Formal Languages and Compilers

Nicoletti Alberto

a.a 2023

General explanation

The following project is based on the original calculator code provided during the labs, to which a set of additional peculiarities were added:

- numeric typing (integer and double)
- additional basic operations (apart from the four original ones)
- logic comparison
- if statements detection
- basic string support
- symbol table implementation
- console feedback

The code itself supports basic functions, although it does not cover each and every case extensively, therefore sometimes syntax errors may occur.

Code description

The code is composed of three files:

- **lex file:** which holds the code for the lexical analyser, that is in charge of reading the input, recognising the tokens and eventually assign a value to a token if specified (for example numbers)
- **yacc file:** which specifies how a specific token is used, allowing to execute different code based on the input received from the lexical analyser
- **c header file:** that contains the actual code for implementing and running the symbol table as well as other additional functions (comparison/conditionals)

Although there is some basic code included in the `.y` file, most of the code is contained in the `.h` file for enhanced readability in both files.

YACC productions

The `.y` file comprises several productions that specify how the tokens produced by the lexical analyser should be handled.

`line`

The line production is the start of the whole parse tree, it is merely in charge of keeping an active input feed by recursively referencing itself. It also handles the safe termination of the program, triggered by the keyword `quit`.

`stmt`

The statement production is in charge of sorting the current input to the correct action one would like to perform, whether this is a conditional statement, an assignment or an actual expression to be evaluated. Additionally it also handles printing, meaning that the user can type the keyword `print` to print the whole symbol table, or by specifying a variable name it can print the data of that exact node. Furthermore the keyword `type` allows one to print the type of a given variable.

`ass`

The assignment statement is in charge of executing the whole assignment operations, whether these are simple assignments, expression assignments (where the value of the expression has to be computed beforehand) or shorthand assignments (combining the two aforementioned types in a single one). This production is supported by the `type` and `shorthand` statements which are simply in charge of specifying respectively the type and shorthand operator declared. Each line of these statements makes a simple call to the `.h` file where the functions are specified, this is mainly because otherwise the `parser.y` file would lose a lot of readability because these function behave differently depending on the different values of the parts of the assignment statement. All statements return a variable node, that way it is possible to access the variable specified (and its values), for more information see `val`.

`cond`

The conditional statement simply computes basic comparison operations, like greater/lesser or equality checks.

`expr`

The expression statement is in charge of computing values of the given variables, once more the lines of this production make a call to the `.h` file where the actual behaviour is specified for the very same reasons that were specified for the `ass` production, since here too it is necessary to determine the types of the inputs and act accordingly to them. This statements also supports increment and decrement operations

`val`

This is a simple middle-point in the parsing tree for when evaluating expression values (that is, converting strings to values of certain types, them being strings, double or integer). Furthermore this method references the `ass` production in order to extrapolate the value contained in a symbol table node.

`ifstmt`
Very simple if-statement implementation that simply prints the content of the body when the given condition is true.

Input description

There are a few reserved words for this program:

- `int` for specifying the integer type
- `double` for specifying the double type
- `string` for specifying the string type*
- `if-then` for declaring an if-then clause
- `type` followed by an `id` prints the type of the given variable
- `print` prints the whole symbol table or, if followed by an `id`, prints a specific variable
- `quit` ends the input session

variable assignment

The assignment statement looks as follows:

```
<type> <id> <operator> <expression>
```

The code accepts different formats for declaring a variable:

- `type` specifies the type of the variable, if not declared the program is able to infer it from the value of the expression provided, if any. If it does not match the type of the expression, the program is able to cast (in case of integer-double) the value or otherwise throws an error;
- `id` stands for a variable name which is always mandatory;
- `operator` the operator may have two forms, either it is a simple `=` which is translated in a simple assignment, or it might be a shorthand operator (like `*=` or `+=`), in that case the value of the node is then extrapolated, processed and then reassigned to the variable itself. In case a shorthand is

used on an uninitialised node, it is treated as a simple assignment instead. `expression` is the value to be assigned to the specified `id`, if the node already has a type, then type check (and eventual conversion) is performed. In case no type is already defined, the type of the expression is then assigned to the blank node. The expression may be another variable or an operation between variables as well.

Examples
The following are valid statements

```
val
val = 7
int val = 9.0 //warning, this performs a type-cast
int val = 9 + val1
```

Examples of not valid statements are

```
int val = "hello"
```

print statement

The print statement is written as follows:

```
print <id>
```

As already specified before, the `id` is an optional parameter, in case it is omitted the entire symbol table (which can hold up to 64 symbols) is printed in an easily readable layout.

The following is an example of the `print` command on the node `val` of value `9.0`.

```
print val
-----------------------------------------------
Node ID: val
Type Declared: yes
Value initialised: yes
Next node: NULL
Value: (Double value) 9.000000
-----------------------------------------------
```

How to run the program

1. Compile the `lexer.l` file using flex:

   ```
   flex -l lexer.l;
   ```

2. compile the parser file `parser.y` using yacc (or bison):

   ```
   yacc -vd parser.y
   ```

3. compile the outputted `y.tab.c` using gcc:

   ```
   gcc y.tab.c -ll
   ```

4. execute the produced output file `a.out`:

   ```
   ./a.out
   ```

And the code should run, on Windows machines may be easier to enable the Windows Subsystem Linux (WSL) to compile the code rather than setting up all the different tools manually in the default command line.
This can be done by either downloading WSL from Microsoft store or by enabling the virtualisation in the BIOS and then installing the desired Linux-distro(in my case was Ubuntu 20.04 LTS) from Microsoft Store