



THE CHINESE UNIVERSITY OF HONG KONG, SHENZHEN

CSC4180

COMPILER CONSTRUCTION

CSC4180 Assignment 1

Student Name:

Bodong Yan

Student Number:

119010369

February 28, 2023

Contents

| | | |
|----------|--------------------------------------|----------|
| 1 | Execution Method and Notice | 2 |
| 2 | Scanner Implementation | 2 |
| 3 | Parser Implementation | 3 |
| 4 | Code Generator Implementation | 4 |
| 4.1 | Overall Design | 4 |
| 4.2 | functions for statement | 5 |
| 4.3 | functions for expression | 6 |
| 4.4 | functions for primary | 7 |
| 5 | Testcases Output | 8 |

1 Execution Method and Notice

Execution and Output The environment needed for my code is simple, just including flex, bison, make and basic C environment. You can build the docker image with the attached Dockerfile, and run the code with the commands in Assignment1's description. As for the output, since the generated MIPS code can be very long, so the generated code will be in both std output and the file "output.asm" for convenience after the compilation finished.

Notice 1 If you find that the code cannot be compiled in the docker image because file "stdio.h" is missing, you can update the apt-get with "apt-get update" and install build-essential with "apt-get install build-essential".

Notice 2 There are some limitations of my compiler. Since the symbol table's design is not so good in the beginning, the total number of symbols are limited to 100 by default, you can edit the struct symbol table to change the limit in file "added.h". However, all the five testcases can be correctly compiled without any edition.

2 Scanner Implementation

The scanner is implemented in file "exp.l" using flex, which will scan all the tokens of the micro language and pass them to the parser.

For this micro language, there are many simple tokens which can be directly represented with strings, such as "begin", "end", "+", and ":=". When these tokens are scanned, the scanner will directly pass the tokens to the parser, for example, "read" will be passed as READ, "(" will be passed as LPAREN, and "+" will be

passed as PLUOP. On the other hand, there are three types of tokens that need to be represented by regular expression including integer, id, and comment. The integer is represented by "[0-9]+" which means the integer is composed by one or more digits. The id is represented by "[A-Z0-9]{1,32}" which means id is composed by at least 1 and at most 32 upper case letters or digits. Then, the comment is represented by "--(.)*\n", which means the comment will be started by "--", followed by any characters and ended within the line.

The scanner will ignore empty space by returning nothing when scanning "[\t\n]". And if "EOF" is scanned, the scanner will call yyterminate() function.

3 Parser Implementation

The parser is implemented in file "exp.y" with bison which will receive tokens from scanner and do the LALR parsing by default.

According to the given CFG defining micro language, there are terminal symbols including all the tokens directly from scanner like READ, WRITE and INTLITERAL. These terminal symbols are defined by "%token" in the definition part. And there are also non-terminal symbols like program, statement, expression and primary, which are defined by "%type". There are four types of non-terminal symbols including expression, expression list, ID list and primary which are needed in code generation part, thus there are four corresponding struct defined in "added" for further operation.

The CFG of micro language should be translated in LR grammars in this part. For example the rule: $\langle \text{statement list} \rangle \rightarrow \langle \text{statement} \rangle \{ \langle \text{statement} \rangle \}$ should be translated to `statement_list: statement | statement_list statement`. In this part,

I basically referred to the code shown in tutorial 3. After the translation, all the 8 types of non-terminal symbols have their rules in LR grammar. In addition, to handle expressions like $A := -3$ or $B := +5$, there are two more matches for primary is added: "PLUOP primary" and "MINUSOP primary". In the end of parser file, implement main() function which calls yyparse().

4 Code Generator Implementation

4.1 Overall Design

This part includes the functions that will be called in parser and generate the MIPS code. The declarations of the functions and the struct needed in this part is in file "added.h", and the implementation of these functions are written in "added.c".

There is a struct called symbol_table composed by an array of struct symbol, size and next empty index, which is used to store all the symbols used in the code. As for the struct symbol, it contains the symbol's ID, status(this value will be 1 if the symbol's value is stored in register, 2 if it's stored in memory), register index that stores the symbol, and the symbol's memory position. Meanwhile, there is a struct for register table that includes an array of register's struct, next available index for save register and next available index for temp register. This register design referred to tutorial 4. The first 8 register in the register array(\$s0 to \$s7) is the save registers, and the next 8 register(\$t0 to \$t7) is the temp registers. The rest three registers(\$t8, \$t9, \$at) are used for system call or loading/saving.

