Notes on my implementation of the second CM20029 coursework

This file contains the second coursework I've prepared for the CM20029: Compilers course. It contains printouts of the .BIS and .PDF file I've prepared for the two parts.

If for any reason you need to access it online, please refer to:
http://people.bath.ac.uk/st221/CM20029/BISON/

Please make sure you always directly go to the cited address, since not only other pages do not contain any link to that place but also because the parent directory is actually password protected.

At that web address, you will find the .BIS and .PDF files (that have been printed out) as well as the .C file generated by BISON 1.875, the .EXE (Windows Executable) file generated using Visual C++ 6.0 SP6 and finally the .TXT files used in testing. Note that not all testing has been done using text files, and some (not all, just the "most interesting" cases) testing and other notes have been included in the .PDF file.

Description

The implementation has one file, bison.bis, which contains everything:

- Included .h files
- Constant definitions
- Global variables
- Functions

BISON has been implemented to be compatible with LEX (and also FLEX), therefore expects the input to be given via a function called yylex(). Also, it will try to output errors using yyerror(), so this function needs to be created as well.

When given an input and the input has been recognized by BISON, the program does the following:

- If it is a variable name or a digit, adds it to the "variable queue" (a list)
- If it is a declaration (so the creation or destruction of variables), does this execution on the variable list. Note that this execution is not reflected to the parse tree.
- If it is an expression or a "variable = expression" type statement, it builds a tree based on the various elements of the expression. After building the tree, also checks for the types of the variables and reorganizes the subtraction and multiplications' signs
- Then, at printing phase, the special characters put for the subtraction and multiplication on different types of variables are replaces by the post-words "N" for "non-defined" (operation between an integer and a float), "D" for decimal (or integer) and "F" for float.

I think this is all I want to say about the way I've implemented my program.

What's good with it?

The code seems to work “perfectly”: it recognizes and builds the good trees for all given inputs I've tried, and also recognizes the types correctly. “Difficult” tasks such as re-assigning variable types do not seem to make the program do unexpected things... Also, since line and character numbers are counted, the program gives helpful output when unexpected input is received.

What's bad with it?

Again, malloc() has been used; and an optimal person would probably have preferred to have arrays of objects and pointers to different places in this array. I do not see any other issues with my program.

Also, the last two tests show a “weird” behavior of the tree printing algorithm; which is quite confusing at a first glance...

Testing

Testing has been done on Windows-based computers, but nothing should prohibit the software from running on *nix machines as well...

- bison1.txt:

```
D a,b;
F x,y,z;
a=(y-z)*(a-b);
U a; F a;
b = x-a;
```

- Gives us this in output:

```
> bison < bison1.txt
;
= a
  *N
    -F y z
    -D a b ;
= b
  -F x a
```

Parsing has been successful: tokens have been recognized, and, the variable “a” is a decimal in the first statement and a float in the second. Also note that in the first expression, the multiplication is of type “unknown” (N) since it is between a float and an integer (as we can see with the “-F” and “-D” signs)

- bison2.txt:

```
D a,b;
F x,z;
a=(y-z)*(a-b);
U a; F a;
b = x-a;
```

- Gives us as output:

```
> bison < bison2.txt
The variable 'y' hasn't been defined and not parsed (E)
;
= a
  *N
    -F E z
    -D a b ;
= b
  -F x a
```

So, the parser tells us that the variable “y” has not been defined this time; and also replaces its token with an “E” (for “error”)... Even if the token is erroneous, the type recognition algorithm gave “float” as the type of the operation between this erroneous token and the variable “z”; which somehow indicates that the erroneous token should be a float...

- bison3.txt:

```
D a,b;
F ,z;
a=(y-z)*(a-b);
```

- Gives us as output:

```
> bison < bison3.txt
The BISON got nervous at you (line 2, char 3: syntax error)
```

Our syntax error has been detected correctly and parsing interrupted...

- bison4.txt:

```
F a,b,c;
a=a-b-c;
```

- Gives us as output:

```
> bison < bison4.txt
;
= a
  -F
    -F a b c
```

Therefore, the grammar's ambiguity has been “removed” by changing our parser a bit: it gives higher priority to the leftmost operation...

Note that the last line is somehow confusing: actually, if we re-write it using a LISP-like format, it gives us:

```
( = a ( -F ( -F a b ) c ) )
```

This is a “weird behavior” of my tree printing algorithm... Sorry about it!

- bison5.txt:

```
F a,y,z,b;
a=y-z*a-b;
```

- Gives us as output:

```
> bison < bison5.txt
;
= a
  -F
    -F y
      *F z a b
```

Therefore, the grammar's ambiguity has been “removed” by changing our parser a bit: it gives higher priority to the multiplication operation...

Again if we re-write the result in a LISP-like format, it gives us:

```
( = a ( -F ( -F y ( *F z a ) ) b )
```