Exceptions are raised with a raise statement.

```scheme
raise <expr>
```

<expr> must evaluate to a subclass of BaseException or an instance of one.

```scheme
try
  <try suite>
except <exception class> as <name>:
  <except suite>
end
```

The `try` suite is executed first. If, during the course of executing the `try` suite, an exception is raised that is not handled otherwise, and if the class of the exception inherits from `Exception`, then handling a `class ZeroDivisionError`:

```scheme
>>> try:
...     x = 1/0
...     except ZeroDivisionError as e:
...         print('handling a class ZeroDivisionError')
...     >>>
```

The `except suite` is executed, with `name` bound to the exception.

```scheme
>>> (define (make-add-lambda n) `(lambda (d) (+ d ,n)))
>>> (define f (mu (x) (+ x y)))
>>> > (g 3 7)
>>> > (define f (mu (x) (+ x y)))
```

Macros exist in many languages, but are easiest to define correctly in a language like Lisp

A macro is an operation performed on the source code of a program before evaluation.

Macros exist in many languages, but are easiest to define correctly in a language like Lisp. Scheme has a `define-macro` special form that defines a source code transformation.

```scheme
(define-macro (twice expr) (twice (print 2)) (begin (print 2) (print 2))
```

The number of groups is the number of unique values of an expression.

A having clause filters the set of groups that are aggregated by weight/legs, group by weight/legs having count(*)>1.

```sql
SELECT [expression] AS [name], [expression] AS [name], ...;
SELECT [columns] FROM [table] WHERE [condition] ORDER BY [order];
```

String values can be combined to form longer strings.

```sql
sqlite> CREATE TABLE phrase AS SELECT "hello, " || " world";
hello, world
```

Basic string manipulation is built into SQL, but differs from Python.

```sql
sqlite> CREATE TABLE phrase AS SELECT "hello, world" AS s;
```

FROM phrase;

The number of groups is the number of unique values of an expression.

A having clause filters the set of groups that are aggregated by weight/legs, group by weight/legs having count(*)>1.
CS 61A Final Exam Study Guide – Page 2

Scheme programs consist of expressions, which can be:
- Primitive expressions: 2 3 3.0 true + quotient ...
- Combinations: (quotient 10 2) (not true)
  Numbers are self-evaluating; symbols are bound to values.

Call expressions have an operator and 0 or more operands.

A combination that is not a call expression is a special form:
- If expression: (if <predicate> <consequent> <alternative>)
- Binding names: (define <name> <expression>)
- New procedures: (define (<name> <<formal parameters>> <body>))

```scheme
(define pi 3.14)
(define (abs x) (- x))
```

Each combination returns its value.

Lambda expressions evaluate to anonymous procedures.

```scheme
(lambda (<formal parameters>) <body>)
```

In the late 1950s, computer scientists used confusing names.
- cons: Two-argument procedure that creates a pair
- cdr: Procedure that returns the second element of a pair
- nil: The empty list

They also used a non-obvious notation for linked lists.
- A (linked) Scheme list is a pair in which the second element is
  nil or a Scheme list.
- Scheme lists are written as space-separated combinations.
- A dotted list has an arbitrary value for the second element of the
  last pair. Dotted lists may not be well-formed lists.

```scheme
> (define x (cons 1 nil))
> (define a 1)
> (cdr x)
```

Symbols normally refer to values; how do we refer to symbols?

```scheme
> (define (a b) (a b))
> (a b)
```

Quotation is used to refer to symbols directly in Lisp.

```scheme
> (list 'a 'b)
```

Syntactic analysis identifies the hierarchical structure of an expression.

Expressions:
- Base cases: symbols and numbers
- Recursive call: scheme_read sub-expressions and combine them

A Scheme list is written as elements in parentheses:

```scheme
(a b)
```

A basic interpreter has two parts: a parser and an evaluator.

```scheme
calculator.py
scheme_reader.py
```

A Scheme list is written as elements in parentheses:

```scheme
(a b)
```

Each <element> can be a combination or atom (primitive).

The task of parsing a language involves coercing a string representation of an expression to the expression itself.

Parsers must validate that expressions are well-formed.

A Parser takes a sequence of lines and returns an expression.

Each call to scheme_read consumes the input tokens for exactly one expression.

Base case: symbols and numbers

Recursive call: scheme_read sub-expressions and combine them

LambdaProcedure instance [parent=g]

Requires an environment for name lookup

Base cases:
- Primitive values (numbers)
- Look up values bound to symbols

Recursive calls:
- Eval(operator, operands) of call expressions
- Apply(procedure, arguments)
- Eval(sub-expressions) of special forms

To apply a user-defined procedure, create a new frame in which
formal parameters are bound to argument values, whose parent
is the env of the procedure, then evaluate the body of the
procedure in the environment that starts with this new frame.

```scheme
(define (f s) (if (null? s) '3 (cons (car s) (f (cdr s)))))
(f (list 1 2))
```

The structure of the Scheme Interpreter

Create new environment and apply

Apply

Base cases:
- Built-in primitive procedures
- Eval(body) of user-defined procedures

The structure of the Scheme Interpreter

Generates new environment and applies

Apply