CMPE 230

Homework-1 Report

Student 1: Hasan Öztürk – 2017400258

Student 2: Alperen Değirmenci – 2017400255

Instructor: Can Özturan

1- Description and the Solution to the Problem

The problem is to translate given codes in a language to A86 codes. Our simple compiler called BITC generates A86 codes for given bitwise operations. We wrote our compiler program in C++.

Our program takes the input line by line and processes it line by line. Every line can have **assignment operations** with either expressions or just hexadecimal numbers or **print statements**.

Each line in the input file are separately put in a string vector. Then from the start to ending, each line is tokenized before processing. Tokenizing is done by removing the whitespaces of the line and resolving the line by numbers, variables, parentheses and operators and so on. Tokenized items are added to a token vector which made up by the **Token** class that has two parameters, **Token type** and **Token string**. To distinguish a line is either assignment operation or print statement, we check the token vector for an **equal sign (=)**.

Processing a line has two parts in the function. First one is tokenization as we mentioned earlier and second one is deciding what to do with that line whether it is an assignment or print statement. If it is a simple print statement, program is not that complex, it just goes for the printing the A86 print codes for the variable and the register. For the assignment operations, our program follows a path like in the **syntax of expression (no left-recursion).** How the code works will be explained in the code section. There are also two types of functions which helps the program to go over the tokens in a given line called **readToken** and **lookAtToken**. **ReadToken** function goes over the **line string** token by token and increments the **current token** (starting from zero) after calling related function. These functions will make our way through tokens on a line and will decide which path program take in the deeper methods while processing the line.

Syntax of expressions in our program contains **expr, term, moreterms, factor and morefactors.** These functions maintain the recursive reading of a line from its tokens and decide which operations precede each other. While these functions call each other, they also start printing A86 codes to the **output** file. To print an A86 code to the output file, function in the deepest level which is **factor** function, also makes the **syntax error** check before transferring codes to the output file. Inside of these functions will also be explained more detailed in the code section.

2- Explanation of the Code

The Main function:

In the main function of the program:

- Checks if there is an input file for the program.
- Takes input file with with ***(int argc, char* argv[])*** and ***ifstream***
- For every line in the input, pushes them to a string vector.
- After the input file is read, creates an output file with ***ofstream.***
- Output file is named **result.asm** (to show the *.asm extension*).

Token Class

Has two parameters: **TokenType** and **TokenString.** TokenTypes are defined in tokenize function and they are **generic**: *equal sign*, *comma*, *open-close parentheses*, *xor*, *not*, *ls*, *rs*, *lr*, *rr*, *variable*, *number*, *or*, *and*. Tokenstrings are the string equivalents of tokentypes. Like: *=, (,), $x, $y, xor, not, 20bc, 2a17…*

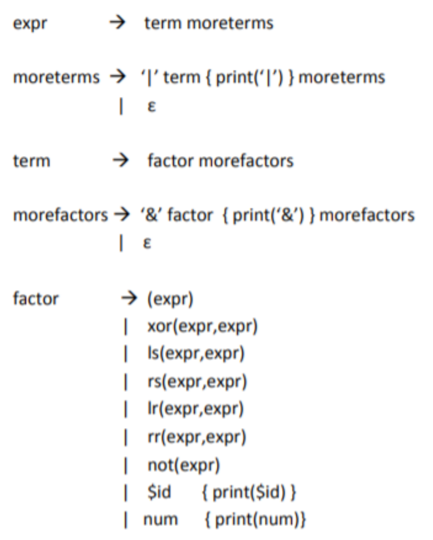
Global Variables

- `vector<string> input`**:** Each line in the input file is stored in this vector line by line.
- `vector<Token> tokenVec`**:** Each token of a line is stored in this vector.
- `string tokenType = ""`**:** Type of the focused token.
- `string tokenString = ""`**:** String value of the focused token.
- `int currentToken = 0` **:** This variable shows the index of the token in a specific line. It's incremented after operations by ***readToken*** function and reset to zero after each line's process is over.

Other Functions

- `void readToken(string& type, string& str)` : This function reads the tokens from the ***tokenVec*** of the current processing line and initializes the variables ***tokenString*** and ***tokenType***. Then increments ***currentToken*** by 1.
- `string lookAtToken()` : Returns the token type of ***currentToken.***
- `vector<Token> tokenize(string line,vector<Token>& v)` **:** Removes the whitespaces in a given line and creates a string for the line. Then in a *for* loop with length of the string, evaluates each character and puts them together if necessary, creates tokens for them with *if* and *else if* statements.
- `void processLine(string line)` : This function calls the *tokenize* function for the parameter string *line* and if there is an equal sign calls *assignment* function, otherwise just prints the codes for print into output file via printing operations.
- `void assignment()` : Assigns an evaluated expression to the given variable at the beginning of the line.

- `void expr(), void term(), void moreterms(), void factor(), void morefactors()`:
  These 5 functions are invoked in the *assignment* function after reading the *variable* to be assigned and *equal sign*. They work recursively just like in the given **syntax of expression** as seen in the picture.



```
expr        →  term moreterms

moreterms   →  '|' term { print('|') } moreterms
            |  ε

term        →  factor morefactors

morefactors →  '&' factor  { print('&') } morefactors
            |  ε

factor      →  (expr)
            |  xor(expr,expr)
            |  ls(expr,expr)
            |  rs(expr,expr)
            |  lr(expr,expr)
            |  rr(expr,expr)
            |  not(expr)
            |  $id     { print($id) }
            |  num     { print(num)}
```

Expr – calls **term** and **moreterms**
Term – calls **factor** and **morefactors**
Moreterms – looks for **or** sign (**|**) and calls **term,** prints out **or** assembly codes and **moreterms** again.
Morefactors – looks for **and** sign (**&**) and calls **factor,** prints out **and** assembly codes and **morefactors** again.
Factor – with successive ***if*** and ***else if*** statements, function looks at token and decides what to do. *Xor*, *not*, *left shift*, *right shift*, *left rotate*, *right rotate*, *open-close parentheses*, *variables and numbers* are all **parsed** in this function. And **syntax errors** are checked in every *if* or *else if* statement.

3- How We Tried the Code

- We first compiled the program with the following **Command Prompt** command:
  g++ main.cpp Token.cpp -o output.exe
- And then for a given **example.bc** file (which contains test cases) we applied the following command:
  output.exe "C:\Users\Alperen\Desktop\cmpe230 finish\example.bc"
- Which generates **result.asm**
- Then we tried our output file on DosBox environment with:

  a86 result.asm

  result.com

- And saw the all test cases running properly.