COMP 442 / 6421
Compiler Design

Tutorial 3
Syntactic Analyser

Instructor:	Dr. Joey Paquet	paquet@cse.concordia.ca
TAs:	Haotao Lai	h_lai@encs.concordia.ca

Lab Instructor

Section: lab hours NNK M------ 20:30-22:20 H819

Name: Haotao Lai (Eric)

Office: EV 8.241

Email: h_lai@encs.concordia

Website: http://laihaotao.me/ta

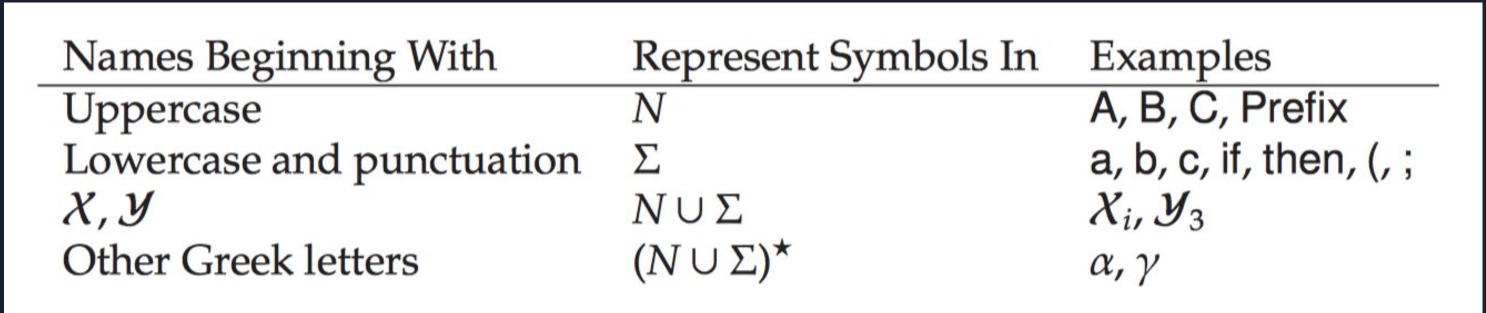
The Goal of Assignment 2

- input: a list of tokens from the first assignment
- output: an abstract syntax tree

1. Convert the given CFG to an LL(1) grammar
   a. Use tools to help your transformation procedure
   b. Remove the grammar from EBNF to non-EBNF representation
   c. Remove ambiguities and left recursions
   d. After each transformation step, verify that your grammar was not broken

2. Implement a LL(1) parser
   a. Recursive descent predictive parsing
   b. Table-driven predictive parsing

Notation



Names Beginning With	Represent Symbols In	Examples
Uppercase	N	A, B, C, Prefix
Lowercase and punctuation	Σ	a, b, c, if, then, (, ;
$\mathcal{X}, \mathcal{Y}$	$N \cup \Sigma$	$\mathcal{X}_i, \mathcal{Y}_3$
Other Greek letters	$(N \cup \Sigma)^\star$	α, γ

If A→γ is a production, then αAβ ⇒ αγβ denotes one step of a derivation using this production.

⇒ denote one step of a derivation of a production

⇒ + derives in one or more steps

⇒ * derives in zero or more steps

Convert CFG an LL(1)G

In context free grammar, all rules are one-to-one, one-to-many or one-to-none. These rule can be applied regardless of context [1]. A CFG can be defined as G(T, N, P, S), it is usually represented by using BNF notation.

Given the following grammar:

S → AA   A → a   A → b

The following input are valid:

aa   ab   ba   bb

[1] *https://en.wikipedia.org/wiki/Context-free_grammar*

Convert CFG an LL(1)G

LL(k) grammar is a formal grammar that can be parsed by an LL parser, which parse the input from left to right and constructs a leftmost derivation of the sentence. The k within the parenthesis is the number of token the parser will lookahead when parsing a sentence.

Given a grammar G with three productions: S → E   E → (E + E)   E → i
and input string: w = ((i + i) + i), the leftmost derivation will be:

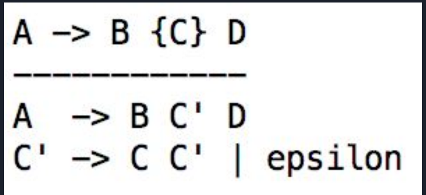
S → E → (E + E) → ((E + E) + E) → ((i + E) + E) → ((i + i) + E) → ((i + i) + i)

Remove EBNF Representation

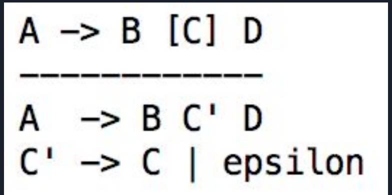
We need to remove two notations introduced by the EBNF format which are repetition and optional of the symbol.

Repetition example: A -> B {C} D
Optional example: A -> B [C] D

They all can be eliminated by introducing an new nonterminal symbol.



```
A -> B {C} D
------------
A  -> B C' D
C' -> C C' | epsilon
```



```
A -> B [C] D
------------
A  -> B C' D
C' -> C | epsilon
```

LL(1) Parsing

In LL(1) parsing, for each combination of a nonterminal and a input token, there should be only one possible production (if it is syntax valid) or no production (if it is a error state).

So we need to make sure after elimination of the EBNF notation, our grammar should be deterministic for each nonterminal symbol. A non-deterministic example can be: A -> B C | B D

Another situation that can lead to the failure of LL(1) parsing is left recursion of the production, an example can be: A -> A a | b. With such an production, you don't know where will be the ending point during the parsing.

