Final Project Documentation

Version 6.10

December 19, 2017

High-Level Description and Documentation for Final Compilers Project

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

1 Documenting Primitive Functions

`(= a b ...)` → int

Returns true if all arguments are equal.

`(> a b)` → int

Returns true if a is greater than b.

`(< a b)` → bool

Returns true if a is less than b.

`(<= a b)` → bool

Returns true if a is less than or equal to b.

`(>= a b)` → bool

Returns true if a is greater than b.

`(+ a b ...)` → int

Returns the sum of all arguments.

`(- a b)` → int

Returns result of subtracting b from a.

`(* a b ...)` → int

Returns product of all numbers.

`(/ a b)` → int

Divides first integer by second. Raises Divide by Zero error if second argument is 0.

`(null? a)` → bool

Returns true if a is empty.

`(cons a b)` → pair

Returns a new pair where the first element is a and the second element is b.

`(car a)` → any

Returns the first element of the pair.

`(cdr a)` → any

Returns the second element of the pair p.

`(list a)` → list

Returns a list with the a as an element in the list.

`(length lst)` → int

Returns the number of elements in the list provided.

`(list-tail lst a)` → list

Returns the list after the first a elements of lst.

`(member a lst)` → list/#f

Checks if a is an element of lst and then returns the tail of the list from a or false if not found.

`(memv v lst)` → list/#f

Checks if a is an element of lst and then returns the tail of the list from a or false if not found using eqv? instead of equals?.

`(map proc lst)` → list

Applies proc to all of the elements of lst. The result is a list with the result of proc applied to each element.

`(append lst ...)` → list

Compiles all elements in all lists supplied in order.

`(foldl proc a l ...+)` → list

foldl applies proc to each element in the list from left to right while containing an accumulator with behavior specified by proc.

`(foldr pro a l ...+)` → list

foldl applies proc to each element in the list from right to left while containing an accumulator with behavior specified by proc.

`(vector? a)` → bool

Returns true if a is a vector.

`(vector v)` → vector

Returns a mutable vector v.

`(make-vector size [v])` → vector

Returns a mutable vector with size slots initialized to v.

`(vector-ref v pos)` → any

Returns the element at index pos from v.

`(vector-set! v pos a)` → #<void>

Sets the element at index pos of vector v to a.

`(vector-length v)` → int

Returns the length of the vector.

`(set v ...)` → set

Creates a set with elements given.

`(set->list s)` → list

Returns a list with the elements from set s.

`(list->set lst)` → set

Creates a set with the elements of lst.

`(list? lst)` → bool

Returns whether lst is a list, pair with list as the subsequent element or an empty list.

`(void? v)` → bool

Returns true if v is constant #<void>.

`(promise? p)` → bool

Returns true if p is a promise.

`(number? n)` → bool

Returns true if n is a number.

`(integer? n)` → bool

Returns true if n is an integer.

`(error s)` → any

Raises an exception that returns a string "error: " with s appended to it.

`(void v ...)` → #<void>

Returns the constant #<void> with all arguments ignored.

`(print p)` → #<void>

Prints p.

`(display d)` → #<void>

Displays datum d.

`(exit e)` → any

passes e to exit handler or returns #<void> if e is not present.

2 Top-level overview of compiler

Scheme input is wrapped in a `begin` and passes through `top-level` from assignment 5.

After passing through `top-level`, all datums become explicitly quoted, `defines` are desugared into `letrec*`, and new bindings are generated for expressions that contain `quasiquotes`, `unquotes`, and `match`.

The `top-level` output is then passed through `desugar` from assignment 2 which converts the input into a small core language including only let forms, lambdas, conditionals, set!, call/cc, and explicit primitive-operation forms.

The resulting output is fed to `assignment-convert` and `alphatize` which replaces `set!` with `make-vector`, `vector-set!`, and `vector-ref` prims.

The grammar is then partitioned by into complex expressions and atomic expressions that are able to be immediately evaluated by ANF conversion.

CPS conversion follows and the current continuation is invoked at return points instead of allowing function calls to return. As a result, `call/cc` is removed and `prims` and `apply-prims` are now let-bound.

`closure-convert` conducts 2 passes on the output to lift the remaining variable references to let-bindings and turns fixed-arity functions into unary functions that take an argument list. One more pass transforms the language into a list of first-order procedures.

Finally, the grammar is turned into a string encoding LLVM IR. The header.cpp file is also transformed into LLVM IR and both are concatenated in a file combined.ll which is compiled with `clang++` to produce a valid binary.

3 Run-time Errors

The tests.rkt have been modified so error messages from the binary are returned. This is how I created tests that fail for certain runtime errors.

Integer Overflow for Addition, Multiplication, and Subtraction

Run integer_overflow_1 and integer_overflow_2. Header.cpp is modified such that the prim functions for +, - and * detect an integer overflow and print "Integer Overflow Error" before halting.

Integer Underflow for Addition, Multiplication, and Subtraction

Run integer_underflow_1 and integer_underflow_2. Header.cpp is modified such that the prim functions for +, - and * detect an integer overflow and print "Integer Overflow Error" before halting.

Division by Zero

Run divide_zero_1 and divide_zero_2. Header.cpp is modified such that the prim functions for / checks whether the second argument is 0. If so, it prints "Divide By Zero Error" before halting.