Review of Scanning & Parsing

COMP360

“I can accept failure, everyone fails at something. But I can’t accept not trying.”

Michael Jordan

Exam Postponed

- The first COMP360 exam scheduled for today has been postponed until Wednesday, February 19, 2020
- The Parsing programming assignment due date has been moved back to Friday, February 21, 2020

Languages and Machines

Language	Machine
Regular Expressions	Deterministic Finite State Automata (DFA)
Context Free	Push Down Automata (PDA)
Context Sensitive	Bounded Turing Machine
Recursively Enumerable	Turing Machine

Output of Compiler Stages

Step	Output
• Lexical Analysis (scanning)	List of tokens
• Syntactic Analysis (parsing)	Parse tree
• Semantic Analysis	Intermediate code
• Optimization	Better Intermediate code
• Code Generation	Machine language

A Scanner can be implemented with a

A. Abacus
B. Finite State Automata
C. Push Down Stack
D. Turing Machine
E. None of the above

Lexical Analysis

- Lexical Analysis or scanning reads the source code (or expanded source code)
- It removes all comments and white space
- The output of the scanner is a stream of tokens
- Tokens can be words, symbols or character strings
- A scanner can be a finite state automata (FSA)

Syntactic Analysis

- Syntactic Analysis or parsing reads the stream of tokens created by the scanner
- It checks that the language syntax is correct
- The output of the Syntactic Analyzer is a parse tree
- The parser can be implemented by a context free grammar stack machine

Simple Language

? This is an arithmetic Language

(5.7 + 3.4) / 6 ? comment

After Lexical Scan

(	5.7	+	3.4	)	/	6

- The lexical scanner creates a list of all tokens in the program
- Comments and whitespace have been removed
- Numbers are single tokens instead of individual characters
- The list could be an array, a linked list or an ArrayList
- The scanner does not check for syntax errors (e.g missing semicolons, unbalanced parenthesis)

Token Object

A token created by the lexical scanner should contain:

- String value; // text of the token
- int type; // type of value
  - constant
  - punctuation
- int line number, column // position in source code

Useful Token methods

It may help if your Token class has methods

- isVar() – true if this is a variable
- isVarConst() – true if this is a variable or a constant
- getValue() – returns the Token string
- add2Token(char letter) – appends the letter to the last token created
- toString() – use to display the tokens

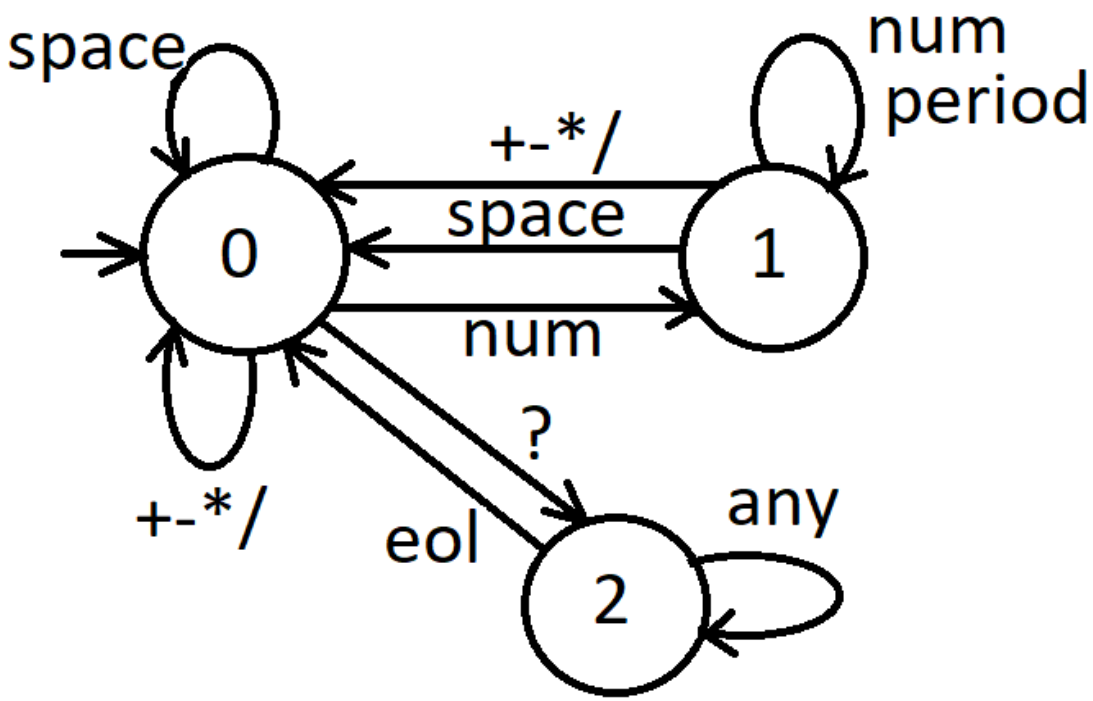
The constructor is missing

```
public class Token{
        String value;  int type;
        int row, col;
        Token(char val, int type) {
           value = String.valueOf(val);
           this.type = type;
        }
```

