Milestone 2

Indian Institute of Technology Kanpur

Semester II, 2023-24

Course: CS335
Instructor: Swarnendu Biswas

Students:

- Paras Sikarwar - 210699
- Ravija Chandel - 210835
- Siddheshwari Ramesh Madavi - 211036

1 Tools Used

- **Flex** is used for writing the lexer.
- **Bison** is used for writing the parser.
- **Dot** is used for creating the Abstract Syntax Tree.

2 Features Implemented

The following features are implemented in addition to Milestone 1:

- symbol table data structure
- performing semantic analysis to do limited error checking,
- generating a semantically equivalent 3AC form of the input program that will be used for target code generation
- including runtime support for function calls.

3 Compilation and Execution Instructions

$ make

$ ./m2.out -input inputfilename.py -output outputdigraphfilename.dot -output3AC output3ACfilename.txt -outputSymTab outputSymTabfilename.csv -verbose

$ dot -Tpdf outputfilename.dot -o graph.pdf

After running these instructions, the user will get the digraph representation of the input program in outputdigraphfilename.dot, the abstract syntax tree of the input program in graph.pdf, the dump of the 3AC of the functions in text format in the file output3ACfilename.txt and a dump of the symbol table of each function in the CSV file outputSymTabfilename.csv.

4 Command Line Options

The following command line options have been implemented:

- **-help**: Gives instructions on how to use the parser and generate the parse tree and get the 3AC intermediate representation of the code
- **-input**: Used to pass the path to the input file containing the Python code to be parsed.

  If path to input file is not passed, a warning message will be displayed and input will be parsed from a default file "input.py".

- **-output**: Used to pass the path to the file where the digraph output will be displayed.

  If path to output file is not passed, a warning message will be displayed and the output will be displayed in the default file "output.dot".

- **(NEW) -output3AC**: Used to pass the path to the file where the 3AC output will be displayed.

  If path to output file is not passed, a warning message will be displayed and the output will be displayed in the default file "3AC.txt".

- **(NEW) -outputSymTab**: Used to pass the path to the file where the symbol table output will be displayed.

  If path to output file is not passed, a warning message will be displayed and the output will be displayed in the default file "symtab.csv".

- **-verbose**: Generates a file "verbose-output.pdf" which contains detailed information about the parser's output stored in "parser.output"

  Usage: ./a.out -help -input input.py -output output.dot -output3AC 3AC.txt -outputSymTab symtab.csv -verbose

  The tags -help and -verbose are optional and can be used as and when required.

5 Creating Symbol Table

The symbol table is created by creating structs to store the structure of a symbol table entry as well as the structure of the symbol table.

We have added functions to insert an entry into the symbol table, lookup an entry from the symbol table, enter a scope, exit a scope and to print the symbol table in the file parser.y. The functions are called within the semantic actions associated with the grammar symbols as and when required.

The global_table variable contains the pointer to the global symbol table for the entire program. The message "scope level changed" is displayed when the scope changes and the symbol table for the new lexically-scoped block follows this message.

The symbol table rows have the name of the identifier as their header. The columns of the symbol table contain Type, Kind, Size and the Line Number of the declaration. The columns depend upon the type of identifier.

6 Intermediate Representation - Three Address Code

The three-address code(3AC) representation of the Python program is created by doing a depth first traversal on the Abstract Syntax Tree and using the quadruple data structure. The code for the child nodes is generated before the parent node.

We have added the files "make_tac.hpp" and "make_tac.cpp" to aid in creation of quadruples and the functions append_tac() and generate_tac() are added in the ast.hpp file to aid in generating the three-adress code from the abstract syntax tree and the function generate_quad() is also added in the ast.hpp file to create the 3AC according to the node type encountered.

Finally, the 3AC of the children is appended to the parent until the root node and the complete 3AC of the program is generated.