The Creation of PL-NEXT

Author: Callum France

EBNF

Using the provided syntax diagrams, EBNF rules were created. One EBNF rule corresponds to one syntax diagram, making 33 EBNF rules for PL-NEXT. These rules are saved in `ebnf_rules.ebnf`.

Lex

After the EBNF rules were created, I began to work on the Lex file. Inside the Lex header, I placed the C libraries that were required.
I then created the two most fundamental data types for PL-NEXT - the `number` and `ident`. Because none of the syntax diagrams needed to consider white space, at this point I also created a Lex rule to ignore all white space.

Despite not being necessitated by the syntax diagrams, I thought that for my own benefit I would add simple comment functionality to my PL-NEXT files. This would give me the option to document my code as I was writing it without needing a separate file (or pen and paper). This was done using a comment 'context' that switched when encountering `/*` and `*/`.

Using the simple regex command `.` which picks up the currently parsed symbol, I created a rule that transfers single symbols directly

into Yacc. The piece of code `return (int) yytext[0];` is retrieving the first symbol of `yytext` that is otherwise unmatched to another keyword.

Next, I created the PL-NEXT keywords. This was done via implicit regex strings in Lex. When each keyword is encountered by the parser, a return statement is given that corresponds to a Yacc token.

Yacc

Once I had created all my keywords in Lex, I began to implement my EBNF statements in Yacc. This required a bit of translation to convert into a form suitable for Yacc. Attention to detail was paid in order to use 'left recursive' statements as opposed to 'right recursive' statements. The left recursion minimizes the amount of items on the stack; if we were to use right recursion, all items would be pushed onto the stack until we get to the very last item, at which point we start popping items. Furthermore, some EBNF rules required 'sub-rules' to be implemented in Yacc. These sub-rules were created in Yacc to increase brevity and allowed repetition. For example, a `statements` rule had to be created in Yacc in order to allow either one or multiple statements to occur.

All of the Lex keywords that utilised return statements had to be registered Yacc tokens. These were declared in the Yacc header. Furthermore, the special `%left` token was used for the math symbols `* / + -` to indicate that if more than one of these symbols occur on the same line, the leftmost symbol gains execution priority. Furthermore,

since `* /` were placed on the line below `+ -`, they gained a higher priority, in line with “BIMDAS” form.

In order to display that a syntax error had occurred, the Yacc header declares `void yyerror(const char *s);`. This function is written in the footer to output the error message and notify the user an error has occurred.

Makefile

Once the Yacc and Lex files had been written, it was appropriate to create a makefile in order to compile both files together at once. The makefile creates the executable `PL-NEXT-SYNTAX`, which can be used to check a grammar file with the following command.

```
$./PL-NEXT-SYNTAX < correct1.next
```

where `correct1.next` is an example of a PL-NEXT grammar file. Note that although a file with the extension `*.next` is given, any filetype is accepted by the syntax checker. The makefile first compiles the lex and yacc files separately using `flex` and `yacc`. Next, `gcc` is used to create `PL-NEXT-SYNTAX`. Executing the make file is trivial -

```
$make
```

Another `clean` section was also created in the makefile. This removes

all files that were created from a `make` command. This is executed as -

```
$make clean
```

The combination of `make` and `make clean` allowed quick changes to the compiler. When a change was made, all I had to do was save the necessary files, delete the compiled parts with `make clean`, and then recreate them with `make`. This was a very effective strategy used to shorten the edit-delete-recompile cycle I often faced in this project.

Debugging

In order to ensure that my Lex and Yacc correctly reflected the provided syntax diagrams, example files were created with the `*.next` file extension. Files labelled `correctX.next` were created to be syntactically perfect, combining many of the EBNF rules correctly within one file. Files labelled `wrongX.next` each targeted one specific grammar outlined in Yacc.

This process lead to many bugfixes being picked up in my code, which needed to be amended in the Yacc and Lex files. The debugging process increased the reliability and standard of the syntax checker created.

Resources

- A guide on regex

    - https://www.rexegg.com/regex-quickstart.html
- Lex and Yacc tutorial
    - https://www.ibm.com/developerworks/aix/tutorials/au-lexyacc/index.html
- Yacc guide
    - https://docs.oracle.com/cd/E19504-01/802-5880/6i9k05dgt/index.html
- Example Yacc and Lex code
    - https://www.epaperpress.com/lexandyacc/calcl.html
    - https://www.epaperpress.com/lexandyacc/calcy.html
- Adding comment rules in Lex
    - https://stackoverflow.com/questions/2130097/difficulty-getting-c-style-comments-in-flex-lex
- Yacc declarations (`%`)
    - https://www.ibm.com/support/knowledgecenter/en/ssw_aix_