A.input symbol
B.punctuation
C.row and column
D.all of the above
E.none of the above

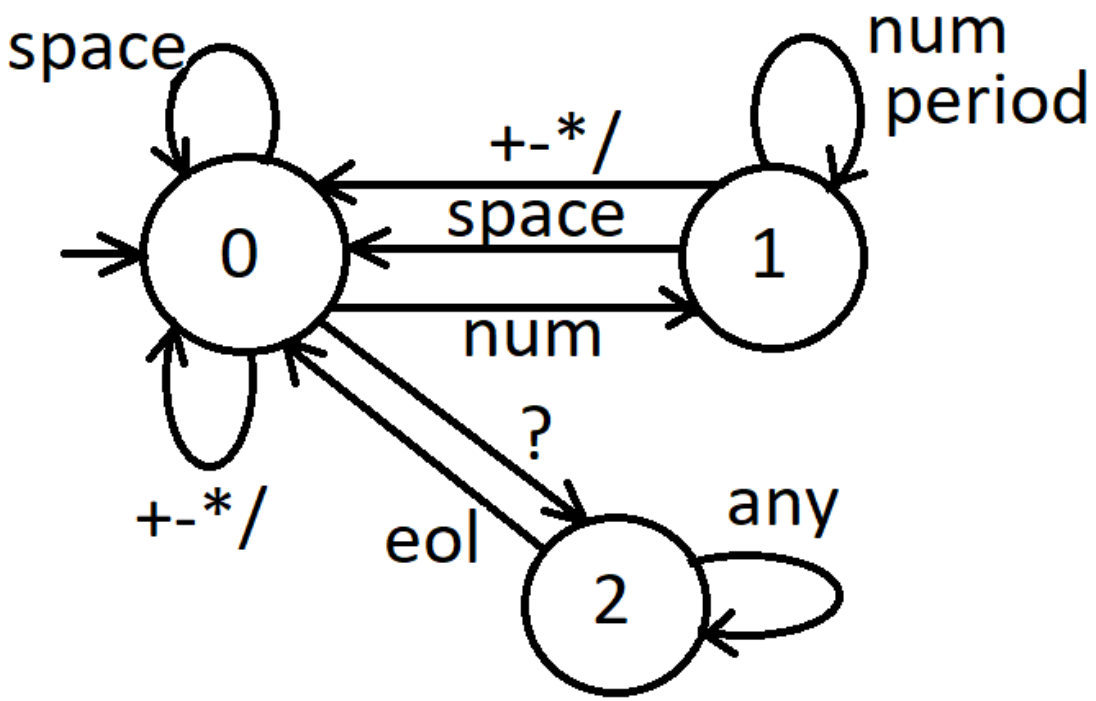
Arithmetic FSA

	0	1	2
number	1	1	2
period	1	1	2
space	0	0	2
EOL	0	0	0
?	2	2	2
+-*/	0	0	2
other	-1	-1	2



Arithmetic Mealy Machine

	0	1	2
number	1/1	1/2	2/0
period	1/1	1/2	2/0
space	0/0	0/0	2/0
EOL	0/0	0/0	0/0
?	2/0	2/0	2/0
+-*/	0/1	0/1	2/0
other	-1/0	-1/0	2/0