Three Roadblocks

Quick review

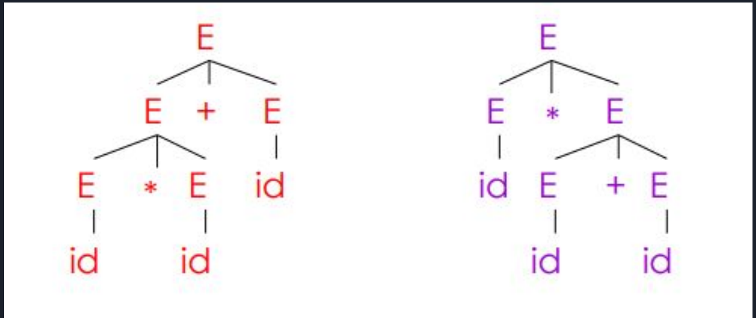
1. Ambiguity
2. Non-deterministic
3. Left recursion

For theoretical detail, see the lecture slide set [syntax analysis: introduction].

Ambiguity Grammar

Grammar: E -> E + E | E * E | id

Input string: id * id + id



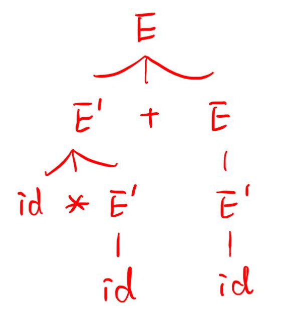
Requirement of the parse tree:

A tree that its in order traversal should give the string same as the input string

Ambiguity Grammar

The solution for ambiguity is rewrite the grammar (that’s exactly what you need to do in assignment 2) to make it unambiguous.

In this case, we want to enforce precedence of multiplication over addition.



original: E -> E + E | E * E | id

modified:

E -> E’ + E | E’

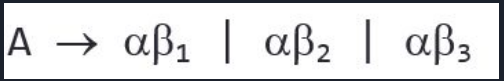
E’ -> id * E’ | id

Note

The modified grammar here is not a LL(1) grammar, the example here just show how to remove ambiguity.

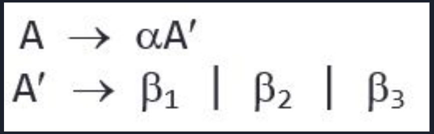
If you look carefully, you will find it is actually a LL(2) grammar

Non-deterministic Grammar



$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$$

1. backtracking can solve this problem, but it is inefficient;
2. introduce a new non-terminal which we refer as left factoring



$$\begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \end{aligned}$$

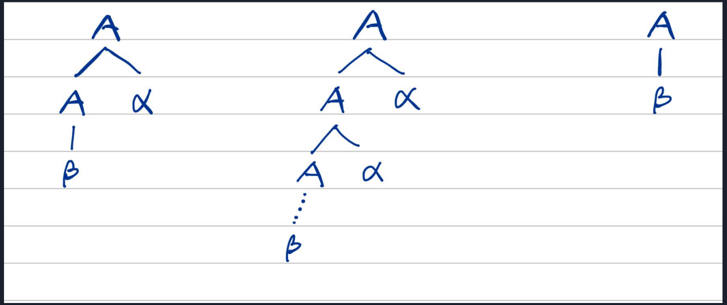
Left Recursion

Grammar: A -> Aα | β

By analyze these three possibilities, our goal is to construct something like: A -> βα*

But we don't allow * in the grammar, so we can replace a* with a new non-terminal A', so we have:

A -> βA'
A' -> αA' | ε



Perform Left Factoring

A -> B C | B D

A -> B’
B’ -> C | D

Eliminate Left Recursion

A -> A a | b

A -> b A'
A' -> a A' | epsilon

Parsing Example

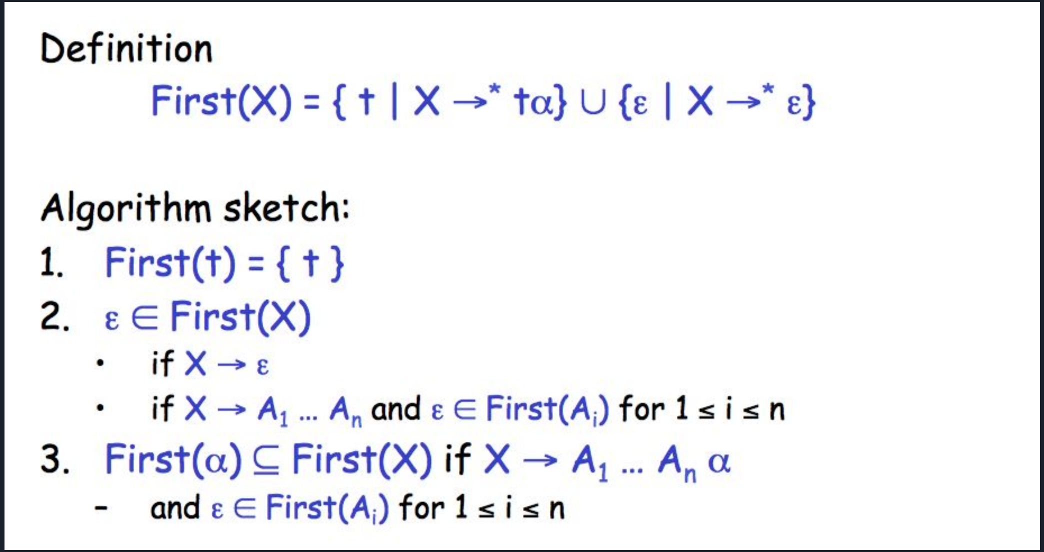
Assume we have the following grammar:

