Functions, Variables and Substitution

CS 350

Dr. Joseph Eremondi

Last updated: July 10, 2024

Overview: Functions

Adding functions to the language

- Want to be able to re-use code, compute in terms of variables

Adding functions to the language

- Want to be able to re-use code, compute in terms of variables
- Two new Curly features

Adding functions to the language

- Want to be able to re-use code, compute in terms of variables
- Two new Curly features
- Function definitions

Adding functions to the language

- Want to be able to re-use code, compute in terms of variables
- Two new Curly features
- Function definitions
  - `{define {f x} {+ x 3}}`

Adding functions to the language

- Want to be able to re-use code, compute in terms of variables
- Two new Curly features
- Function definitions
  - `{define {f x} {+ x 3}}`
- Function calls

Adding functions to the language

- Want to be able to re-use code, compute in terms of variables
- Two new Curly features
- Function definitions
  - `{define {f x} {+ x 3}}`
- Function calls
  - Sometimes called *function applications*

Adding functions to the language

- Want to be able to re-use code, compute in terms of variables
- Two new Curly features
- Function definitions
  - `{define {f x} {+ x 3}}`
- Function calls
  - Sometimes called *function applications*
  - `{f 10}`, produces `13`

Adding functions to the language

- Want to be able to re-use code, compute in terms of variables
- Two new Curly features
- Function definitions
  - `{define {f x} {+ x 3}}`
- Function calls
  - Sometimes called *function applications*
  - `{f 10}`, produces `13`
  - To start: single argument, inputs and outputs number

Syntax: Definitions

- New datatype for function definitions

Syntax: Definitions

- New datatype for function definitions

Syntax: Definitions

- New datatype for function definitions

```
(define-type FunDef
  (mkFunDef [name : Symbol]
            [arg : Symbol]
            [body : Expr]))
```

Parser: Definitions

- New parser

Parser: Definitions

- New parser
  - Note: function definition is not an expression

Parser: Definitions

- New parser
  - Note: function definition is not an expression
  - Need to elaborate body after parsing

Parser: Definitions

- New parser
  - Note: function definition is not an expression
  - Need to elaborate body after parsing

Parser: Definitions

- New parser
  - Note: function definition is not an expression
  - Need to elaborate body after parsing

```
(define (parse-fundef [s : S-Exp]) : FunDef
  (cond
    [(s-exp-match? `{define {SYMBOL SYMBOL} ANY} s)
     (mkFunDef
        (s-exp->symbol
           (first (s-exp->list (second (s-exp->list s)))))
         (s-exp->symbol
            (second (s-exp->list (second (s-exp->list s)))))
         (elab (parse (third (s-exp->list s)))))]
    [else (error 'parse-fundef "invalid input")]))
```

New Expression Syntax

- We need a way to:

New Expression Syntax

- We need a way to:
  - Call a function

New Expression Syntax

- We need a way to:
  - Call a function
  - Refer to the parameter of a function inside its body

New Expression Syntax

- We need a way to:
  - Call a function
  - Refer to the parameter of a function inside its body

New Expression Syntax

- We need a way to:
  - Call a function
  - Refer to the parameter of a function inside its body

```
(define-type Expr
  (NumLit [n : Number])
  (Plus [left : Expr]
        [right : Expr])
  (Times [left : Expr]
         [right : Expr])
  (If0 [test : Expr]
       [thenBranch : Expr]
       [elseBranch : Expr])
  (Var [x : Symbol])
  (FunCall [f : Symbol]
           [arg : Expr]))
```

- Also add variables and calls to surface syntax

Interpreting Variables

- What is the meaning of a variable in a program?

Interpreting Variables

- What is the meaning of a variable in a program?
  - Variable is just a placeholder for whatever the value is given to the function

Interpreting Variables

- What is the meaning of a variable in a program?
  - Variable is just a placeholder for whatever the value is given to the function
- Interpreting a variable is an **error**

Interpreting Variables

- What is the meaning of a variable in a program?
  - Variable is just a placeholder for whatever the value is given to the function
- Interpreting a variable is an **error**
  - Similar to “out of scope” or “undefined variable” errors

Interpreting Variables

- What is the meaning of a variable in a program?
  - Variable is just a placeholder for whatever the value is given to the function
- Interpreting a variable is an **error**
  - Similar to “out of scope” or “undefined variable” errors
- Could statically check if variable was out of scope

