TP May 13, 2020

Formal Languages – Exercise on Syntactic Analysis

The files associated with this exercise can be found here:

`http://www.lsv.fr/~schwoon/enseignement/langages-formels/2020/`

Your answers to the questions will be marked. You can work in groups of two. Answers must be submitted until May 23 before midnight, in the form of an archive (zip or tgz) containing the following files:

- your `.l` file for Question 1;
- your `.y` and `.l` file for Question 2;
- possibly a `Readme` file if you have additional comments.

The mark of the exercise counts for a fifth of the overall mark.

In this exercise, we will use the programs *flex* for lexical analysis and *bison* for syntactic analysis. The two work in a similar manner : the user defines rules that are then translated into C code.

As a running example, we will use the following grammar with terminals **int**, `(`, `)`, verb-+-, `*`, which treats simple arithmetic expressions:

$$E \to E + T \mid T \qquad T \to T * F \mid F \qquad F \to (E) \mid \textbf{int}$$

1 Lexical analysis

An exhaustive documentation of *flex* can be found on line:

`https://westes.github.io/flex/manual/`

or on your local machine using `info flex`. While this document explains the most basic aspects, it is recommended to go to the above sources for more detailed explanations; the most useful sections will be from 4 to 8, as well as 15.

Note : the examples of this section can be found in the directory `lexical`.

Generally, an input file for flex consists of three sections separated by a line containing two percent signs:

```
Definitions
%%
Rules
%%
Code
```

The Rules consist of regular expressions associated with blocks of C code. The C code is executed whenever an instance of the associated regular expression was found in the input. If the input matches multiple choices, flex always chooses the longest match possible, and gives priority to expressions further up in the list. Parts of the input that do not match any regular expression are simply copied to standard output.

The Definition section can serve to define shortcuts for regular expressions, which can make them more readable. One can also include additional C code, delimited between markers `%{` and `%}`, e.g. to define global variables or functions.

Finally, the last section can contain additional C code. Technically, the Rules section is translated into a function `yylex`, which can be called repeatedly to scan part of the input; it will return whenever the code associated with your regular expressions tells it to, or at the end of the input. By default, `yylex` reads from standard input, but it can also be made to read from files or strings, see the examples.

Consider the file `example.l` which extracts the terminals for our running example. On the command line, it suffices to `make example.c` to produce the corresponding C code, or simply `make example` to do that, and compile that program at the same time. The programme accepts input from standard input or a file. The file `string.l` shows how to read from strings instead. Finally, the example `flux.l` shows how to associate additional informations with the extracted terminals. (*bison* will provide a similar, but more sophisticated interface for this purpose.)

Question 1. Modify `exemple.l` to provide the following extensions.

1. Currently, non-treated characters are copied to the screen. Find a way to stop this from happening.

2. If the input fontains the keywords `if`, `then` or `else`, they should be printed in capital letters.

3. Add code to count the number of lines in the text, and print that number to the screen when the program ends.

4. Add the possibility to specify numbers in hexadecimal, e.g., `0x2a` for 42. The function `strtol` can be useful for this.

Flex can be used to provide a flow of terminals towards another function. For this, one includes `return` statements in the C code. When `yylex` returns 0, this signifies 'end of input', whereas positive values are reserved for terminals. This mechanism is exploited by bison.

2 Syntactic analysis

Bison provides an LALR parser and is made to interface with flex. An exhaustive documentation can be found here:
`https://www.gnu.org/software/bison/manual/html_node/index.html`
or on your machines with `info bison`.
Note : The examples for this section are found in the directory `syntaxique`.

A first example. The first example consists of files `expr.y` and `exprlex.l` (compile using `make exprlex.c` plus `make expr`). As before, the executable accepts standard input, or a file name given as parameter.

An input file for bison (such as `expr.y`) contains three sections, mostly as in flex, but where the Rules section defines a grammar. In our example, one finds a simplified grammar for expressions. The keyword `%union` defines the files of a variable `yylval` which the lexical analyser can use to attach additional information to terminals (in bison lingo, terminals are called *tokens*). Moreover, an instance of `yylval` can be associated with every non-terminal. The declarations starting with `%token` and `%type` tells the parser which terminals and non-terminals are using which field of the union (in the example, there is only one integer type). Individual characters are treated as tokens even without explicit declaration.

