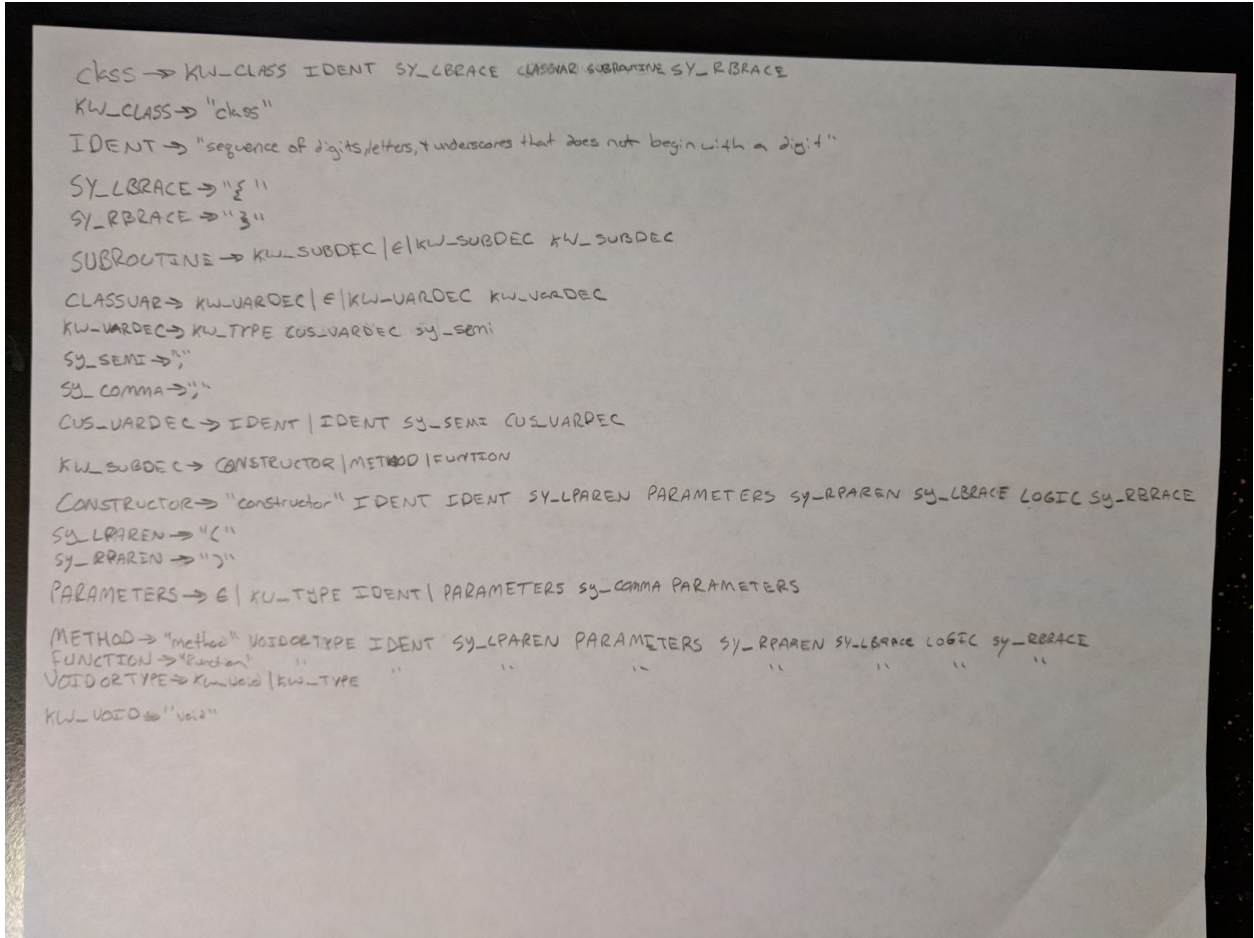


Ryan Darras PJ04 project writeup

Preface

First things first, I apologize for not having this assignment done completely. This is relatively uncharacteristic of me, but between work/other projects something had to get put on the back burner. Thankfully Dr. Bahn drops the lowest grade for extra credit! As for any code or actual generation of the xml files, I didn't do anything. I did, however, work out a grammar that, while I know likely has mistakes because I can't test it, should demonstrate that I at least know where to go for this assignment. I realize I might not get many (if any) points for this submission, but I'd rather show that I at least looked at the assignment and thought about it than just have a non-submission.

