Final Project README

Noah Zingler
114489678
CMSC 430

Overview

This compiler takes in a Scheme file and turns it into binary and compares the evaluations between them to see that they still produce the same output. The code goes through multiple steps, utilizing the output language of one step and producing an output language for the next step.

Note that all assignment files were taken from provided references

top-level.rkt makes up the top-level of the compiler by quoting all datums, desugaring defines into letrec*, and adding implicit begin forms explicitly. It also desugars quasiquotes and unquotes as well as match statements. This is the first function that is applied to the input expression.

```
e ::= (define x e)
    | (define (x x ... defaultparam ...) e ...+)
    | (define (x x ... . x) e ...+)
    | (letrec* ([x e] ...) e ...+)
    | (letrec ([x e] ...) e ...+)
    | (let* ([x e] ...) e ...+)
    | (let ([x e] ...) e ...+)
    | (let x ([x e] ...) e ...+)
    | (lambda (x ... defaultparam ...) e ...+)
    | (lambda x e ...+)
    | (lambda (x ...+ . x) e ...+)
    | (dynamic-wind e e e)
    | (guard (x cond-clause ...) e ...+)
    | (raise e)
    | (delay e)
    | (force e)
    | (and e ...)
    | (or e ...)
    | (match e match-clause ...)
    | (cond cond-clause ...)
    | (case e case-clause ...)
    | (if e e e)
    | (when e e ...+)
    | (unless e e ...+)
    | (set! x e)
    | (begin e ...+)
    | (call/cc e)
    | (apply e e)
    | (e e ...)
    | x
    | op
    | (quasiquote qq)
    | (quote dat)
    | nat | string | #t | #f

cond-clause ::= (e) | (e e e ...) | (else e e ...)
case-clause ::= ((dat ...) e e ...) | (else e e ...)
match-clause ::= (pat e e ...) | (else e e ...)
; in all cases, else clauses must come last
dat is a datum satisfying datum? from utils.rkt
x is a variable (satisfies symbol?)
defaultparam ::= (x e)
op is a symbol satisfying prim? from utils.rkt (if not otherwise in scope)
op ::= promise? | null? | cons | car | + | ...  (see utils.rkt)
qq ::= e | dat | (unquote qq) | (unquote e) | (quasiquote qq)
     | (qq ...+) | (qq ...+ . qq)
;; (quasiquote has the same semantics as in Racket)
pat ::= nat | string | #t | #f | (quote dat) | x | (? e pat) | (cons pat pat)
| (quasiquote qqpat)
qqpat ::= e | dat | (unquote qqpat) | (unquote pat) | (quasiquote qq)
        | (qq ...+) | (qq ...+ . qq)
;; (same semantics as Racket match for this subset of patterns)
```

Above is the input language for top-level. Notice that now all lets and lambdas in the output language below only have one body expression.

```
e ::= (letrec* ([x e] ...) e)
    | (letrec ([x e] ...) e)
    | (let* ([x e] ...) e)
    | (let ([x e] ...) e)
    | (let x ([x e] ...) e)
    | (lambda (x ...) e)
    | (lambda x e)
    | (lambda (x ...+ . x) e)
    | (dynamic-wind e e e)
    | (guard (x cond-clause ...) e)
    | (raise e)
    | (delay e)
    | (force e)
    | (and e ...)
    | (or e ...)
    | (cond cond-clause ...)
    | (case e case-clause ...)
    | (if e e e)
    | (when e e)
    | (unless e e)
    | (set! x e)
    | (begin e ...+)
    | (call/cc e)
    | (apply e e)
    | (e e ...)
    | x
    | op
    | (quote dat)

cond-clause ::= (e) | (e e) | (else e)  ; in all test cases
case-clause ::= ((dat ...) e) | (else e)  ; else clauses always come last
dat is a datum satisfying datum? from utils.rkt
x is a variable (satisfies symbol?)
op is a symbol satisfying prim? from utils.rkt (if not otherwise in scope)
op ::= promise? | null? | cons | car | + | ...  (see utils.rkt)
```

desugar.rkt desugars the language drastically by turning exception, dynamic-wind, first-class primitives, guard and raise, delay and force, and many more into a language that includes only the let form, lambdas, conditions, set!, call/cc, and explicit primitive-operation forms.

