CS 335 Semester 2023–2024-II: Milestone 2

A. Atulya Sundaram, 210001
Anuj Singhal, 210166
Goutam Das, 210394

Compilation and Execution Instructions

The `milestone1` directory has 3 subdirectories

- `src`: containing the project's source files, i.e. the scanner(`lex.l`), parser(`parser.y`) and a `makefile`.
  To compile the code, you can simply use the given `makefile`, make sure that `flex`, `bison` and `g++` are installed on the system

  ```
  $ make all
  ```

- `doc`: Includes documentation of the project (i.e. this file)

The implementation has been tested to successfully compile and execute on

- `flex 2.6.4`
- `bison 3.8.2`
- `g++ 11.4`

`make` command creates an executable named `parser`, to parse a file using parser, the usage instructions are as follows:

```
USAGE:
[-input <path-to-inputfile>]: specify the file path to parse, if not given, takes
input from stdin
[-csv <path-to-outputfile>]: specify the file path to write the symbol table csv, if not
given, defaults to symtab.csv
[-tac <path-to-outputfile>]: specify the file path to write the 3AC code, if not
given, defaults to 3AC.txt
[-verbose]: prints verbose messages to stdout, error messages are anyways printed
[-help]: print usage instructions
```

An example command to run parser on file named `test1.py`:

```
$ ./parser -input test1.py
```

Symbol Table

The symbol table is generated in an CSV file named `sym_table.csv` in the current directory. For every testcase file, we create a new `sym_table.csv`, overwriting the old content if any. The symbol table contains the symbol table entries of each function in the file to be compiled. The entries are in the following format:

```
Class: <class-name>
Function: <function-name>
Lexeme, Type, Token, LineNo, Size
<entry-lexeme>, <entry-type>, <entry-token>, <entry-lineno>, <entry-size>
```

- `<function-name>` is of the form `<parent_name>.<function_name>@<param1_type>,..` where `<class_name>` is added in case the function is a method of a class, else it will be `global`, `<function_name>` if the name of the function in the program, and `<param[i]_type` represents the type of the parameter that the function takes in as input. Note that the class is an implicit parameter in a method, thus we will also have `<class_name>` as the first parameter.. In case takes no parameters, we will have it of the form `<parent_name>.<function_name>`.
- `entry-lexeme`: Lexeme of symbol table entry.
- `entry-type`: Type of symbol table entry. It can be `int`, `float`, `list[int]`, `list[float]`, `list[str]`, `list[<class>]` where `<class>` is a class of the program
- `entry-token`: Token that is used to define the entry, in general it will be `NAME`, as all the variable used in a function use that token.
- `<entry-lineno>`: Line number where the entry is declared.
- `entry-size`: Size of the entry

For example, the function:

```
1.    class A():
2.
3.
4.    def hello()->int:
5.        z:int = 100
```

will have an entry in the CSV of the type:

```
Class: A
Function: A.hello@A
Lexeme, Type, Token, LineNo, Size
z, int, NAME, 5, 4
```

Error Handling

In case of errors, the error message is printed to stderr, the abstract syntax tree is generated from the token seen till now but it may contain extra, disconnected nodes for due to incomplete productions.
The syntax errors reported by the parser are as follows.

- In case of an invalid token which is not in the language, the scanner can report the error as follows:

  ```
  Error: Unrecognized token <token> at line <line_number>
  ```

- In case of a syntax error, the parser reports the error along with the token at which the error was detected as follows

  ```
  SYNTAX ERROR: Unexpected token: <token> at line <line_number>
  ```

  Also based on the type of token, it can print the corresponding token type as well like

  ```
  SYNTAX ERROR: Unexpected string literal: <string> at line <line_number>
  OR
  SYNTAX ERROR: Unexpected indent at line <line_number>
  ```

We have also handled errors in the semantics of the program. Each of these errors report the line number of the error in the beginning of the form:

```
Error at line no: <line_number>
```

The errors handled in our semantic checks are:

- In case the variable has been redeclared:

  ```
  variable <var_name> already declared at line no <prev_line_no>
  ```

- In case of type mismatch during an operation `<op>`, with types `<type_1>` and `<type_2>`:

  ```
  TYPE_ERROR: Incompatible types <type_1> and <type_2< at "<op>" at line no <line_no>
  ```

- In case the function is redeclared, we get the following error:

  ```
  Error at line no <line_no>: can't declare function in this scope again.
  function <func_name> already declared at line no <prev_line_no> with same parameter
  types
  ```

- In case the variable has not been declared before use:

  ```
  Variable <var_name> is not declared before use
  ```

- In case the return type of a function does not match with the return type it was declared with:

  `Return type of function <func_name> is <curr_type> but actual return type <actual_type>`

- In case of an invalid unary operation `op` on a variable:

  `Invalid Unary Operation <op> with type <type>`

- In case function is called with incorrect parameter types:

  `TYPE_ERROR: No declaration of function <func_name> with these parameter types <line_nc`

- All form of type errors are handled in the form of an error with the prefix, `TYPE ERROR:`. As there are numberous cases handled, we shall omit mentioning them.

- Scoping errors have also been handled, with a few mentioned above

3AC IR Generation

We have handled the generation of the 3AC along with the parsing and thus generate it in the first scan. The 3AC generated is dumped in a text file called `3AC.txt` for each program, overwriting previous content of the file in case it already exists.

The `3AC` code contains the following forms, note that `x,y and z` can be either variables used in the program or temporaries of the form `#t_i`:

- `beginfunc <function_name>` : Beginning of the function with `<function_name>` following the format prescribe above.

- `x = popparam` : Pop a parameter entry in a function from the stack and assign it to a `x`.

- `x = y <op> z` : Operating on `y` and `z` with the variable `<op>`, which can be an arithmetic operator or a relational one.

- `param x` : Push the parameter `x` onto the stack.

- `call <func_name> <no_of_param>` : call the function `<func_name>` which contains the number of paramters, which have been pushed into the stack by `param`.

- `*(x) = y` : assigning the value of `y` to the address of `x`.

- `x = *(y)` : dereferencing `y` which is a pointer.

- `if_false x goto @Labeli` : Go to the label with number `i` if `x` is false

- `goto @Labeli` : Unconditionally go to the label with number `i`

- `label:@Labeli` : Form for the label with number `i`

- `shiftpointer +x`: Shift the pointer of the stack up by x bytes (before calling the function and pushing in parameters.

- `shiftpointer -y` : Shift the pointer of the stack down by y bytes (used after calling the function).