CS335: Milestone 2

Aman 210104   Aniket Suhas Borkar 210135   Siddharth Kalra 211002

31st March 2024

1 Compilation and Execution Instructions

The lexical analyser and parser have been written in `C++` using flex and bison respectively. In order to automate the compilation process, a Makefile has been created. The code is structured as follows:

- `lexer.l` contains the flex specification.
- `parser.y` contains the bison grammar rules.
- `3ac.h`, `symbol_table.h` and `symbol_table.cpp`, `3ac.cpp` contain helper function declarations and definitions respectively, and are included in both the flex and bison specifications.
- `Makefile`

The code makes use of the `C++` STL extensively by including the `<bits/stdc++.h>` header file, and must be compiled using `g++`. The parser supports the following execution options:

- `-input <filename>`: This option is used to give the input `.py` file to be compiled. By default, the input is read from `stdin`.
- `-output <filename>`: This option is used to specify the file to which the 3AC is to be stored. By default, the output is saved to a file named `"tac.txt"`.
- `-verbose`: This option prints all the steps of the parsing process to `stderr` and provides debugging information.
- `-help`: This option prints out the usage instructions.

In short, to run the parser on a file `"trial.py"` and generate the 3AC, the following commands must be executed:

```
make
./parser -input trial.py -output tac.txt
```

2 Type checking

- We have allowed implicit type conversion and type checking in following cases:
  - Initialization or assignment of a variable/list.
  - Parameters passed to a function.
  - Value returned from a function.
  - Expression evaluation.
- In these cases, the only type conversions allowed are from `bool` to `int`, `int` to `bool`, `float` to `int`, and `int` to `float`.
- A function call is allowed only if each of the actual parameters are convertible to formal parameter types.
- Expression formed from division operator will always have ”float” as the inferred type, regardless of type of operands (given they are valid).
- In expressions, only upcasting is allowed, similar to standard python rules.
- Following are the implementation details for type checking:
  - Symbol table entry has a field ”type” which stores type of variable/function.
  - For functions, its type is a vector of types of its parameters. Return type is also stored for functions.
  - Each node in AST stores the type inferred for the corresponding expression
  - For each expression in grammar, we first check whether the types are compatible for the operator (that is, convertible to the types allowed by operator).
  - Type of the expression is then inferred based on standard rules.
  - If type conversion is needed, 3AC code for type conversion for the corresponding temporaries are also generated.

3 Symbol Table

We've implemented a hierarchical system for managing symbol tables, utilizing a global symbol table alongside individual local symbol tables for each class and function. These symbol tables are stored as CSV files and contain crucial information pertaining to variable, function, and class declarations within them. Specifically, each entry in the CSV file includes the Lexeme, Category, Type, Line number, and Return type for function declarations. These CSV files are dumped in the same folder as the executable.

- We are creating a new symbol table for each new scope - a class or a function.
- For `if-else`, `for` and `while` loop blocks, we have not created a new symbol table. We are using the scope of their parent as their scope.
- For every declaration, we are checking whether the symbol (variable, class, or function) is previously declared or not. We throw an error if it is already declared.

- We are storing following information for each declaration : Lexeme, Category, Type, Line number, Width, Offset in symbol table, return type for functions, dimensions for lists and other important fields for implementing specific functionalities.
- For uses of variables, functions and Classes, we are checking whether the symbol is previously declared or not. For use without declaration, we are throwing error.

4 Three Address Code IR

We make the 3AC IR by utilising the information from the symbol tables in the same pass, i.e. we implement a single pass parser-type checker and IR generator. The 3AC generated is stored in-memory in a vector of quadruples. A quadruple contains information about the type of the 3AC instructions, its two operands and a target. Code snippets to generate 3AC instructions have been inserted at appropriate places in the semantic actions.

- Temporaries: The temporaries are generated with names of the form `tn` where `n` is a number. These are used whenever a name or literal is found and its rvalue is evaluated. Further, they are also used when evaluating expressions.
- Functions: When a function call is being made, the caller saves the caller saved registers using the `save registers` instruction, pushes the parameters on to the stack, makes the call, removes the return value from the stack, shrinks the stack to remove the activation record of the called function, and finally restores the caller saved registers using `restore registers`. At the callee side, we include `beginfunc x` and `endfunc` instructions at the start and end of the function body respectively. Here, `x` stands for the space required for storing the local variables of the function. The `beginfunc` instruction accordingly adjusts the stack pointer to make space for these locals, and also saves the callee saved registers. At function return, the callee reclaims the stack and restores the callee saved registers using the `leave` instruction. It stores the return value `x` using a `return x` and returns to the caller. All of these are pseudo-instructions to be replaced by their actual x86 implementations in the third milestone.
- Type Conversions: Whenever needed and allowed, type conversion instructions `cvt_*_to_*` are generated. Allowed coercions are `bool` to `int` and vice versa, `int` to `float` and vice versa. For lists, limited type coercion support has been provided, i.e. type casting is done only when a variable is assigned as a list in `annassign`.
- Augmented assignments: Instructions of the type `a += b` are broken into `a = a + b`, and the 3AC is generated accordingly.
- Lists: Lists have been implemented by allocating memory dynamically by calling the function `allocmem`, and then assigning the values to the memory locations individually. A variable of type list only stores the pointer to this memory location. Further, a higher dimensional list of dimension d is a list of pointers to lists of dimension $d - 1$. The first 8 bytes of the memory of a list store the size of the list. This is used to implement the `len()` functionality. Array access is done by dereferencing these pointers.

- Strings: At the current level of abstraction, strings have simply been indicated as a constant. Later on they could be replaced by pointers to strings stored in the `rodata` region.
- Loops and conditionals: `for` and `while` loops are supported along with the `else` constructs. `if-elif-else` blocks are also supported. Appropriate labels are generated where needed to allow conditional and unconditional jumps.

5 Wrapper Script

The submission also includes a wrapper script named `test.sh` in the `milestone2/scripts` directory that will create 3ACs and symbol table CSV dumps of the sample python codes present in `milestone2/tests`. All the outputs (including the CSV dumps) would be generated in the `milestone2/out` directory. Run the script using the following command from the `milestone2/scripts` directory:

```
cd milestone2/scripts
chmod +x test.sh
./test.sh
```

6 References

The reference for the testcase which has been taken from the internet is as follows:

1. Editorial of problem: ”Solve the Sudoku” in python for `test3.py`