Interpreting Variables

- What is the meaning of a variable in a program?
  - Variable is just a placeholder for whatever the value is given to the function
- Interpreting a variable is an **error**
  - Similar to “out of scope” or “undefined variable” errors
- Could statically check if variable was out of scope
  - Might do later in the course

Interpreting Function Calls

- Function call:

Interpreting Function Calls

- Function call:
  - Looks up body of function

Interpreting Function Calls

- Function call:
  - Looks up body of function
  - Replaces variable with value given

Interpreting Function Calls

- Function call:
  - Looks up body of function
  - Replaces variable with value given
  - Evaluates the body after that replacement

Interpreting Function Calls

- Function call:
  - Looks up body of function
  - Replaces variable with value given
  - Evaluates the body after that replacement
- Interpreter needs context now

Interpreting Function Calls

- Function call:
  - Looks up body of function
  - Replaces variable with value given
  - Evaluates the body after that replacement
- Interpreter needs context now
  - List of function definitions

Interpreting Function Calls

- Function call:
  - Looks up body of function
  - Replaces variable with value given
  - Evaluates the body after that replacement
- Interpreter needs context now
  - List of function definitions

Interpreting Function Calls

- Function call:
  - Looks up body of function
  - Replaces variable with value given
  - Evaluates the body after that replacement
- Interpreter needs context now
  - List of function definitions

```
(define (interp [expr : Expr]
                [defs : (Listof FunDef)]) : Number
  ....)
```

How can we replace a variable

Substitution

- More recursion!

Substitution

- More recursion!
- “Replace all occurrences of the variable `x` with the expression `s` inside of the expression `t`”

Substitution

- More recursion!
- “Replace all occurrences of the variable `x` with the expression `s` inside of the expression `t`”
- Do this by traversing the expression recursively

Substitution

- More recursion!
- “Replace all occurrences of the variable `x` with the expression `s` inside of the expression `t`”
- Do this by traversing the expression recursively
- Critical operation in programming languages

Cases for Substitution

- Variable y: check if $y = x$ i.e. it's the variable the one we're replacing

Cases for Substitution

- Variable y: check if $y = x$ i.e. it's the variable the one we're replacing
  - If it is, produce s

Cases for Substitution

- Variable y: check if $y = x$ i.e. it's the variable the one we're replacing
  - If it is, produce s
  - Otherwise, produce y again

Cases for Substitution

- Variable y: check if y = x i.e. it's the variable the one we're replacing
  - If it is, produce s
  - Otherwise, produce y again
- Everything else: recursively substitute in the sub-expressions

Cases for Substitution

- Variable y: check if $y = x$ i.e. it's the variable the one we're replacing
  - If it is, produce s
  - Otherwise, produce y again
- Everything else: recursively substitute in the sub-expressions
  - Will have more complex cases later

Code for Substitution

Code for Substitution

```
;; `(subst x s t)` replaces all occurences of `x` in `t` with `s`.
;; We use this to implement function calls
(define (subst [toReplace : Symbol]
               [replacedBy : Expr]
               [replaceIn : Expr]) : Expr
  (type-case Expr replaceIn
    ;; Base case: we're replacing a variable in an expression
    ;; where that expression is a variable
    [(Var x)
     ;; Check if it's the variable we're replacing
     (if (equal? x toReplace)
         replacedBy ;; If so, produce what we're replacing it with
         (Var x))] ;; else produce the original variable
    ;; Number is a leaf, no sub-expressions
    ;; so return it unchanged
    [(NumLit n) (NumLit n)]
    ;; ...
```

Code for Substitution (ctd)

Code for Substitution (ctd)

```
;; Plus has two sub-expressions,
;; so we replace the variable in both sub-expressions
[(Plus l r)
 (Plus (subst toReplace replacedBy l)
       (subst toReplace replacedBy r))]
;; other operations work similarly
[(Times l r)
 (Times (subst toReplace replacedBy l)
        (subst toReplace replacedBy r))]
[(Ifo test thn els)
 (Ifo (subst toReplace replacedBy test)
      (subst toReplace replacedBy thn)
      (subst toReplace replacedBy els))]
;; Have to decide how to handle namespaces
;; For now, functions and variables are different namespaces
;; so we don't ever replace a function name in subst
[(Call funName arg)
 (Call funName (subst toReplace replacedBy arg))])
```