The struct corresponding to the four non-terminal symbols in parser include expr_struct, expr_list_struct, primary_struct, and id_list_struct. However, there is nothing needed in the struct for id list and expression list, since the functions that

need id list or expression can use id or expression to complete their jobs, which will be described in detail later. As for the expression and primary struct, they contains the same things including its type(the value will be 1 if the type is integer, 2 if the value is stored in register, 3 if the value is stored in memory), its value, register index and memory position.

Finally, there is an integer that record the next available position in memory. When saving a word, it will provide its value and minus 4 after the saving operation.

4.2 functions for statement

There are three matches for statement including assignment, read function, and write function.

Function assignment() These function will assign the value of an expression to an ID. First, it will check whether the ID has been saved in symbol table or not. If it's in symbol table, check its status and the expression's type to do the assignment operation. For example, if the symbol is saved in register and the expression is an integer, the code will be "addi register_of_symbol \$zero expression_value", or if the symbol is saved in memory and the expression is saved in register, the code will first assign the expression to register \$t8 and save \$t8 in memory at the symbol's memory position. This part should consider all the possibility to generate corresponding code. And if the ID is not in symbol table, then this function will first save the ID in the symbol table, and check if there is available save register. If empty save register exists, then save expression to the register, or if no empty save register is left, save the expression in memory. In the same time, the symbol's status should be update according to the operation.

Function read_func() Actually, there are nothing in this function, as the function for read operation is implemented in function: `id_to_id_list()` and `id_list_id_to_id_list()`. It can be observed that only read statement will use ID list, so the two functions for ID list will do the read operation for the single ID which will achieve the read operation for all the ID in ID list. The read operation in that two functions is implemented with the three steps: first set register `$v0` be 5 and do the system call to read an integer from user which will be stored in `$v0`, finally assign `$v0` to the target ID. The assignment operation is similar to that in function `assignment()`.

Function write_func() Also, there are nothing in this function since it can be observed that only write statement will use expression list, so the write operation is implemented in function: `expr_to_expr_list()` and `expr_list_expr_to_expr_list()`. These two functions will do the write operation on the single expression which will eventually achieve write operation for all the expressions in the expression list. The write operation is implemented with the three steps: first set `$v0` be 1 and save the expression into register `$a0` according to expression's type, then do the system call to print the value in `$a0`.

4.3 functions for expression

Function primary_to_expression() Since the struct of primary have the same components with the struct of expression, this function will just assign all the components of primary to expression.

Function expr_op_prim_to_expression() This function will assign the result of (expression PLUOP/MINUSOP primary) to a new expression. First it will check if there is temp register available, if available temp register exists, the result will be stored in that temp register, if no empty register exists, the result will be stored

in memory. And the generated code will also depend on the type of the input expression and primary, since each one of them has three types, there are totally 9 combinations to consider.

4.4 functions for primary

Function `lp_expression_rp_to_primary()` Since the struct of primary have the same components with the struct of expression, this function will just assign all the components of expression to primary.

Function `id_to_primary()` This function is similar to assignment function. It will first check if the id is already in symbol table, if it is in symbol table, then update primary's component according to the symbol's status. If the ID is not saved in symbol table, the function will find a save register or memory position to save the ID(initialized to be 0) and update primary's component according to the symbol's status.

