
		`Number`	*NumExp*
	|	`(+ Exp Exp`$^+$`)`	*AddExp*
	|	`(- Exp Exp`$^+$`)`	*SubExp*
	|	`(* Exp Exp`$^+$`)`	*MultExp*
	|	`(/ Exp Exp`$^+$`)`	*DivExp*
	|	`Identifier`	*VarExp*
	|	`(let ((Identifier Exp)`$^+$`) Exp)`	*LetExp*
	|	`( Exp Exp`$^+$`)`	***CallExp***
	|	`(lambda (Identifier`$^+$`) Exp)`	***LambdaExp***
`Number`	::=	`Digit`	*Number*
	|	`DigitNotZero Digit`$^+$	
`Digit`	::=	`[0-9]`	*Digits*
`DigitNotZero`	::=	`[1-9]`	*Non-zero Digits*
`Identifier`	::=	`Letter LetterOrDigit`*	*Identifier*
`Letter`	::=	`[a-zA-Z$_]`	*Letter*
`LetterOrDigit`	::=	`[a-zA-Z0-9$_]`	*LetterOrDigit*

Figure 5.1: Grammar for the Funclang Language. Non-terminals that are not defined in this grammar are exactly the same as that in Definelang.

Examples: Combine with Let and Define

```
(let
  (( identity  (lambda (x) x)))          //Naming the function
  ( identity  1)                         //Function call
)
```

```
$ (define square (lambda (x) (* x x)))
$ (square 1.2)
1.44
```

Exercise: Lambda Expression

Write some Lambda Expressions with Let and Define

Exercise: Lambda Expression

```
$ (define identity (lambda (x) x))
$ (identity 2)
2
$ (define test (lambda (x y z ) (* x y z)))
$ (test 1 2 3)
6
$ (let ((x (lambda (x) (+x 1)))) (x 3))
4
```

Note. lambda is the function definition keyword.

Control Structure

- if expression: three mandatory expressions – the condition, then, and else expressions
- comparison expression: $>$, $<$, $=$

Control Structure: Grammar

Exp ::=		*Expressions*
	`Number`	*NumExp*
|	`(+ Exp Exp`$^+$`)`	*AddExp*
|	`(- Exp Exp`$^+$`)`	*SubExp*
|	`(* Exp Exp`$^+$`)`	*MultExp*
|	`(/ Exp Exp`$^+$`)`	*DivExp*
|	`Identifier`	*VarExp*
|	`(let ((Identifier Exp)`$^+$`) Exp)`	*LetExp*
|	`( Exp Exp`$^+$`)`	*CallExp*
|	`(lambda (Identifier`$^+$`) Exp)`	*LambdaExp*
|	`(if Exp Exp Exp)`	***IfExp***
|	`(< Exp Exp)`	***LessExp***
|	`(= Exp Exp)`	***EqualExp***
|	`(> Exp Exp)`	***GreaterExp***
|	`#t` | `#f`	***BoolExp***

Figure 5.6: Extended Grammar for the Funclang Language. Non-terminals that are not defined in this grammar are same as that in figure 5.1.

Note. (1 1) is a valid expression from grammar point of view, but, it will throw a semantic error. More on this later!

Exercise: Lambda Expression

Write some Lambda Expressions with if then else

```
(if (= x 10) (let ((x 20)) ((lambda (y) (* y y)) x )) 0)
```

Pair and List

1. pair: 2 tuple (fst, snd)
2. list: empty list, or 2 tuple
3. a list is a pair, a pair is not necessarily a list

$$List := (list) \mid (cons\ \mathtt{val}\ List),\ \ \mathtt{where\ val} \in \mathtt{Value}$$

List and its built-in functions in FuncLang

- list: creating a list, e.g., (list 1 1 1 1 1)
- null?: check if a list is a null, returns #t if that argument is an emptylist else return #f
- car: get the first element of a pair, e.g., (car (list 11 1))
- cdr: get the second element of a pair, e.g., (cdr (list 342, 331, 327))
- cons:
  - constructing a pair, e.g., (cons 2 3) (cons 541 (list 342))
  - lists can also be constructed by using the cons keyword, as is shown here: > (cons 1 (list))
    (1)