E → T + E | T
T → int | int * T | (E)

Left-factored grammar:

E →T X
T → (E) | int Y
X →+ E| ε
Y →* T | ε

First Set



Definition

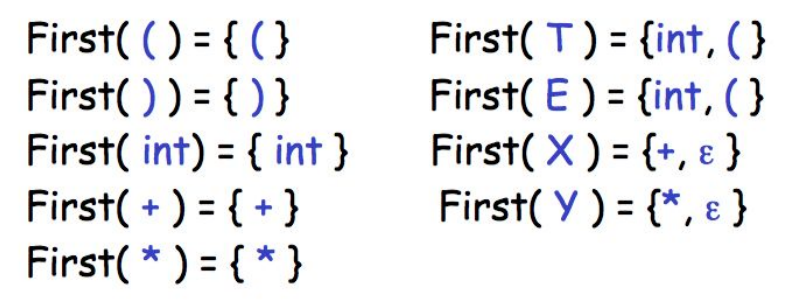
$$\mathrm{First}(X) = \{\, t \mid X \rightarrow^{*} t\alpha \} \cup \{\varepsilon \mid X \rightarrow^{*} \varepsilon\}$$

Algorithm sketch:

1. $\mathrm{First}(t) = \{\, t \,\}$
2. $\varepsilon \in \mathrm{First}(X)$
    - if $X \rightarrow \varepsilon$
    - if $X \rightarrow A_1 \ldots A_n$ and $\varepsilon \in \mathrm{First}(A_i)$ for $1 \leq i \leq n$
3. $\mathrm{First}(\alpha) \subseteq \mathrm{First}(X)$ if $X \rightarrow A_1 \ldots A_n\ \alpha$
    - and $\varepsilon \in \mathrm{First}(A_i)$ for $1 \leq i \leq n$

First Set

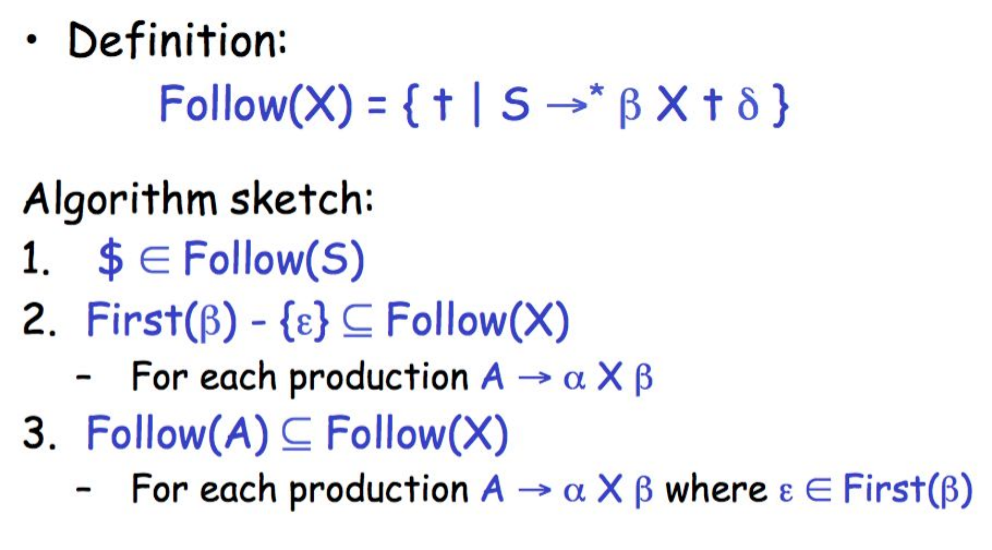
E →T X
T → (E) | int Y
X →+ E| ε
Y →* T | ε



First(( ) = { (}
First( )) = {) }
First(int) = { int }
First(+) = { + }
First(*) = { * }

First(T) = {int, (}
First(E) = {int, (}
First(X) = {+, ε }
First(Y) = {*, ε }

Follow Set



- Definition:

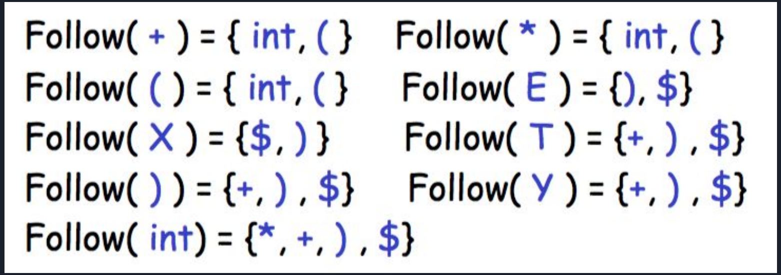
$$\text{Follow}(X) = \{ t \mid S \to^* \beta X t \delta \}$$

Algorithm sketch:

1. $\$ \in \text{Follow}(S)$
2. $\text{First}(\beta) - \{\varepsilon\} \subseteq \text{Follow}(X)$
   - For each production $A \to \alpha X \beta$
3. $\text{Follow}(A) \subseteq \text{Follow}(X)$
   - For each production $A \to \alpha X \beta$ where $\varepsilon \in \text{First}(\beta)$

Follow Set

E →T X
T → (E) | int Y
X →+ E| ε
Y →* T | ε



Follow(+) = { int, (}   Follow(*) = { int, (}
Follow(( ) = { int, (}   Follow(E) = {), $}
Follow(X) = {$,) }   Follow(T) = {+,) , $}
Follow( )) = {+,) , $}   Follow(Y) = {+,) , $}
Follow(int) = {*, +,) , $}

First Set and Follow Set

example 1:

S -> A B C D E
A -> a | ε
B -> b | ε
C -> c
D -> d | ε
E -> e | ε

First Set and Follow Set

example 2:

S -> B b | C d
B -> a B | ε
C -> c C | ε

First Set and Follow Set

example 3:

S -> A C B | C b B | B a
A -> d a | B C
B -> g | ε
C -> h | ε

the note shows what I did in the lab can access: http://laihaotao.me/ta/w18_comp442_fst_flw_set.pdf

How to come up with the proper grammar?

