CMPE 230 PROJECT I REPORT

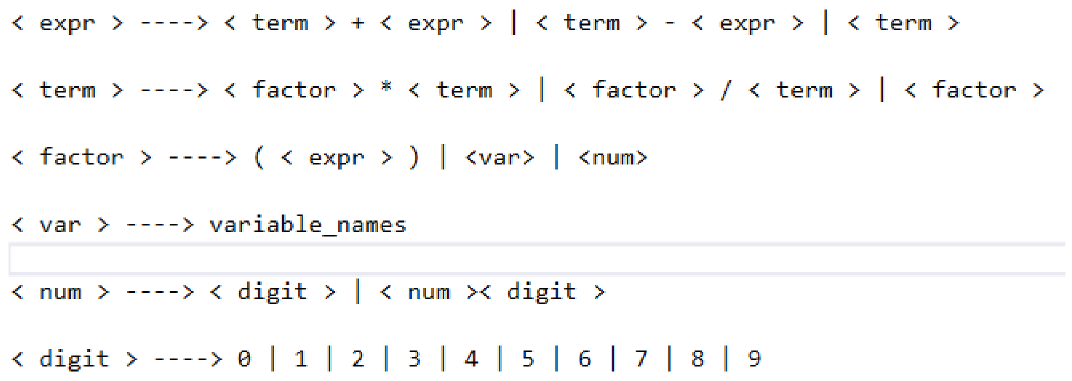
Miraç Batuhan Malazgirt & Hasan Baki Küçükçakıroğlu

April 28, 2021

1 Problems and How Did We Solve Them

One of the first problems that we encountered in the project was syntax errors, white spaces, and comments. We solved the white space problem by erasing all of the white spaces that the input code has at the beginning of the program. Also, we erase all of the input strings that come after char '#' to get rid of comments. After all of these, we check the entire program for syntax errors and if it has syntax errors then we print appropriate LLVM code to indicate syntax error. Therefore we make sure ourselves about our input code is syntactically true before starting parsing.

The second problem we encountered was about how are we going to handle expressions. Expressions have + - / * operators and also may contain parentheses. Thus we were confused about operator precedence at the beginning of the process. We began to search for solutions to the operator precedence problem. The solution we were looking for has come from CMPE 260 " Principles of Programming Languages ". We learned that the highest precedence operator must be written at the end of the BNF Notation and vice versa for the lowest precedence operator. Therefore based on that information we prepared the following draft in BNF notation for our program.



```
< expr > ----> < term > + < expr > | < term > - < expr > | < term >

< term > ----> < factor > * < term > | < factor > / < term > | < factor >

< factor > ----> ( < expr > ) | <var> | <num>

< var > ----> variable_names

< num > ----> < digit > | < num >< digit >

< digit > ----> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

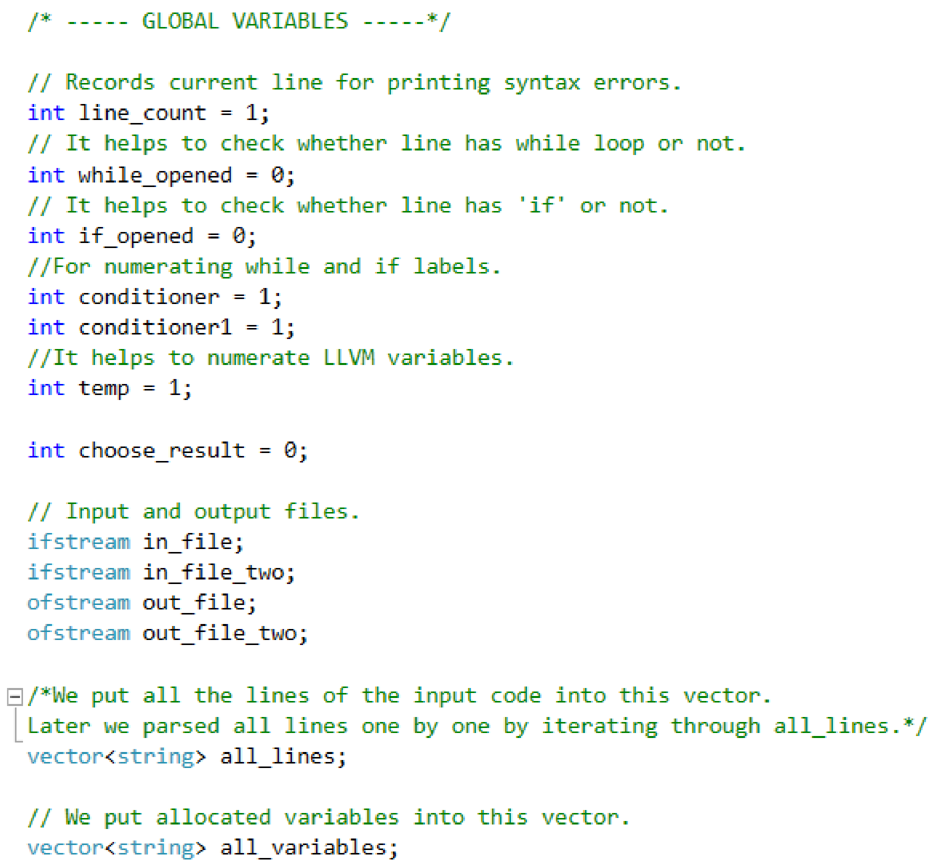
We were able to handle operator precedence by using the BNF description which we have described above.

The last problem we solved was about 7 things: while, if, print, choose, assignment statements, an empty line and }. We realized that every line in a given input code has to be one of seven possibilities. So we printed appropriate LLVM code to print syntax error if one line does not comply with one of seven possibilities that we have talked about.

Solution of the how-to handle of While, if, print, an empty line and } were pretty straightforward. We just printed the appropriate LLVM code on our file. However, choose was a little tricky. Luckily we were able to handle it as well by using recursion techniques.

2 Implementation and Our Code

We tried to write our code as modular as it gets. Our functions divide into three main categories. The following image shows all of the methods, global variables that we have declared and our code architecture.



