Semantics (VI)

Loop Semantics

- Another type of statements that heavily relies on blocks is loop.
- A typical while loop is composed of an expression, test, and a loop body, a block of statements.

$$Loop = Expression\ test; Statement\ body$$

- The **meaning of a while loop depends on the meaning of the expression**.

$$M(Loop\ l, State\ state) = \begin{cases} M(l, M(l.\text{body}, state)), & \text{if } M(l.\text{test}, state) \text{ is true} \\ state, & \text{otherwise} \end{cases}$$

  - If the expression is **true**, it is the **meaning of the** **body** of the loop.
  - Otherwise, the **meaning is the** **state**.
- Note that, **l.body** here **is a statement**. It can be an assignment, an expression, a conditional, a skip, or a block.
  - It also means that it is okay that if there is an assignment, expression, or conditional after the expression *without using curly braces*.
  - If there are more than one statement, curly braces should be used to form a block.

```
# Loop
class Loop:
    def __init__(self, test, body):
        self.test = test
        self.body = body

# Meaning function of Loop
# M(Statement loop, State state)
def M_Loop(loop, state):
    if M_Expression(loop.test, state).value == False:
        return state
    return M_Loop(loop, M_Statement(loop.body, state))

# Meaning function of Statement
# M(Statement statement, State state)
def M_Statement (statement, state):
    if isinstance(statement, Assignment):
        return M_Assignment(statement, state)
    elif isinstance(statement, Conditional):
        return M_Conditional(statement, state)
    elif isinstance(statement, Skip):
        return M_Skip(statement, state)
    elif isinstance(statement, Block):
        return M_Block(statement, state)
    elif isinstance(statement, Loop):
        return M_Loop(statement, state)
    else:
        raise Error("ERROR: wrong type of statement”)
```

```
def main6():
    # while (a < 5) {
    #   a = a + 1;
    #   b = b - 1;
    # }
    # {<a, 1>, <b, 10>}
    vara = Variable('a')
    varb = Variable('b')
    state = State()
    state.setValue(vara, 1)
    state.setValue(varb, 10)
    test = BinaryExpression(vara, '<', Value(5))
    expr1 = BinaryExpression(vara, '+', Value(1))
    expr2 = BinaryExpression(varb, '-', Value(1))
    assignment1 = Assignment(vara, expr1)
    assignment2 = Assignment(varb, expr2)
    block = Block([assignment1, assignment2])
    loop = Loop(test, block)
    newState = M_Loop(loop, state)
    print(newState)


if __name__ == "__main__":
    #main1()
    #main2()
    #main3()
    #main4()
    #main5()
    main6()
```

- To extend the simple interpreter, we need a loop class, a meaning function for loop, a main function to test the code, and extend the meaning function of statement for the loop.
- For loop semantics
  - The abstract syntax for a for loop in Clite requires a a number of different fields: an initial statement, an expression, the body statement, and the post statement (just like the for loop in C and Java).

$$For = Statement\ initial;\ Expression\ test;\ Statement\ body;\ Statement\ post$$

  - The meaning of the for loop is a combination of things.
    - The initial statement modifies the state prior to evaluation of the test expression.
    - The body and post statements modify the state only if the test expression is true.
    - The post statement modifies the state after the application of the body to the state.

  - Because the initial section executes only once, we have to use two functions to represent the meaning of the for loop.

$$M(For\ f, State\ state) = M(f,\ M(f.initial,\ state))$$

$$M(For\ f, State\ state) = \begin{cases} state, & M(f.text,\ state) = False \\ M(f,\ M(f.post,\ M(f.body,\ state))), & otherwise \end{cases}$$

I/O Semantics

- I/O includes file I/O and standard I/O (read from keyboard and write to screen).
  - Some languages have I/O mechanisms built into the language syntax (PHP, Prolog, and Python)
    - In Python, we can use print, raw_input, open, close, read, write, etc. directly.
  - Many languages use a built-in library of functions to handle all I/O (C, Fortran, Lisp)
    - In C, input and output are not part of the C language itself, but defined in standard library <stdio.h>.

C Formatted Standard Output - printf

- `printf` converts, formats, and prints its arguments on the standard output under control of the format. [Show print.c, and ask the output.]

```c
/**
 * print.c
 */

#include <stdio.h>

int main () {
     char c = 97;
     printf("%c\n", c); //a

     int a = 15, b = 9;
     printf("%X, %o\n", a, b); //F, 11

     return 0;
}
```

C Formatted Standard Input - scanf

- `int scanf (char *format, arg1, arg2, …)`
  - reads characters from the standard input
  - interprets them according to the specification in format
  - stores the results through the remaining arguments, each of which must be a pointer

```c
/**
 * rudimentary.c
 */

#include <stdio.h>

int main () {
     double sum, v;

     sum = 0;
    // compile time warning and run time segment fault; should be scanf("%lf", &v)
     while (scanf("%lf", v) == 1)
       printf("\t%.2f\n", sum +=v);
     return 0;
}
```

- stops when it exhausts its format string, or when some input fails to match the control specification
- returns the number of successfully matched and assigned input items; can be used to decide how many items were found.

C Unformatted Standard In and Standard Out

- `int getchar (void)` // read one character at a time from the standard input (keyboard, normally)
- `int putchar (int)` // put the character int to the standard output (screen by default)

- Show lower.c and run it on terminal

```
/**
 * lower.c
 * Convert input to lower case
 */

#include <stdio.h>
#include <ctype.h>

int main () {
     int c;

     while ((c = getchar()) != EOF)
       putchar(tolower(c));
     return 0;
}
```

```
$ gcc lower.c –o lower
$ ./lower
ABCD (press enter)

The output is
$ ./lower
ABCD
abcd (ctl+Z to EOF)
```

- Change the input/output direction
  - If a program prog uses `getchar`, the the command line `prog < infile` causes `prog` to read characters from `infile` instead.
  - The command line `otherprog | prog` runs the two programs `otherprog` and `prog`, and pipes the standard output of `otherprog` into the standard input for `prog`.
  - If `prog` uses `putchar`, `prog > outfile` will write the standard output to `outfile` instead.
- Show lower.c and run it on terminal

```c
/**
 * lower.c
 * Convert input to lower case
 */

#include <stdio.h>
#include <ctype.h>

int main () {
     int c;

     while ((c = getchar()) != EOF)
       putchar(tolower(c));
     return 0;
}
```

```c
/**
 * test.c
 * A simple code to generate a upper case string
 */

#include <stdio.h>

int main () {
     printf("WORLD\n");
     return 0;
}
```

```
/**
 * test.txt
 */
H
E
L
L
O
```

Show test.txt and run

```
$ ./lower < test.txt
h
e
l
l
o
```

Show the output to rslt.txt

```
$ ./lower < test.txt > rslt.txt
$ cat rslt.txt
h
e
l
l
o
```

Show test.c and run pipes

```
$ gcc test.c -o test
$ ./test | ./lower
world
```

Run the code like following, ask the output

```
$ ./lower < test.txt | ./lower
h
e
l
l
o
```