Compiler Project Report

Ali Parvizi
Negin Shirvani

Dr. Eghbal Mansouri
5th of Jan 2022

INTRODUCTION

The project is meant to read a program in python from a text file and write a three address code for it in C and save it as a text file.

Attention : you have to have ply installed on your system so you can run the code.

Files

1. lexer.py (ply.lex)
2. parser.py (ply.yacc)
3. parser.out
4. parsetab.py (this file is automatically generated from parser)
5. program.c
6. program.py.txt

lexer.py

In this file we defined all tokens and non-tokens(Strings, key words, names, parentheses brackets, comments, etc) using ply library and wrote a function to distinguish whatever we might have in a code as comments, numbers, newline, etc.
So we get the text and brake it into tokens
More details are commented on in the file.

parser.py

Parser takes the tokens and builds expressions out of them.

Here we defined how our expressions should be due to different situations (grammar rules):

```
statements : statements statement

             |empty

statement : assignment

            |if |while|for|expr|print|CONTINUE|BREAK

print : PRINT LPRAN STRING_LITERAL print_args RPRAN

Print_args : expr | empty

for : FOR ID IN RANGE LPRAN params RPRAN COLON LBRACE statements RBRACE

params : num_or_id

         | num_or_id SEP num_or_id

         | nume_or_id SEP num_or_id SEP num_or_id

while : WHILE expr COLON LBRACE statements RBRACE

if : IF expr COLON LBRACE statements RBRACE elif

elif : ELIF expr COLON LBRACE statements RBRACE elif | else

else : ELSE COLON LBRACE statements RBRACE | empty

assignment : ID EQ expr

             | ID PLUS_EQUAL expr

             | ID MINUS_EQUAL expr

             | ID TIMES_EQUAL expr

             | ID DIV_EQUAL expr
```

```
expr : MINUS expr %prec UMINUS

expr : PLUS expr %prec UPLUS

expr : expr PLUS expr | expr TIMES expr | expr DIV expr | expr MOD expr |

expr AND expr | expr OR expr | expr LT expr | expr LTE expr | expr GT expr |

expr GTE expr | expr EQU expr | expr NEQU expr | ID | NUMBER | TRUE | FALSE

expr : LPRAN expr RPRAN

num_or_id : NUMBER | ID

empty :
```

Now we write the three address code for the code after parsing it as below :

We create a symbol table, then we create a parse tree for our code and then we write a three address code using the parse tree.

In the code it is defined how do we write a three address code for the code we have for instance :

```
def tac_while(self, line):
    condition = line[1]
    statements = line[2]

    condition_tac_str, condition_root = self.get_tac(condition)
    statements_tac_str = self.tac_program(statements)

    start_label = self.get_label()
    end_label = self.get_label()

    # the second time that we need to evaluate the condition
    # no variable definitions will be needed
    condition_tac_str_repeat = condition_tac_str.replace('float ', '')

    structure = f"{start_label}: \n" \
                f"if (!{condition_root}) goto {end_label};\n" \
                f"{statements_tac_str}\n" \
                f"{condition_tac_str_repeat}\n" \
                f"goto {start_label};\n" \
                f"{end_label}:;\n"

    return condition_tac_str + structure, None
```

```
def tac_for(self, line):
    for_var = line[1]
    start, end, step = line[2]
    statements = self.tac_program(line[3])

    start_label = self.get_label()
    end_label = self.get_label()

    structure = ""

    # define for variable if not defined before
    if for_var not in self.symbol_table:
        structure += f"float {for_var};\n"
        self.symbol_table[for_var] = 'float'

    # initialize the for variable
    structure += f"{for_var} = {start};\n"
    op = '>=' if step > 0 else '<='

    structure += f"{start_label}:\n" \
                 f"if ({for_var} {op} {end}) goto {end_label};\n" \
                 f"{statements}\n" \
                 f"{for_var} += {step};\n" \
                 f"goto {start_label};\n" \
                 f"{end_label}:;\n"
    return structure, None
```

parser.out

The output of the parsing part is saved in this file and we can see all the rules and grammars in this part. We'll see a small part of the output bellow:

(More can be seen in the file)

```
Rule 0     S' -> statements
Rule 1     statements -> statements statement
Rule 2     statements -> empty
Rule 3     statement -> assignment
Rule 4     statement -> if
Rule 5     statement -> while
Rule 6     statement -> for
Rule 7     statement -> expr
Rule 8     statement -> print
Rule 9     statement -> CONTINUE
Rule 10    statement -> BREAK
Rule 11    print -> PRINT LPRAN STRING_LITERAL print_args RPRAN
Rule 12    print_args -> expr
Rule 13    print_args -> empty
Rule 14    for -> FOR ID IN RANGE LPRAN params RPRAN COLON LBRACE statements RBRACE
Rule 15    params -> num_or_id
Rule 16    params -> num_or_id SEP num_or_id
Rule 17    params -> num_or_id SEP num_or_id SEP num_or_id
Rule 18    while -> WHILE expr COLON LBRACE statements RBRACE
Rule 19    if -> IF expr COLON LBRACE statements RBRACE elif
Rule 20    elif -> ELIF expr COLON LBRACE statements RBRACE elif
Rule 21    elif -> else
Rule 22    else -> ELSE COLON LBRACE statements RBRACE
Rule 23    else -> empty
Rule 24    assignment -> ID EQ expr
Rule 25    assignment -> ID PLUS_EQUAL expr
Rule 26    assignment -> ID MINUS_EQUAL expr
Rule 27    assignment -> ID TIMES_EQUAL expr
Rule 28    assignment -> ID DIV_EQUAL expr
Rule 29    expr -> MINUS expr
Rule 30    expr -> PLUS expr
Rule 31    expr -> expr PLUS expr
Rule 32    expr -> expr MINUS expr
Rule 33    expr -> expr TIMES expr
Rule 34    expr -> expr DIV expr
Rule 35    expr -> expr MOD expr
Rule 36    expr -> expr AND expr
Rule 37    expr -> expr OR expr
```

The question is how to generate a parse tree, syntax tree and then a three address code using the tree:

We start from the start symbol and put it as the root.Each leaf of the tree is labeled by a token or it might be empty. Nonterminals are used as interior nodes.
Syntax tree is generated using the grammars we wrote based on how python works. (the grammars are written above).
Now we walk through the syntax tree to generate the three-address-code.

Conclusion

We used the ply library to create a lex file and then we tokenized our code, gave it to parser and then created a parse tree and then generated a three address code for it in C language.

More details are commented on the files. So it would be easier to track.

Thanks for your time