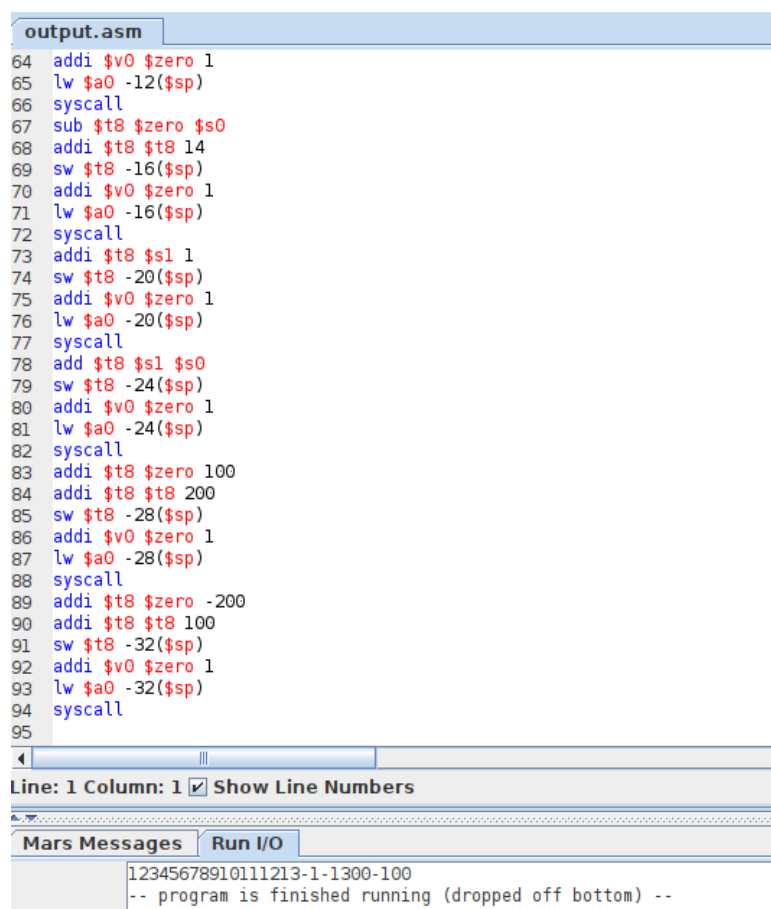
Function `integer_to_primary()` The function will update the primary's value to be the integer and type to be 1.

Function `op_primary_to_primary()` This function is designed to handle the expressions like $A := -3$ or $B := +5$, if the operation is PLUOP, update the result primary's component to be the same as that of the input primary. If the operation is MINUSOP, and if the input primary is an integer, the result primary's value will be the opposite number and the type will be 1, or if the input primary is in register or memory, the function will find a register or memory position to store the opposite value of the input primary and update the result primary's component accordingly.

5 Testcases Output

This part will show the output of the five given test cases by using Mars. There is no empty space between each printed value, but all the values are the same as the given expected output.

Figure 1: Output of test1



The screenshot shows the Mars MIPS simulator interface. The top pane displays assembly code for 'output.asm' with line numbers 64 to 95. The code performs a series of arithmetic and memory operations using registers \$v0, \$a0, \$t8, and \$s0. The bottom pane shows the 'Mars Messages' window with the output of the program, which is a long string of digits followed by a message indicating the program has finished running.

```

64 addi $v0 $zero 1
65 lw $a0 -12($sp)
66 syscall
67 sub $t8 $zero $s0
68 addi $t8 $t8 14
69 sw $t8 -16($sp)
70 addi $v0 $zero 1
71 lw $a0 -16($sp)
72 syscall
73 addi $t8 $s1 1
74 sw $t8 -20($sp)
75 addi $v0 $zero 1
76 lw $a0 -20($sp)
77 syscall
78 add $t8 $s1 $s0
79 sw $t8 -24($sp)
80 addi $v0 $zero 1
81 lw $a0 -24($sp)
82 syscall
83 addi $t8 $zero 100
84 addi $t8 $t8 200
85 sw $t8 -28($sp)
86 addi $v0 $zero 1
87 lw $a0 -28($sp)
88 syscall
89 addi $t8 $zero -200
90 addi $t8 $t8 100
91 sw $t8 -32($sp)
92 addi $v0 $zero 1
93 lw $a0 -32($sp)
94 syscall
95

