**COMP.2030 HW 4: Symbol Table Due 10/26 (Mon) 11:59 PM**

In HW 3, successive characters in an input string are grouped into coherent units, called tokens. In the resulting tabToken, a series of tokens and their types are stored from the input string. HW4 processes tokens in the token table and produces a Symbol Table (symTab).

In a Symbol Table, variable names used in the program and their values are saved. Each entry in the symbol table consists of (a) the TOKEN (maximum 8 bytes or two words), (b) its VALUE (one word), and (c) the status (one word).

The value of a variable is **the address** of the input string where the variable is used. The LOC (location counter) keeps track of the addresses. LOC is initialized to 0x0400 and is incremented by 4 each time a new input string is processed.

The status of a symbol has three flags: F – the first occurrence of the symbol; A – already defined; D – a new definition. A symbol is defined from the current input string, if the symbol is found in the label field of the input string. The flag A keeps track if the symbol is already defined, and therefore its value in the symbol table has a valid address.

The status field uses only the last three bits: (First occurrence, Already defined, DEFN).

The basic structure of the assignment is as follows.

LOC = 0x0400

nextLine:

Read an input string and save token and type in tabToken [HW 3]

If (tabToken[0][0] == ‘#’) goto exit

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

nextTok: if (tabToken[i+1][0] != ‘:’) goto operator

TOKEN = tabToken[i][0]

valVar = VAR (TOKEN, 1) // store label in symTab

i +=2 // skip ‘:’

operator: i++ // do nothing with operator

isComma = true

chkVar: if (tabToken[i][0] == ‘#’) goto dump // ‘#’

if (!isComma || tabToken[i][1]!=2) goto nextVar

TOKEN = tabToken[i][0]

valVar = VAR (TOKEN, 0)

nextVar: isComma = (tabToken[i][0] == ‘,’)

i++

goto chkVar

dump: clear inBuf

clear tabToken

print symTab

LOC +=4

goto nextLine

exit: stop the program

VAR () is passed the current TOKEN (or a pointer (index) to the current TOKEN), and a flag DEFN. VAR () returns a value as determined by the outcome of symbol table action routines, symACT0(), symACT1(), …, symACT5().

VAR () first calls LookUp() to search for the TOKEN in **the symTab**. LookUp() returns -1 if the token is not found, the index of the token in the symbol table, otherwise.

If the token is not found, the newStatus shall be (F,A,D)=(1, 0, DEFN), and you save the token in the symbol table. In either case, the index to the symbol table entry of the token is saved in symIndex. If the token is found, however, the newStatus is dependent on the previous values of FAD in the symbol table.

VAR (TOKEN, DEFN){

symIndex = srchSymTab(TOKEN) // -1 if not found; index if found

if (symIndex < 0) { // First occurrence

newStatus = 0x4 | DEFN;

symIndex = saveSymTab(TOKEN, newStatus);

}

else {

oldStatus = symTab[symIndex][2];

// determine the new status (F flag = 0)

newStatus = oldStatus & 0x2 | ((oldStatus & 0x1) << 1); // set A flag

newStatus = newStatus | DEFN; // set D flag

symTab[symIndex][2]=newStatus;

}

retVal = symACTS(newStatus, symIndex); // from the jump table

retVar: return retVal

}

Based on the newStatus, the assembler updates the symbol table and return a value according to the following.

|  |  |  |  |
| --- | --- | --- | --- |
| 4’s (F) | 2’s (A) | 1’s (D) | action |
| 0 | 0 | 0 | symACT0: This is a forward reference. Store LOC in VALUE field of the symbol table. |
| 0 | 0 | 1 | symACT1: A previously used TOKEN is defined for the first time. Store LOC in VALUE field of the symbol table. |
| 0 | 1 | 0 | symACT2: A previously defined TOKEN is used. |
| 0 | 1 | 1 | symACT3: A previously defined TOKEN is defined again. Print “Double Definition Error.” |
| 1 | 0 | 0 | symACT4: TOKEN seen for the first time as a forward reference. Store LOC in VALUE field. |
| 1 | 0 | 1 | symACT5: TOKEN seen for the first time as a definition. Store LOC in VALUE field of the symbol table. |
| 1 | 1 | 0 | N/A |
| 1 | 1 | 1 | N/A |

Namely, the newStatus value triggers a function call indexed by the value. Since there are six statuses to consider, a jump table is often used to process actions corresponding to the new status value.

symACTS: b symACT0

b symACT1

b symACT2

b symACT3

b symACT4

b symACT5

The last instruction in each symACTs should be a branch instruction with **the target address of ret\_VAR** to return from VAR ().

Each action routine will carry out what is described in the ‘action’ column of the table below. In order to call the right routine, the following MIPS instructions can be used, assuming that $s1 has the new status value (you are free to use another register other than $s1).

la $s0, symACTS

sll $s1, $s1, 2

add $s0, $s0, $s1

jr $s0

retSymAct:

**Note**

MIPS has a single level of function calls, as the return address is saved in a single register of $ra. Thus, when VAR () calls LookUp(), for example, the return address from VAR () has to be saved prior to the call to LookUp().

**Note**

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

the symbol table will be printed as follows.

hex2char:   sw $t0, saveReg($0)

**symTab: hex2char 0x0400 0x5**

**saveReg 0x0400 0x4**

li $t9, 3 # $t9: counter limit

**symTab: hex2char 0x0400 0x5**

**saveReg 0x0400 0x4**

jal hex2char

**symTab: hex2char 0x0400 0x2**

**saveReg 0x0400 0x4**

saveReg: or $t0, $t1, $0

**symTab: hex2char 0x0400 0x2**

**saveReg 0x040C 0x1**

**hex2char function**

#

# hex2char:

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

# Convert a hex 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 four bits for the next hex digit in the last four 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