Our Programming Language

15CSE220 :: SICP

R.Sreekumar, Ph.D.

July 30, 2020

1 Expressions

Expressions are the basic building blocks. You can type expressions in the REPL, the expression is evaluated and the result is printed.

```
> 142
142
>
```

In this case, '142' is an symbol or primitive expression that consists of numerals which represent the number 142 in base 10.

Expressions can be anything, but the interpreter must know how to evaluate it.

```
> something
. . something: undefined;
 cannot reference undefined identifier
>
```

If the interpreter cannot identify an expression (or symbol) it throws an error. Otherwise, 'something' is a valid expression if we provide a meaning for them.

2 Combining expressions

Expressions representing numbers can be combined with a primitive expression (or procedure) such as +, /, * to get a combound expression.

```
> (+ 34 65)
99
> (* 5 99)
495
>
```

When we write a compound expression, we enclose them in open and close brackets. This is the only syntax in the language we are going to learn.

A *combination* is nothing but list of delimited expressions, with left most expression is the *procedure* and the rest are *parameters*.

In our case, the left most expression is an operator and the rest are operands.

In a compound expression, the first expression is evaluated with other expressions as operands for that expression.

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

This may confuse at first, as it significantly deviates from our normal convention.

```
int a = 34 + 65
```

But it has got several advantages. For example, as shown below, a procedure can accommodate arbitrary number of arguments.

```
> (+ 2 4 6 8)
20
>
```

Another advantage of this notation is that combinations can be nested.

```
> (+ (* 3 (- 10 6)))
12
```

Actually, there is no limit for such nested expressions.

For example,

```
> (+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
```

is a valid expression. However, reading this expression may be difficult, so we use proper delimitation for human understanding known as *pretty-printing*.

The same expression can be written as:

```
> (+ (* 3
        (+ (* 2 4)
           (+ 3 5)))
     (+ (- (- 10 7)
           6)))
45
```

This will increase the readability of the program. By default, Dr. Racket provides indentation and parenthesis match.

3 The first primitive

We are ready to use the first primitive in our language.

```
(define size 2)
```

define is a built-in primitive used for defining variables and functions (procedures).

As we have mentioned in our first lecture, we will have very few primitives to remember. However, there is a word of caution. The scheme **Programmers Guide** released in 2018 has many primitives defined.

We need not worry about that, because through out our course we will be using the primitives discussed here only except for two or three which come in the next chapter.

Another thing to remember is that **Lisp** is a meta programming language, and if we don't find a primitive, it is easy to define them.

The complete code as in Dr. Racket is:

```
> (define size 2)
> size
2
> (* 4 size)
8
```

In the similar way, we can define our first procedure also.

```
> (define (square x)
    (* x x))
> (square 10)
100
```

Our language is almost complete, but we need some conditionals also.

4 Conditional Expressions and Predicates

Often during the execution of process we need to take decisions. You are familiar with the constructs **if-then-else, for, while, do-while,** etc.

Our language has only two conditionals: They are **cond** and **if**.

It is easy to remember when we have multiple decisions to take, we use **cond** and when we have only two decisions to take we use **if**. **cond** can be considered as **switch-case** in the normal programming languages.

We write our first procedure with **cond**. The procedure is to get the absolute value of a given number.

We know that,

```
abs(x) = x  if x > 0
       = 0  if x = 0
       = -x if x < 0
```

This can be easily written as:

```
> (define (absolute x)
    (cond ((> x 0) x)
          ((= x 0) 0)
          ((< x 0) (- x))))
> (absolute -3)
3
```

The reason for using **absolute** instead of **abs** is that **abs** is already defined in **Dr.Racket**.

Coming back to the code, **cond** has three conditions and for each condition, we use `(<predicate> <expression)`

The general form is:

```
(cond ((<predicate>) (<expression>))
      ((<predicate>) (<expression>))
      ...
      ((<predicate>) (<expression>)))
```

We will use only when we know all the possibilities clearly. If we want to include all other possibilities, we use **else** as the last statement.

As an example, our **absolute** function can be written as:

```
> (define (absolute x)
    (cond ((> x 0) x)
          ((= x 0) 0)
          (else (- x))))
> (absolute -3)
3
>
```

If we have exactly 2 possibilities, we can use **if** statement.

```
> (define (absolute x)
    (if (< x 0) (- x)
        x))
> (absolute -3)
3
```

We can also have **and**, **or** and **not**.

```
(and (> x 0) (> y 0))
(or (> x 0) (> y 0))
(not (> x 0))
```

Before closing this session, we see a trivial but not important primitive which all of you are egarly awaiting.

It is to print to screen.

```
> (display "hello world")
hello world
```

We won't be using this primitive for some reason. Because all our procedures will return a value and they represent an abstraction layer for building bigger applications.

To put it another way, **display** is a primitive with **side effect**, that means, it talks to external system while executing.