Save to Evernote

Tuesday Lecture Week 8

Updated Feb 27, 2018

Storage management and scheme

objects in scheme are allocated dynamically and are never freed like Java, Python, ML, but NOT like C and C++
types are latent, not manifest
dynamic type checking for latent types
static type checking for manifest types
python also has latent types
java, ml, c and c++ have more like manifest types
static scoping, can look at each identifier and look at who did it, so scheme is like java, python, c, ml, what language is scheme unlike? some have dynamic scoping , fi you look at identifier, you might not know where declaration might be, you have to run the program to find out how scoping works, one example is lisp. in lisp if you'er trying to look up the value of an identifer, you look at the caller's identifiers
contrast to this, to static scoping which is used in most languages -- static scoping if we have nested functions works as follows: look at current function, if you dont find it, you loo in definer's, then in definer's definer, etc. static scoping means you follow the static chain,where every frame points at its definer's frame
with dynamic scoping, follow dynamic frame, find in caller's frame, etc
both make sense and when they decided to define lisp, they decided to do dynamic scopin because its easier and cheaper. because you dont need to maintain static frames. static scoping has higher probability and reliability
another advantage of static scoping is efficiency
if you know that the variable is in your definer's function, its cheap to find the value of a statically scoped variable
w/ dynamica scoping, everytime you look up a variable not in this function,you enter a loop and taht loop might take a while
so dynamic coping is less efficient and less reliable

if your caller set path in that environment, you inherit that
the shell environment also uses dnamic scoping
advantage: mmeans you can set a variable in your progarm and adjust behaviro of everyhitng you run wtihout that software knowing or caring
so you get extra flexibility with dynamic scoping
not something you can easily do wtih static scoping
lets you reach into code in ways you couldn't do before
w/ a global variable, you declare it, someone incldues it, and so that they can see it, then they statically know the variable exists
we could have modified the shell to use static scoping, and then everyone to include PATH, but its a hassle
we wnat it to be simple and flexible so we do dynamic scoping

```
(define y 12)
(define f(lambda(x) (+x y)))
(define g( lambda(y) (f 17))))
(g 6)
```

when g runs, the y in second line we talk about is the y that's 12 in first line in scheme
if we did this in lisp, then this y in second line is the y in the third line
lambda is a long version

scheme uses call by value, which is almost like all the languages we've seen before. one language that doesn't is prolog
evaluate all the arguments to teh functions firs tand hten take copies of those values and then the function can play w/ those values
scheme folks want you to know they dont use the fancy ones we'll talk about later there's other calling functions
scheme has a wide variety of built in object types, including a type some language don't have, called procedures
in scheme, procedures are first class values, some called functions, and we dont treat them specially in scheeme, theyre just objects like any other objects
no special syntax for them the idea is that you shouldn't think of procedures as being in a separate namespace, theyre just objects and they're values
in particular, theres a variety of procedures, include continuations these are special procedures that are kinda like cut, reach in and altar flow of control. that's what continuations do in scheme, they let you change how scheme interpreter behaves. you can call them whenever, and start using scissors on interpreter.
scheme has a very simple syntax, has a property that's very useful: has a straightforward

representation as data
this is one of the key features of the language, easy to write scheme programs that generat
other scheme programs
style of generation is different from higher order programming in ML
you're writing a higher order function w/ a different feel than ML
w/ scheme, you can just write
say you have this
(let a (x y)(cons x y))
we can say here's a piece of data
can do this w/ java or c++
key advantage of lisp and scheme over others, backgbone of AI. you want your programs t
be smarter, so you write programs that generate programs
your programs mutate, they cahnge themselves, which is easy to do in lisp and scheme
when lisp was first designed, they couldn't decide on the syntax and they were in a rush tc
get it out
so simple syntax w/ parentheses
in scheme, tail recursion optimization is required
constraint on the implementation, implementation is required to aviod overflowing the stac
as long asyour code is tail recursive
tail recursion optimization says if the last thing your function does is calls anotehr funtion,
f calls g, and it returns whatever g returns, then the compiler is reqired to optimize the call
such a way that we dont grow the stack
scheme has high level arithmetic, they try to get at least integer arithmetic and rational to
work correctly
dont have to worry about integer overflow
no integer overflow
if you want to be computing w/ fractions, you can do so without having any rounding error
get exact integer artihemtic
so 1/3 is a number, equals 1/3 EXACTLY, no rounding error
multiple precision arithmetic, uses strings of smaller numbers instead of one large string
suppose i keep getting bigger and bigger integers
say i take a big number and square it
# of words needed to represent is gona double, then you run out of memory
integer overflow in scheme is the same thing as memory exhaustion
so do heavy arithmetic in C but represent in Sceheme?
Scheme does have floating point, adn it has complex