Examples: list with functions

```
(define cadr
  (lambda (lst)
    (car (cdr lst))
  )
)

(define caddr
  (lambda (lst)
    (car (cdr (cdr lst)))
  )
)
```

Examples: list and its built-in functions in FuncLang

```
$ (list 1 2 3)
(1 2 3)
$ (cons 1 2)
(1 2)
$ (cons 1 (list 2))
(1 2)
$ (define L (list 1 2 3))
$ (car L)
1
$ (cdr L)
(2 3)
```

Examples: list and its built-in functions in FuncLang

```
$ (car (list))
funclang.Value$Null cannot be cast to funclang.Value$pairVal
$ (cdr (list))
funclang.Value$Null cannot be cast to funclang.Value$pairVal
$ (list)
$ (cdr (list 1))
()
$ (car (list 1))
1
$ (null? (list))
#t
```

Grammar with List

`Exp`	::=		*Expressions*
		`Number`	*NumExp*
	|	`(+ Exp Exp`$^+$`)`	*AddExp*
	|	`(- Exp Exp`$^+$`)`	*SubExp*
	|	`(* Exp Exp`$^+$`)`	*MultExp*
	|	`(/ Exp Exp`$^+$`)`	*DivExp*
	|	`Identifier`	*VarExp*
	|	`(let ((Identifier Exp)`$^+$`) Exp)`	*LetExp*
	|	`( Exp Exp`$^+$`)`	*CallExp*
	|	`(lambda (Identifier`$^+$`) Exp)`	*LambdaExp*
	|	`(if Exp Exp Exp)`	*IfExp*
	|	`(< Exp Exp)`	*LessExp*
	|	`(= Exp Exp)`	*EqualExp*
	|	`(> Exp Exp)`	*GreaterExp*
	|	`#t` | `#f`	*BoolExp*
	|	`(car Exp)`	***CarExp***
	|	`(cdr Exp)`	***CdrExp***
	|	`(null? Exp)`	***NullExp***
	|	`(cons Exp Exp)`	***ConsExp***
	|	`(list Exp`*`)`	***ListExp***

Figure 5.8: Extended Grammar for the Funclang Language. Non-terminals that are not defined in this grammar are same as that in figure 5.1.

Exercise: functions, list and control structure

Write programs with list, functions and control structures

Examples : all together

```
$(if (null? l) (car l) (cdr l))
$ (cdr (cdr (list 1 2 3)))
(3)
$ (car (cdr (list 1 2 3)))
2
$ (cdr (cdr (cdr (list 1 2 3))))
()
$ (cdr (list))
funclang.Value$Null cannot be cast to funclang.Value$PairVal
$ (car (list))
funclang.Value$Null cannot be cast to funclang.Value$PairVal
$ (cdr (cdr (cdr (cdr (list 1 2 3)))))
funclang.Value$Null cannot be cast to funclang.Value$PairVal
$ (cons 3 (list))
(3)
$ (define f (lambda (x) (* x x)))
$ (list 1 2 f)
(1 2 (lambda ( x ) (* x x )))
$ (list 1 2 (f 5))
(1 2 25)
```

Recursive Function

- Recursive function mirror the definition of the input data type

$$List := (list) \mid (cons\ \texttt{val}\ List),\ \ \texttt{where val} \in \texttt{Value}$$

```
(define append
  (lambda (lst1 lst2)
    (if (null? lst1) lst2
      (if (null? lst2) lst1
        (cons (car lst1) (append (cdr lst1) lst2))
      )
    )
  )
)
```

Define a function sum that sums the number 1 to n.

```
$ (define sum (lambda (n) (if (= n 1) 1 (+ n (sum (- n 1))))))
$ (sum 1)
1
$ (sum 2)
3
$ (sum 3)
6
```

FuncLang Programming Exercise

- Define a function that computes the factorial of a given integer n
- Define a function sumsquares that takes two integers as a parameter and computes the sum of square of numbers from the first number to the second number.