Similar to flex, every production can be associated with a piece of code, which is to be executed when the parser *reduces* that rule. Since the parsing is bottom-up, the `yylval` values for the right-hand side of a production are already known when the reduction happens. This makes it easy to construct the result of the entire syntax tree, as in this example for integer expressions:

```
E : E '*' E       { $$ = $1*$3; }
  | E '+' E       { $$ = $1+$3; }
  | '(' E ')'     { $$ = $2; }
  | INT
```

In the bits of C code, `$$` signifies the value associated with the left-hand side non-terminal; thanks to previous definition, the parser knowns that its type is an integer. The expressions `$1` etc recover the values associated with the first, second etc symbol on the right-hand side. If no C code is provided, the standard action is `$$ = $1;`, which is the case for the last production in our example.

Note: A common beginner's mistake is to leave the code section empty when one wants the compiler to do nothing at all during a reduction. However, the C compiler may then complain when the types of `$$` and `$1` do not match. In these cases it is better to explicitly provide an empty piece of code (`{ }`).

Fixing conflicts. Play around with the example. You will notice that the usual priority between multiplication and addition is not respected, thanks to our "simplified" grammar. At compile time, bison actually complains about multiple shift/reduce conflicts. When faced with such a problem, it is useful to inspect the state table of the parser, which can be produced using `bison -v expr.y`; this produces a file called `expr.output`. Conflicts are clearly marked in that file; in this case one would see that items like $E \to E + E.$ and $E \to E * E.$ cause shift/reduce conflicts for both `+` and `*`, which are arbitrarily resolved by shifting. What this means is that an expression of the form $E \circ_1 E \circ_2 E$ is evaluated as $E \circ_1 (E \circ_2 E)$. It should be clear that in some cases, this is either inconsequential or in fact the intended semantics, but in other cases it is not. The upshot of this is that conflicts in your grammars should *always* be treated as errors and be eliminated. Bison provides multiple ways to do this.

The first method to fix those conflicts is to modify the grammar along the lines of our running example, see page 1. This works but quickly creates overly complicated grammars (imagine we treated not 2 but 20 operators...). For the case of arithmetic and logical expressions, there exists a tailored solution to guide bison in correctly resolving those conflicts: one can add the following lines to the Definitions:

```
%left '+'
%left '*'
```

This declares both operators to be left-associative, and the order signifies that multiplication binds more strongly than addition. (Recompile and verify the state table to check that it works as intended!)

Finally, we remark that `exprlex.l` now supplies the terminals declared in `expr.y` (such as `INT`) and uses `yylval` to provide additional attributes. In fact, bison compiles its grammar into a function called `yyparse`, which continually calls `yylex` to obtain the next terminal. Thus, `exprlex.c` is included by `expr.y` to provide it with the `yylex` function.

A small programming language. The files `lang.y` and `langlex.l` contain an interpreter of a very simple programming language with boolean variables (compile it as before). The file `counter.my` gives an example of that language. Your task is to extend the language in several ways. For most of these tasks, one must extend multiple things: the lexical analyser, the grammar, and the associated interpreter.

Question 2.

1. Add a boolean equivalene operator (`<=>`), with the right priority.
2. Add a control structure `if/then/else/fi` where the `else` part is optional. How is the 'dangling else' problem treated? (Look up the problem on Wikipedia if you haven't heard of it before.)
3. Add integer variables to the languages. This comprises several subtasks:
   (a) Integer variables should be declared analogously to booleans with a list like this: `int a,b,c;`. A program may have only integer variables, or only boolean variables, or both, or none of them.
   (b) Add instructions of the form `a := expr`, where `a` is an integer variable and `expr` an integer expression with at least multiplication, addition, and parantheses.
   (c) Extend the boolean expressions with comparisons (at least `==` and `<`).
   (d) `print` should accept both types of variables.
   (e) A small type inference system will be needed to check that variables used in expressions are of the right type.

Additional remarks Shift/reduce and reduce/reduce conflicts must be eliminated from the grammar.

The addition of the integers is clearly the main part of the work, and it requires to carefully think about the data structures that you are going to use. Nonetheless, as an indication, the entire solution can be obtained by adding no more than 2K of code overall, or around 60 lines of code.