**COMP.2030 HW 6: Symbol Table Due 11/10/23 (Fri) 11:59 PM**

In HW 4, related characters in an input string are grouped into coherent units, called tokens. The resulting table tabToken holds a series of tokens and their types as read from the input string. HW5 processes tokens in the token table and produces a “Symbol Table” stored at tabSym.

In a Symbol Table, each occurance of the variable names (also known as “symbols”) used in the program are stored along with some information associated with each one. Each entry in the symbol table consists of (a) the symbol TOKEN (maximum 8 bytes or two words), (b) its VALUE (one word), and (c) the status (one word).

The VALUE of a variable is an address representing the “line number” of the input string where the variable is used. Intuitively, think of the sequence of input strings given to your program as the lines of some assembly code, so when an instruction refers to a symbol the line number it occurs on will be stored as the VALUE of that occurrence. To keep track of addresses where instructions (input strings) are stored, a location counter “LOC” is incremented by 4 each time a new input string (holding the next instruction) is processed. In this assignment, LOC is a global variable initialized to the address 0x0400.

The STATUS of a symbol’s occurrence depends on how that symbol is being used in the particular instruction given by the input string. The STATUS should be set to

* 1 if it is the definition of the symbol as a label of an instruction.
* 0 if it is a reference to a symbol defined elsewhere, like as the target to a jump.

For example, suppose we are processing the 3rd input (so LOC=0x0408) and get the input string

“loop: ble $t0, 0, done”

This string has two symbols in it: “loop” and “done”. The “loop” symbol is being defined in this statement as the label for the ble instruction so it’s status should be 1, whereas the “done” symbol is referring to a label defined elsewhere so its status should be 0. After processing this line, LOC will be updated to 0x040C and the Symbol Table tabSym should contain two new entries:

|  |  |  |
| --- | --- | --- |
| TOKEN | VALUE | STATUS |
| “loop” | 0x0408 | 1 |
| “done” | 0x0408 | 0 |

The basic structure of the assignment is as follows. The function VAR(&curToken, DEFN) takes two arguments — first is a pointer to the current token (that is, the address of the current token inside tabToken), and second a flag DEFN — that it uses along with the global location value LOC to store a new entry in the Symbol Table tabSym.

In C, the structure of the Symbol Table generating algorithm looks like this:

struct Token { char \*token; int type; };

struct Symbol { char \*label; int value; int status; };

Token \*tabToken; Symbol \*tabSym;

char\* inBuf; int i; bool paramStart;

int LOC = 0x0400;

while (true) { // read more lines forever

read\_line(inBuf); // read input string into inBuf

makeTokenTable(inBuf, tabToken); // save token & type [HW 4]

if (tabToken[1].token[0] == ‘:’) {

// Very first token defines a new label when second token is ‘:’

VAR(&tabToken[0].token, 1); // store label definition on this line

i = 2; // skip label, ‘:’, and instruction

} else {

// Otherwise, first token is the instruction name, skip it

i = 1;

}

paramStart = true;

while (true) { // check each token in line

if (tabToken[i].token[0] == ‘#’) {

// End of the line, stop processing tokens

break;

} else if (paramStart && tabToken[i].type == 2) {

// Found a label reference at the start of an instruction parameter

VAR(&tabToken[i].token, 0); // store label reference on this line

} else {

// Look for a comma, it signals the start of next parameter

paramStart = ‘,’ == tabToken[i].token[0];

}

i++;

}

print\_symbol\_table(tabSym);

clear\_string(inBuff);

clear\_token\_table(tabToken);

LOC += 4;

}

In Pseudo-C, the structure of the Symbol Table generatoring algorithm looks like this:

LOC = 0x0400;

nextLine:

Read input string, save token & type in tabToken [HW 4]

i = 0; // index to tabToken[][]

if (tabToken[1][0] != ‘:’)

goto instruction;

labelDef:

curToken = &tabToken[0];