Input language:


```
e ::= (letrec* ([x e] ...) e)
    | (letrec ([x e] ...) e)
    | (let* ([x e] ...) e)
    | (let ([x e] ...) e)
    | (let x ([x e] ...) e)
    | (lambda (x ...) e)
    | (lambda x e)
    | (lambda (x ...+ . x) e)
    | (dynamic-wind e e e)
    | (guard (x cond-clause ...) e)
    | (raise e)
    | (delay e)
    | (force e)
    | (and e ...)
    | (or e ...)
    | (cond cond-clause ...)
    | (case e case-clause ...)
    | (if e e e)
    | (when e e)
    | (unless e e)
    | (set! x e)
    | (begin e ...+)
    | (call/cc e)
    | (apply e e)
    | (e e ...)
    | x
    | op
    | (quote dat)
```

Output language:

```
e ::= (let ([x e] ...) e)
    | (lambda (x ...) e)
    | (lambda x e)
    | (apply e e)
    | (e e ...)
    | (prim op e ...)
    | (apply-prim op e)
    | (if e e e)
    | (set! x e)
    | (call/cc e)
    | x
    | (quote dat)
```

After desugaring, the expression is passed through simplify-ir, which removes lists and their respective prims and transforms them into lambda functions.

Input language (output from assignment 2; satisfies `ir-exp?`):

```
e ::= (let ([x e] ...) e)
    | (lambda (x ...) e)
    | (lambda x e)
    | (apply e e)
    | (e e ...)
    | (prim op e ...)
    | (apply-prim op e)
    | (if e e e)
    | (set! x e)
    | (call/cc e)
    | x
    | (quote dat)
dat is a datum satisfying datum? from utils.rkt
x is a variable (satisfies symbol?)
op is a symbol satisfying prim? from utils.rkt (if not otherwise in
scope)
op ::= promise? | null? | cons | car | + | ...  (see utils.rkt)
```

Language after `assignment-convert` (and `alphatize`):

```
e ::= (let ([x e] ...) e)
    | (lambda (x ...) e)
    | (lambda x e)
    | (apply e e)
    | (e e ...)
    | (prim op e ...)
    | (apply-prim op e)
    | (if e e e)
    | (call/cc e)
    | x
    | (quote dat)
```

cps.rkt contains three phases, the first of which performs assignment-convert and alphatize. assignment-convert removes set! from the language by boxing all mutable local variables (putting them into vectors). alphatize makes all names unique to a single point binding and removes all shadowing. The input and output languages are shown above

The second phase converts to administrative normal form using a function anf-convert. This gives an explicit order of evaluation to evaluation of any subexpressions by administratively let-binding them to a temporary name. Let forms with multiple right-hand sides are also flattened into multiple let forms. This turns the grammar into atomic expressions (ae), which can be immediately evaluated, and complex expressions (e), which are not trivially known to terminate. The input language is the output of phase one and the output language is the input for phase three.

Language after ANF conversion:

```
e ::= (let ([x e]) e)
    | (apply ae ae)
    | (ae ae ...)
    | (prim op ae ...)
    | (apply-prim op ae)
    | (if ae e e)
    | (call/cc ae)
    | ae
ae ::= (lambda (x ...) e)
     | (lambda x e)
     | x
     | (quote dat)
```

The third phase calls a function cps-convert that will convert to continuation-passing style. This means a continuation is explicitly passed and no function call ever returns, instead the current continuation is invoked at return points. At this phase, call/cc can effectively be removed. We leave prim and apply-prim as special forms that can (and now must) be on the right-hand side of a let because they need not extend the continuation. The input language is shown above and the output language is shown below.

Language after CPS conversion:

```
e ::= (let ([x (apply-prim op ae)]) e)
    | (let ([x (prim op ae ...)]) e)
    | (let ([x (lambda (x ...) e)]) e)
    | (let ([x (lambda x e)]) e)
    | (let ([x (quote dat)]) e)
    | (apply ae ae)
    | (ae ae ...)
    | (if ae e e)
ae ::= (lambda (x ...) e)
     | (lambda x e)
     | x
     | (quote dat)
```


```
e ::= (let ([x (apply-prim op ae)]) e)
    | (let ([x (prim op ae ...)]) e)
    | (let ([x (lambda (x ...) e)]) e)
    | (let ([x (lambda x e)]) e)
    | (let ([x (quote dat)]) e)
    | (apply ae ae)
    | (ae ae ...)
    | (if ae e e)
ae ::= (lambda (x ...) e)
     | (lambda x e)
     | x
     | (quote dat)
```


```
dat is a datum satisfying datum? from utils.rkt
x is a variable (satisfies symbol?)
op is a symbol satisfying prim? from utils.rkt (if not already removed)
```

