Nathan Schnitzer
Project: Exception Handling
CMSC430

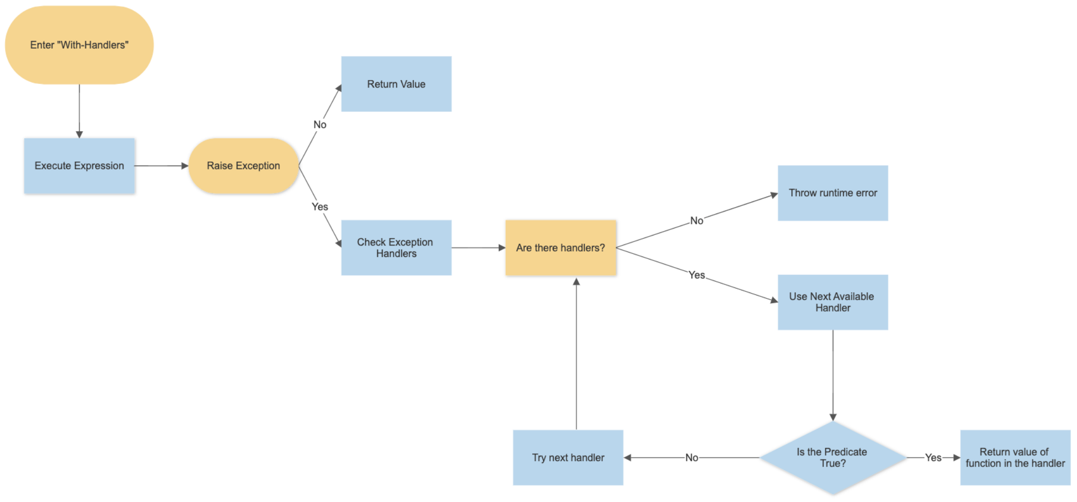
Design Philosophy

I think going right to the AST and parser immediately and beginning to implement how the structure of how with-handlers and raise will be defined is pretty stupid and short-sighted. I would almost guarantee that my first instinct of defining the structure from the project description would lead to either me having to redo the AST and parser later on to avoid a major workaround or lead to me having to do some pretty gross workarounds in the compiler. I absolutely do not have the time for that this semester, so I decided to take the less impulsive, smarter approach. I decided to get a rough approximation of how I wanted my compiler to implement an exception handler and then form my design to meet this rough control flow.

Right off the bat, I had a pretty good vision of how I wanted the assembly of with-handlers to function and how the control flow would… well flow. I knew that I wanted the exception handlers to live in their own space of memory where exception handlers could be treated like a stack. This way we could easily add and remove nested exception handlers while easily being able to access handlers defined earlier in the program. It would also be easy to tell when you ran out of exception handlers and when to throw a runtime error. I also wanted to treat each with-handlers instance as their own container or even black box function of sorts. A with-handlers instance would have a list of exception handlers and an expression to execute. The with-handlers instance will return either a valid value or it will raise an exception. With-handlers will return a valid value when one of the following occurs:

1. The expression runs without raising an exception
2. An exception is raised during the execution of the expression, but one of the exception handlers handled the exception and returned a valid value.

An exception will only be raised by an with-handlers statement if an exception is raised by the expression which the handlers do not satisfy. After a with-handlers instance finishes executing, the exception handlers associated with that instance will be no longer accessible (aka stack/exception pointer moves).



Once I had this rough approximation for what the control flow was going to look like, I knew that I had enough information to begin implementing this in the AST and parser.

AST and Parser Development

In all honesty, the hardest part for me was deciding how to define the structures in the AST and how to parse them. At first I thought it would be easy since I didn't think that the predicates could be user defined. But once it was confirmed that user defined predicates were allowed, it made it a lot more complicated. I thought that I was going to have to set up a bunch of different control flow structures for each type of predicate that could be defined (existing predicates in the language, calls to user defined functions, and lambda functions). And then, on top of that, I had to determine if something even was a predicate and how my program should handle that – is it a compile-time error or runtime error? I opted for the latter. At this point, I had to decide how I was going to get the compiler to know how to handle the different ways that the predicates can be defined. I decided that the best way to handle this was at the level of the parser.

Long Live Lambda!