Grammar



Handwritten grammar rules on a piece of paper:

```
CLASS → KW_CLASS IDENT SY_LBRACE CLASSVAR SUBROUTINE SY_RBRACE
KW_CLASS → "class"
IDENT → "sequence of digits/letters, & underscores that does not begin with a digit"
SY_LBRACE → "{"
SY_RBRACE → "}"
SUBROUTINE → KW_SUBDEC | ε | KW_SUBDEC KW_SUBDEC
CLASSVAR → KW_VARDEC | ε | KW_VARDEC KW_VARDEC
KW_VARDEC → KW_TYPE CUS_VARDEC SY_SEMI
SY_SEMI → ";"
SY_COMMA → ","
CUS_VARDEC → IDENT | IDENT SY_SEMI CUS_VARDEC
KW_SUBDEC → CONSTRUCTOR | METHOD | FUNCTION
CONSTRUCTOR → "constructor" IDENT IDENT SY_LPAREN PARAMETERS SY_RPAREN SY_LBRACE LOGIC SY_RBRACE
SY_LPAREN → "("
SY_RPAREN → ")"
PARAMETERS → ε | KW_TYPE IDENT | PARAMETERS SY_COMMA PARAMETERS
METHOD → "method" VOIDORTYPE IDENT SY_LPAREN PARAMETERS SY_RPAREN SY_LBRACE LOGIC SY_RBRACE
FUNCTION → "function" " " " " " " " "
VOIDORTYPE → KW_VOID | KW_TYPE " " " " " " " "
KW_VOID → "void"
```

LOGIC \rightarrow KW-LET | KW-DO | KW-IF | KW-WHILE | KW-RETURN | LOGIC LOGIC
 KW-LET \rightarrow "let" IDENT SY-EQ ASSIGNMENT
 SY-EQ \rightarrow "="
 ASSIGNMENT \rightarrow IDENT | INTEGER | STRING | CONCATENATION | MATH | ASSIGNMENT
 INTEGER \rightarrow "any integer"
 STRING \rightarrow "any string"
 CONCATENATION \rightarrow STRING " " STRING | STRING "+" CONCATENATION
 MATH \rightarrow IDENT SY-OP IDENT | IDENT SY-OP MATH | INTEGER SY-OP INTEGER | INTEGER SY-OP MATH |
 SY-OP \rightarrow "+" | "-" | "*" | "/" | "%" | "<" | ">"
 INTEGER SY-OP IDENT | IDENT SY-OP INTEGER
 KW-DO \rightarrow "do" IDENT SY-LPAREN INPUT PARAMS SY-RPAREN SY-SEMI
 INPUT PARAMS \rightarrow KW-CONST | INTEGER | STRING | IDENT | MATH | CONCATENATION | INPUT PARAMS SY-COMMA INPUT PARAMS
 KW-IF \rightarrow "if" SY-LPAREN ARGUMENT SY-RPAREN SY-LBRACE LOGIC SY-RBRACE |
 ARGUMENT "if" SY-LPAREN ARGUMENT SY-RPAREN SY-LBRACE LOGIC SY-RBRACE "else" SY-LBRACE SY-RBRACE |
 "else" KW-IF
 ARGUMENT \rightarrow IDENT | BOOLEAN | BOOLEAN-STATEMENT
 BOOLEAN \rightarrow "true" | "false"
 BOOLEAN-STATEMENT \rightarrow ASSIGNMENT SY-OP ASSIGNMENT | ASSIGNMENT SY-OP SY-OP ASSIGNMENT | ASSIGNMENT SY-OP SY-OP ASSIGNMENT
 KW-WHILE \rightarrow "while" SY-LPAREN ARGUMENT SY-RPAREN SY-LBRACE LOGIC SY-RBRACE
 KW-RETURN \rightarrow "return" SY-SEMI | "return" ASSIGNMENT SY-SEMI

This description will be very broad, and I realize there exists a few issues in the above screenshots that I will be contradicting in this section. I'm writing more about what I understand, less about what I actually did. In order for a language to be LL(1), the parser can look at most 1 token ahead to find out what it needs to know. This can get tricky, but it can be solved by splitting the productions up so that you never run into a situation where the program gets confused because it doesn't know what step to take next. For example, KW_SUBDEC knows that the next input will start with the character 'c', 'm', or 'f'. Because of this, it will never get confused and will always be able to recursively percolate down into either CONSTRUCTOR, METHOD, or FUNCTION respectively. Alternatively, in the images above my MATH production has productions inside of it that would require you to look at least 2 tokens ahead to discover what they are.

To Code the Grammar

The way I thought of programming this felt different in my head compared to the way we used grammars in class (even though I know it is essentially the same idea). For example, the “CLASS” production has a single production in it, but that single production involves other productions ($S \rightarrow ABCDEF$). Some of these, A for example, is simply KW_CLASS which only has a single literal to it. The others, such as E in this situation, recursively dive down into SUBROUTINE. Now SUBROUTINE simply breaks it down into 0 or more KW_SUBDECs. This gives our CLASS the availability to have 0 or more constructors/methods/functions inside of it. Since we are doing this recursively, that remaining F at the end of the CLASS production (SY_RBRACE) will save itself until everything returns to the first called production, which was CLASS, and then it will do its own function. In the end, you are going to have to groups of functions. One group is going to be a group that are deciders; they decide what is actually being read next. The second group are the printing functions, that will print the respective xml to the file.

Conclusion

Once again, I apologize for not getting any implementation done for this project. I should have started on it earlier than a few days ago. While I feel like this would take me a good bit to implement (~15 hours) I feel confident that I could do it. Considering I am using this as my “extra credit” assignment, feel free to give whatever points you feel necessary. I would completely understand a 0 as I don’t have the bulk of it done.