CS335: COMPILER DESIGN

Milestone 2

HDS PYTHON COMPILER

Harsh Bihany (210406)
Danish Mehmood (210297)
Siddhant Suresh Jakhotiya (211030)

April 2, 2024

1 Running the files

The implementation of Parser and Lexer is in Flex+Bison, and is written in C++. Further, DOT is used to create a graphical representation of the AST and exporting it to a PDF file. The symbol table and the 3AC code have been output in `.csv` and `.txt` files respectivily. User should install all the above (if not already installed) by running the following commands:

```
$ sudo apt-get update
$ sudo apt-get install flex
$ sudo apt-get install bison
$ sudo apt-get install graphviz
```

A script file `pyrun.sh` has been made to help run the program against a test case. The command line options provided by our program are as follows:

- `-i` or `-input`: Filename from which the compiler reads the Python program.
- `-o` or `-output`: Filename to which the compiler outputs the `DOT` file.
- `-v` or `-verbose`: Prints additional checkpoints to show progress.
- `-h` or `-help`: Shows the usage for the script.
- `-c` or `-clean`: Performs `make`, with given options (if any) and then performs `makeclean`.

Each of the commands mentioned above is optional and can be utilized in any sequence. The script takes `../tests/test1.py` as default input and `graph.pdf`, `symbol_table.csv`, `three_address_code.txt` as the default output for the graph, symbol_table and 3AC respectivily , does not print additional messages and does not perform `make clean` after execution.

An example of a successful default execution is as follows:

```
$ ./pyrun.sh
```

An example of successful execution using the command line options provided above against the test case `test3.py`, the graph output being redirected to `graph1.pdf` and with additional debug messages, is as follows:

```
$ ./pyrun.sh -input ../tests/test3.py -o graph1.pdf -verbose -c
```

2 Codebase

The codebase can be found at our GitLab ID `hds-cs335`, inside the repository `python-compiler-2024`. The whole codebase is inside the directory `milestone2` which is on the `main` branch of the repository mentioned before. The file structure of the directory `milestone1` is as follows:

```
milestone2/
|-- src/
|   |-- include
|       |-- _3AC.hpp
|       |-- node.hpp
|       `-- symbol_table.hpp
|   |-- Makefile
|   |-- main.cpp
|   |-- _3AC.cpp
|   |-- symbol_table.cpp
|   |-- node.cpp
|   |-- pylex.l
|   |-- pyparse.y
|   `-- pyrun.sh
|-- tests/
|   |-- test1.py
|   |-- test2.py
|   |-- test3.py
|   |-- test4.py
|   `-- test5.py
`-- doc/
    `-- hds_milestone2_report.pdf
```

The same file structure has also been following in the zip file uploaded to **Canvas**.

Description for different files is as follows:

- **pylex.l** : This `Flex` file houses the code for the lexical scanner. It scans the input file and passes the `tokens` to the `parser`. It also keeps track of the line number for error reporting.
- **pyparse.y** : This `Bison` file contains the code for the parser. It drives the lexer and generates the `Abstract Syntax Tree`.
- **pyrun.sh** : This file is the `script` that helps run the compiler against a testcase. It also implements the command line options `-input`, `-output`, `-verbose` and `-help`. It concludes with a `make` command that executes the `Makefile`, which then runs the program with the supplied arguments to generate the output PDF.
- **main.cpp** : This `C++` file has the code to drive the `parser` and generate the `dot` script.
- **_3AC.cpp/_3AC.hpp** : These files contain the definition of the `struct 3AC` and also include the global **IR** vector. Generated **3AC** codes are added to this vector, which is then used to produce the final IR output file.

- **symbol_table.cpp/symbol_table.hpp** : These files contain the definitions for the `struct symbol_table`, symbol table entries, and other necessary fields for the symbol_table. This setup aids in creating the global symbol table.
- **node.cpp / node.hpp** : These files house the function definitions of `struct node` methods and functions for `AST` generation.
- Additionally, the files `test1.py`, `test2.py`, `test3.py`, `test4.py`, `test5.py` are the test cases provided by us (these were not explicitly asked for in the problem PDF).

3 Symbol Table creation

The construction of the symbol table is being done while the Abstract Syntax Tree is produced from parsing the program (`milestone1`). For accurate symbol table construction, it's crucial to verify that each entry added is unique where required (such as variables within the same scope), is both declared prior to use, and possesses the appropriate datatype as stipulated by the grammar.

