Computing Science (CMPUT) 325

Nonprocedural Programming

Department of Computing Science
University of Alberta

Fun - A Simple Functional Language

- A program in the `Fun` language is a collection of functions
- Functions are defined over lists and atoms
- We do computation by evaluating functions on given argument(s)
- We use a math-like syntax (for now)

Fun Example

$$f(x, y) = x * x + y$$

- The symbol = is read as **is defined as**
- f(x,y) is called the **lefthand side**
- The function definition is on the **righthand side**

Function Evaluation

- The function definition means that the two sides are equal
- As in math, we can **apply** a function by **replacing equals by equals**
- Consider $f(x, y) = x * x + y$
- The definition says that for **any** x and y, the lefthand side can be replaced by the righthand side
- In logic, this corresponds to having **forall quantifiers** for each of the variables:
- $\forall x \forall y : f(x, y) = x * x + y$

Function Evaluation (continued)

- $f(x, y) = x * x + y$
- We can build an interpreter which mechanically evaluates a function application
- $f(2, 3) \longrightarrow 2 * 2 + 3$
- This is a one-step evaluation of function application $f(2, 3)$
- An interpreter needs to do two things:
  - replace f by its definition (its righthand side)
  - substitute the variables x, y with the given arguments 2 and 3
- Note: I use the $\longrightarrow$ symbol here to mean "the result of the evaluation is".

Terminology

- Syntax: which texts are valid programs? Here, simple math-like functions
- Semantics: “meaning” of a program.
- Execution: Evaluation of function applications. Here, based on replacing equals by equals

Terminology about Functions

- **function** A mapping from domain to co-domain
- **function definition** What the function does
- **function application** Evaluate function for specific arguments

Objects in the Fun Language

- **Atoms**: primitive, inseparable, including integers and real numbers
  - a, id73, helloWorld, -15, 3.1415
- **Lists**: defined inductively:
  - () is a list, called an empty list
  - If $x_1, ..., x_n$ are lists or atoms, then $(x_1 ... x_n)$ is a list
  - Nothing else is a list

Examples of Lists

- Note: The definition allows lists to be nested to arbitrary depth
- (a (b) c (d))
- (a ((b) c (d) ((((e))))))
- ((((((((((())))))))))(((())))))

Manipulating Lists - Selectors and Constructor

- Given a (possibly complex) list, how to access the data in it?
- How to create new lists?
- We only need three **primitive** functions for this
- Primitive functions are those we assume to be "given", i.e. we do not worry about how they are defined

Selector: `first`

- `first` returns the first element in a list
- We will often abbreviate it as `f`
- `first( (a1 ...  an) )` $\longrightarrow$ `a1`
- Note: brackets ( ) are used both for lists and for function calls. Yes this is confusing at first. But you will get used to it.
- It is an error to call `first` with an argument that is not a list, or with an empty list

Examples for `first`

- `first( (a b c) )` $\longrightarrow$ `a`
- `first( (((a) b) c) )` $\longrightarrow$ `((a) b)`
- `first( (((()))) )` $\longrightarrow$ `((()))`
  - Note: in last example, function `first` is called with the argument `(((())))`.
  - This argument is a list with a single element.
  - That element is again a nested list, namely `((()))`, which is the result returned by function `first`.

Selector: `rest`

- `rest` returns all but the first element in a list
- We often abbreviate it as `r`
- `rest( (a1 a2 ... an) )` $\longrightarrow$ `(a2 ... an)`
- It is an error to call `rest` with an argument that is not a list, or with an empty list

Examples for `rest`

- `rest( (a b c) )` $\longrightarrow$ `(b c)`
- `rest( (a b) )` $\longrightarrow$ `(b)` Note: **not just b**, but list `(b)`
- `rest( (a) )` $\longrightarrow$ `()`
  - Note: `(a)` is a list with one element, namely a. The rest of the list is empty.

