Project

The final assessment for this course consists of an individually completed project.

Final deliverables are due by the end of the time schedule for the class's final exam as set by the registrar.

There are several projects to choose from, described below.

Compared to assignments, the project is more open-ended. You will need to select from a project description below and then select which language you'd like to target with your project. As starter code, you can use the source code of any of the course languages. How you implement your project is up to you. It may involve changes to all aspects of the language implementation: the parser, the compiler, and the run-time system (however, we do not require an interpreter implementation). No tests are provided, so we recommend you write your own and suggest focusing on tests *before* trying to implement these features.

In addition to the source code for your project, you must write a 2-page document in PDF format, which gives a summary of your work and describes how your project is implemented.

1 Multiple return values

Racket, Scheme, and even x86 support returning more than one value from a function call. Implement Racket's `let-values` and `values` forms to add multiple return values.

You may choose to implement this feature for any language that is Iniquity or later for a maximum 95% of the possible points. For 100% you'll need to implement the feature for Loot or later.

Here are the key features that need to be added:

- `(values e1 ... en)` will evaluate `e1` through `en` and then "return" all of their values.

- `(let-values ([(x1 ... xn) e]) e0)` will evaluate `e`, which is expected to be an expression that produces *n* values, which are bound to `x1` through `xn` in the body expression `e0`.

Here are some examples to help illustrate:

Examples

```
> (let-values ([(x y) (values 1 2)]) (+ x y))
3
> (let-values ([(x) (values 1)]) (add1 x))
2
> (let-values ([() (values)]) 7)
7
> (define (f x)
    (values x (+ x 1) (+ x 2)))
> (let-values ([(x y z) (f 5)])
    (cons x (cons y (cons z '()))))
'(5 6 7)
> (add1 (values 5))
6
> (let ((x (values 5)))
    (add1 x))
6
```


Any time an expression produces a number of values that doesn't match what the surrounding context expects, an error should be signaled.

Examples


```
> (add1 (values 1 2))
result arity mismatch;
 expected number of values not received
  expected: 1
  received: 2
> (let-values ([(x y) 2]) x)
result arity mismatch;
 expected number of values not received
  expected: 2
  received: 1
  in: local-binding form
  arguments...:
   2
```

The top-level expression may produce any number of values and the run-time system should print each of them out, followed by a newline:

Examples

```
> (values 1 2 3)
1
2
3
```

Note there is some symmetry here between function arity checking where we make sure the number of arguments matches the number of parameters of the function being called and the “result arity” checking that is required to implement this feature. This suggests a similar approach to implementing this feature, namely designating a register to communicate the arity of the result, which should be checked by the surrounding context.

You will also need to design an alternative mechanism for communicating return values. Using a single register (`'rax`) works when every expression produces a single result, but now expressions may produce an arbitrary number of results and using registers will no longer suffice. (Although you may want to continue to use `'rax` for the common case of a single result.) The solution for this problem with function parameters was to use the stack and a similar approach can work for results too.

1.1 Returning multiple values to the run-time system or `asm-interp`

In implementing `values`, there are two design decisions you have to make:

1. How are values going to be represented during the execution of a program?
2. How are values going to be communicated back to the run-time system and/or asm-interp when the program completes?

The answers to (1) and (2) don’t necessarily have to be the same.

Note that you can go a long way working on (1) without making any changes to the run-time system or `unload-bits-asm.rkt` (which is how the result of `asm-interp` is converted back to a Racket value). You can basically punt on (2) and work on (1) by writing tests that use multiple values within a computation, but ultimately return a single value, e.g. `(let-values ([(x y) (values 1 2)] (cons x y)))`.

As for (2), here is a suggestion that you are free to adopt, although you can implement (2) however you’d like so long as when running an executable that returns multiple values it prints the results in a way consistent with how Racket prints and that if using `asm-interp`, your version of `unload/free` produces multiple values whenever the program does.