Solution

- (define fac (lambda (n) (if (= n 1) 1 (* n (fac (- n 1))))))
- $ (define f (lambda (n m) (if (> n m) 0 (+ (* n n) (f (+ n 1) m)))))
  $ (f 0 2)
  5
  $ (f 1 2)
  5
  $ (f 2 3)
  13

High Order Function - take function as an input

- a function that accepts a function as an argument or return a function as value

```
(define addthree
   (lambda (x)(+ x 3)))

(define applyonone
   (lambda (f) (f 1)))

$(applyonone addthree)
4
$(addthree applyonone) //error
$ (addthree (applyonone addthree))
7

(define addtwovalues (lambda (x y) (+ x y)))
$ (applyonone addtwovalues) //error

(define applyonetwo (lambda (f) (f 1 2))
$ (applyonetwo addtwovalues)
3
```

High Order Function - return a function

```
(lambda
   (c)
   (lambda (x) c)
)

( (lambda
   (c)
   (lambda (x) c)
  )
1
)

( ( (lambda
   (c)
   (lambda (x) c)
  )
1
)
2)
```

High Order Function - using function to represent data structure and its operations

```
(define pair
    (lambda (fst snd)
      (lambda (op)
        (if op fst snd)
      )
    )
)
(define apair (pair 3 4))
(define first (lambda (p) (p #t)))
$ (first apair)
```

- what is apair?
- what is first?

High order function: problem solving

- parameterize functions: defining reusable algorithmic structures, e.g., a higher-order function that accepts an operation and a list and applies the operation on each element of the list.
- identify: what is high order function (given by the problem)? what is op used as paramaterized algorithms? what are the parameters of high order function that op will apply to? op will perform on which data structure and paramaters? element of a list? a list?
- high order function will (repeatedly) apply op on its other parameters
- if the high order function is recursive, what is the initial condition, and what is the subproblem of n-1.
- high order function for data structures: paramaters are members, the operators, .e.g, getfirst, getsecond, on the data structure are op: a constructor for creating pairs, an observer for getting the first element of the pair, and another observer for getting the second element of the pair.

Exercise: High Order Function

Define a function *filter* with the signature: (define filter (lambda (test op lst) ...)) The function takes two inputs, an operator test op that should be a single argument function that returns a boolean, and lst that should be a list of elements. The function outputs a list containing all the elements of "lst" for which the test op function returned #t.

```
$ (define gt5? (lambda (x) (if (> x 5) #t #f)))
$ ( filter gt5? ( list ))
()
$ ( filter gt5? ( list 1))
()
$ ( filter gt5? ( list 1 6))
(6)
$ ( filter gt5? ( list 1 6 2 7))
(6 7)
$ ( filter gt5? ( list 1 6 2 7 5 9))
(6 7 9)
```

Solution: High Order Function

- (define gt5? (lambda (x) (if (> x 5) #t #f)))
- (define filter (lambda (op l) (if (null? l) (list) (if (op (car l)) (cons (car l) (filter op (cdr l))) (filter op (cdr l)))))))
- // Try applying filter with similar function as gt5?
  (define iszero (lambda (x) (if (= x 0) #t #f)))

Currying

[the term Currying is from Haskell Curry] Model multiple argument lambda abstractions as a combination of single argument lambda abstraction

```
(define plus
   (lambda (x y) (+ x y)))

(define plusCurry
   (lambda (x)
     (lambda (y)
       (+ x y)
     )
   )
)
```

Revisit Syntax

What is new?
Funclang - Functions

- Lambda expression
- Call expression
- Function with a name
- High order function, including currying
- List and built-in functions
  - cons
  - list
  - car
  - cdr
  - null?
- if cond truestmt falsestmt
  - #t, #f
  - $<$ Exp Exp
  - $=$ Exp Exp
  - $<$ Exp Exp

Language design decisions for functions

- Do we require a function name? or do we allow anonymous functions? (**first-class function** functions are variables of the function type)
- Do we require an explicit return?
- Do we allow to write a function in the function body (nested function)?
- Do we allow high order functions? (Consider C function pointers)
- Do we allow default values in the parameters?
- Do we support variant parameters? foo(int x, ...)
- An alternative syntax for CallExp:
  (LambdaExp Exp^{+})
  Should we perform syntactic or semantic checks to report an invalid expression (1 1)?
- There are errors that we cannot use CFG to check but only can check them in evaluators, e.g., checking the numbers of formal paramaters and actual parameters must be equal for a CallExp.

