Welcome to CS302
Paradigms of Programming

Must Read textbooks:- SICP (pdf available online)

CS302 is a computer science course.

CS302 is a computer science course.

Really?

CS302 is a computer science course.

Really?

- Is it science?

CS302 is a computer science course.

Really?

- Is it science?
- Is it about computers?

CS302 is a computer science course.

Really?

- Is it science?
- Is it about computers?
- Is it fiction?

CS302 is a computer science course.

Really?

- Is it science?
- Is it about computers?
- Is it fiction?
- Is it engineering?

CS302 is a computer science course.

Really?

- Is it science?
- Is it about computers?
- Is it fiction?
- Is it engineering?

This course will aim at formalizing intuitions about process (to be precise 'computational processes'). How should one go about doing things?

A computational process is indeed much like a sorcerer's idea of a spirit. It cannot be seen or touched. It is not composed of matter at all.

This course will aim at formalizing intuitions about process (to be precise 'computational processes'). How should one go about doing things?

A computational process is indeed much like a sorcerer's idea of a spirit. It cannot be seen or touched. It is not composed of matter at all.

However, it is/appears very real. Why?

A computational process, in a correctly working computer, executes programs precisely and accurately.

Well-designed computational systems, like well-designed automobiles or nuclear reactors, are designed in a modular manner, so that the parts can be *constructed, replaced, and debugged* separately.

The modules are ideal modules in the sense that if the behaviour of each of the modules is known, then the behaviour of any combination of such ideal modules can also be accurately predicted, unlike modules of real world objects say for example an amplifier.

A procedure is a pattern of rules which directs a process. We need an appropriate language for describing processes, and we will use for this purpose the programming language Lisp.

We are going to use the programming language Scheme (a dialect of Lisp) to teach you some big ideas in computer science.

The ideas are mostly about *control of complexity* that is, about how to develop a large computer program without being swamped in details.

Scheme has unique features that make it an excellent medium for studying important programming constructs and for relating them to the linguistic features that support them.

The most significant of these features is the fact that Scheme's descriptions of processes, called *procedures*, can themselves be represented and manipulated as data.

There are powerful program-design techniques that rely on the ability to blur the traditional distinction between "passive" data and "active" processes.

Query: "I am right now at CV Raman guest house. How do I reach the class where CS302 is being conducted ?"

Imperative response
Use the outer peripheral road to reach A9 building. Find the road adjacent to the guard post and walk along the academic block road to until you reach the end of the road. Walk 3 flights of stairs of the last building on your right. Find the first room on the ground floor.

vs

Declarative response
Come to A17-1A.

Query: “I am right now at CV Raman guest house. How do I reach the class where CS302 is being conducted ?”

Imperative response
Use the outer peripheral road to reach A9 building. Find the road adjacent to the guard post and walk along the academic block road to until you reach the end of the road. Walk 3 flights of stairs of the last building on your right. Find the first room on the ground floor.

vs

Declarative response
Come to A17-1A.

The first response gives a clear sequence of instructions to be performed in order to reach the target location whereas the second response only gives the target location. The second response does not restrict the path one needs to take to reach the destination.

This difference is crucial.

A powerful language should provide the means for combining simple ideas to form more complex ideas. Every powerful language has three mechanisms for accomplishing this:

- **primitive expressions:-** represent the simplest entities the language is concerned with,
- **means of combination:-** building compound entities from simpler ones
- **means of abstraction:-** naming compound entities and manipulating them as units.

A powerful language should provide the means for combining simple ideas to form more complex ideas. Every powerful language has three mechanisms for accomplishing this:

- **primitive expressions:-** represent the simplest entities the language is concerned with,
- **means of combination:-** building compound entities from simpler ones
- **means of abstraction:-** naming compound entities and manipulating them as units.

These entities can be procedures or data.

Later we shall observe that they are really not so distinct.

- A Scheme expression is a construct that returns a value, such as a variable reference, literal, procedure call, or conditional.

- A Scheme expression is a construct that returns a value, such as a variable reference, literal, procedure call, or conditional.
- Expression types are categorized as primitive or derived.

- A Scheme expression is a construct that returns a value, such as a variable reference, literal, procedure call, or conditional.
- Expression types are categorized as primitive or derived.
- Primitive expression types include variables and procedure calls.

- A Scheme expression is a construct that returns a value, such as a variable reference, literal, procedure call, or conditional.
- Expression types are categorized as primitive or derived.
- Primitive expression types include variables and procedure calls.
- Derived expression types are not semantically primitive, but can instead be explained in terms of the primitive constructs. They are redundant in the strict sense of the word, but they capture common patterns of usage, and are therefore provided as convenient abbreviations.