You can return a vector of results at the end of `entry`. This means after the instructions for the program, whatever values are produced are converted from the internal representation of values (i.e., your design for (1)) to a vector and the address (untagged) is put into `rax` to be returned to the run-time system and/or `asm-interp`.

Now both the run-time system and `unload-bits-asm.rkt` need to be updated to deal with this change in representation for the result.

In `main.c`, the part that gets the result and prints it:

```
val_t result = entry(heap);
print_result(result);
if (val_typeof(result) != T_VOID)
  putchar('\n');
```

can be changed to getting the vector and printing each element:

```
val_vect_t *result = entry(heap);
for (int i = 0; i < result->len; ++i) {
  print_result(result->elems[i]);
  if (val_typeof(result->elems[i]) != T_VOID)
    putchar('\n');
}
```

You'll also need to update the signature of `entry` in `runtime.h` to:

```
val_vect_t* entry();
```

You'll also need to make a similar change to `unload/free` in `unload-bits-asm.rkt`, which plays the role of the run-time system when writing tests that use `asm-interp`.

Instead of:

```
; Answer* -> Answer
(define (unload/free a)
  (match a
    ['err 'err]
    [(cons h v) (begin0 (unload-value v)
                        (free h))]))
```


You'll want:

```
; Answer* -> Answer
(define (unload/free a)
  (match a
    ['err 'err]
    [(cons h vs) (begin0 (unload-values vs)
                         (free h))]))

(define (unload-values vs)
  (let ((vec (unload-value (bitwise-xor vs type-vect))))
    (apply values (vector->list vec))))
```


Let's say you make these changes to the run-time system and `unload/free` before you make any changes to the compiler and now you want to adapt the compiler to work with the new set up (before trying to do anything with `values`). You can add the following at the end of `entry`, just before the `(Ret)`:

```
; Create and return unary vector holding the result
(Mov r8 1)
(Mov (Offset rbx 0) r8)  ; write size of vector, 1
(Mov (Offset rbx 8) rax) ; write rax as single element of vector
(Mov rax rbx)            ; return the pointer to the vector
```

In order to return more values, you'd construct a larger vector.

2 Exceptions and exception handling

Exceptions and exception handling mechanisms are widely used in modern programming languages. Implement Racket's `raise` and `with-handlers` forms to add exception handling.

You may choose to implement this feature for any language that is Iniquity or later for a maximum 95% of the possible points. For 100% you'll need to implement the feature for Loot or later.

Here are the key features that need to be added:

- `(raise e)` will evaluate *e* and then "raise" the value, side-stepping the usual flow of control and instead jump to the most recently installed exception handler.
- `(with-handlers ([p1 f1] ...) e)` will install a new exception handler during the evaluation of *e*. If *e* raises an exception that is not caught, the predicates should be applied to the raised value until finding the first *pi* that returns true, at which point the corresponding function *fi* is called with the raised value and the result of that application is the result of the entire `with-handlers` expression. If *e* does not raise an error, its value is the value of the `with-handler` expression.

Here are some examples to help illustrate:

Examples


```
> (with-handlers ([string? (λ (s) (cons "got" s))])
    (raise "a string!"))
'("got" . "a string!")
> (with-handlers ([string? (λ (s) (cons "got" s))]
                  [number? (λ (n) (+ n n))])
    (raise 10))
20
> (with-handlers ([string? (λ (s) (cons "got" s))]
                  [number? (λ (n) (+ n n))])
    (+ (raise 10) 30))
20
> (let ((f (λ (x) (raise 10))))
    (with-handlers ([string? (λ (s) (cons "got" s))]
                    [number? (λ (n) (+ n n))])
      (+ (f 10) 30)))
20
> (with-handlers ([string? (λ (s) (cons "got" s))]
                  [number? (λ (n) (+ n n))])
    'nothing-bad-happens)
'nothing-bad-happens
> (with-handlers ([symbol? (λ (s) (cons 'reraised s))])
    (with-handlers ([string? (λ (s) (cons "got" s))]
                    [number? (λ (n) (+ n n))])
      (raise 'not-handled-by-inner-handler)))
'(reraised . not-handled-by-inner-handler)
```

Notice that when a value is raised, the enclosing context is discard. In the third example, the surrounding `(+ [] 30)` part is ignored and instead the raised value `10` is given the exception handler predicates, selecting the appropriate handler.

