Compiler Design(CS 335), 2023-24
Indian Institute of Technology Kanpur
Milestone 2

Student Names: Abhinav Garg, Kriti, Ashutosh Kumar
Roll Numbers: 210029, 210534, 210221
Date: April 4, 2024

Milestone

2

Introduction

For milestone 2, we have extended our compiler from milestone 1 to include additional features such as symbol table management, semantic analysis, generation of three-address code (3AC), and runtime support for function calls. The primary goal of this milestone is to prepare the groundwork for generating x86_64 code from Python input.

Compiling and Executing Instructions

The optional commands provided by our program are given below:

- **-i :** Input file specification (default is command line) which the compiler uses for reading the Python program. The input file must immediately follow the -i tag.
- **-o :** Generates the AST corresponding to the input code in out.pdf
- **-v :** Uses YYDEBUG to show the states of the stack maintained for LALR Parsing
- **-h :** Instruction regarding usage instructions and options

By default, the program does not have an input file. The -o tag by default creates out.pdf. A sample execution example is provided below:

```
$ cd src

$ make

$ ./slytherin.o -i ../tests/test1.py -o

$ make clean
```

File Structure

- All the source files are in the milestone1/src directory.
- flex.l contains the lexer implementation in flex
- parser.y contains the paser implementation in Bison
- node.cpp contains the parse tree implementation
- The five test cases are in a directory milestone1/tests. The testcases are named as “test¡serialnumber¿.py”.
- The submission includes a PDF file under the milestone1/doc directory.

Compiler Enhancements

0.1 Symbol Table Data Structure

We have implemented a hierarchical two-level symbol table structure comprising:

- A global symbol table mapping function names to their respective local symbol tables.
- Local symbol tables for each function containing information about parameters and local variables.

0.2 Semantic Analysis

Our semantic analysis implementation performs the following checks:

- Ensure variables are used within the scope of their declarations.
- Verify type correctness in computations involving variables.
- Match actual and formal arguments of functions.

0.3 Generate 3AC

We generate three-address code (3AC) for syntactically and semantically correct programs. We've defined 3AC instructions corresponding to various Python features, including function calls, parameter passing, and return statements. The 3AC is stored in an in-memory data structure for potential optimizations during code generation.

Output

For correct input programs, our compiler produces the following outputs:

1. A dump of the symbol table of each function as a CSV file, containing columns for syntactic category, lexeme, type, and line number.
2. A dump of the 3AC of the functions in text format.

For erroneous input programs, meaningful error messages are provided to identify the cause of rejection. Error handling includes type errors, scoping errors, and non-context-free restrictions on the language.

Conclusion

In milestone 2, we have successfully extended our compiler to support symbol table management, semantic analysis, and 3AC generation. These enhancements lay the foundation for generating x86_64 assembly code in the next milestone.