Actions

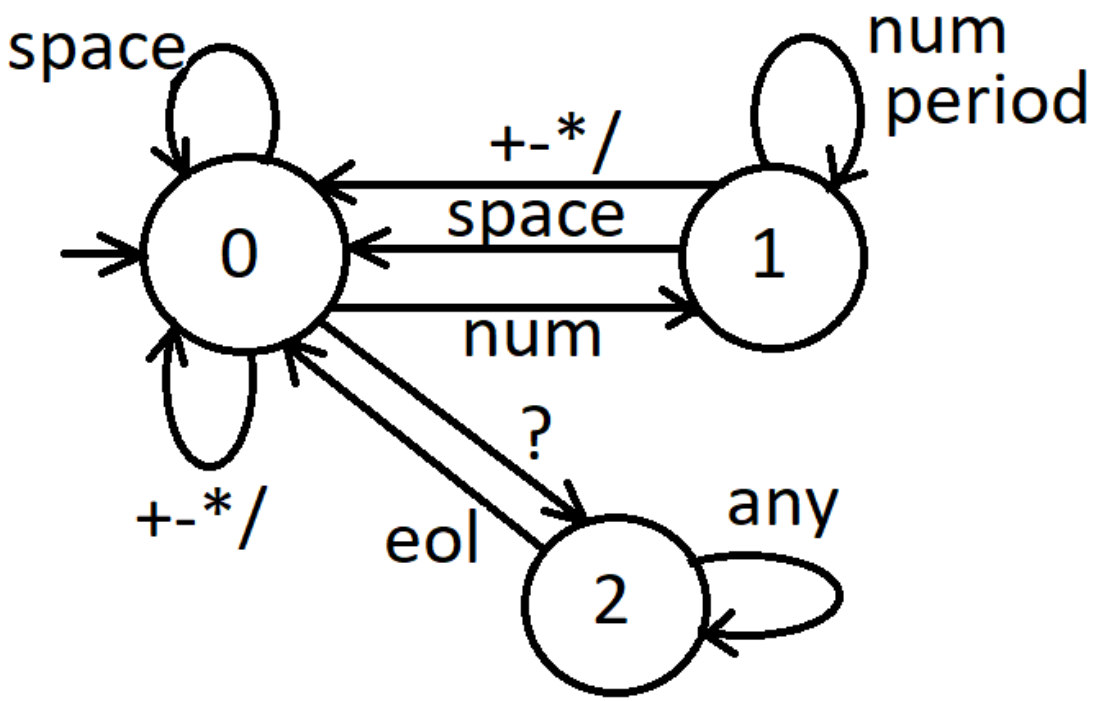
0 = do nothing

1 = create token with input character

2 = add input to token

Arithmetic Action Table

	0	1	2
number	1	2	0
period	1	2	0
space	0	0	0
EOL	0	0	0
?	0	0	0
+-*/	1	1	0
other	0	0	0



Actions

0 = do nothing

1 = create token with input character

2 = add input to token

Converting the State Table to Java

```
int[][] stateTable = {
    /* state 0,  1,  2 */
            {  1,  1,  2 }, // 0 number
            {  1,  1,  2 }, // 1 period
            {  0,  0,  2 }, // 2 space
            {  0,  0,  0 }, // 3 EoL
            {  2,  2,  2 }, // 4 ?
            {  0,  0,  2 }, // 5 +-*/
            { -1, -1,  2 }};// 6 other
```

Convert Input Character to Index

```
if (Character.isDigit(symbol)) {
    inx = 0;
} else if (symbol == '.') {
    inx = 1;
} else if (symbol == ' ' || symbol == '\t') {
    inx = 2;
} else if (symbol == '\n') {
    inx = 3;
} else if (symbol == '?') {
    inx = 4;
}else if ("+-*/".contains(symbol)) {
    inx = 5;
} else {
    inx = 6;
```

FSA Program

```
state = 0
while not end of file
        symbol = read next input character
        inx = number based on type of character
        action = actionTable[inx][state ]
        state   = stateTable[inx][state ]
        if state = -1 then error
        switch ( action )
                case 1: new Token(symbol); put on list
                case 2 : append symbol to last token
```

If you are in a comment state and you input a number, what action should you take?