Thinking about the implementation, what this means is that a portion of the stack needs to be discarded, namely the area between the current top of the stack and the stack that was in place when the `with-handlers` expression was evaluated.

This suggestions that a `with-handlers` expression should stash away the current value of `'rsp`. When a `raise` happens, it grabs the stashed away value and installs it as the current value of `'rsp`, effectively rolling back the stack to its state at the point the exception handler was installed. It should then jump to code that will carry out the applying of the predicates and right-hand-side functions.

Since `with-handlers` can be nested, you will need to maintain an arbitrarily large collection of exception handlers, each of which has a pointer into the stack and a label for the code to handle the exception. This collection should operate like a stack: each `with-handlers` expression adds a new handler to the handler stack. If the body expression returns normally, the top-most handler should be removed. When a raise happens, the top-most handler is popped and used.

3 Basic syntax macros

Many languages (such as C and Racket) provide a feature called *syntax macros* that make it possible for programmers to implement their own syntactic sugar. They are conceptually similar to functions, except that their input and output are syntax and they are evaluated during compilation rather than at run-time.

Implement a simplified macro system for Iniquity or later according to the following specifications.

3.1 Adding macro definitions

You will add macro definitions to the program before all the function definitions. It will look like this:

```
(define-macro (m0 mx0 ...) s0)
(define-macro (m1 mx1 ...) s1)
...
(define (f0 fx0 ...) e0)
(define (f1 fx1 ...) e1)
...
e
```

We do not need to extend the syntax of expressions, except to recognize the invocation of a macro form (which look the same as function applications). An example of a concrete program might be:

```
(define-macro (and a b)
  (if a (if b b #f) #f))

(add1 (and #t 42))
```

This program defines the macro `and`, which is simply two nested `if` expressions. This `and` follows the same conventions as Racket's `and`, which returns `#f` if any of the arguments are `#f` and returns the last argument otherwise.

3.2 Implementing the basic macro functionality

Your macros should be evaluated during or after parsing and before the resulting AST is passed to `Prog`. We recommend you make changes to the `parse` function in `parse.rkt` to correctly return a `Prog` only after the macros have been evaluated.

In this simplified macro system, you only need to support macros that act like functions. These are defined with this syntax:

```
(define-macro (macro-name params ...)
  replacement)
```

That is, users will use the `define-macro` form with a name and a list of zero or more parameters to define the number of arguments to the macro, and then a program fragment to use when replacing occurrences of the macro.

In the previous example, the result should be the same as if the user had just written this program in the first place:


```
(add1 (if #t (if 42 42 #f) #f))
```


Defined macros should be usable within functions and the main program expression, but they do not have to be usable within macros themselves.

4 Implementing iterators

Racket provides some specialized iteration forms that accumulate the results of the iteration into collections. You will implement four of these forms (though we expect much of the code will be able to be shared, once you figure it out).

4.1 Basic list-producing iterator

The simplest of these iteration forms is `for/list`. Here is a very simple example:

Examples

```
> (for/list ([x (cons 1 (cons 2 (cons 3 '())))])
    (add1 x))
'(2 3 4)
```

This expression will produce the same result as `(list 2 3 4)`. It iterates over the list `(cons 1 (cons 2 (cons 3 '())))` and assigns each value to `x` in turn, and then builds a list for each of those values by doing `(add1 x)`.