- A Scheme expression is a construct that returns a value, such as a variable reference, literal, procedure call, or conditional.
- Expression types are categorized as primitive or derived.
- Primitive expression types include variables and procedure calls.
- Derived expression types are not semantically primitive, but can instead be explained in terms of the primitive constructs. They are redundant in the strict sense of the word, but they capture common patterns of usage, and are therefore provided as convenient abbreviations.
- Numerical constants, string constants, character constants, boolean constants etc.. are some examples of primitive expressions.

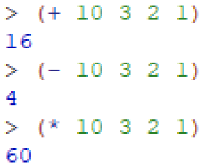
- A Scheme expression is a construct that returns a value, such as a variable reference, literal, procedure call, or conditional.
- Expression types are categorized as primitive or derived.
- Primitive expression types include variables and procedure calls.
- Derived expression types are not semantically primitive, but can instead be explained in terms of the primitive constructs. They are redundant in the strict sense of the word, but they capture common patterns of usage, and are therefore provided as convenient abbreviations.
- Numerical constants, string constants, character constants, boolean constants etc.. are some examples of primitive expressions.

Expressions such as these, formed by delimiting a list of expressions within parentheses in order to denote procedure application, are called **combinations**.

The leftmost element in the list is called the *operator*, and the other elements are called *operands*.

Expressions such as these, formed by delimiting a list of expressions within parentheses in order to denote procedure application, are called **combinations**.

The leftmost element in the list is called the *operator*, and the other elements are called *operands*.



```
> (+ 10 3 2 1)
16
> (- 10 3 2 1)
4
> (* 10 3 2 1)
60
```

The value of a combination is obtained by applying the procedure specified by the operator to the arguments that are the values of the operands.

Even with complex expressions, the interpreter always operates in the same basic cycle:

It reads an expression from the terminal, evaluates the expression, and prints the result.

This mode of operation is often expressed by saying that the interpreter runs in a *read-eval-print loop or REPL in short*.

Notice that it is not necessary to explicitly instruct the interpreter to print the value of the expression.

The convention of placing the operator to the left of the operands is known as *prefix* notation.

Main advantage is that it can accommodate procedures that may take an arbitrary number of arguments.

The convention of placing the operator to the left of the operands is known as *prefix* notation.

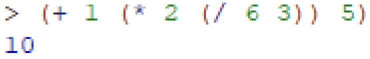
Main advantage is that it can accommodate procedures that may take an arbitrary number of arguments.

A second advantage of prefix notation is that it extends in a straight-forward way to allow combinations to be nested, that is, to have combinations whose elements are themselves combinations:

The convention of placing the operator to the left of the operands is known as *prefix* notation.

Main advantage is that it can accommodate procedures that may take an arbitrary number of arguments.

A second advantage of prefix notation is that it extends in a straight-forward way to allow combinations to be nested, that is, to have combinations whose elements are themselves combinations:

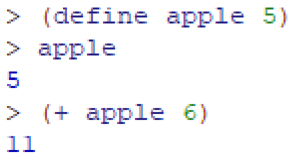


```
> (+ 1 (* 2 (/ 6 3)) 5)
10
```

The real reason, however stems from its relationship with lambda calculus.

A programming language must provide way(s) to use names as a means to refer to computational objects.

define is scheme's simplest means of abstraction, for it allows us to use simple names to refer to the results of compound operations. In general, computational objects may have very complex structure.



```
> (define apple 5)
> apple
5
> (+ apple 6)
11
```

The ability to associate values with symbols and later retrieving them means that the interpreter must maintain some sort of memory to keep track of the *name-object* pairs.

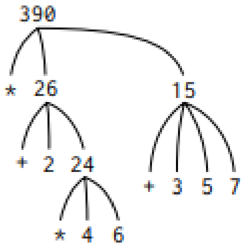
This memory is called the *environment* (more precisely the global environment).

A computation may involve a number of different environments used by various local procedures.

To evaluate a combination, do the following:

1. Evaluate the subexpressions of the combination.
2. Apply the procedure that is the value of the leftmost subexpression (the operator) to the arguments that are the values of the other subexpressions (the operands).

Notice that the procedure to evaluate a combination is *recursive*.