```

Line: 1 Column: 1 ☒ Show Line Numbers

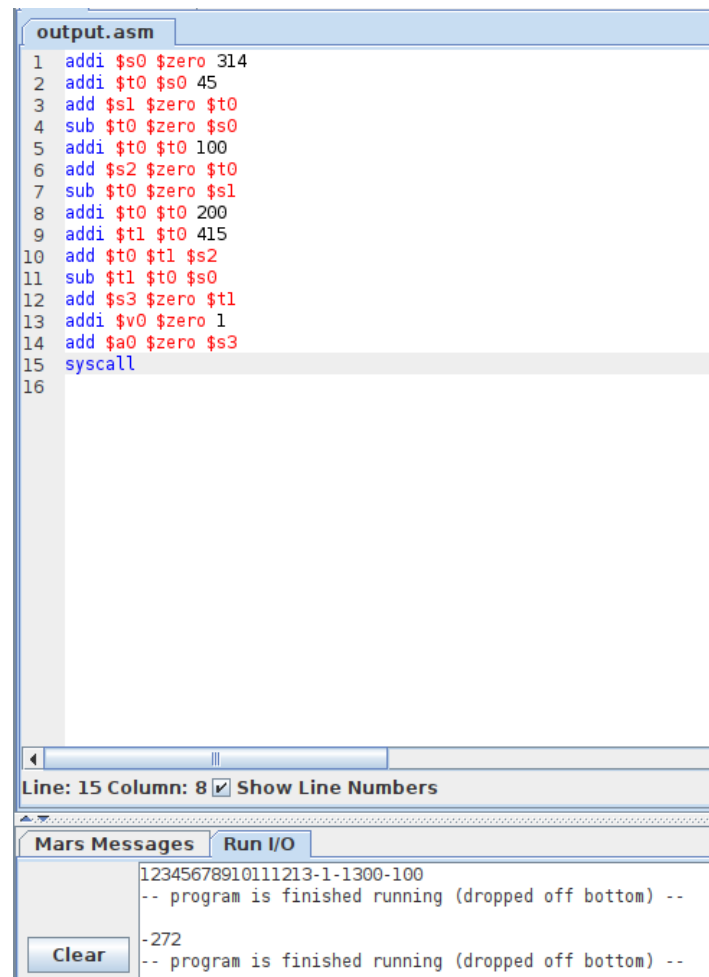
Mars Messages Run I/O

```

12345678910111213-1-1300-100
-- program is finished running (dropped off bottom) --

```

Figure 2: Output of test2



The screenshot shows the MARS MIPS simulator interface. The main window displays assembly code in a file named `output.asm`. The code consists of 16 lines, with the last line being `syscall`. Below the code window, the status bar indicates "Line: 15 Column: 8" and has a checked box for "Show Line Numbers". At the bottom, there are two tabs: "Mars Messages" and "Run I/O". The "Mars Messages" tab is active, showing a list of messages including a long hexadecimal string, a message about the program being finished running, and a message about the program being finished running. A "Clear" button is located next to the messages.

```
1  addi $s0 $zero 314
2  addi $t0 $s0 45
3  add $s1 $zero $t0
4  sub $t0 $zero $s0
5  addi $t0 $t0 100
6  add $s2 $zero $t0
7  sub $t0 $zero $s1
8  addi $t0 $t0 200
9  addi $t1 $t0 415
10 add $t0 $t1 $s2
11 sub $t1 $t0 $s0
12 add $s3 $zero $t1
13 addi $v0 $zero 1
14 add $a0 $zero $s3
15 syscall
16
```