closure-convert.rkt is the last part of the compiler, and contains two phases in order to convert the language into LLVM IR. The first phase runs output from cps-convert through a function closure-convert which expects inputs that satisfy cps-exp? and should produce outputs that satisfy proc-exp?. This phase removes all lambda abstractions and replaces them with new make-closure and env-ref forms. Remaining atomic expressions other than variable references are lifted to their own let bindings. Finally all function calls must be non-variadic. This can be done by turning all fixed-arity functions into variadic functions and then all variadic functions into unary functions that take an argument list explicitly. The input language is shown above and the output language is shown below.

```
p ::= ((proc (x x ...) e) ...)
e ::= (let ([x (apply-prim op x)]) e)
    | (let ([x (prim op x ...)]) e)
    | (let ([x (make-closure x x ...)]) e)
    | (let ([x (env-ref x nat)]) e)
    | (let ([x (quote dat)]) e)
    | (clo-app x x ...)
    | (if x e e)
dat is a datum satisfying datum? from utils.rkt
x is a variable (satisfies symbol?)
op is a symbol satisfying prim? from utils.rkt (if not already removed)
nat is a natural number satisfying natural? or integer?
```

The second phase converts this first-order procedural language into LLVM IR that can be combined with a runtime written in C to produce a binary. proc-¿llvm is implemented so that it takes a program p and produces a string encoding valid LLVM IR. Applying eval-llvm on this string should return the correct value.

This interpreter will make sure that the runtime, header.cpp, is compiled to a file header.ll before it concatenates the llvm code onto compiled LLVM IR and saves the result to combined.ll. This file is then compiled and run using clang++ combined.ll -o bin; ./bin.

In **tests.rkt**, I kept the functionality from Assignment 4 where the tests could be run to test closure-convert and to-llvm seperately. I modified the make-test function so that it compared the evaluation of either closure-convert or to-llvm with eval-top-level to see if each evaluation was the same.

In **utils.rkt**, I added a function test-proc-¿llvm-error to eval a llvm without evaluating its proc first to check for raised errors. I also added a trigger for a change made in **desugar.rkt** regarding raising an error for dividing by zero.

Documentation for Primitive Operations:

= : (= x y ...+) → boolean?
Returns #t if all of the arguements are numerically equal, #f otherwise. Equality can be between numbers of different percision, such as 1.0 and 1 being equal.

> : (> x y ...+) → boolean?
Returns #t if the arguements in the given order are strictly decreasing, #f otherwise.

< : (< x y ...+) → boolean?
Returns #t if the arguements in the given order are strictly increasing, #f otherwise.

<= : (<= x y ...+) → boolean?
Returns #t if the arguements in the given order are non-decreasing, #f otherwise.

>= : (>= x y ...+) → boolean?
Returns #t if the arguements in the given order are non-increasing, #f otherwise.

+ : (+ x ...) → number?
Adds the integers from left to right. If only one integer is present, the integer adds to 0 (integer is returned). If no arguments are provided, returns 0.

− : (− x y ...) → number?
Subtracts the integers from left to right. If only one integer is present, the integer subtracts from 0 (integer is returned as negative).

∗ : (∗ x y ...) → number?
Multiplies the first integer by each subsequent integer. If only one integer is present, the integer is multiplied by 1.

/ : (/ x y ...) → number?
Divides the first integer by each subsequent integer. If only one integer is present, the integer divides by 1. Can raise

a "division by zero" exception (string).

cons? : (cons? *v*) → boolean?
Returns #t if *v* is a pair, #f otherwise.

null? : (null? *v*) → boolean?
Returns #t if *v* is the empty list, #f otherwise.

cons : (cons *x* *y*) → boolean?
Returns a newly allocated pair whose first element is *x* and second element is *d*.

car : (car *p*) → any
Returns the first element of the pair *p*. In lists, this returns the first element of the list.

cdr : (cdr *p*) → any
Returns the second element of the pair *p*. In lists, this returns the tail of the list.

list : (list *v* ...) → list?
Returns a newly allocated list containing the *v*s as its elements.

first : (first *lst*) → any
Returns the first element of a list. Same as (car *lst*).

second : (second *lst*) → any
Returns the second element of a list.

third : (third *lst*) → any
Returns the third element of a list.

forth : (forth *lst*) → any
Returns the forth element of a list.