The following utility functions have been developed for symbol tables:

- `void add_entry(st_entry* entry)`: Inserts a new entry into the symbol table that is presently in scope.
- `st_entry* get_entry(string name)`: Looks for an entry within the currently scoped symbol table, traversing up through the symbol table hierarchy as necessary until the entry is located.
- `int delete_entry(string name)`: Removes the specified entry from the symbol table that is currently in scope.

A symbol table is generated every time a new scope is entered, such as when defining a class, a function. This symbol table compiles a comprehensive list of identifiers and maintains a hierarchy of child symbol tables for nested scopes. Its attributes include details about the scope level, the count of nested scopes, and a reference to its parent symbol table, if any. Moreover, the global symbol table encompasses all class definitions; each class symbol table, in turn, includes all its methods, and a method's symbol table contains all its parameter definitions.

3.1 Scope Hierarchy handling

The scope of every new symbol table is determined by its parent scope. Lets understand how we have handled scope hierarchy using the following example program:

```
global_var: str = "Global"

class A():
    def __init__(self, a:int, b:int):
        self.sum: int = a + b
        self.prod: int = a * b
    def foo(self, x: int) -> int:
        self.x = x
        return self.prod + self.sum - x

def bar(a:int) -> int:
    return a**3
```

In the given example program, a global symbol table is constructed containing entries for `class A` and the function `bar`. These entries are linked to new symbol tables dedicated to each. Specifically, the symbol table for `class A` includes entries for the data members `self.sum`, `self.prod`, `self.x`, the `__init__` function, and the `foo` function. It's important to highlight that our compiler is designed to manage scopes beyond just the `__init__` function but instead it can handle any function, showcasing an additional capability we've integrated. Furthermore, we've established a structured order within the class symbol table: starting with self-referential code, followed by functions, arranged according to their appearance in the code. The `__init__` and `foo` functions within `class A`'s symbol table direct to further symbol tables, which catalog parameters and local variables for these functions. Likewise, the global symbol table entry for the function bar connects to its own symbol table, listing parameters and local variables specific to `bar`.

3.2 Type Checking

Type checking is conducted concurrently with the generation of 3AC (Three Address Code), which is detailed in the following section. Additionally, we provide support for coercion (implicit type casting) between integer and float, as well as between integer and boolean. For instance, if a function expects an integer as a formal argument but receives a float as the actual argument, a 3AC code snippet for coercion is automatically generated.

4 Generating 3AC IR

The quadruple data structure is utilized to generate the 3AC (Three Address Code) representation. By performing a depth-first traversal on the Abstract Syntax Tree (AST), code for the child nodes is generated prior to that of the parent node. To enhance the readability of the 3AC, relative referencing is employed in code generation, with absolute line numbers being printed instead of using labels. Following an incremental translation approach, the quadruple produced for each node is added to the global Intermediate Representation (IR) vector.

Below is a description of the various types of Three Address Code (3AC) instructions:

- `Q_UNARY`: Generates the 3AC for unary operations such as unary minus (-), unary plus (+), and bitwise NOT ($\sim$).
- `Q_BINARY`: Generates the 3AC for arithmetic, boolean, and relational operations.
- `Q_COERCION`: Produces 3AC for implicit type casting, useful in evaluating expressions with compatible data types. This includes type conversion features.
- `Q_ASSIGN`: Generates the 3AC for assignment statements.
- `Q_DEREFERENCE`: Creates 3AC for the dereference operator, beneficial for operations like array referencing.
- `Q_PRINT`: Generates the 3AC for the 'print' function.
- `Q_JUMP`: Produces 3AC for unconditional jumps, such as goto statements.
- `Q_COND_JUMP`: Generates the 3AC for conditional jumps, used in if-else blocks and loops.
- `Q_RETURN`: Creates 3AC for return statements from functions.
- `Q_FUNC_CALL`: Generates the 3AC for function calls, for example, ‘call(p, n)‘.
- `Q_PUSH_PARAM`: Produces 3AC for pushing parameters required by a function.
- `Q_POP_PARAM`: Generates the 3AC necessary for a function to pop its parameters.
- `Q_ALLOC`: Creates 3AC for the pseudo-instruction of allocating memory.
- `Q_BLANK`: Generates a blank 3AC code, mainly for beautification purposes.

5 References

- [DRAGv2] A. Aho, R. Sethi, and J. Ullman. Compilers: Principles, Techniques, and Tools, 2nd edition pdf