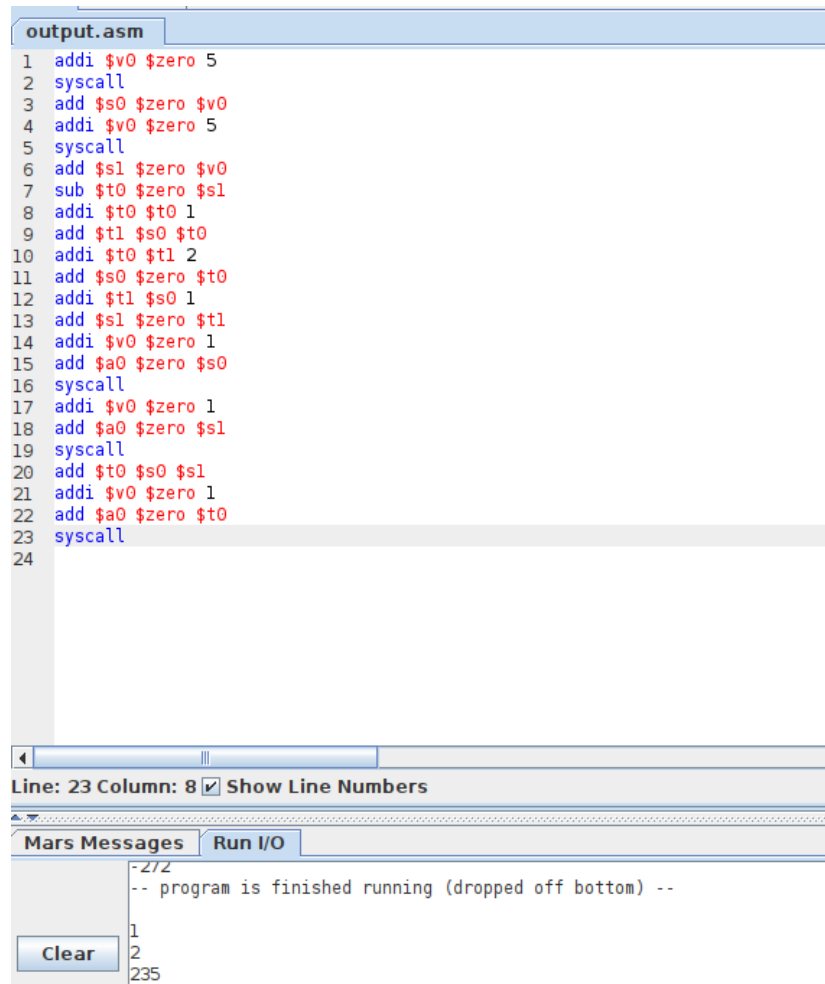
Line: 15 Column: 8 ☒ Show Line Numbers

Mars Messages Run I/O

12345678910111213-1-1300-100
-- program is finished running (dropped off bottom) --
-272
-- program is finished running (dropped off bottom) --

Clear

Figure 3: Output of test3



```
1  addi $v0 $zero 5
2  syscall
3  add $s0 $zero $v0
4  addi $v0 $zero 5
5  syscall
6  add $s1 $zero $v0
7  sub $t0 $zero $s1
8  addi $t0 $t0 1
9  add $t1 $s0 $t0
10 addi $t0 $t1 2
11 add $s0 $zero $t0
12 addi $t1 $s0 1
13 add $s1 $zero $t1
14 addi $v0 $zero 1
15 add $a0 $zero $s0
16 syscall
17 addi $v0 $zero 1
18 add $a0 $zero $s1
19 syscall
20 add $t0 $s0 $s1
21 addi $v0 $zero 1
22 add $a0 $zero $t0
23 syscall
24
```

