Machine Problem 4: Grammar Nazi 2.0

Rubinos, Tyna S.
BS Computer Science II

Variable and Function Declarations

```
int b = 56                              ,                       c =                                 21  ;
float                                                   x;
```

These are okay...

Spaces are one thing to take account of when declaring variables. Besides the usual single space after every word (i.e. `int x = 5;`), C accepts multiple spaces in a variable and function declaration, with the sole exception of the assignment, meaning `int x = 69. 76;` or `float y = 6 4 2 1;` are both *invalid*.

```
double ss = 'c';
char tt = 66;
```

These are okay, too!

Characters also have their equivalent *ASCII values*, which are their numerical equivalents. If I printed the value of variable `ss` and `tt`, I would get a number and a letter, respectively. Initially, I wanted to exclude accepting values like the ones above, but I realized that I wouldn't need to make another DFA specifically for the `char` data type. I did some checking, and values like `88.34` are still accepted by `char`. I presume it just cuts off the floating point as with `int`.

```
double double = 77;
```

...But these are NOT okay!

Another thing to consider is checking the *variable names*. The variable must begin with either an `_` (underscore) or any letter, then followed by any number of underscores or alphanumeric characters. C does not allow language-specific keywords to be variable names (meaning `for`, `while`, `else` are invalid variable names). However, this makes the DFA even more complicated than it already was, so I decided to make only primitive data types (`int`, `char`, `double`, `float`) the illegal variable names for this declaration.

```
char mom                                                (int, char gsd, float AHHHHH)       ;
int test(int a,int b,int c,int d);
double squareRoot(float, char a, int b);
```

Some very valid functions.

Function declarations still need to start with primitive data types, with the inclusion of `void`. Some variables do not have parameters (i.e. `int function();`), while some can operate with just the data type (`double function(int, char);`). Once again, I must take note of spaces and separators when checking a function. As for variable name limitations, I decided to include `void` as well in this state machine.

```
void tean(void a);
void arm(void);
```

Welcome to the void!

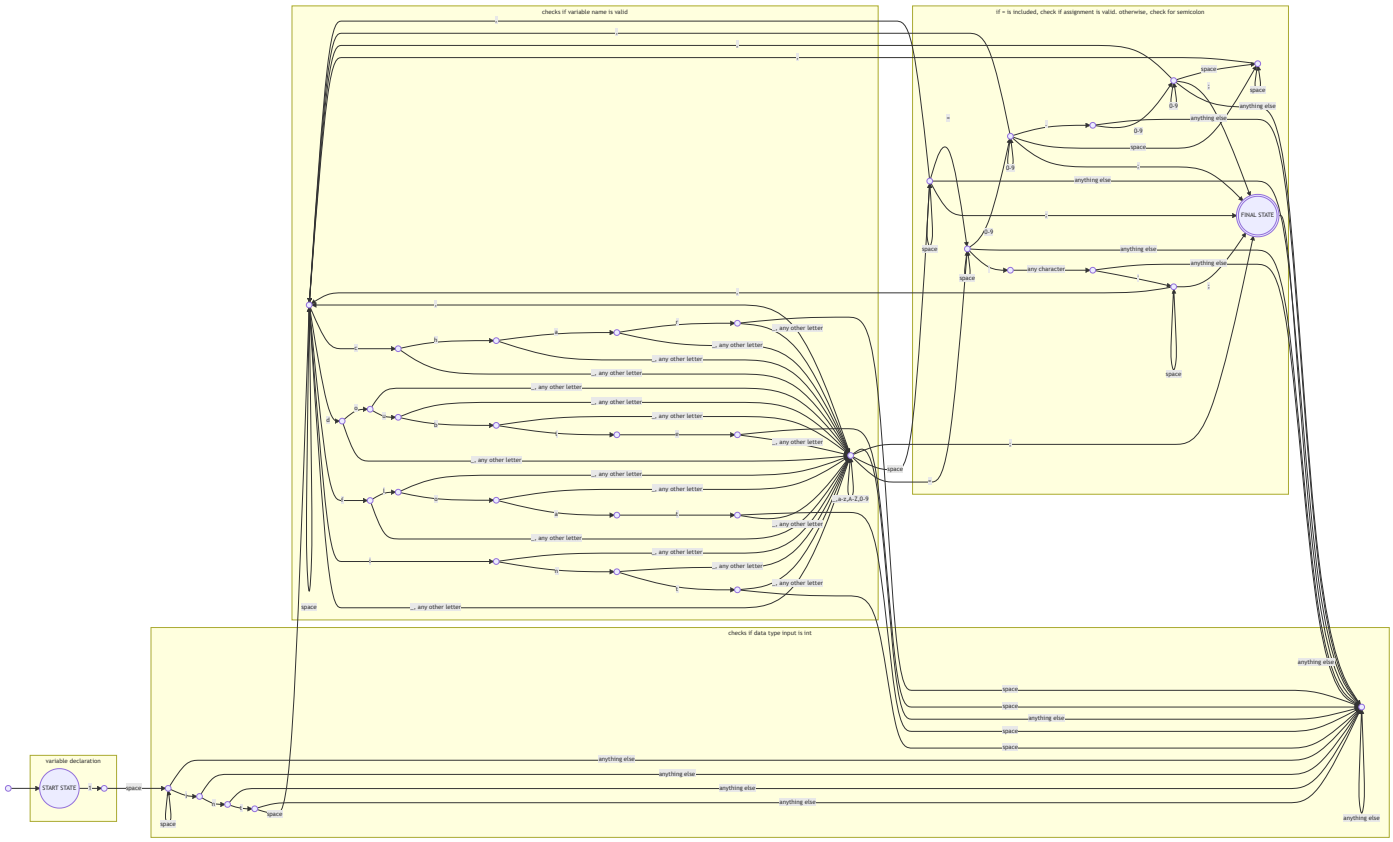
I should also take note of putting `void` between the parentheses. Functions with that include the `void` parameter must only contain that and nothing else (i.e. `void function1(void);` is *valid*, while `void function(void, int);` is *invalid*).

