FuncLang

September 22, 2016

Overview

- FuncLang
- Syntax
- Semantics

Abstraction in Programming Languages

- Variable: fixed abstraction – you cannot change functionality

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

- Function (procedure, method): **parameterization for computation** – you can reuse the functionality for different concrete input: the ability to define a procedure, and the ability to call a procedure.

Lambda Abstraction and Call

- A tool for defining anonymous function, a language features

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

- Compare to the notion of procedures and methods in ALGOL family of languages: C, 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 the notion of procedures and methods in ALGOL family of languages: C, C++, C#, Java (semantics):
  Procedures and methods: proxy of the location of the subsection of code section
  - adjust the environment
  - jump to the location

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

Examples: Lambda abstraction

```
(
 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
)
```

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
```

In-class Exercise

- Write five lambda abstraction and calls, discuss with your neighbors
- Select your favorite functions

Built-in Functions List

1. List: constructor for a list; parameters: values used to initialize a list, e.g., (list 1 1 1 1 1)
2. car: takes a pair or a list as an argument, returns the first element, e.g., (car (list 11 1))
3. cdr: take a pair or a list as an argument, returns the second element, e.g., (cdr (list 342))
4. cons: If the second value is a list, it produces a new list with the first value appended to the front of the second value list. Otherwise, it produces a pair of two argument values. e.g., (cons 541 (list 342))
5. null? The function null? takes a single argument and evaluates to #t if that argument is an emptylist.

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
4. Lists are constructed by using the cons keyword, as is shown here:
   $>$ (cons 1 (list))
   (1)

Examples: Using Built-In Functions

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

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

In-class Exercise

- Write three functions related to list and discuss with your neighbors
- Select your favorite functions

Recursive Function

- Recursive function mirror the definition of the input data type

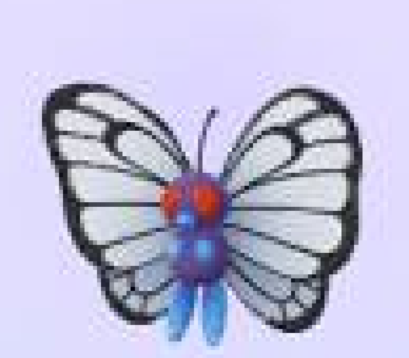
$$List := (list) \mid (cons\ \texttt{val}\ List),\ \ \texttt{where val} \in \texttt{Value}$$

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

In-class Exercise

- Write one recursive function in funclang and discuss with your neighbors
- Select your favorite functions

Pokemon?



High Order Function

First-class function

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

```
(define addthree
   (lambda (x)(+ x 3))
)
(define returnone
   (lambda (f) (f 1))
)
$(returnone addthree)
```

High Order Function

Function definition:

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

Function call:

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

In-class Exercise

- Write three high order functions and discuss with your neighbors
- Select your favorite functions

Function Works with Data Structures

```
(define pair
    (lambda (fst snd)
      (lambda (op)
        (if op fst snd)
      )
    )
)
(define abeautifulpair (pair 1 2))
(define first (lambda (p) (p#t)))
$ (first pair)
```

Currying

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)
   )
  )
)
```

Write FuncLang programs?

Syntax

What is new?

- Lambda expression
- Call
- Function with a name
- High order function and Currying

Grammar for FuncLang

```
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
```

Value of a Lamdba Expression

- 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`	::=	`(FunVal var`$_0$`,.., var`$_n$` e env)` where `var`$_0$`,.., var`$_n$ $\in$ Identifier, `e` $\in$ Exp, `env` $\in$ Env	*FunVal*

Value of a Lamdba Expression

VALUE OF LAMBDAEXP

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

Evaluate Call Expressions

```
(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.

Value of a Call Expression

VALUE OF CALLEXP

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

Dynamic Errors

- 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

Review and Further Reading

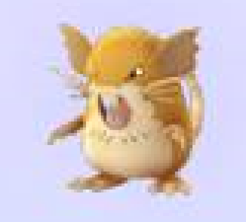
FuncLang: lambda expression and call

- Syntax: lambda expression and call (AST node, visitor interface)
- Semantics: funval, dynamic errors

Further reading:

- Rajan: CH 5, Sebesta Ch 9, 10

How to Further Improve the Expressiveness of FuncLang



Control Structure

- if expression: three mandatory expressions – the condition, then, and else expressions
- comparison expression: $>$, $<$, $=$ (not prefix form)

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.

How to Extend the Semantics for the Grammar?

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

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, $\texttt{e} \in$ Exp, $\texttt{env} \in$ Env	*FunVal*
`DynamicError`	::=	`(DynamicError s)`, where $\texttt{s} \in$ the set of Java strings	*DynamicError*

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

Semantic Rules

Value of GreaterExp

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

Value of EqualExp

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

Value of LessExp

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

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

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

Support List and its Operations

- list: creating a list
- cons: constructing a pair
- null?: check if a list is a null
- car: get the first element of a pair
- cdr: get the second element of a pair

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*
|	(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.

How to Compute Values

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

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

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

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

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

A corollary of the relation is:

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

How to Compute Values

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`