Semantics (IX)

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.

\[
\text{Loop} = \text{Expression test; Statement body}
\]

- The meaning of a while loop depends on the meaning of the expression.
  - If the expression is true, it is the meaning of the body of the loop.
  - Otherwise, the meaning is the state.

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

- Note that, \text{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.
- 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.

```python
# 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:
        print("ERROR: wrong type of statement")
```

# 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:
        print("ERROR: wrong type of statement")
For loop semantics

- The abstract syntax for a for loop requires a number of different fields: an initial statement, an expression, the body statement, and the post statement.

\[ \text{For} = \text{Statement initial}; \text{Expression test}; \text{Statement body}; \text{Statement post} \]

- 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.

- The for loop syntax in C/Java is special, because commas are used to differentiate different statements in the initial and post sections. However, the semantic interpretation is to treat them as blocks, in which case they are Statements.

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

\[
M(\text{For } f, \text{State state}) = M(f, M(f.initial, state))
\]

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

- 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;
}

/**
 * test.c
 * A simple code to generate a upper case string
 */
#include <stdio.h>

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

```plaintext
H
E
L
L
O
```
$ gcc lower.c -o lower
$ ./lower
ABCD (press enter)

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

Show test.txt and run
$ ./lower < test.txt
hello

Show the output to rslt.txt
$ ./lower < test.txt > rslt.txt
$ cat rslt.txt
hello

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
hello