JCaml: An Extension of Loot

By Jaraad Kamal.

Overview

This is a programming language that extends a stripped down lisp like language that is very similar to racket. This language constitutes a CMSC 430 final project. It extends from a language called `loot`. The main added feature is the implementation of `try-catch` for errors.

Usage

Creating an Error

When creating an error use the `error` function. This function expects a `string` for the parameter. Variables can be used to store errors, an error is not propagated until the `raise` function is called.

```
(error "message")
```

Getting an Error Message

All errors have a message. They can be retrieved with the `get-message` function. This function will take an `error` and return the `string` message associated with it.

```
(get-message e)
```

Checking type

The `error?` function will check if a given value is of the error type.

```
(error? e)
```

Raising an Error

When raising an error use the `raise` function. This function expects an `error` for the parameter

```
(raise e)
```

Try-Catch

When using `try-catch` the first parameter constitutes the code that could potentially cause an error. The second parameter constitutes the variable name given to any caught errors. The third paremeter constitutes the code that will be executed in the event the first block raised an error. All errors (apart from parsing errors) can be caught with the `try-catch` function.

```
(try-catch (raise (error "message")) err (get-message err))
```

Standard Error Messages

```
"ERROR: primitive <1/2/3> error"      # for invalid types
"ERROR: make-vector"                  # make-vector gets negative length
"ERROR: vector-ref"                   # vector-ref gets out of bounds index
"ERROR: make-string"                  # negative length
"ERROR: string-ref"                   # out of bounds
"ERROR: vector-set"                   # invalid index
"ERROR: lookup error"                 # variable or function does not exist
"ERROR: error: need string"           # error value can only be made with a string
"ERROR: raise: type error"            # raise can only be called with a type error
"ERROR: apply: not a procedure"       # trying to apply a function on a non-
function.
```

Implementation

An error is broken up into two different pointer types. One is an `Error-v`. This is a glorified string pointer and is what the programmer will interact with when using `get-message` or `raise`. The other type is `Error` this is used only inside the assembly and is not accessible to the programmer. It represents an error that was thrown and an indication to propagate the error. This is also a glorified string pointer.

ast.rkt

Addition of the following nodes

- `(struct Error (e))` - This is a node that represents an uncuaght and thrown error.
- `(struct Error-v (e))` - This is a node that represents an error type that can be saved in a variable.
- `(struct Raise (e))` - This is a node that represents raising an error
- `(struct Get-Message (e))` - This is a node that represents getting the message from an error
- `(struct Try-Catch (t x c))` - This is a node represents a try catch block. The first expression `t` is evaluated. If it results in an error then the `c` expression is evaluated. The environment for `c` will have access to a new variable with the id `x`.

types.h / types.rkt

Added new pointer types for an `Error` and `Error-v` type.

- `Error-v` ends in 6 (`error-v-type-tag`)
- `Error` ends in 7 (`error-type-tag`)

print.c

Adding functions to print an `Error-v` and `Error`.

interp-prims.rkt

First, all primitives that recieve improper types will raise an `error`. Some primitives will also throw specific error messages (see above). Addtionally I added a new primtive `error?` which will check if a given value is of type `Error-v`.

interp.rkt

Propagation

When interpetting an expression will check if the value is of type `Error`. If it is then it will pass it up (like the `'err` in previous implementation).

Lookup Errors

Instead of defaulting to a racket match error when a function or varaible is not found in the environment, it will return an `Error` value.

Raise

When interpreting `raise`, the program will check if the nested expression is of value `Error-v`.

Get-Message

For `Get-Message` the program will just retrieve the contained error message.

Try-Catch

The program will frist evaluate the try expression. If the value did not result in something of type `Error`, it will return that. If it did end up being type `Error` will convert the `Error` into an `Error-v`, then it will evlauate the catch expression with an extended environment. Where the id is bound the the caught `Error-v`.

compile-ops.rkt

Added a `compile-error` function that basically replicates the `compile-str` function but uses `error-type-tag` (7).

Overall similar to `interp-prims.rkt`. Modified assertions so that if any of them fail it will compile an `error` using the `compile-error` function and a given default message. Now if assertions pass it will jump past any of the actual primitive code and compile an error.

If the result was already an error it will jump past compile error code and will jump to the end (so `rax` keeps the already saved error)

compile.rkt

Propagation

First the assembly checks if the value in `rax` has the `error-type-tag`. If it does then it will just jump to the end of the expression without any execution. This will "propagate" the error because it remains in `rax`.

Lookup Errors

When a lookup fails it will just compile an `error` with a lookup error message. This pushes lookup errors from compile-time to runtime.

Arity Errors

When a function gets an improper number of arguments and `arrity error` is sent. This is done in a similar way to `iniquity+`. The `r11` register holds the number of arguments passed in. During function execution if this is different from expected the function code is skipped over, the stack is cleaned up (using `r11` for number of words to clear) and an error is compiled in `rax`.

Errors when Evaluating Parameters

This is for the follwing example:

```
#lang racket
(define (f x) x)
(f (add1 #f))
```

The `compile-es` function is modified so that if any of the parameters end up being an error when compiling them, it will clean up the stack and remove all the previous values and put the error into `rax`.

Raise

First will compile the inner expression. If it has the `error-v-type-tag` it will change the tag to `error-type-tag`. This effectively "raises" an error because `rax` has an error in it.

Get-Message

Because `Error-v` is basically a string, it will just remove the `error-v-type-tag` and add a `str-type-tag`.

Try-Catch

The program will compile the first expression. It will then check if the value in `rax` has an `error-type-tag`. If it does not have the tag then it will jump to the end. Otherwise it will change the tag from `error-type-tag` to `error-v-type-tag` and push it to the stack so it can be accessed as a variable.

Other Changes

Modified `unload-bits-asm.rkt` to accomodate the new values. Modified `test-runner.rkt` to include all unit tests for `JCaml`.