Milestone-2

Pratham Sahu Aditya Bangar Abir Rajbongshi

February 2023

Contents

1 Running the Files

The Parser and Lexer are implemented using Flex and Bison, which are tools for generating scanners and parsers. This implementation is written in C++. Additionally, DOT is utilized for creating Abstract Syntax Trees (ASTs). If flex, bison or graphviz is not installed use the following commands.

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

A make file has been provided to compile and make the executable.

```
make
```

To run the executable, use the following command:

```
    ./pycomp
```

The following command line arguments can be used to run the executable:

```
    -- help : to display the help message
    -- verbose : to display the verbose output, on the terminal
    -- input : to specify the input file
    -- output : to specify the output dot file
    -- graph : to specify the output type for the AST, it will be
        of the same name as the output file. It can be either png
        or pdf.
```

The csv for symbol table and the 3AC code will be generated in the same directory as the input file.

2 Symbol Table and Scoping

The hierarchy of symbol tables are constructed by adding S and L attributed translations to the grammar rules. We maintain a stack of symbol tables which defines the current scope while parsing. This allows us to enter and exit scopes as we encounter new scopes in the code, and enter variables into the correct scopes. We have implemented the following features in the following way:

- **Global Symbol Table**: We maintain a global symbol table which contains the global variables and functions. This is the first symbol table that is created when the parser is initialized. It also contains the symbol tables for all the classes in the program.
- **Class Symbol Table**: We built the symbol table for classes by adding a new class to the symbol table stack. This class contains the name of the class and a map of the variables in the class. It also contains as entries the functions in the class. We also maintain a pointer to the parent class, so that we can inherit the variables and functions from the parent class. We maintain the pointers of descendant classes as well.
- **Function Symbol Table**: We built the symbol table for functions by adding a new function to the symbol table stack. This symbol table contains the name of the function, the return type of the function, the arguements of the function and the local variables of the function. The parent of this symbol table is the class symbol table in which the function is defined.
- **Local Symbol Table**: We build a local symbol table for control flow elements like for, while, if. This allows us to get a greater control of the scopes. This also helps us implement nesting of scopes. The variables in the scopes are merged when the scope is exited.
- **Merging of Local Symbol Tables**: When a scope is exited, we merge the local symbol table with the parent symbol table. This allows us to access the variables in the parent scope.
- **Symbol Table Lookup**: During symbol table lookup, we first get the current scope using the symbol table stack we were maintaining. We then walk this upwards towards its parents making the assumption that scopes are inherited in a parent-child relationship.

3 Type Checking

We perform the following type checks in our implementation: The symbol table is queried while performing the type checks. All this check is done during the parsing

- Declare before use of variables. During the initialization of a variable, it is stored in the symbol table, and every time we use a variable, we perform a symbol table lookup to check if it has been declared.
- Type checking of expressions. While using operators, we check if the types of the operands are compatible.
- Type checking in relational operators. While using relational operators, we check for compatibility. We have written functions for the same inside src/symbol_table.cpp. The function names should be explanatory.
- Type checking in assignment statements. We check that variables, array references are assigned to the correct type.
- Checking return type of functions. We check that the return type of the function is the same as the type of the expression being returned.
- Checking the number of arguements in function calls
- Checking the type of arguements in function calls
- We perform merging of types in case of binary operations, assignments if they are compatible
- During array dereferencing, we check if the index is of integer type. We also check wether it in the bounds of the array, in case it is a constant literal.

4 Generating IR - 3AC

We use quadruple to generate the intermediate representation. This is then converted into printable triplet form for the purpose of output. We have implemented the following features in the following way:

- For literals and identifiers, we just allot the addr of the node to be the value of the literal or the identifier. We do not create any new temporary variables for them.
- For if-then-elif-else statements, we generate labels for the if, elif and else statements and jump to the appropriate label based on the condition.
- For while loops, we generate labels for the start and end of the loop and jump to the start of the loop based on the condition.

- For function calls, we generate a label for the function and jump to the function label.
- In the case of BREAK and CONTINUE statements, we generate labels for the start and end of the loop and jump to the appropriate label. This is done scope wise, and hence it can also support nesting of for loops.
- For array accesses, we first get the address of the current index and then dereference it into a temporary.
- For array declerations, we allocate space for the array and store the base address of the array in the entry pointed by the symbol table.
- For function doclerations, we add a begin_func and end_func pseudo instruction to mark the start and end of the function.
- For unary functions, we generate a temporary variable and store the result of the unary operation in the temporary variable.
- For binary functions, we generate a temporary variable and store the result of the binary operation in the temporary variable.
- For augmented assignments, we convert it into a binary operation and assignment.
- For function calls, we push the arguements on the stack and then call the function. After this, we pop the return values from the stack.

5 Runtime Environment

For this milestone we have implemented a simple form of activation records.

While calling a function, we allocate space for the following:

- Return Address(Assumed to be saved by the call instruction)
- rbp(Old rbp)(Assumed to be saved on every function call by the callee just after the call instruction)
- local variables of the callee function(we do this by incrementing the stack pointer inside the callee function).

The parameters are passed as is using a pseudo instruction call push_param which pushes the parameters on the stack, and we then manipulate the stack pointer accordingly.(Note: They are pushed in opposite order so that they can be popped in order by the callee function).

The arguements are used by the calle function by just using pop_param pseudo instruction which uses these arguements.

The return value is popped from the stack from the caller and is stored on the stack by the callee. The pseudo instruction leave is responsible for cleaning up the locals and restoring the old rbp value.

The return address for the callee is assumed to implicitly stored by the call function and the callee function also pushes the old rbp value on the stack.

The space for saved registers is architecture dependent and has not been allocated in our implementation to make it architecture independent.

In x86_64, 6 registers are used for passing the arguements, rest are passed via tha activation record. However, for simplicity, we assume that all arguements are passed via tha stack.