Example of Formalising a Grammar for use with Lex & Yacc

Here is a sample of a data-file that we want to try and recognise. It is a list of students and information about them.

```
CIS W7  Abramson,Paul B          CS3001 CS3071 CS3102 CS3132 CS3311 CS3322 CS3361 CS3900 EM2490
CE  X1  O'Rourke,Daniel M        CS3001 CS3041 CS3052 CS3071 CS3082 CS3111 CS3322 CS3900 EM2490
AI  Y6  Naismith,Gregory S       CS3001 CS3052 CS3071 CS3082 CS3311 CS3322 CS3361 CS3900 EM2490
CIS X4  Sanders,Alexander P      CS3001 CS3071 CS3102 CS3132 CS3311 CS3322 CS3361 CS3900 EM2490
SI  __  Varney,Samantha          CS3041 CS3052 CS3071 CS3082 CS3251 CS3311 CS3432 CS3900 EM2490
CS  W1  Smith,Mark               CS3001 CS3052 CS3071 CS3082 CS3212 CS3232 CS3241 CS3251 CS3900 EM2490
MI  I3  Cooper,Paul              CS3311 CS3561 CS3572 CS3902 MK3322 HM3111 HM3132 HM3142 HM3152 HM3121
MI  I4  Smythe,Helen Ruth        CS3561 CS3572 CS3902 MK3322 MK3311 DI3111 HM3132 HM3152 HM3121 HS3132
CE  Z5  Grant,Ellie              CS3052 CS3071 CS3082 CS3111 CS3311 CS3322 CS3361 CS3900 EM2490
```

For each student,
the first item is a school (e.g. `CIS`),
the next is a tutorial group (e.g. `W7`),
the next is the student's name (e.g. `Abramson,Paul B`)
and finally there is list of modules that they are taking (e.g. `CS3001 CS3071 CS3102 CS3132 CS3311 CS3322 CS3361 CS3900 EM2490`):

```
student : school group name modules
        ;
```

The whole file consists of one or more students:

```
%start file
%%

file    : student
        | file student
        ;
```

The list of modules can be empty (e.g. if they haven't registered yet) or contain any number of module names:

```
modules :
        | modules module
        ;
```

Alternatively, I could have written the grammar for student and modules so that the module list always contains at least one module name but a student description need not contain a list of modules:

```
student : school group name modules
        | school group name
        ;
modules : module
        | modules module
        ;
```

Similarly, I could have written the grammar to combine the rules for file and students:

```
file    : school group name modules
        | file school group name modules
        ;
```

However, neither of these two changes is likely to be a good idea, as when we come to write the associated actions (e.g. to build parse trees) we will end up writing the same code twice, e.g. once to deal with "school group name modules" at the start of a file and once to deal with the same information in "file school group name modules"
You should only do something like this if there is good reason e.g. if you needed to perform different actions for students who have chosen some modules ("school group name modules") and for students who haven't chosen any modules ("school group name"), rather than just having an empty (e.g. NULL) module list in the second case.

Another possible rewrite is to use right recursion instead of left recursion for file and/or modules (for this data it doesn't seem to matter, but I prefer to use left recursion with yacc to avoid any chance of the stack overflowing, and again you should have a good reason to use right recursion):

```
file    : student
        | student file
        ;
```

```
modules :
        | module modules
        ;
```

This is the whole of the yacc grammar, but we still need to write some lex rules to recognise various tokens:

```
%token school group module name
```

A school is some number of upper-case letters:

```
[A-Z]+                          {if(trace)ECHO; return school;}
```

("trace" is used to control whether all input characters are ECHOed to output, and needs to be declared and initialised.)

A group can be an upper-case letter followed by a digit, or two underline characters (e.g. if it is unknown because the student has not yet been put in a group):

```
[A-Z][0-9]                      {if(trace)ECHO; return group;}
"__"                            {if(trace)ECHO; return group;}
```

or we could have written this as:

```
([A-Z][0-9]|"__")               {if(trace)ECHO; return group;}
```

A module name is two upper-case letters, followed by 3 or 4 digits:

```
[A-Z][A-Z][0-9][0-9][0-9][0-9]  {if(trace)ECHO; return module;}
[A-Z][A-Z][0-9][0-9][0-9]       {if(trace)ECHO; return module;}
```

or we could have written this as:

```
[A-Z][A-Z][0-9][0-9][0-9][0-9]? {if(trace)ECHO; return module;}
```

or even as:

```
[A-Z][A-Z][0-9]{3,4}            {if(trace)ECHO; return module;}
```

As with the yacc, it simplifies the actions (especially when we come to e.g. build parse trees) if we have just one rule for module names or for group names. However, we may genuinely need to have different actions in the different cases - e.g. if we want to detect errors (see below).

A student's name starts with an upper-case letter, and then continues with upper- and lower-case letters, commas, spaces, hyphens and apostrophes.

```
[A-Z][-A-Za-z', ]*              {if(trace)ECHO; return name;}
```

As we have included spaces in characters accepted by this rule, we need to worry whether the rule will gobble up more than we expect, by using up any spaces after the name and then going on to read the list of modules. In fact the different fields are separated by tab characters rather than spaces for just this reason, so this isn't a problem.

