* Now, let’s put several topics together from our studies this semester and have you use the Java language to build a very simple recursive descent parser for a very simple grammar.
* We will use the SMASM grammar used in your lexical analyzer assignment and build a recursive descent parser that processes a file of tokens (from a SMASM Language program) to see if that program is syntactically correct.
* **Requirements:**
  1. Recall, each grammar rule becomes a function in a recursive descent parser, you would include the grammar rule that you are implementing in the respective function in the function header documentation.
  2. Each method/function should include in its header documentation the grammar rule that it is implementing along with the other items shown for documentation requirements.
  3. Check each rule to ensure that it is in an appropriate format for a recursive descent parser. If there is a grammar rule that needs to be transformed, work it out on your own on paper and clearly indicate that rule and its transformation in your main program comments! Then, you will encode the new transformed rule(s). So, be sure to implement those new rules rather than the old rule that had a problem with it.
     1. **Is there direct left recursion?**
     2. **Are the first sets mutually disjoint for the terms of each rule?**
  4. Functions should announce, print a message, that they are in a particular function – for example, when a function is entered, for a particular grammar rule, print a simple output statement **(to the screen and to an output file)** that says (with respect to each function)
     1. ‘Entering function:<function name>’ or ‘Entering Function: <function name>’
  5. Similarly, when a function (grammar rule) successfully completes, a message should be printed (**to the screen and to an output file**)
     1. ‘Leaving function: <function name> Success’
  6. Or, if the function did not successfully complete, a message should be printed to the screen and to an output file
     1. Leaving function: <function name> Unsuccessful
  7. A suggestion, for your own debugging efforts, upon entering a function the function should also print out the values of its incoming parameters (with appropriate labels) to the screen and to an output file. These are useful in debugging but also in seeing how your recursive descent parser works.
  8. Your program should use the file Prog4XYTokens.txt as the input file of tokens and write output to: Prog4XYout.txt. The XY are your first and last initials.
  9. We are building a parser for the SMASM language. In the last assignment you built a lexical analyzer for the language. In general, the output of your last program (your lexical analyzer) consisted of the tokens from a SMASM program. That file of tokens serves as the input file to your parsing program. Note: I will provide you with a file of tokens that you can use in this assignment. Although, you can certainly build your own file of tokens to do additional tests. I will test your parsing code with more than one example input file. Some should parse successfully and some will have SMASM syntax errors and your parser should respond appropriately.
  10. Your program will use the input file of tokens and **your recursive descent parser will print to the screen that it is checking (that filename), so do include the name of the file in that print statement to the screen,** to see if it is a syntactically correct SMASM Language program.
  11. **Please call your main function main()**
  12. If the tokens in the file entered by the user successfully parses, your program should print a message that indicates that it parsed successfully and is legal.
  13. If the tokens in the input file do not successfully parse, your program should print a message that the program in the input file (print that file name) did not successfully parse and is not a legal program.
  14. The code must be written in a .java file with appropriate documentation for each function/method as we have done this semester.
      1. You must save your code in a textfile with a .java extension so that you can submit it through Schoology.
      2. I will be grading your work through the BlueJay IDE. I strongly suggest that you use BlueJ to develop this solution. If you use another IDE, I strongly suggest that you run your code in BlueJ as that is how I will grade your work. Note you will be sending me your Java file and I will be loading and running your program to determine your grade for this assignment. You should test it yourself with a sample file of tokens.
  15. HINTS: HINTS: put in print statements in your functions when you are debugging.
  16. **Documentation requirements**: Each function/method (and main program) should have a function header in a style similar to what you have done this semester. Here is the example from a previous assignment. You should use a similar style for your python function documentation. As mentioned earlier, your method header should include the grammar rule that the method is implementing as well.
      1. Remember to include the grammar rule in your method headers for the methods implementing grammar rules. Like we saw in our textbook in chapter 4.
      2. ALSO – When your function requires a control construct, you should consider a comment to explain what that code will be doing.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; ALCM

; Programmer: Steve Jodis

; Date Created: 3/6/2016

; Last modified: 3/6/16

;

; Arguments: prev – an integer, representing the last value of the linear congruential algorithm

; b – a seed value for the lcm random number generator algorithm

; m – a seed value for the lcm random number generator algorithm

;

; Returns: the next value from the lcm random number generator

; Example Usage: (ALCM prev b m)

; Description: This function employs the linear congruential method of random number generation……

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Implementation Notes:** Create each function one at a time at the beginning of your file. Get one function working before moving on to the next one. You will start with the START symbol, but you can initially code just part of it to get your program to parse a very simple program in the SMASM language. Then add more grammar rules to complete the parser. Start with the simplest grammar rule. Build your solution one grammar rule at a time (ie, one function at a time).

After the function section of your code you can put the statements for your main program code. Note that I will test your code with other input that should successfully parse and some that should not parse successfully.

**Language Spring 310 – Spring 2024**

**SMASM Language**

**Language Name: SMASM**

**Case Sensitivity:** The language is not case sensitive

**Extension:** SMA (programs in the SMASM language use .sma as the file extension)

**Statement Terminator: ‘.’ (the period)**

**Note: This was preprocessed out in the first programming assignment of the semester.**

**Comments: None – if added in the future would be handled in pre-processing**

**Statements are separated by ‘.’ One continuous line – these are pre-processed out and have not effect on parser program.**

**Grammar Description:**

This is a preliminary grammar for a simple assembly language to perform simple arithmetic.

The architecture of the assembly language SMASM is:

There are six registers, a..f but a is a special register known as the 'accumulator'

- In your virtual machine, **all registers should be initialized to zero at the start of a program**.

The commands of the SMASM language are:

ld [val,reg] load the value into the register

sub [reg1,reg2] result of reg2 - reg 1 stored to a

st [reg1,reg2] store reg1 into reg2

sq [reg] square the value in reg with result stored to a

add [reg1,reg2] reg 1 + reg 2 with result stored to a

rt [reg] square root of value in reg with result stored to a

**Sample Program**

ld[7,b]

ld[10,c]

sub[b,c]

sq[a]

st[a,d]

ld[14,b]

ld[20,c]

sub[b,c]

sq[a]

st[a,e]

add[a,b]

st[a,f]

rt[f]

st[a,e]

**Grammar:**

START -> { COMMANDS }

COMMANDS -> COMMAND ‘[‘ (REGISTER | VALUE ) [ ‘,’ REGISTER] ‘]’

COMMAND -> ‘st’ | ‘ld’ | ‘sub’ | ‘add’ | ‘sq’ | ‘rt’

REGISTER -> ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’

VALUE -> DIGITS

DIGIT -> ‘0’ | .. | ‘9’

DIGITS -> DIGIT {DIGIT}

**Sample Input File (based on first two lines of sample program – without errors)**

ld

[

7

,

b

]

ld

[

10

,

c

]

**Sample Input File (based on first two lines of sample program – with errors)**

ld

[

7

,

,

b

]

ld

[

10

,

c

]