How to extend the semantics for FuncLang?

- Any new types of values to be added?
- Semantic rules?
- How to implement it?

New Values for FuncLang

- Lambda expression is function, it has values, and can be passed as parameters, return from a function and stored in the environment

`Value`	::=		*Values*
		`NumVal`	*Numeric Values*
	|	`FunVal`	*Function Values*
`NumVal`	::=	`(NumVal n)`	*NumVal*
`FunVal`	::=	$(\texttt{FunVal}\ \texttt{var}_0, \ldots, \texttt{var}_n\ \texttt{e}\ \texttt{env})$	*FunVal*
		where $\texttt{var}_0, \ldots, \texttt{var}_n \in \text{Identifier}$,	
		$\texttt{e} \in \text{Exp}$, $\texttt{env} \in \text{Env}$	

New Values for FuncLang: Implementation

```
class FunVal implements Value {
  private Env _env;
  private List<String> _formals;
  private Exp _body;
  public FunVal(Env env, List<String> formals, Exp body) {
    _env = env;
    _formals = formals;
    _body = body;
  }
  public Env env() { return _env; }
  public List<String> formals() { return _formals; }
  public Exp body() { return _body; }
}
```

Figure 5.4: FunVal: A New Kind of Value for Functions

```
Value visit (LambdaExp e, Env env) {
  return new Value.FunVal(env, e.formals(), e.body());
}
```

Evaluate a Lambda Expression

$$\frac{\texttt{(FunVal } \texttt{var}_i\texttt{,for i = 0...k } \texttt{exp}_b\texttt{ env) = v}}{\texttt{value (LambdaExp } \texttt{var}_i\texttt{,for i = 0...k } \texttt{exp}_b\texttt{) env = v}}\ \textsc{Value of LambdaExp}$$

Evaluate a Call Expression

```
(define identity
   (lambda (x) x)
)
$(identity i)
```

1. *Evaluate operator.* Evaluate the expression whose value will be the function value. For example, for the call expression `(identity i)` the variable expression `identity`'s value will be the function value.

2. *Evaluate operands.* For each expression that is in place of a formal parameter, evaluate it to a value. For example, for the call expression `(identity i)` the variable expression `i`'s value will be the only operand value.

3. *Evaluate function body.* This step has three parts.

   a) Find the expression that is the body of the function value,

   b) create a suitable environment for that body to evaluate, and

   c) evaluate the body.

Evaluate a Call Expression

VALUE OF CALLEXP

$$\frac{\begin{array}{c}\texttt{value}\ \texttt{exp}_b\ \texttt{env}_{k+1}\ \texttt{=}\ \texttt{v}\\ \texttt{value exp env = (FunVal}\ \texttt{var}_i\texttt{,for i = 0...k}\ \texttt{exp}_b\ \texttt{env}_0\texttt{)}\\ \texttt{value}\ \texttt{exp}_i\ \texttt{env = v}_i,\ \text{for } \mathrm{i} = 0...\mathrm{k}\\ \texttt{env}_{i+1}\ \texttt{= (ExtendEnv}\ \texttt{var}_i\ \texttt{v}_i\ \texttt{env}_i\texttt{)},\ \text{for } \mathrm{i} = 0...\mathrm{k}\end{array}}{\texttt{value (CallExp exp}\ \texttt{exp}_i\texttt{,for i = 0...k) env = v}}$$

Dynamic Errors in FuncLang

Errors that cannot be found using grammar rules:

- number of formal parameters and actual parameters do not match (context-sensitivity part of the language, cannot been find by the grammar)
- if exp (operator) does not return a function value