Access Within Nested Lists

- Using compositions of calls to `first` (or `f`) and `rest` (or `r`), we can get any component (atom or sublist) from a given list, no matter how deeply it is nested.
- Example: From the list L = (a (b) (c d)), how can we get atom c?
  - `r(L) = ((b) (c d))`
  - `r(r(L)) = ((c d))`
  - `f(r(r(L))) = (c d)`
  - `f(f(r(r(L)))) = c`
- For brevity we can omit some parentheses and define a new function, e.g. `ffrr(L)`

Constructor

- Construct a list when given first element `X` and rest of list `(a1 ... an)`
- `cons(X, (a1 ... an) )` $\longrightarrow$ `(X a1 ... an)`
- Empty list as second argument of cons:
- `cons(a,()) = (a)`
- Using cons, you can construct any nested list
- Example: construct list `L = (a (b))`
- `cons(a, cons(cons(b, () ), () ) )`

Example - Details

- construct list `L = (a (b))`
- `cons(b, () ) = (b)`
- `cons((b), () ) = ((b))`
- `cons( a, ((b)) ) = (a (b))`
- `cons(a, cons(cons(b, () ), () ) )`

Primitive Functions

- In theory there is a way to define any function from scratch
  - (We will talk about this topic later)
- In practice it is useful to assume we have some more **primitive functions**
- The next slide lists them

Primitive Functions List

- Arithmetic, e.g. $+, -, *, /$
- Comparison functions, e.g. $<, >$
- `if then else`
- `null(x)`: true if x is an empty list, false otherwise
- `eq(x,y)`: true if x and y are the same atom, false otherwise
- `atom(x)`: true if x is an atom

Primitive Functions - Comments

- We can build a powerful language from just a few primitives
- Many other useful, even frequently used functions can be defined from the given primitives.
- Similar idea in hardware: RISC (reduced instruction set computer)

Function Definition in Fun

- No notion of variable-as-storage
- **No assignment statement** as used in procedural languages
- No loop constructs such as `while` and `repeat`
- **RECURSION** as the only mechanism to define non-trivial functions

Example: `count` Function

- `count(L)` returns the number of elements in L,
- assuming L is a list
- Definition:

```
count(L) =
   if null(L) then 0
   else 1 + count(r(L))
```

Applying `count` and Execution Trace

```
count( (a b c d) )
⟶ 1 + count( (b c d) )
⟶ 1 + 1 + count( (c d) )
⟶ 1 + 1 + 1+ count( (d) )
⟶ 1 + 1 + 1 + 1 + count( () )
⟶ 1 + 1 + 1 + 1 + 0
⟶ 4
```

- Note this was not a full trace.
- Did not show details of “replacing equals by equals” and of evaluating primitive functions
- Start of full trace on next slide

Start of Full Trace for `count`

```
count( (a b c d) )

if null( (a b c d) ) then 0 else 1 + count(r( (a b c d)

if False then 0 else 1 + count( r( (a b c d) ) )

1 + count( r( (a b c d) ) )))

1 + count( (b c d) )
......
```

Example - Append

- `append(L1,L2)` - append two lists L1 and L2
- Examples:
  `append( (1 2), (a b c) )` $\longrightarrow$ `(1 2 a b c)`
  `append( ((1) 2), (a (b c)) )` $\longrightarrow$
  `((1) 2 a (b c))`

Definition of Append

```
append(L1,L2) =
   if null(L1) then L2
   else cons(f(L1), append(r(L1), L2))
```

Trace of Append

```
append((1 2), (a b c))
⟶ cons(1, append((2), (a b c)))
⟶ cons(1, cons(2, append((), (a b c))))
⟶ cons(1, cons(2, (a b c)))
⟶ cons(1, (2 a b c))
⟶ (1 2 a b c)
```

A Different Definition of Append

- Another idea to implement `append(L1,L2)`:
- Get the last element of L1
- Use `cons` to put it in front of L2
- Now append:
  - The remainder of L1, without the last element
  - The new cons'ed list with last element and L2
- How does this process terminate?
  - In each new call, L1 has one element less than before
  - If L1 is empty, just return L2

A Different Definition of Append (Continued)

- It makes sense here to break the solution into different smaller functions
- Remember the principle: **one function should do one thing**
- Here we use three functions: `last, removeLast, append`
- `last` returns the last element of a non-empty list

```
last(L) =
   if null(r(L)) then f(L)
   else last(r(L))
```

A Different Definition of Append (Continued)

- `removeLast` returns a copy of the list, but without its last element

```
removeLast(L) =
   if null(r(L)) then ()
   else cons(f(L), removeLast(r(L)))

append(L1,L2) =
   if null(L1) then L2
   else append(removeLast(L1),
         cons(last(L1), L2))
```

A Different Definition of Append - Comments

- Note the non-destructive style of programming. Instead of modifying the input list, we build a new output list
- This sounds very inefficient, but often is OK.
- In practice, optimized functional languages can avoid some copying by structure sharing.