For both variable and function declarations, I will not include any pointers. This means that `char *ilovealbedo;` and `char testfunc(char* one, int* two);` will be considered *invalid*.

Equivalent State Diagrams

I made these very convoluted state diagrams using `mermaid.js` (with help from its live editor). It's very confusing and it will make you cry. I know I did.

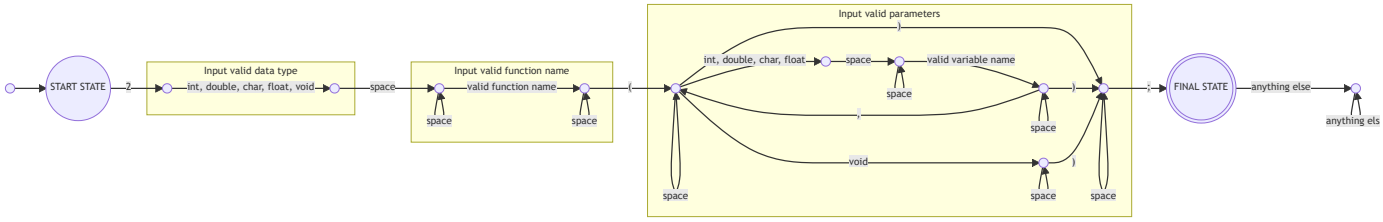
I started out with just the `int` variable declaration, and it resulted in this state diagram. A better file can be found here.



...And that's just for int!

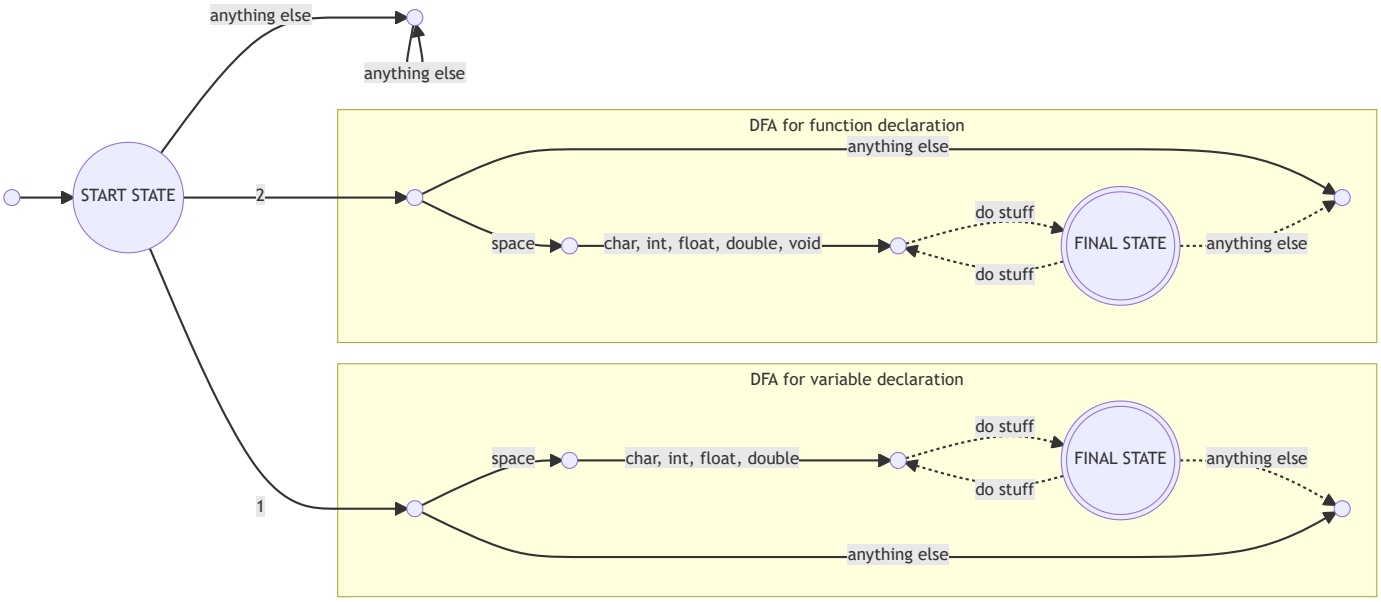
My initial plan was to use the same state diagram for `int`, `double`, and `float`, then create a separate diagram when the machine reads a `char`. However, since ASCII characters can be interpreted as numbers and vice versa depending on the data type used, it would make sense to just use the same state machine for all.