syntax of scheme:

syntax of scheme:

letters and identifiers in scheme

'a means yield the symbol of a without the value of a

quasi quoting

it works like quoting, except youre allowed to stop quoting in the middle of the expression and start evalutaitng

`(+,x y)

what this means is please return a list that's three elements

evaluat e the sub expression

define and lambda isn't in the syntax? the first element is an identifier, that identiferi is a keyword

complete low level sytnax of scheme,

scheme keywords:

define id E

you can say i'd like to define a function called caaar

(define caaar(lambda (x)(car(car(car(x)))))

(lambda(x y) E)

evaluates those expression w/ variables bond

this lambda expression doesn't evaluate E, but just remmbers ot be evaluated later

special case:

(lambda () E) in which case this is a function, just calling E and returning E. called a thunk

(f) wiht no arguments. if f is a thunk, then we get the value that the thunk returns

these are not the only forms

you can define a lambda expression in which the first thing is not paren, but an identifer lik

(lambda x E)

this is a procedure that's very ....accepts any number of arguments, passed as a list into x

so when this function is called, x is bound to a list of values these values are the values tha

were passed to this function

there's a fancier flavor of this:

(lambda x y, z) E) x is bound to first arg, y to second arg, z is bound to a list of 3rd, 4th, etc

z might be empt if you call function w/ 2 arguments

example of how you might want to use this is if you want to define a printf function to be lambda and it takes a format string followed by any number of args and then it goes and does its thing

(define printf (lambda (format, args)))

printf takes one argument and the other arguments
proper list -- has empty list at the end
improper list -- follow that chain and you end up with something not nil

(define list (lambda x x))
basic data structure in lisp and scheme is pairs not lists
when we say we have a list, internally what we have is a couple of paris
since we wnat to be able to do pairs in general, want to be able to have a syntax for pairs t
build, we want to have a syntax for this:
in scheem, every value is an object, all represented by pointers to actual data. you only see
the pointers not the actual data
(define(list.x)x) is syntactic sugar for (define list(lambda x x))

(define (ngargs, as)) if i called (nargs 5 7 () #f) should be 4
first we check to see if as is 0
(if (null? as)
(if I T E)
anything other than #f counts as true
ther'es a fancier special form for the keyword: (cond I T) (I2 T2) /first I that returns true, it'
go into T
can always write cond in terms of if

(and E1..En) kind of operators like the && in C and Java
the minute it finds one that's false, it sotps evaluating and returns #f
wlaks through expressions right to left and when it finds a false one, then stops evaluating
and returns #f
if all of them evaluate to true, then it returns the value of En (the last one
conditional and
the difference is that in C, && always returns a boolean but here it can return false or
antyhing
of (E1....En) returns value of first true Ei. the fact that they return arbitrary expressions
instead of boolean, could be useful

((LET X1)(Y E2)) (+ (*2X)Y))
we dont really need let
its kind of like cond, but not like if
same as saying ((lambda (x y) (+ (* 2 x) y)) E1 E2)

when you do let, the order is the same as the order in text, so its easier
read code the way its executed, which is why almost all scheme programmmers use let instead of lambda
because of this equivalence, the scope of x extends

named let -- syntax looks like this: (let f (()()) unlike order let, in a named let, f is bound to function, and that function is one has this as a body and formal aramters