However, `for/list` is even more interesting because it allows for iterating over multiple sequences at a time:

Examples

```
> (for/list ([x (cons 1 (cons 2 (cons 3 '())))]
             [y (cons #t (cons #f '()))])
    (cons x y))
'((1 . #t) (2 . #f))
```

This will produce `(list (cons 1 #t) (cons 2 #f))`. Note that the `3` is never iterated over because the `y` sequence is too short. The `for/list` form will only iterate as many times as the shortest sequence it's given.

Although `for/list` *produces* a list, it doesn't have to only take lists as its iteration sequences: strings and vectors are also valid sequences in Racket. Let's write a more interesting example:

Examples

```
> (define (get-three s)
    (for/list ([c s]
               [n (cons 1 (cons 2 (cons 3 '())))])
      (cons c n)))
```


We can call this function with each of the different kinds of iterators:

Examples

```
> (get-three (cons 1 (cons 2 (cons 3 '()))))
'((1 . 1) (2 . 2) (3 . 3))
> (get-three "hello")
'((#\h . 1) (#\e . 2) (#\l . 3))
> (get-three (make-vector 2 #t))
'((#t . 1) (#t . 2))
```


Here are the steps we recommend you follow:

- Add new AST node(s) to the compiler.
- Update the parser. We recommend looking at how `let` bindings are parsed.
- Extend the compiler to handle `for/list` with only a single iteration sequence. Get this working first so you can be sure you understand how it works.
- Extend the compiler to handle `for/list` with an arbitrary number of iteration sequences.

4.1.1 Parsing hint

If you like, you can copy the below AST node definition into `ast.rkt`:

```
(struct ForList (xs its e) #:prefab)
```

And then you can add the following to `parse.rkt`:

```
(define (parse-e s)
  (match s
    ; ...
    [(list 'for/list (list (list (? symbol? xs) its) ...) e)
     (ForList xs (map parse-e its) (parse-e e))]
    ; ...))
```

Note that there are many ways to handle this part of the project, and if you don't like this way you are more than welcome to use your own parsing code.

4.2 Basic vector-producing iterator

Racket also provides `for/vector`, which works identically to `for/list` except that it returns a vector:

Examples

```
> (for/vector ([x (cons 1 (cons 2 (cons 3 '())))]
               [y (cons #t (cons #f '()))])
    (cons x y))
'#((1 . #t) (2 . #f))
```


Implement this form as well.

4.3 Nesting iterators

In addition to `for/list` and `for/vector`, Racket provides `for*/list` and `for*/vector`. These work very similarly to the basic versions, except that all of the iteration is nested rather than sequential. Compare:

Examples

```
> (for/list ([x (cons 1 (cons 2 (cons 3 '())))]
             [y (cons #t (cons #f '()))])
    (cons x y))
'((1 . #t) (2 . #f))
> (for*/list ([x (cons 1 (cons 2 (cons 3 '())))]
              [y (cons #t (cons #f '()))])
    (cons x y))
'((1 . #t)
  (1 . #f)
  (2 . #t)
  (2 . #f)
  (3 . #t)
  (3 . #f))
```

The first expression uses `for/list`, so it iterates over the sequences at the same time. The second expression uses `for*/list`, so it iterates over the sequences in a nested manner.

Implement `for*/list` and `for*/vector`.

5 Design your own

You may also design your own project, however, you will need to submit a one-page write-up that documents what you plan to do and how you will evaluate whether it is successful. You must submit this document and have it approved by the instructor by April 19.

6 Submitting

Submissions should be made on Gradescope.

Your submission should be a zip file containing the following contents:

```
info.rkt
summary.pdf
<lang>/
```

where `<lang>` corresponds to the language you have chosen to implement for your project, e.g. `iniquity`, `loot`, etc.

The `info.rkt` should contain the following information:

```
#lang info
(define project '<project>)
(define language '<lang>)
```


where `<project>` indicates which project you are doing. It should either be `values`, `exceptions`, `garbage-collection`, or `custom`. The `<lang>` should be `iniquity`, `loot`, etc. and should be the same as the directory that contains the implementation.