- You receive the initial grammar in EBNF in assignment 2 description already
- You need to remove the EBNF since AtoCC kfgEdit cannot understand this form
- Perform left factoring (if necessary)
- Remove left recursion (if exist, unfortunately, they exist in the given grammar)

It is strongly suggested that every time you make a single transformation step, that you use AtoCC to check whether your transformation broke the grammar or not.

Don’t try to correct many errors in one shot, it is easy to get lost. Plus, if you make a mistake in one transformation step and you carry on without checking, your further transformation will be made on a wrong grammar and thus be invalid.

Installing AtoCC

Installing AtoCC

- AtoCC can be downloaded at the following web site:
  - http://www.atocc.de
- You can either download an installer, or precompiled applications.

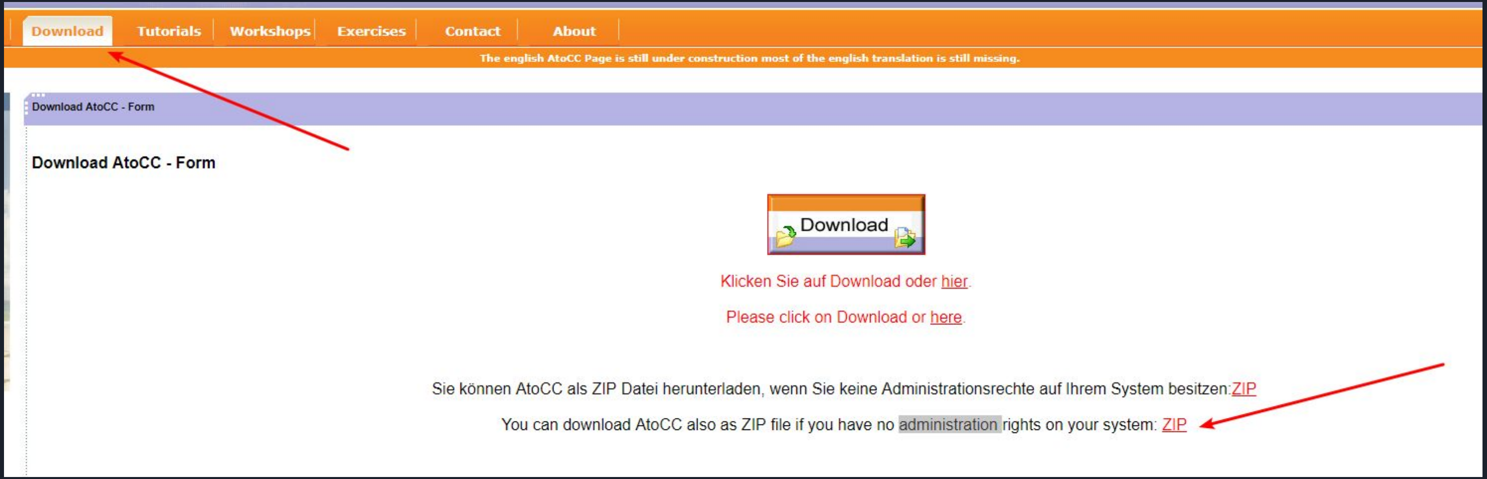
You don’t have Windows machine?

Check the following link out:

http://atocc.de/AtoCCFAQ/index.php?option=com_content&task=category&sectionid=11&id=25&Itemid=34

Works, for example, for macOS High Sierra version 10.13.1

Install AtoCC without administration rights ?



The english AtoCC Page is still under construction most of the english translation is still missing.

Download AtoCC - Form

Download AtoCC - Form

Klicken Sie auf Download oder hier.

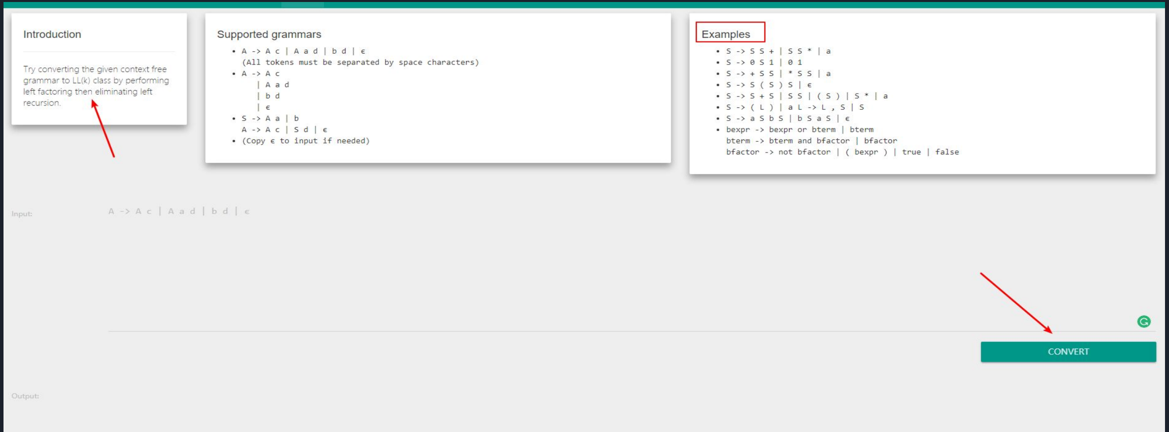
Please click on Download or here.