Trace of Second Version of Append

```
append( (1 2), (a b c) )
⟶ append(removeLast( (1 2) ),
           cons(last( (1 2) ), (a b c) ))
⟶ append( (1), cons(2, (a b c) ))
⟶ append( (1), (2 a b c) )
⟶ append(removeLast( (1) ),
           cons(last( (1) ), (2 a b c) ))
⟶ append( (), cons(1, (2 a b c) ))
⟶ append( (), (1 2 a b c) )
⟶ (1 2 a b c)
```

Remarks on Two Versions of Append

- The first solution is more elegant than the second
- It is also more efficient
- In general, access/change to the front of a list (first, cons) is better than access/change to the end

Example - Reverse List

- `reverse(L)`: reverse the elements in L
- E.g. reverse((a b c)) $\longrightarrow$ (c b a)
- Definition:

```
reverse(L) =
   if null(L) then L
   else append(reverse(r(L)), cons(f(L), ()))
```

- Think about: Why use `append` instead of `cons` in last line?

Trace of Reverse

```
reverse( (a b c) )
⟶ append(reverse((b c), (a)))
⟶ append(append(reverse(c), (b)) (a))
⟶ append(append(append(reverse(()),
          (c)), (b)), (a))
⟶ append(append(append((), (c)), (b)), (a))
⟶ append(append((c), (b)), (a))
⟶ append((c b), (a))
⟶ (c b a)
```

Example - Binary Tree

- Goal: implement a binary tree data structure and some operations, such as inserting elements
- Two main tasks:
  - Decide how trees are represented by lists
  - Implement an **abstract data type** for binary trees, and the operations on them, as a set of functions
- The user will work with trees using only these functions. The user is protected from the details of our data representation
- We will build up a data structure and functionality bottom-up, step by step, similar to what we have done for lists

Tree Representation

- One possible representation scheme:
- Empty tree: represented by the atom `nil`
- Non-empty tree: three-element list,
  `(left-subtree node-value right-subtree)`
- Food for thought:
  - Can you think of a different representation?
  - Are there any problems with storing the value `nil` itself?

Tree Representation - Examples

- `(nil 5 nil)`: A tree consisting of a single node with node value 5
- Example:
  `((nil 2 nil) 4 ((nil 5 nil) 6 (nil 8 nil)))`
  - Tree with 5 nodes
  - Root value 4
  - Left subtree has one node with value 2
  - Right subtree has three nodes: root 6, left 5, right 8

Selectors, Constructors, and Tests for Binary Tree

- Selectors: `leftTree, rightTree, nodeValue`
  - `leftTree(Tr) = f(Tr)` ... left subtree of `Tr`
  - `rightTree(Tr) = f(r(r(Tr)))`
  - `nodeValue(Tr) = f(r(Tr))`
- Constructors: `consNilTr, consTree`
  - `consNilTr() = nil` ... return an empty tree
  - `consTree(L, V, R) = cons(L, cons(V, cons(R, ())))` ... construct tree with given subtrees L,R and value V
- `isEmpty(Tr) = eq(Tr, nil)` ... return True if `Tr` is empty tree

Building an Abstract Tree Data Type

- The functions from last slide are the only ones that need direct knowledge of our tree representation
- Everything else can be implemented in terms of these basic functions - providing such a base set of functions is the essence of implementing an abstract data type in functional programming
- Note the analogy with lists, where we built many other useful functions from basic functions `first, rest, cons, null`
- If we ever decided to change our tree representation, we only need to change the few basic functions

Example: `insert` into Tree

- Assume our trees contain integer values and are sorted such that:
- All values in left subtree $<$ node value $<$ all values in right subtree
- No value appears more than once
- Now we define an `insert` function that maintains the sorted property
- `insert(Tr, Int)`: Inserts integer `Int` into binary tree `Tr`.

Definition of `insert`

```
insert(Tr, Int) =
   if isEmpty(Tr)
      then consTree(consNilTr(), Int, consNilTr())
   else if Int = nodeValue(Tr) ...  Int already in
      then Tr
   else if Int < nodeValue(Tr)
      then consTree(insert(leftTree(Tr), Int),
         nodeValue(Tr),
         rightTree(Tr))
   else consTree(leftTree(Tr),
      nodeValue(Tr),
      insert(rightTree(Tr), Int))
```

Summary of Simple Functional Language

- We defined `Fun`, a simple math-like functional language
- Built-in data types: atoms and lists
- A few primitive functions allow us to easily define other useful functions
- We built an abstract data type for binary trees:
- We chose a representation of trees by lists
- We implemented a few basic functions to work with such trees, then defined other functions using the basics