fifth : (fifth *lst*) → any
Returns the fifth element of a list.

length : (length *lst*) → integer?
Returns the number of elements in a list.

list-tail : (list-tail *lst pos*) → any
Returns the list after the first *pos* elements of *lst*. If the list has fewer than *pos* element, then an error will be thrown saying that the index is too large for the list. The *lst* argument can also be a chain of pairs.

drop : (drop *lst pos*) → any
Exactly like list-tail.

take : (take *lst pos*) → list?
Returns a list whose elements are the first *pos* elements of *lst*. If *lst* had fewer than *pos* elements, then an error will be thrown saying that the index is too large for the list. The *lst* argument can also be a chain of pairs.

member : (member *v lst*) → (list? *or* #f)
Locates the first element of *lst* that is equal? to *v*. If such an element exists, the tail of *lst* starting with that element is returned. Otherwise, the result is #f.

memv : (memv *v lst*) → (list? or #f)
Like member, but finds an element using eqv?.

map : (map *proc lst* ...+) → list?
Applies *proc* to the elements of the *lsts* from the first elements to the last. The *proc* argument must accept the same number of arguments as the number of supplied *lsts*, and all *lsts* must have the same number of elements. The result is a list containing each result of *proc* in order.

append : (append *lst* ...) → list? —— (append lst ... v) → any
When given all list arguments, the result is a list that contains all of the elements of the given lists in order. The last argument is used directly in the tail of the result.

The last argument need not be a list, in which case the result is an improper list.

foldl : (foldl *proc init lst* ...+) → any
Like map, foldl applies a procedure to the elements of one or more lists. Whereas map combines the return values into a list, foldl combines the return values in an arbitrary way that is determined by *proc*.

If foldl is called with n lists, then *proc* must take n+1 arguments. The extra argument is the combined return values so far. The *proc* is initially invoked with the first item of each list, and the final argument is *init*. In subsequent invocations of *proc*, the last argument is the return value from the previous invocation of *proc*. The input *lsts* are traversed from left to right, and the result of the whole foldl application is the result of the last application of *proc*. If the *lsts* are empty, the result is *init*.

Unlike foldr, foldl processes the *lsts* in constant space (plus the space for each call to *proc*).

foldr : (foldl *proc init lst* ...+) → any
Like foldl, but the lists are traversed from right to left. Unlike foldl, foldr processes the *lsts* in space proportional to the length of *lsts* (plus the space for each call to *proc*).

vector? : (vector *v* ...) → vector?
Returns a newly allocated mutable vector with as many slots as provided *v*s, where the slots are initialized to contain the given *v*s in order.

make-vector : (make-vector *size* [*v*]) → vector?
Returns a mutable vector with *size* slots, where all slots are initialized to contain *v*.

vector-ref : (vector-ref *vec pos*) → any
Returns the element in slot pos of *vec*. The first slot is position 0, and the last slot is one less than (vector-length *vec*).

vector-set! : (vector-set! *vec pos v*) → void?
Updates the slot *post* of *vec* to contain *v*.

vector-length : (vector-length *vec*) → nonnegative-integer?
Returns the length of *vec* (i.e., the number of slots in the vector).

set : (set *e* ...) → set?
Produces a set containing the elements *e*. If no elements are provided, returns an empty set.

set->list : (set->list *st*) → list?
Produces a list containing the elements of *st*.

list->set : (list->set *lst*) → set?
Produces a set containing the elements of *lst*.

set-add : (set-add *st v*) → set?
Produces a set that includes *v* plus all elements of *st*.

set-union : (set-union *st*0 *st* ...) → set?
Produces a set of the same type as *st*0 that includes elements from *st*0 and all of the *st*s.

set-count : (set-count *st*) → nonnegative-integer?
Returns the number of elements in *st*.

set-first : (set-first *st*) → any?
Produces an unspecified element of *st* that is the first in the set. Multiple uses of set-first on st produce the same result.

set-rest : (set-rest *st*) → set?