`Value`	::=		*Values*
		`NumVal`	*Numeric Values*
	|	`FunVal`	*Function Values*
	|	`DynamicError`	*Dynamic Error*
`NumVal`	::=	`(NumVal n)`	*NumVal*
`FunVal`	::=	$(\texttt{FunVal var}_0, .., \texttt{var}_n\ \texttt{e env})$	*FunVal*
		where $\texttt{var}_0, .., \texttt{var}_n \in$ Identifier,	
		$\texttt{e} \in$ Exp, $\texttt{env} \in$ Env	
`DynamicError`	::=	`(DynamicError s)`,	*DynamicError*
		where `s` $\in$ the set of Java strings	

Implementation: Evaluating a Call Expression

```
Value visit (CallExp e, Env env) {
  //Step 1: Evaluate operator
  Object result = e.operator().accept(this, env);

  if (!( result instanceof Value.FunVal))
    return new Value.DynamicError("Operator not a function");
  Value.FunVal operator = (Value.FunVal) result;
  List<Exp> operands = e.operands();

  //Step 2: Evaluate operands
  List<Value> actuals = new ArrayList<Value>(operands.size());
  for(Exp exp : operands)
    actuals.add((Value)exp.accept(this, env));

  //Step 3: Evaluate function body
  List<String> formals = operator.formals();
  if (formals.size()!=actuals.size())
    return new Value.DynamicError("Argument mismatch in call ");
  Env fenv = appendEnv(operator.env(), initEnv);
  for (int i = 0; i < formals.size(); i++)
    fenv = new ExtendEnv(fenv, formals.get(i), actuals.get(i));
  return (Value) operator.body().accept(this, fenv);
}
```

Control Structure: Extending Value

`Value`	::=		*Values*
		`NumVal`	*Numeric Values*
	|	`BoolVal`	*Boolean Values*
	|	`FunVal`	*Function Values*
	|	`DynamicError`	*Dynamic Error*
`NumVal`	::=	`(NumVal n)`	*NumVal*
`BoolVal`	::=	`(BoolVal true)`	***BoolVal***
	|	`(BoolVal false)`	
`FunVal`	::=	`(FunVal` $\texttt{var}_0$`,..,` $\texttt{var}_n$ `e env)` where $\texttt{var}_0$`,..,` $\texttt{var}_n$ $\in$ Identifier, `e` $\in$ Exp, `env` $\in$ Env	*FunVal*
`DynamicError`	::=	`(DynamicError s)`, where `s` $\in$ the set of Java strings	*DynamicError*

Figure 5.7: The set of Legal Values for the Funclang Language with new **boolean value**

Control Structure: Semantic Rules

Value of GreaterExp

$$\frac{\texttt{value exp}_0\ \texttt{env = (NumVal n}_0\texttt{)} \qquad \texttt{value exp}_1\ \texttt{env = (NumVal n}_1\texttt{)} \quad \texttt{n}_0\ \texttt{>}\ \texttt{n}_1\ \texttt{= b}}{\texttt{value (GreaterExp exp0 exp}_1\texttt{) env = (BoolVal b)}}$$

Value of EqualExp

$$\frac{\texttt{value exp}_0\ \texttt{env = (NumVal n}_0\texttt{)} \qquad \texttt{value exp}_1\ \texttt{env = (NumVal n}_1\texttt{)} \quad \texttt{n}_0\ \texttt{==}\ \texttt{n}_1\ \texttt{= b}}{\texttt{value (EqualExp exp0 exp}_1\texttt{) env = (BoolVal b)}}$$

Value of LessExp

$$\frac{\texttt{value exp}_0\ \texttt{env = (NumVal n}_0\texttt{)} \qquad \texttt{value exp}_1\ \texttt{env = (NumVal n}_1\texttt{)} \quad \texttt{n}_0\ \texttt{<}\ \texttt{n}_1\ \texttt{= b}}{\texttt{value (LessExp exp0 exp}_1\texttt{) env = (BoolVal b)}}$$

Control Structure: Semantic Rules

VALUE OF IFEXP - TRUE

$$\frac{\texttt{value exp}_{cond}\ \texttt{env = (BoolVal true)} \qquad \texttt{value exp}_{then}\ \texttt{env = v}}{\texttt{value (IfExp exp}_{cond}\ \texttt{exp}_{then}\ \texttt{exp}_{else}\texttt{) env = v}}$$