A. do nothing
B. create token with input character
C. add input to token
D. create empty token
E. None of the above

BNF Fundamentals

- Terminal symbols are tokens or symbols in the language. They come from the lexical scanner
- Nonterminal symbols are “variables” that represent patterns of terminals and nonterminal symbols
- A BNF consists of a set of **productions** or **rules**
- A language description is called a **grammar**

BNF Structure

- Nonterminal symbols are sometimes enclosed in brackets to differentiate them from terminal symbols
- I will use **bold font** for terminal symbols and no brackets
- Productions are of the form:

  nonterminal → nonterminals or terminals

- Multiple definitions are separated by | meaning OR

  whatever → this | that

Recursive Descent

- Recursive descent is a top down parsing method
- It is relatively easy algorithm to implement
- Functions or methods are written to represent each BNF production
- Recursive descent does not use parse tables

BNF to Recursive Descent

- Each production becomes a separate method
- Usually the method name matches the name of the non-terminal
- For each non-terminal on the right side, the method for the non-terminal is called
- For terminal symbols, the program checks if the next token is the proper symbol

Input tokens

- The input to the parser is a list of tokens from the lexical scanner (called **`tokens`** in the examples)
- I used an integer (called **`curtok`**) to keep track of the current location in the input that is being parsed
- When a method matches a BNF production, it increments **`curtok`** to move past that input token

Example Grammar

expr → term

| term **+** expr

| term **-** expr

term → factor

| factor ***** factor

| factor **/** factor

factor → **variable**

| **(** expr **)**

expr function

expr → term | term + expr | term - expr

```
boolen expr() {
 if (!term()) return false;
 if (tokens[curtok] == '+'
        || tokens[curtok] == '-') {
    curtok++;     // consume token
    if (!expr()) throw parse error;
 }
 return true;
}
```

Possible Snowflake BNF

1. SF → parmstmt code ret
2. parmstmt → **parm** varlist **;**
3. varlist → **var**
   | **var** varlist
4. varclist → **varconst**
   | **varconst** varclist
5. ret → **return var ;**
6. pattern → **varconst**
   | **varconst | pattern**

Possible Snowflake BNF (cont)

7. code → line
   | line code
8. line → assign
   | loop
9. assign → **var** **=** varclist **;**
   | **var** pattern **=** varclist **;**
10. loop → **while** **varconst** pattern **{** code **}**

Write the Recursive Descent parser method for SF, the Snowflake root

SF → parmstmt code ret

Recursive Descent parser method for

SF → parmstmt code ret

```
boolean SF() throws ParseException {
        if (!parmstmt()) throw new ParseException;
        if (!code()) throw new ParseException;
        if (!ret()) throw new ParseException;
        return true;
}
```

Throwing ParseException

- Our compiler is not very forgiving
  - It stops at the first error found
- The scanner and parser must both produce reasonable error messages with the error location
- Throwing an exception is an easy way to quit

```
throw new java.text.ParseException("Error Msg",
                                   getLocation());
```

Error Location

- Tokens must include the column and line number
  - Increment the column for each character read
  - At a new line, increment the line number and column = 0
- Since the second parameter of the ParseException constructor takes an int, you can combine row & col

```
public int getLocation() {
        return 1000 * tokens[curtok].row +
                      tokens[curtok].column;
}
```

Undo

- The recursive descent methods “consume” input tokens as they are executed
- Each method looks at the first token after what was matched by earlier methods
- If a method “fails”, it must leave the tokens on the input in the same position they were at the start of the method
- A rollback method may be necessary
- It appears Snowflake parsers do not need to back up

cat -> dog goat Method with Recovery

```
boolean cat() {
        int saveLoc = curtok;
        if ( dog() ) {
                if ( goat() ) {
                        return true;
                }
                curtok = saveLoc;
                return false;
        }
        return false;
}
```

Exam Postponed

- The first COMP360 exam scheduled for today has been postponed until Wednesday, February 19, 2020
- The Parsing programming assignment due date has been moved back to Friday, February 21, 2020