As the rules we have written for student-names, school-names, group-names and module-names are all very similar, we need to pause for a moment to convince ourselves that each rule will only pick up what it is supposed to. (**You shouldn't need to do this for the Faust DFSA language, as I have deliberately chosen the different tokens to be easy to distinguish.**) In this example, although each token starts with [A-Z], the second or third character seems to be enough to distinguish most of them e.g. a group has a digit as its 2nd character, whereas the others have characters.

Unfortunately, the rule for a student's name could easily pick up the school-name as well:

```
[A-Z][-A-Za-z', ]*              {if(trace)ECHO; return name;}
[A-Z]+                          {if(trace)ECHO; return school;}
```

We get some help from flex, as it will warn us when one rule blocks another completely as above but it won't warn us if there is just some (unexpected) overlap between what rules might pick up.

We could fix the order of the two rules, so that the rule for school-name comes first and so takes precedence over that for student's name:

```
[A-Z]+                          {if(trace)ECHO; return school;}
[A-Z][-A-Za-z', ]*              {if(trace)ECHO; return name;}
```

Alternatively, and safer if we can achieve it, we can change the rules so there is no possibility of confusion. In this case, we can change the rule for a student's name to insist that the second character is either lower-case or an apostrophe:

```
[A-Z][a-z'][-A-Za-z, ]*         {if(trace)ECHO; return name;}
[A-Z]+                          {if(trace)ECHO; return school;}
```

Note that we might have introduced a source of errors here - we are implicitly assuming that no-one has a name consisting of a single letter. In fact, the closest we have come to a problem of this kind was a student with a 2-letter surname, but even he had a 4-letter first name, making 7 characters (including the comma) in total. (We have also had problems with students with absolutely identical names, for which there is no solution other than to use e.g. their registration number instead as a unique identifier.)

As usual, we discard white space and guard against any unexpected characters:

```
[ \t\n]                         {if(trace)ECHO;}
.                               {ECHO; yyerror("unexpected character");}
```

If the input had been more confusing (e.g. if we couldn't distinguish between some or all of the different kinds of names) it might have been necessary to pass the tab and/or newline characters from the lex to the yacc to help to recognise the input correctly. (There aren't any tabs between module names, only spaces.) Luckily, we don't need to do so in this case.

```
student : /*school*/name tab /*group*/name tab /*student*/name tab modules eoln
        ;
modules :
        | modules /*module*/name
        ;
```

We could check that a module name is plausible by e.g.:

```
[A-Z][A-Z][1-6][0-9][0-9][0-2]  {if(trace)ECHO; return module;}
[A-Z][A-Z][0-9][0-9][0-9][0-9]  {printf("warning - module name looks suspicious: %s\n", yytext);
                                 return module;}
[A-Z][A-Z][0-9][0-9][0-9]       {printf("warning - module name missing semester digit: %s\n", yytext);
                                 return module;}
```

Similarly, we could have a list of plausible school names, and output a warning if the school didn't match them:

```
"CS"|"CIS"|"CE"|"AI"|"SI"|"MI"  {if(trace)ECHO; return school;}
[A-Z]+                          {printf("warning - school name not recognised: %s\n", yytext);
                                 return school;}
```

Note that this is not a complete list of possible school names - the real list is much longer, and changes every few years.

Here is (one version of) the complete Yacc and Lex code

Yacc:

```
%start file
%token school group module name
%%

file    : student
        | file student
        ;
student : school group name modules
        ;
modules :
        | modules module
        ;

%%
int trace=0;
yyerror etc. as usual
```

Lex:

```
%{
#include "y.tab.h"
extern int trace;
%}
%%

[A-Z]+                          {if(trace)ECHO; return school;}
[A-Z][0-9]                      {if(trace)ECHO; return group;}
"__"                            {if(trace)ECHO; return group;}
[A-Z][A-Z][0-9][0-9][0-9][0-9]  {if(trace)ECHO; return module;}
```

```
[A-Z][A-Z][0-9][0-9][0-9]       {if(trace)ECHO; return module;}
[A-Z][a-z'][-A-Za-z, ]*         {if(trace)ECHO; return name;}
[ \t\n]                         {if(trace)ECHO;}
.                               {ECHO; yyerror("unexpected character");}

%%
yywrap etc. as usual
```

What do we do next?

The next step for the Faust DFSA language exercise, to build parse trees, is explained in detail in the lab description. (general information about parse trees)

For the example of lists of students and their module choices, the next step would probably be to do something with the actual names of students and modules etc., which at the moment we are just discarding. We would probably want to pass the names recognised by lex on to yacc, and then add code to yacc e.g. to count the number of students in each tutorial group, or the number taking each module, etc.

To inform yacc (and lex, via the generated y.tab.h file) that we are going to be using strings to hold the names of schools, groups, students and modules, we need to change the start of the yacc program:

```
%start file
%union {char *string}
%token <string> school group module name
```

We then need to modify each piece of code in lex that "return"s a "school" or "group" or "module" or "name" to copy the actual characters of the name into yylval from yytext. For example, we can change the code in lex that handles a school-name to be:

```
[A-Z]+                          {if(trace)ECHO;
                                 yylval.string = (char *)malloc(yyleng+1);
                                 strcpy(yylval.string, yytext);
                                 return school;}
```

More information about grammars including using lex without yacc