Sie können AtoCC als ZIP Datei herunterladen, wenn Sie keine Administrationsrechte auf Ihrem System besitzen:ZIP

You can download AtoCC also as ZIP file if you have no administration rights on your system: ZIP

These are portable executables, but they often crash, so save your work frequently!

Automated grammar transformation tools



Introduction

Try converting the given context free grammar to LL(k) class by performing left factoring then eliminating left recursion.

Supported grammars

```
• A -> A c | A a d | b d | ϵ
  (All tokens must be separated by space characters)
• A -> A c
     | A a d
     | b d
     | ϵ
• S -> A a | b
  A -> A c | S d | ϵ
• (Copy ϵ to input if needed)
```

Examples

```
• S -> S S + | S S * | a
• S -> 0 S 1 | 0 1
• S -> + S S | * S S | a
• S -> S ( S ) S | ϵ
• S -> S + S | S S | ( S ) | S * | a
• S -> ( L ) | a L -> L , S | S
• S -> a S b S | b S a S | ϵ
• bexpr -> bexpr or bterm | bterm
  bterm -> bterm and bfactor | bfactor
  bfactor -> not bfactor | ( bexpr ) | true | false
```

Input: A -> A c | A a d | b d | ϵ

Output:

- CyberZHG’s Compiler construction toolkit:

  https://cyberzhg.github.io/toolbox/

- Can help you apply specific transformations

- Use in conjunction with kfgEdit

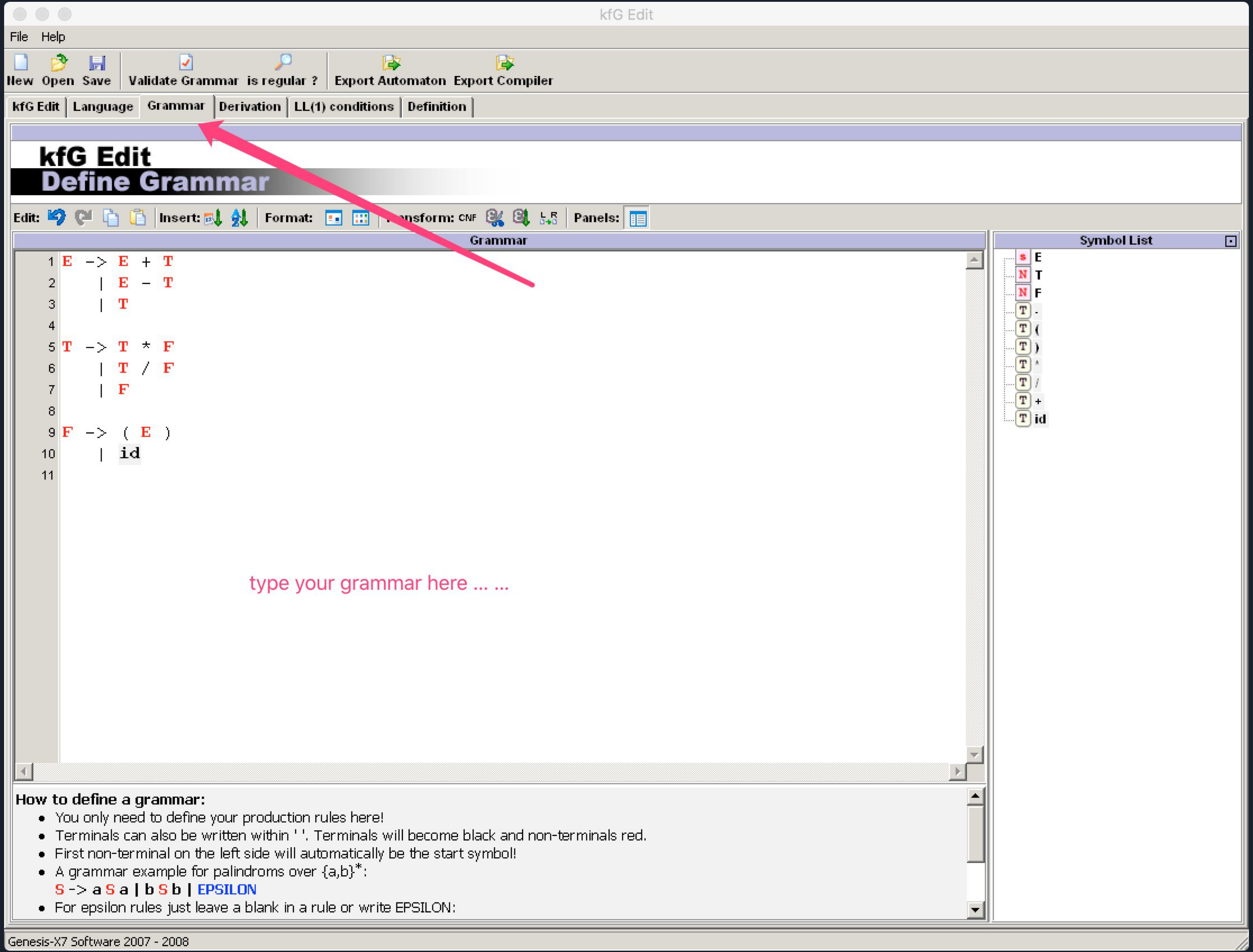
- However, it does not use the same grammar representation conventions

Example

--- How to use AtoCC for verification

AtoCC kfgEdit

- Tool that allows you to analyze your grammar and locate possible ambiguities in the grammar.
- After you grammar is entered, it also allows you to enter a string representing a token stream and verify if this token stream is derivable from the grammar. If it is, it generates a parse tree and a derivation for it.