Produces a set that includes all elements of *st* except (set-first *st*).

set-remove : (set-remove *st* v) → set?
Produces a set that includes all elements of *st* except v.

list? : (list? v) → boolean?
Returns #t if v is a list: either the empty list, or a pair whose second element is a list. This procedure effectively takes constant time due to internal caching (so that any necessary traversals of pairs can in principle count as an extra cost of allocating the pairs).

void? : (void? v) → boolean?
Returns #t if v is the constant #<void>, #f otherwise.

promise? : (promise? v) → boolean?
Returns #t if v is a promise, #f otherwise.

procedure? : (procedure? v) → boolean?
Returns #t if v is a procedure, #f otherwise.

number? : (number? v) → boolean?
Returns #t if v is a number, #f otherwise.

integer? : (integer? v) → boolean?
Returns #t if v is a integer, #f otherwise.

error : (error *sym*) → any
Raises the exception exn:fail, which contains an error string. This form creates a message string by concatenation "error: " with the string form of *sym*.

void : (void v ...) → void?
Returns the constant #¡void¿. Each v argument is ignored.

print : (print *datum* [*out*]) → void?
Prints *datum* to *out*. The rationale for providing print is that display and write both have specific output conventions, and those conventions restrict the ways that an environment can change the behaviour display and wrote procedures. No output conventions should be assumed for print, so that environments are free to modify the actual output generated by print in any way.

display : (display *datum* [*out*]) → void?
Displays *datum* to *out*, similar to write, but usually in such a way that byte- and character-based datatypes are written as raw bytes or characters.

write : (write *datum* [*out*]) → void?
Writes *datum* to *out*, normally in such a way that instances of core datatypes can be read back in.

exit : (exit [v]) → any
Passes v to the current exit handler. If the exit handler does not escape or terminate the thread, #<void> is returned.

eq? : (eq? $v1$ $v2$) → boolean?
Return #t if $v1$ and $v2$ refer to the same object, #f otherwise.

eqv? : (eqv? $v1$ $v2$) → boolean?
Two values are eqv? if and only if they are eq?, unless otherwise specified for a particular datatype.

equal? : (equal? $v1$ $v2$) → boolean?
Two values are equal? if and only if they are eqv?, unless otherwise specified for a particular datatype.

not : (not v) → boolean?
Returns #t if v is #f, #f otherwise.

Run-time Failure checks

Most of the run-time exceptions are caught by the with-handlers part of the racket-compile-eval and racket-proc-eval, where they return a specific error to show what the exception is. For my error tests, I have them add to the score of tests passed if those tests fail with an exception to show that a 100% means that everything is working.

Function has too little or too many arguments
I caught the wrong number of arguments exception in the with-handlers part of the racket-compile-eval and racket-proc-eval functions in utils.rkt. If the test contained a prim with too few arguments or too many arguments, a exn:fail:contract:arity? exception would be caught and the string "Arguement mismatch Exception:" is printed along with the exception. An error is also thrown which shows that the test fails.

The tests that I used to check for too few or too many arguments were error-too-few-args-simple.scm, error-too-few-args.scm, error-too-many-args-simple.scm, and error-too-many-args.scm.

Non-function value is applied
I did not catch this exception, but I wrote error-non-function-applied-1.scm as a test to trip the caught exception and it successfuly fails.

Use of not-yet initialized letrec or letrec* variable
I caught the use of not-yet initialized variables in the with-handlers part of the racket-compile-eval and racket-proc-eval functions in utils.rkt. If the test contained a prim with too few arguments or too many arguments, a exn:fail:contract:variable exception would be caught and the string "Use of not-yet initialize variable Exception:" is printed along with the exception. An error is also thrown which shows that the test fails.

I wanted to implement a solution to check this at the binary level by utilizing assignment-convert's mutated-variables function in a similar way to where while iterating over th variables it could check if the variable has been initialized yet in the scope.

The tests I used to check for not-yet initialized variables were error-not-yet-init-1.scm and error-not-yet-init-2.scm.

Division by zero
I caught the division by zero exception in the with-handlers part of the racket-compile-eval and racket-proc-eval function of **utils.rkt**. If a test contains prim divide where a zero is the first and only argument or present in the arguments after the first one, a exn:fail:contract:divide-by-zero? exception would be caught and the string "Divide by 0 Exception:" is printed along with the exception. An error is also thrown which shows that the test fails.

In **desugar.rkt**, I tried to pass through a prim halt if a divide by 0 case occured so the string "Divide by 0 error" would be the output of the binary after llvm-eval. Unfortunately I only got a return of #<eof> when the code was evaluated with the raise in it.

The tests that I used to check for division by zero were error-divide-by-0-simple.scm and error-divide-by-0.scm.

I did not complete either Part 3 or Part 4 of the project. I was intending to try and implement strings and characters but ran out of time.

I, Noah Zingler, pledge on my honor that I have not given or received any unauthorized assistance on this assignment.