Midterm Exam

Midterm exam grades are not necessarily a reflection of your final grade; homework is more important!

If you are worried, come see me!

Announcements

HW5 solutions
Announcements

HW6 out today, due next Wednesday, April 11.

Announcements

Typo on HW6: if you want a new partner, notify me (via email) by Wed, April 4 with your partner's name.

Announcements

Grades for HW3 programming portion, HW4, HW5 will be back soon.

Refresher: First-class functions

- A language with first-class functions treats functions no differently than any other value:
- You can assign functions to variables:
  ```scala
  val f = fn x => x + 1
  ```
- You can pass functions as arguments:
  ```scala
  fun g h = h 3
  g f
  ```
- You can return functions:
  ```scala
  fun k x = fn () => x + 3
  ```
- First-class function support complicates implementation of lexical scope.
First Class Functions

- To implement support for first class functions, we need two additional data structures:
  - Access links
  - Closures
- The implementation difficulty of maintaining lexical scope for first class functions is called the *funarg problem*.

Access link

- An *access link* is a pointer from the current activation record to the activation record of the closest lexical scope.
- In other words, the access link in the activation frame for a function \( f \) points to where \( f \) was defined.
- Why do we need access links? So that the language can determine the values of free variables in a function.

Closure

- A *closure* is a tuple that represents a function value. One tuple value points to a function’s code and the other value points to the activation record of the point of definition of the function (i.e., closest lexical scope).

Example

```plaintext
val x = 4
fun f y = x * y
fun g h = let val x = 7 in (h 3) + x end
```
```plaintext
g f
```
Desugared Example

```haskell
let val x = 4 in
  let f = fn y => x * 4 in
  let g = fn h => let val x = 7 in (h 3) + x in
  g f
end
end
```

Blocks Define Activation Records

```
val x = 4
fun f y = x * y
fun g h = let val x = 7 in (h 3) + x
g f
```

call stack

---

```haskell
val x = 4
fun f y = x * y
fun g h = let val x = 7 in (h 3) + x
g f
```

call stack
val x = 4
fun f y = x * y
fun g h = let val x = 7 in (h 3) + x
  
g f

val x = 4
fun f y = x * y
fun g h = let val x = 7 in (h 3) + x
  
g f
Blocks Define Activation Records

```
val x = 4
fun f y = x * y
fun g h = let val x = 7 in (h 3) + x
```

Activation Records in Functional Langs

```
let val g =
    let
        val x = 1
        fun f () = x + 1
    in f
end
in g() end
```

How is this function evaluated? Do we have a problem when we call g()?
Upward funargs

```ml
let val g = 
    let val x = 1
    fun f () = x + 1
    in f end 
in g() end
```

1. Push let block for `g` onto call stack. We don't yet know `g`'s value.
2. Push let block for `x` and `f`.
3. Return `f`. We have a problem!

4. The fix: delay deallocating record until we are done using it. Instead of using stack, just heap allocate frames and use garbage collector!

5. Now we can call `g()` and it will work correctly.

Safety

- SML is a “safe” language.
- What does that mean?
- It means that execution behavior is determined solely by the program, not:
  a. the implementation of the language, or
  b. the design of the hardware
Safety

- How is safety achieved?
- Type checking rules out manifestly incorrect constructs.
  "hello" - "world"
- However, type checking cannot rule out all errors.

fun sum (xs: int list) =
  foldl (fn (x,acc) => x + acc) 0 xs
fun mean (xs: int list) =
  (sum xs) div (List.length xs)
- For these kinds of errors, we use "exceptions."

Exceptions

- In ML (and in Java), exceptions have three parts:
  a. Exception declaration:
    exception MyException of string
  b. Exception use:
    raise MyException "Don’t send me back to school!"
  c. Exception handling:
    handle MyException msg => msg ^ ”? Fine. Here’s your tuition bill. Pay it yourself.”

Exceptions

- More generally...
  a. Exception declaration:
    exception <exception name> [of <type>]
  b. Exception use:
    raise <exception name> [expr]
  c. Exception handling:
    handle <pattern>

A real example

fun sum (xs: int list) =
  foldl (fn (x,acc) => x + acc) 0 xs
fun mean (xs: int list) =
  (sum xs) div (List.length xs)
- mean [] handle Div => 0;
val it = 0 : int
A real example

exception ZeroLength
fun sum (xs: int list) =
  foldl (fn (x,acc) => x + acc) 0 xs
fun mean (xs: int list) =
  if List.length xs = 0 then
    raise ZeroLength
  else (sum xs) div (List.length xs)
-
mean [] handle
  Div => 0
| ZeroLength => 1 (* … for fun *)
val it = 1 : int

Exceptions aren't just for errors

- Exceptions are actually a special form of goto.
- You can use them to return data to any calling function on the stack.

Exceptions for efficiency

datatype tree =
  Leaf of int
| Node of tree * tree

fun prod (Leaf x) = x
| prod (Node(x,y)) = prod x * prod y
val t = Node(Node(Leaf 1, Leaf 2), Leaf 3)
-
prod t;
val it = 6 : int

Exceptions for efficiency

- What if...
  val t = Node(Node(Leaf 0, Leaf 2), Leaf 3)
  - prod t;
  val it = 0 : int
- Somewhat inefficient, isn't it?
Exceptions for efficiency

exception Zero

fun prod (Leaf x) = 
  if x = 0 then raise Zero else x
|  prod (Node(x,y)) = prod x * prod y
val t = Node(Node(Leaf 0, Leaf 2), Leaf 3)
- prod t handle Zero => 0;
val it = 0 : int

Exceptions are dynamically scoped

- Remember: variable bindings are statically (lexically) scoped.
- Exceptions are dynamically scoped.
- Remember that I said `raise` is like `goto`?
- Where would this `raise` “go to”? We haven’t even used `prod` yet!

```
fun prod (Leaf x) = 
  if x = 0 then raise Zero else x
|  prod (Node(x,y)) = prod x * prod y
val t = Node(Node(Leaf 2, Leaf 0), Leaf 3)
prod t handle Zero => 0;
```

```
val t = Node(Node(Leaf 2, Leaf 0), Leaf 3)
prod t handle Zero => 0;
```
Exceptions are dynamically scoped

\[
\text{val } t = \text{Node(Leaf 2, Leaf 0), Leaf 3)}
\]

\[
\text{prod } t \text{ handle Zero } \Rightarrow 0;
\]

- Pop ("unwind") the stack until handler is found.

raise Zero

\[
\text{val } t = \text{Node(Leaf 2, Leaf 0), Leaf 3)}
\]

\[
\text{prod } t \text{ handle Zero } \Rightarrow 0;
\]
Exceptions are dynamically scoped

val t = Node(Node(Leaf 2, Leaf 0), Leaf 3)
prod t handle Zero => 0;

• Pop ("unwind") the stack until handler is found.

Exceptions are dynamically scoped

val t = Node(Node(Leaf 2, Leaf 0), Leaf 3)
prod t handle Zero => 0;

• Pop ("unwind") the stack until handler is found.

Exceptions are dynamically scoped

val t = Node(Node(Leaf 2, Leaf 0), Leaf 3)
prod t handle Zero => 0;

• Pop ("unwind") the stack until handler is found.

Exceptions are dynamically scoped

val t = Node(Node(Leaf 2, Leaf 0), Leaf 3)
prod t handle Zero => 0;

• Pop ("unwind") the stack until handler is found.

Activity

What is the value of the following expression?

exception X
( let fun f(y) = raise X
    and g(h) = h(1) handle X => 2 in
    g(f) handle X => 4 end)
handle X => 6