VAR(curToken, 1); // store label in **tabSym**

i = 2; // skip ‘:’ and instruction

instruction:

i = 1; // skip instruction

paramStart = true;

chkForVar:

if (tabToken[i][0] == ‘#’) // end ‘#’ character

goto dump;

if (!paramStart || tabToken[i][1] != 2)

goto chkForComma;

curToken = &tabToken[i];

VAR(curToken, 0);

goto nextToken;

chkForComma:

paramStart = ‘,’ == tabToken[i][0];

nextToken:

i++;

goto chkForVar;

dump:

print tabSym;

clear inBuf;

clear tabToken;

LOC +=4;

goto nextLine;

In this assignment, VAR() stores the curToken, its value (LOC), and DEFN flag into a new entry in the tabSym.

**Notes**

* If your HW4 is not completed and tabToken cannot be produced from an input string, you can test your code by manually entering values into tabToken one entry (token string and token type) at a time corresponding to an input string. For example, for the input string ‘li $t9, 3’ you need to read strings ‘libbbbbb’ (‘b’ stands for a blank) and integer 2 for the first entry of tabToken, and ‘$bbbbbbb’ and 4 for the 2nd entry, etc.
* **You should include a function which prints the symbol table one row at a time, similarly to the printToken function in HW4.**
* Your program will be tested according to the tokens generated by the following set of MIP instructions:

hex2char:   sw $t0, saveReg($0) #

li $t9, 3 # $t9: counter limit

jal hex2char #

saveReg: or $t0, $t1, $0 #

For the test input strings, the symbol table will be printed as follows.

hex2char:   sw $t0, saveReg($0)

**tabSym: hex2char 0400 1**

**saveReg 0400 0**

li $t9, 3 # $t9: counter limit

**tabSym: hex2char 0400 1**

**saveReg 0400 0**

jal hex2char

**tabSym: hex2char 0400 1**

**saveReg 0400 0**

**hex2char 0408 0**

saveReg: or $t0, $t1, $0

**tabSym: hex2char 0400 1**

**saveReg 0400 0**

**hex2char 0408 0**

**saveReg 040C 1**

Note:

When you print symbol table, four hex digits of the symbol value need to be printed and you may use hex2char function below.

**hex2char function**

# hex2char: Function used to print a hex value into ASCII string.

# Convert a hex value in $a0 to char hex in $v0

# (0x6b6a in $a0, $v0 should have 'a''6''b''6')

#

# 4-bit mask slides from right to left in $a0.

# As corresponding char is collected into $v0,

# $a0 is shifted right by 4 bits for the next hex digit in the last 4 bits

#

# Make it sure that you are handling nested function calls in return addresses

.data

saveReg: .word 0:3

.text

hex2char:

# save registers

sw $t0, saveReg($0) # hex digit to process

sw $t1, saveReg+4($0) # 4-bit mask

sw $t9, saveReg+8($0)

# initialize registers

li $t1, 0x0000000f # $t1: mask of 4 bits

li $t9, 3 # $t9: counter limit

nibble2char:

and $t0, $a0, $t1 # $t0 = least significant 4 bits of $a0

# convert 4-bit number to hex char

bgt $t0, 9, hex\_alpha # if ($t0 > 9) goto alpha

# hex char '0' to '9'

addi $t0, $t0, 0x30 # convert to hex digit

b collect

hex\_alpha:

addi $t0, $t0, -10 # subtract hex # "A"

addi $t0, $t0, 0x61 # convert to hex char, a..f

# save converted hex char to $v0

collect:

sll $v0, $v0, 8 # make a room for a new hex char

or $v0, $v0, $t0 # collect the new hex char

# loop counter bookkeeping

srl $a0, $a0, 4 # right shift $a0 for the next digit

addi $t9, $t9, -1 # $t9--

bgez $t9, nibble2char

# restore registers

lw $t0, saveReg($0)

lw $t1, saveReg+4($0)

lw $t9, saveReg+8($0)

jr $ra