Evaluation of (* (+ 2 (* 4 6)) (+ 3 5 7))

Observe that the repeated application of the first step brings us to the point where we need to evaluate, not combinations, but primitive expressions such as numerals, built-in operators, or other names.
We take care of the primitive cases by stipulating that

- the values of numerals(i.e 54, 2.6) are the numbers that they name,
- the values of built-in operators (i.e. $+$, *) are the machine instruction sequences that carry out the corresponding operations, and
- the values of other names (i.e apple) are the objects associated with those names in the environment.

The environment plays a crucial role in determining the meaning of the symbols in expressions. As these symbols may have different meaning within various procedures.

Notice that the evaluation rule (i.e. operator followed by operands to be processed) given above does not handle definitions.

For instance, evaluating (define apple 5) does not apply define to two arguments, one of which is the value of the symbol *apple* and the other of which is 5, since the purpose of the define is precisely to associate *apple* with a value.

Notice that the evaluation rule (i.e. operator followed by operands to be processed) given above does not handle definitions.

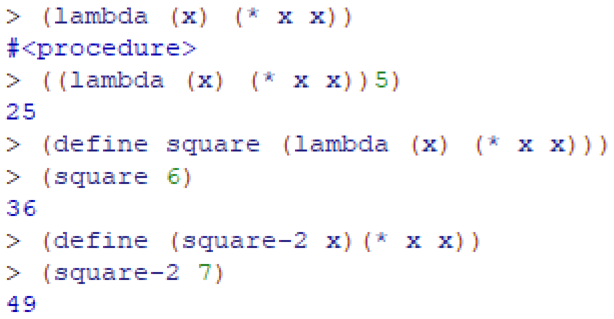
For instance, evaluating (define apple 5) does not apply define to two arguments, one of which is the value of the symbol *apple* and the other of which is 5, since the purpose of the define is precisely to associate *apple* with a value.

Such exceptions to the general evaluation rule are called *special forms*. Each special form has its own evaluation rule.

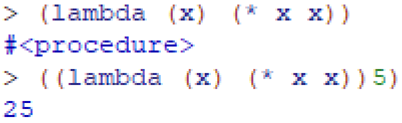
The various kinds of expressions (each with its associated evaluation rule) constitute the *syntax* of the programming language.

In comparison with most other programming languages, Scheme has a very simple syntax; that is, the evaluation rule for expressions can be described by a simple general rule together with specialized rules for a small number of special forms.

Procedure definition is a powerful abstraction technique by which a compound operation can be given a name and then referred to as a unit.

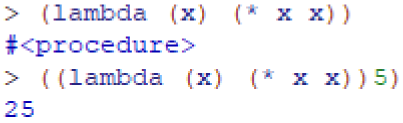


```
> (lambda (x) (* x x))
#<procedure>
> ((lambda (x) (* x x))5)
25
> (define square (lambda (x) (* x x)))
> (square 6)
36
> (define (square-2 x)(* x x))
> (square-2 7)
49
```



```
> (lambda (x) (* x x))
#<procedure>
> ((lambda (x) (* x x))5)
25
```

- The first command uses notation from lambda calculus to define a procedure which takes a variable x and returns its square.



```
> (lambda (x) (* x x))
#<procedure>
> ((lambda (x) (* x x))5)
25
```

- The first command uses notation from lambda calculus to define a procedure which takes a variable x and returns its square.

- The second command shows how to use that procedure by providing an input to that procedure which then produces the desired output.

For convenience, one needs to assign a name to these procedures.

```
> (define square (lambda (x) (* x x)))
> (square 6)
36
```

There is an alternative way to define such a procedure. Special syntactic forms that are simply convenient alternative surface structures for things that can be written in more uniform ways are sometimes called **syntactic sugar**.

```
> (define (square x) (* x x))
> (square 5)
25
```

The general form of a procedure definition is

(define *(<name> <formal parameters>) <body>*)

- <name> is a symbol to be associated with the procedure definition in the environment.
- The <formal parameters> are the names used within the body of the procedure to refer to the corresponding arguments of the procedure.
- The <body> is an expression that will yield the value of the procedure application when the formal parameters are replaced by the actual arguments to which the procedure is applied.

The <name> and <formal parameters> are grouped within parentheses, just as they would be in an actual call to the procedure being defined. i.e (square x)

Compound procedures are used in exactly the same way as primitive procedures.

For example, the “square” procedure behaves as though it is a primitive procedure and any compound procedure built using square also behaves like a primitive procedure.

To evaluate a combination whose operator names a compound procedure, the interpreter follows much the same process as for combinations whose operators name primitive procedures i.e.

To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.