

This learning group assignment will have a single question for everyone to work on. I will be checking your “solutions” as usual at the next lecture.

In the language of your choice¹ write a basic utility application that does the following:

1. Read in a grammar (G),
2. Determine the **non-terminals** and **terminals** of the language,
3. Have the basic **internal routines** to support the calculations of **first sets**, **follow sets**, and determining if a non-terminal allows the derivation $A \Rightarrow^* \lambda$ (“derives to λ ”). Note the distinction: **you are not being asked to write the routines to calculate these basic grammar symbol sets**, but you should have the primitive operations in place that will support their development.

Do not over stretch here, keep your solution simple. Don’t add unneeded features, just get this done for the next lecture. You may do this in the language of your choice, but a higher level language, preferably one you are completing your course projects in, is ideal. Run to your favorite language with **built-in** lists and associative arrays and go, go, go. Here are some obvious details:

1 Grammar

Your script should parse a grammar file conforming to the following interpretation of white space delimited tokens:

- Non-terminal symbols **contain** a capital letter $A-Z$.
- \rightarrow (two characters) denotes the beginning of a production rule with a LHS non-terminal.
- $|$ (pipe character) represents alternation of LHS production rules.
- $\$$ (dollar sign) represents the end of token sequence.
- λ (lambda) represents, you know, λ .
- Terminal symbols are **all other white space delimited tokens**, eg: $++$, $\}$, $-->$, $\$a\rightarrow terminal|symbol/\lambda/$.
- Production rules with specific LHSs will occur only at the beginning of lines.
- RHS alternation of production rules ($|$) can occur on the same line as the previous rule for the same LHS or on the next line.
- Empty lines are ignored

For now, your script does not need to verify the integrity or consistency of a grammar (that may come later).

An example grammar is on page 3.

1.1 Coding Hints

- I’m not asking (nor expecting) you to write this with a `Lex scanner.1`. The CFG syntax described above is very straightforward, and I implemented my logic in one medium sized `while-do-done` loop. On the inside I looked ahead two tokens for \rightarrow to know when a next token was a new LHS (I simply ignored newlines during parsing).

Another approach would be to first split the entire file contents on the string \rightarrow ; now the last token of all but the last partition is the LHS of the next partition, reorganize the strings as such and then split your RHSs by $|$. After this you’re pretty darn close to being done with the parsing...

- CFGs are essentially a list of rules, I would suggest keeping them as such.

¹The forward looking student might be sure to choose something available on the `alamode` machines, so this effort can be folded into a future graded work.

- RHSs of productions rules are also lists, hmmm, choose a language with easy lists perhaps :).

2 Output

Your utility application (script?) should output the following information when provided a grammar file:

- The grammar as read, with rules numbered for easy identification
- The start symbol of grammar (the only rule with \$)
- The terminals and non-terminals of the language

3 Testing

Return to your group with **at least one** non-trivial but verifiable grammar that each of your group members can use to test and compare their results. **I highly recommend some type of quick and dirty shared repository for reviewing your group's work during the next learning group time.**

3.1 Additional Hints

- Begin with a very trivial ($S \rightarrow a \ \$$ perhaps) language for testing, add to it slowly as your testing verifies correctness.
- Example output is on the next page. (I wrote my solution in Python on a Linux box, no big surprise there...).
- I chose to have my `grammar` script take a command-line argument. You can have yours do the same, or read `stdin`, or use a GUI, or even run on a Windows machine. This is the **small stuff**, don't sweat the small stuff.

assignlist.cfg

```
startGoal -> A $
A -> T A
    | lambda
T -> Var equal E
E -> a plus b
    | s times t
E -> zero | Var
Var -> e | f | g
    | h | j | k
```

Example output

```
Grammar Non-Terminals
startGoal, A, T, Var, E
Grammar Symbols
a, b, e, equal, f, g, h, j, k, plus, s, t, times, zero, $, startGoal, A, T, Var, E

Grammar Rules
(1)  startGoal -> A $
(2)  A -> T A
(3)  A -> lambda
(4)  T -> Var equal E
(5)  E -> a plus b
(6)  E -> s times t
(7)  E -> zero
(8)  E -> Var
(9)  Var -> e
(10) Var -> f
(11) Var -> g
(12) Var -> h
(13) Var -> j
(14) Var -> k

Grammar Start Symbol or Goal: startGoal
```