I think a lot of people forget that lambda calculus is just as much of a computational model as a turing machine is, and because of the universality of lambda calculus, we should be able to simulate anything we do in a turing machine in terms of lambda calculus. So, there is nothing stopping me from implementing this in terms of lambdas. We already implemented lambda functions in loot, so it's not like I will have to implement them from scratch either. I might as well piggy back off all the hard work that Jose did! After all, it is good practice to reuse code. So, we are going to turn everything inside of the handlers into lambda expressions that take 1 argument.

In general: if a lambda expression that takes 2 arguments is passed, it will not compile. If you want to use a primitive or function that takes more than 1 argument (or no arguments), you must put it in a lambda expression that takes 1 argument and has the reference to the other arguments inside the lambda expression.

For the predicate: If the function or primitive you are using takes 1 argument, you should simply pass the name of the function. Additionally, it should be noted that the compiler will not check if the lambda expression returns a boolean (if it is a predicate) and will fail at runtime if it is not.

For the handler: the expression which is executed if the predicate is true must be in the form of a lambda expression which takes 1 argument. It will not compile if it is not. If you want to simply return some value or pass it to another function, you must wrap that in a lambda expression.

So, each error handler will contain 2 lambda expressions that take 1 argument. If the first lambda expression resolves to true, the second lambda expression will execute and return that result. Otherwise, it will move onto the next error handler.

These are examples of **valid** arguments to be passed as handlers:

```
[(lambda (x) (string? x))      (lambda (x) (cons “got” x))]
[string?                       (lambda (x) (cons “got” x))]
[exPred?                       (lambda (x) x)]  (assuming exPred? is defined in scope)
[(lambda (x) (eq? x 4))        (lambda (x) #t)]
```

These are examples of **invalid** arguments to be passed as handlers:

```
[(eq? 4)                       (lambda (x) x)]
[(lambda (x y) (string? x))    (lambda (x) (cons “got” x))]
[(lambda () #f)                (lambda (x) 0)]
[empty?                        (lambda (x y) (cons x y))]
[(lambda (x) (empty? x)        (lambda () ‘())]
[(lambda (x) (empty? x)        ‘()]
```

The resulting AST of with-handlers looks like the following: An Execution Manager (ExMgr) that contains a list of pairs of lambda expressions, aka handlers, and contains an expression.

Interpreter Implementation

From there it was pretty straightforward to implement this into the interpreter. I implemented it similar to how we implemented user defined functions – there would be a list of handlers that would be passed into the environment of the expression being executed. So, when a raise expression is called, it would iterate through the handlers, applying the first function to see if the predicate is true and then returning the value of the other function if it is. Otherwise it returns ‘err and the other handlers are checked. If there are no more handlers, an error is thrown. There are some more intricacies but I am already on page 3 and I haven't even talked about the compiler implementation yet, so you can find some of those goodies by reading the code.

Compiler Implementation

The compiler was also pretty straightforward to implement as I was just treating the functions in the handlers as lambda expressions.

The only difference is that I needed to keep separate references of the closures in another area of memory. I decided to create a whole other area of memory just for the exception handler closures, so I modified the runtime by allocating a new area of memory and then passing the reference of that to entry. The pointer to this area of memory will be stored in register r12. And a counter of the currently scoped exception handlers is stored in r14. So, there is a function called compile-hdlr-lam which compiles the lambdas inside the handlers. The only difference between this and the normal compile-lam is that it puts the pointer to the closure on the stack at r12. If a handler's predicate and function are successfully put on the r12 stack, the r14 register is incremented. The code for handling raise is very similar to compile-app-non-tail. In a nutshell it is the same idea except that it has a loop that goes through the available handlers. Inside that loop, the predicate is executed. If the result is true, the value is passed to the function and that is returned. If the result is false, it moves on to the next handler. If it's neither true or false, the program throws a runtime error.

Pretty simple. All I have to do is make sure that I properly set up the stack before I call the functions in the handler. Should be a breeze, right?

Fuck Lambda!

I am writing this part 2 days after running into a brick wall. I should have kept my mouth shut. I had a really difficult time fixing a bug where I could not reference anything outside of the handler, and I couldn't figure out why the program seemed to want an extra return address in the stack. It turns out that I was simply giving a reference to the lambda definition and not the closure. And then once I realized that, it took me a little while more to figure out that I wasn't using the type-proc bit mask in the right place or at all sometimes. After using gef and stepping my way through the execution for a few hours, I was finally able to get it right. Hallelujah!

Thank you guys for a great semester. I really enjoyed the projects and I think that I learned a lot from this class. Also, I like racket a lot more than I would have thought I would at the beginning of the semester.