I made a different state diagram for function declarations. Unlike the previous state diagram, I had to make this a simpler FSM. I may have missed some valid inputs. I also realized one thing: I could always make a state transition table instead. A better file can be found here.



Hopefully this looks more understandable

The figure below shows a very rough draft for what my finite state machine would look like, following the machine problem's constraints. Though it isn't shown here, I decided that I wanted to check the variable declaration character-by-character, instead creating tokens to check specific words. A better file can be found here.

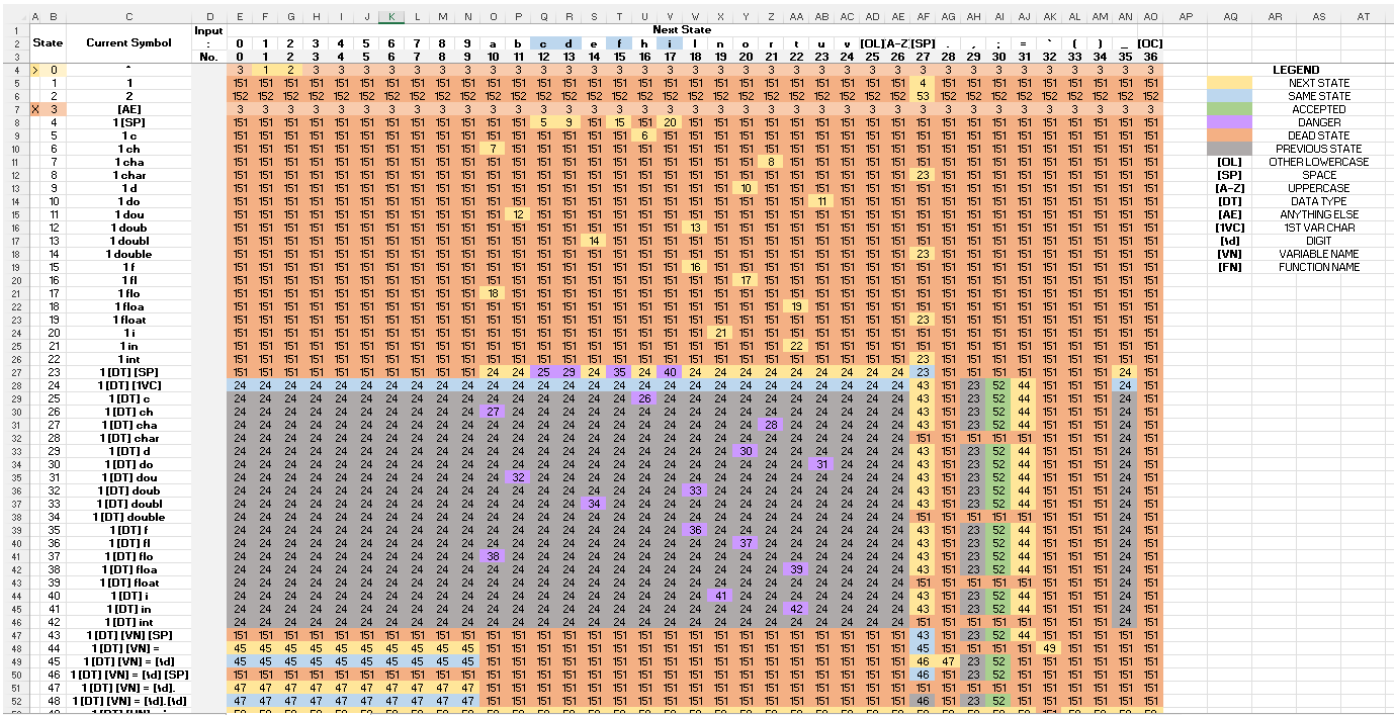


Assuming the DFA is similar to one above, machine must first check if data type is valid, followed by a space. Otherwise it reaches a dead state.