VALUE OF IFEXP - FALSE

$$\frac{\texttt{value exp}_{cond}\ \texttt{env = (BoolVal false)} \qquad \texttt{value exp}_{else}\ \texttt{env = v}}{\texttt{value (IfExp exp}_{cond}\ \texttt{exp}_{then}\ \texttt{exp}_{else}\texttt{) env = v}}$$

Semantics of List: Extending the Values

`Value`	::=		*Values*
		`NumVal`	*Numeric Values*
	|	`BoolVal`	*Boolean Values*
	|	`FunVal`	*Function Values*
	|	`PairVal`	*Pair Values*
	|	`NullVal`	*Null Value*
	|	`DynamicError`	*Dynamic Error*
`NumVal`	::=	`(NumVal n)`	*NumVal*
`BoolVal`	::=	`(BoolVal true)`	*BoolVal*
	|	`(BoolVal false)`	
`FunVal`	::=	(FunVal $\texttt{var}_0$, . . , $\texttt{var}_n$ e env)	*FunVal*
		where $\texttt{var}_0$, . . , $\texttt{var}_n$ $\in$ Identifier,	
		e $\in$ Exp, env $\in$ Env	
`PairVal`	::=	(PairVal $\texttt{v}_0$ $\texttt{v}_1$)	***PairVal***
		where $\texttt{v}_0$, $\texttt{v}_1$ $\in$ `Value`	
`NullVal`	::=	`(NullVal)`	***NullVal***
`DynamicError`	::=	`(DynamicError s)`,	*DynamicError*
		where `s` $\in$ the set of Java strings	

Figure 5.9: The set of Legal Values for the Funclang Language with new pair and null values

Semantics for List Operations

$$\texttt{value (ListExp exp}_0 \texttt{ ... exp}_n\texttt{) env = (ListVal val}_0 \texttt{ lval}_1\texttt{)}$$

$$\texttt{where exp}_0 \texttt{ ... exp}_n \in \texttt{Exp} \quad \texttt{env} \in \texttt{Env}$$

$$\texttt{value exp}_0 \texttt{ env = val}_0\texttt{, ..., value exp}_n \texttt{ env = val}_n$$

$$\texttt{lval}_1 \texttt{ = (ListVal val}_1 \texttt{ lval}_2\texttt{), ...,}$$

$$\texttt{lval}_n \texttt{ = (ListVal val}_n \texttt{ (EmptyList))}$$

A corollary of the relation is:

$$\texttt{value (ListExp) env = (EmptyList)}$$

Note.
exp_0 ... lval_1
exp_1 ... lval_2
...
exp_n = val_n

Semantics for List Operations

The value of a `CarExp` is given by:

`value (CarExp exp) env = val`

`where exp` $\in$ `Exp` `env` $\in$ `Env`

`value exp env = (ListVal val lval) where lval` $\in$ `ListVal`

The value of a `CdrExp` is given by:

`value (CdrExp exp) env = lval`

`where exp` $\in$ `Exp` `env` $\in$ `Env`

`value exp env = (ListVal val lval) where lval` $\in$ `ListVal`

The value of a `ConsExp` is given by:

`value (ConsExp exp exp') env = (ListVal val lval)`

`where exp,exp'` $\in$ `Exp` `env` $\in$ `Env` `value exp env = val`

`value exp' env = lval`

The value of a `NullExp` is given by:

`value (NullExp exp) env = #t if value exp env = (EmptyList)`

`value (NullExp exp) env = #f`

`if value exp env = (ListVal val lval') where lval'` $\in$ `ListVal`

`where exp` $\in$ `Exp` `env` $\in$ `Env`

Review

FuncLang: Function

- Abstraction, parameterize computations
- Lambda Expression, Call Expression
- Combine with Let and Define: functions are also variables
- if then else: Condition $>$, $>$, $=$
- list: (car, cdr, null?, cons, list)
  Understanding of pair and list: List is a pair, pair is not a list
- syntax: CFG, semantic: operational
- recursive function, high order function, currying
- Values: NumVal, FunVal, PairVal, NullVal, BoolVal, UnitVal, Dynamic Errors