Line: 23 Column: 8 ☒ Show Line Numbers

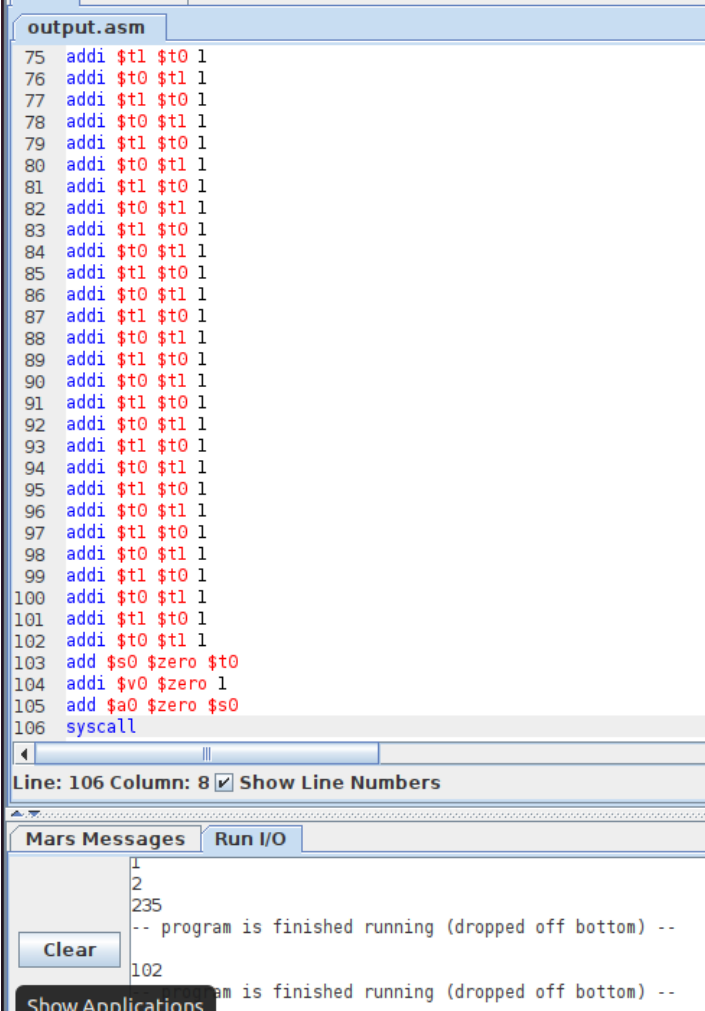
Mars Messages Run I/O

-- program is finished running (dropped off bottom) --

1
2
235

Clear

Figure 4: Output of test4



The screenshot shows the Mars MIPS simulator interface. The main window displays assembly code for a file named `output.asm`. The code consists of 32 instructions, numbered 75 to 106. Instructions 75 through 102 are `addi $t1 $t0 1` and `addi $t0 $t1 1` in an alternating pattern. Instructions 103 through 105 are `add $s0 $zero $t0`, `addi $v0 $zero 1`, and `add $a0 $zero $s0` respectively. Instruction 106 is `syscall`. Below the code window, the status bar shows "Line: 106 Column: 8" and a checked "Show Line Numbers" option. At the bottom, the "Mars Messages" window is open, showing a list of messages. The messages include "1", "2", "235", "-- program is finished running (dropped off bottom) --", "102", and "am is finished running (dropped off bottom) --". There is a "Clear" button and a "Show Applications" button at the bottom left of the messages window.

```
75 addi $t1 $t0 1
76 addi $t0 $t1 1
77 addi $t1 $t0 1
78 addi $t0 $t1 1
79 addi $t1 $t0 1
80 addi $t0 $t1 1
81 addi $t1 $t0 1
82 addi $t0 $t1 1
83 addi $t1 $t0 1
84 addi $t0 $t1 1
85 addi $t1 $t0 1
86 addi $t0 $t1 1
87 addi $t1 $t0 1
88 addi $t0 $t1 1
89 addi $t1 $t0 1
90 addi $t0 $t1 1
91 addi $t1 $t0 1
92 addi $t0 $t1 1
93 addi $t1 $t0 1
94 addi $t0 $t1 1
95 addi $t1 $t0 1
96 addi $t0 $t1 1
97 addi $t1 $t0 1
98 addi $t0 $t1 1
99 addi $t1 $t0 1
100 addi $t0 $t1 1
101 addi $t1 $t0 1
102 addi $t0 $t1 1
103 add $s0 $zero $t0
104 addi $v0 $zero 1
105 add $a0 $zero $s0
106 syscall
```

Line: 106 Column: 8 ☒ Show Line Numbers

Mars Messages Run I/O

1
2
235
-- program is finished running (dropped off bottom) --
102
am is finished running (dropped off bottom) --

Clear Show Applications

Figure 5: Output of test5

The screenshot displays the MARS MIPS simulator interface. The top pane shows the assembly code for 'output.asm', with lines 253 through 284. The code consists of a loop that adds the value in register \$t0 to register \$t1, then loads a new value from memory into \$t8, and increments the loop counter in \$t0. The loop continues until \$t0 reaches 1, at which point it calls the syscall instruction.

```
253 add $t1 $t0 $t8
254 lw $t8 -116($sp)
255 add $t0 $t1 $t8
256 lw $t8 -120($sp)
257 add $t1 $t0 $t8
258 lw $t8 -124($sp)
259 add $t0 $t1 $t8
260 lw $t8 -128($sp)
261 add $t1 $t0 $t8
262 lw $t8 -132($sp)
263 add $t0 $t1 $t8
264 lw $t8 -136($sp)
265 add $t1 $t0 $t8
266 lw $t8 -140($sp)
267 add $t0 $t1 $t8
268 lw $t8 -144($sp)
269 add $t1 $t0 $t8
270 lw $t8 -148($sp)
271 add $t0 $t1 $t8
272 lw $t8 -152($sp)
273 add $t1 $t0 $t8
274 lw $t8 -156($sp)
275 add $t0 $t1 $t8
276 lw $t8 -160($sp)
277 add $t1 $t0 $t8
278 lw $t8 -164($sp)
279 add $t0 $t1 $t8
280 lw $t8 -168($sp)
281 add $t1 $t0 $t8
282 addi $v0 $zero 1
283 add $a0 $zero $t1
284 syscall
```

The bottom pane shows the 'Mars Messages' window, which contains the following output:

```
102
-- program is finished running (dropped off bottom) --
1
1326
am is finished running (dropped off bottom) --
```

A 'Clear' button is visible next to the output messages.