```
/* ----- GLOBAL VARIABLES -----*/

// Records current line for printing syntax errors.
int line_count = 1;
// It helps to check whether line has while loop or not.
int while_opened = 0;
// It helps to check whether line has 'if' or not.
int if_opened = 0;
//For numerating while and if labels.
int conditioner = 1;
int conditioner1 = 1;
//It helps to numerate LLVM variables.
int temp = 1;

int choose_result = 0;

// Input and output files.
ifstream in_file;
ifstream in_file_two;
ofstream out_file;
ofstream out_file_two;

/*We put all the lines of the input code into this vector.
Later we parsed all lines one by one by iterating through all_lines.*/
vector<string> all_lines;

// We put allocated variables into this vector.
vector<string> all_variables;
```

```
/* ------- DECLARATION OF THE METHODS ------- */
// Methods divide into 3 main categories: Helper Methods, Syntax Checkers, Parser Methods.


/* --- Helper Methods --- */
void print_module();
void take_all_lines();
bool is_number(string s);

/* --- Syntax Checkers ---*/
void paranthesis_count(int i);
void equal_sign_count(int i);
void white_space_eraser(int i);
void while_if_checker(int i);
void print_checker(int i);
void comment_eraser(int i);
bool isValidVar(string var);
void is_it_all_good(int i);
void syntax_checker(int i);

/* --- Parser Methods --- */
string assign_parser(string line);
string add_op(string line);
string sub_op(string line);
string mul_op(string line);
string div_op(string line);
string par_op(string line);
void parser(string line);
void choose_handler(string exp)
string choose_finder(string line)
```

The meaning of the functions that we have used is the following, respectively:

print_module():

Prints the beginning of the LLVM file.

take_all_lines():

Takes all the lines of the given input code and puts it into all_lines vector.

is_number():

Takes the given string and decides whether it is number or not. If false then it is a variable.

paranthesis_count(int i):

Checks that whichever open parentheses in a line has a closed parentheses. If every open parentheses has a closed parentheses does not print syntax error, prints and exits the program otherwise.

equal_sign_count(int i):

Checks that given line has more than one equal sign.

white_space_eraser(int i):

Erases all of the white spaces in a line.

while_if_checker(int i):

Checks that while or if statement syntactically true. If they are not syntactically true then prints a syntax error message and exits the program.

print_checker(int i):

Decides given line of the print statement is syntactically okay. If not then prints syntax error.

comment_eraser(int i):

Erases every comment in the given input code which means erasing everything after the char # .

isValidVar(string var):

Decides whether variable name is syntactically correct or not.

is_it_all_good(int i):

Checks that if the given line has unpermitted char. Prints syntax error if it has.

syntax_checker(int i):

Parent method for all SYNTAX CHECKER methods.

assign_parser(string line):

Assignment parser for assignment statements. Takes left of the = operator and checks whether it is a valid variable or not. Then takes the right of the = operator and puts it into add_op method. Then parsing begins. At the end of the method it writes the LLVM code that equalizes given variable with given expression.

add_op(string line):

If line contains a choose statement than sends the line to the choose_finder function and continues after. Right recursive. Iterates through chars of the line and stops when it see a + operand. Then divides expression into two part as term and expr. Continues it until all +'s are checked. Then calls sub_op function.

sub_op(string line):

Iterates through chars of the line and stops when it see a - operand. Then divides expression into two part as term and expr. Continues it until all -'s are checked. Then calls mul_op function.

mul_op(string line):

Right recursive. Iterates through chars of the line and stops when it see a * operand. Then divides expression into two part as factor and term. Continues it until all *'s are checked. Then calls div_op function.

div_op(string line):

Right recursive. Iterates through chars of the line and stops when it see a / operand. Then divides expression into two part as factor and term. Continues it until all /'s are checked. Then calls par_op function.

par_op(string line):

Right recursive. If zeroth index of the line is parentheses then: Removes outer parentheses. And calls the add_op method. If zeroth index of the line is not parentheses then that means it is a variable or number. Controls whether it is a number or variable. If number returns the line. If variable then controls whether it is a valid variable or not. If not prints syntax error. If it is a variable then it pushes back variable into all_variables vector.

parser(string line):

First checks if line has a equal sign char or not. If yes then sends the line to the assign_parser method. If no, then continues to parsing the line. There are five possibilities. A line can be: While statement, if statement, print statement, } or an empty line. Prints the necessary LLVM code for the 5 possibilities.

choose_handler(string exp):

Handles choose expressions.

choose_finder(string line):

Finds choose statements in a line.