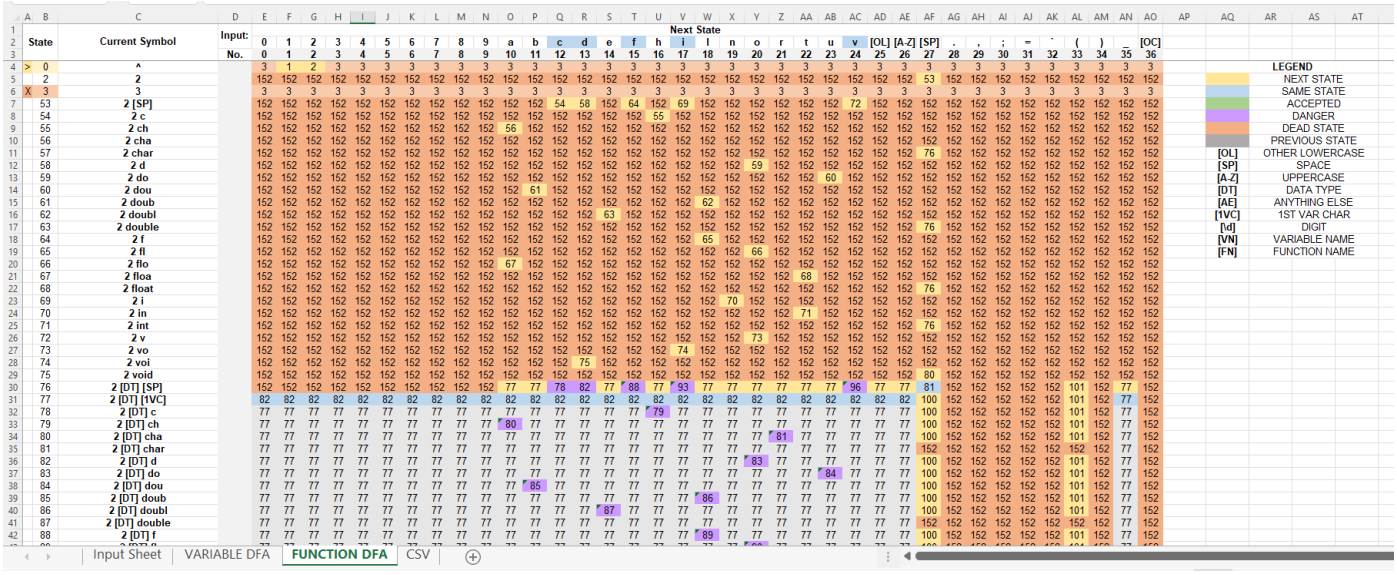
The DFA State Transition Table

Just when I thought it'd be easier if I made a transition table, I made this behemoth of a spreadsheet *for every possible character* (at least for this exercise). I have many regrets. Maybe I should've done tokens instead. But oh well, I've already started it. There's no turning back. I sent a whole Excel file (which can also be accessed here) dedicated for both variable and function declarations.

I'm sorry sir, I should've listened to you when you mentioned tokens.



Spreadsheet for variable declarations only. The spreadsheet for function declarations is just as massive.



These are for function declarations.

Anyways, there's no way I can make that whole thing fit in one Python file. Instead, I made a `.csv` file and imported the `numPy` module, where I can easily convert the file into a 2D array. I can now the 2D array as is and check string inputs.

Some Limitations

Even with how big it is, there's still some limits to how the DFA works. I've already included it in the comments, but I'd like to write it here as well:

1. Only illegal variable names are primitive data types (and `void` for function declaration).
    - C-specific keywords like `else`, `if`, `return`, etc. were not identified in the DFA
    - This means `1 int if = 68;` and `1 char void;` will be *valid*.

- `double double;` and `2 int char();` will be *invalid*.
- This could've been avoided if I just used tokens.

2. Variable declarations allow for one character to be converted as a number. I have included them all, even if some are invalid.
   - `1 float check = '\';` and `1 char test = ''';` will be *valid*
   - Again, this could've been avoided if I just used tokens.