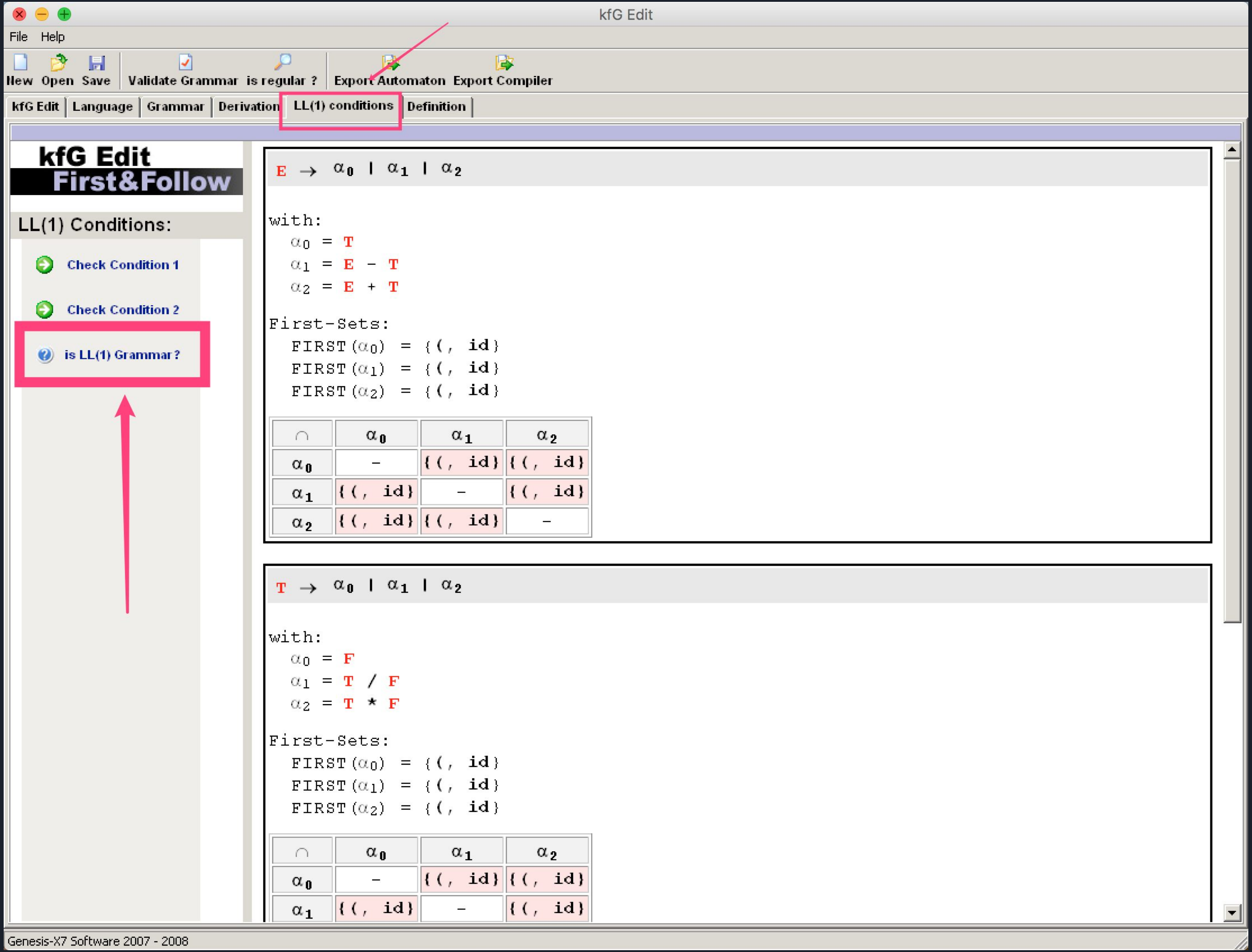


```
E -> E + T
   | E - T
   | T

T -> T * F
   | T / F
   | F

F -> ( E )
   | id

```

type your grammar here ... ...

How to define a grammar:

- You only need to define your production rules here!
- Terminals can also be written within ' '. Terminals will become black and non-terminals red.
- First non-terminal on the left side will automatically be the start symbol!
- A grammar example for palindroms over {a,b}*:
  S -> a S a | b S b | EPSILON
- For epsilon rules just leave a blank in a rule or write EPSILON:



LL(1) Conditions:

$E \rightarrow \alpha_0 \mid \alpha_1 \mid \alpha_2$

with:

α_0 = T
α_1 = E - T
α_2 = E + T

First-Sets:

FIRST(α_0) = {(, id}
FIRST(α_1) = {(, id}
FIRST(α_2) = {(, id}

∩	α_0	α_1	α_2
α_0	-	{(, id}	{(, id}
α_1	{(, id}	-	{(, id}
α_2	{(, id}	{(, id}	-

$T \rightarrow \alpha_0 \mid \alpha_1 \mid \alpha_2$

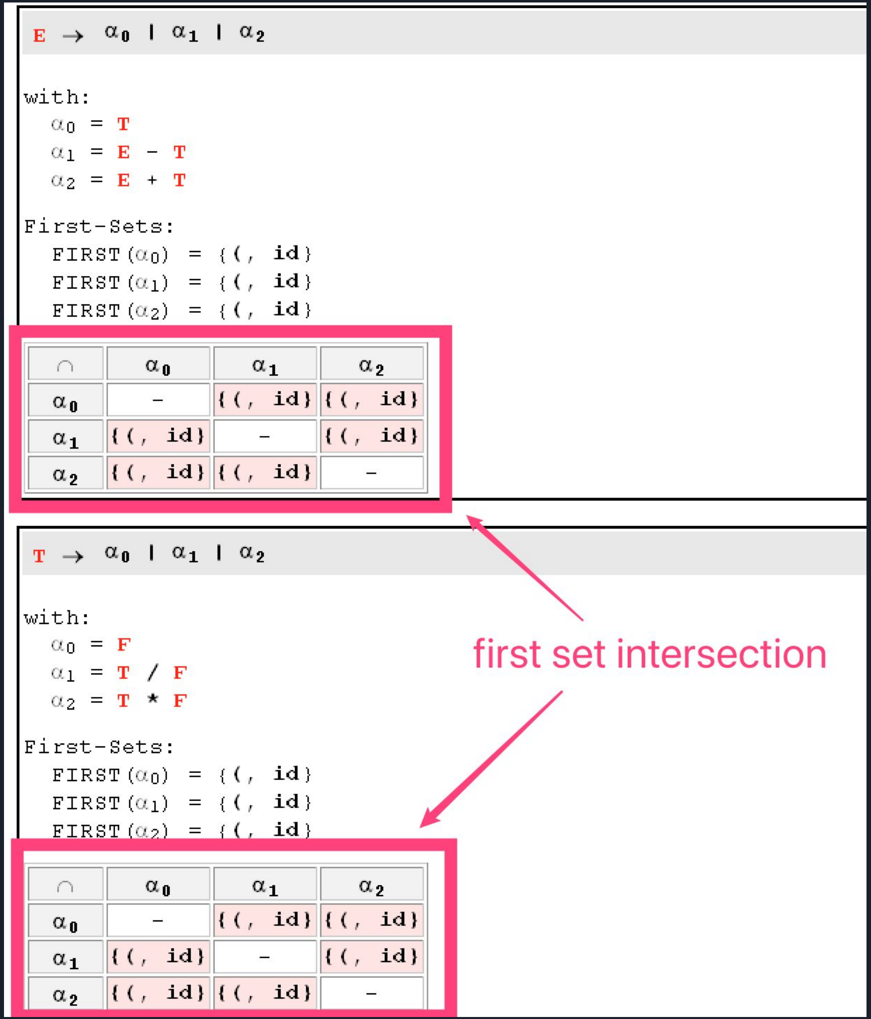
with:

α_0 = F
α_1 = T / F
α_2 = T * F

First-Sets:

FIRST(α_0) = {(, id}
FIRST(α_1) = {(, id}
FIRST(α_2) = {(, id}

∩	α_0	α_1	α_2
α_0	-	{(, id}	{(, id}
α_1	{(, id}	-	{(, id}



E → α_0 | α_1 | α_2

with:
  α_0 = T
  α_1 = E - T
  α_2 = E + T

First-Sets:
  FIRST(α_0) = {(, id}
  FIRST(α_1) = {(, id}
  FIRST(α_2) = {(, id}

∩	α_0	α_1	α_2
α_0	-	{(, id}	{(, id}
α_1	{(, id}	-	{(, id}
α_2	{(, id}	{(, id}	-

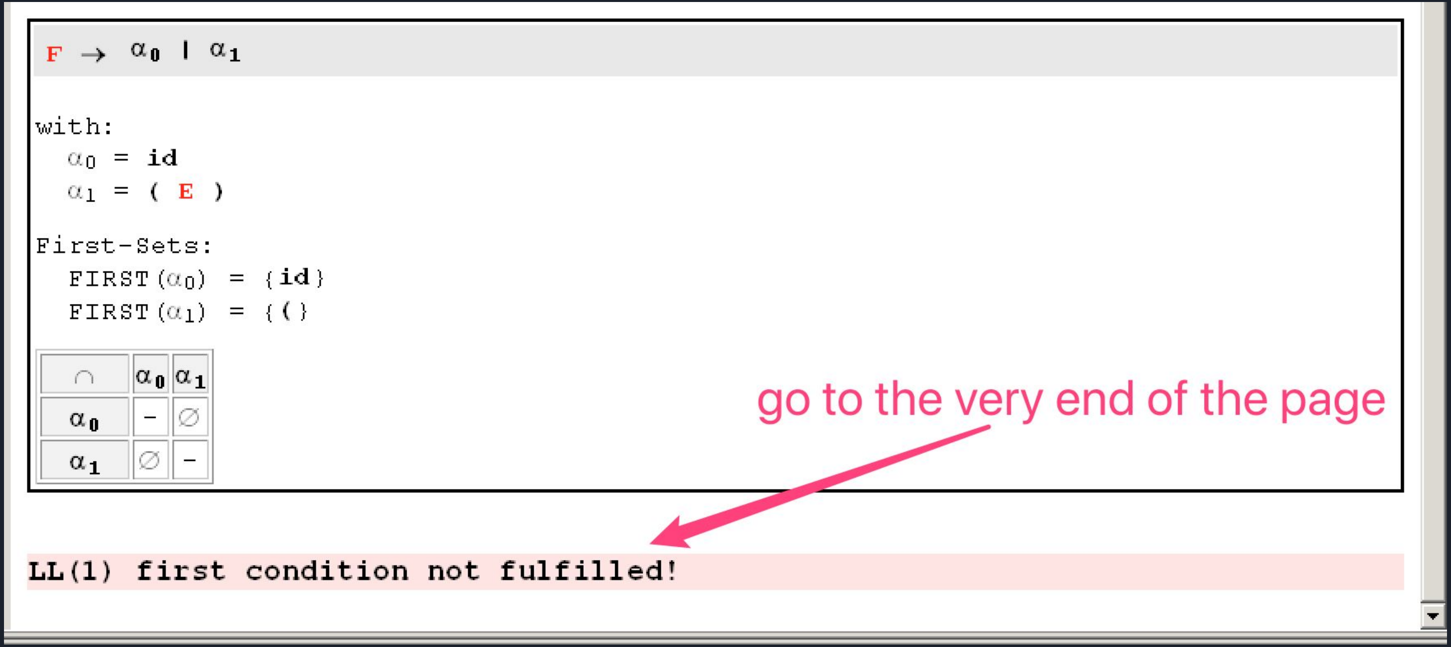
T → α_0 | α_1 | α_2

with:
  α_0 = F
  α_1 = T / F
  α_2 = T * F

First-Sets:
  FIRST(α_0) = {(, id}
  FIRST(α_1) = {(, id}
  FIRST(α_2) = {(, id}

∩	α_0	α_1	α_2
α_0	-	{(, id}	{(, id}
α_1	{(, id}	-	{(, id}
α_2	{(, id}	{(, id}	-

first set intersection



$F \rightarrow \alpha_0 \mid \alpha_1$

with:

α_0 = **id**

α_1 = **(** E **)**

First-Sets:

FIRST(α_0) = {**id**}

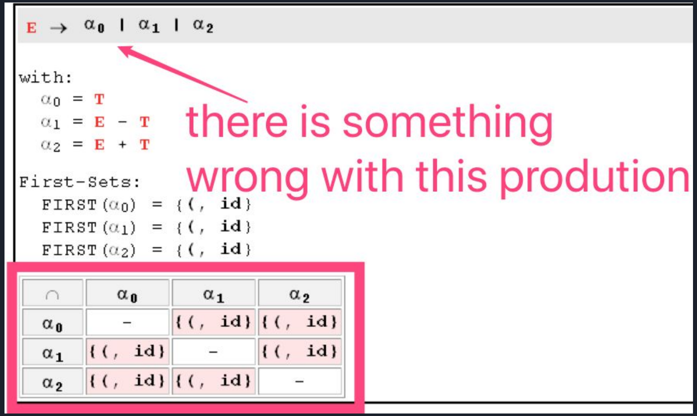
FIRST(α_1) = {**(**}

$\cap$	α_0	α_1
α_0	–	∅
α_1	∅	–

go to the very end of the page

LL(1) first condition not fulfilled!

What you should do?



$E \rightarrow \alpha_0 \mid \alpha_1 \mid \alpha_2$

with:

α_0 = T

α_1 = E - T

α_2 = E + T

First-Sets:

FIRST(α_0) = {(, id}

FIRST(α_1) = {(, id}

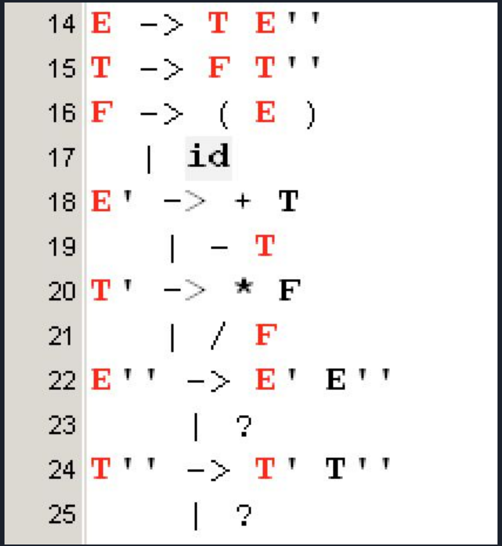
FIRST(α_2) = {(, id}

∩	α_0	α_1	α_2
α_0	-	{(, id}	{(, id}
α_1	{(, id}	-	{(, id}
α_2	{(, id}	{(, id}	-

there is something wrong with this prodution

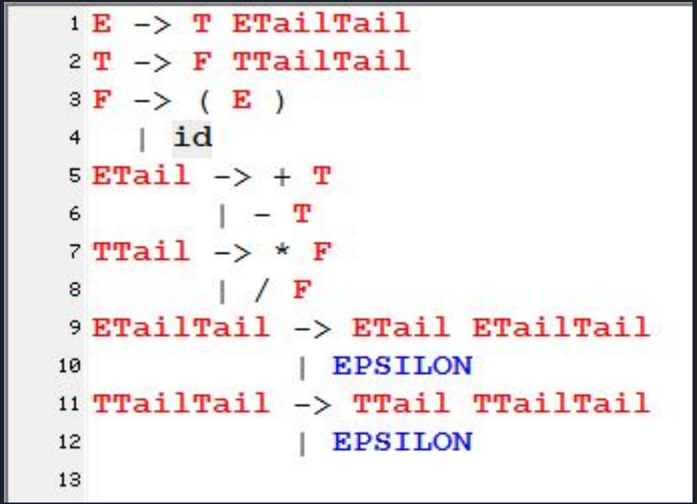
1. Locate a specific error and identify the faulty productions (shown in red)
2. Copy the related productions into the grammar transformation tool mentioned above (https://cyberzhg.github.io/toolbox/cfg2ll).
3. Copy the correction from the tool and paste it into AtoCC
4. Do some modification to adapt to AtoCC format
5. Check the grammar again

Note: Don't try to solve more than one production at a time. When you solve one production's error, use the tool to check to make sure you are not bringing new errors.



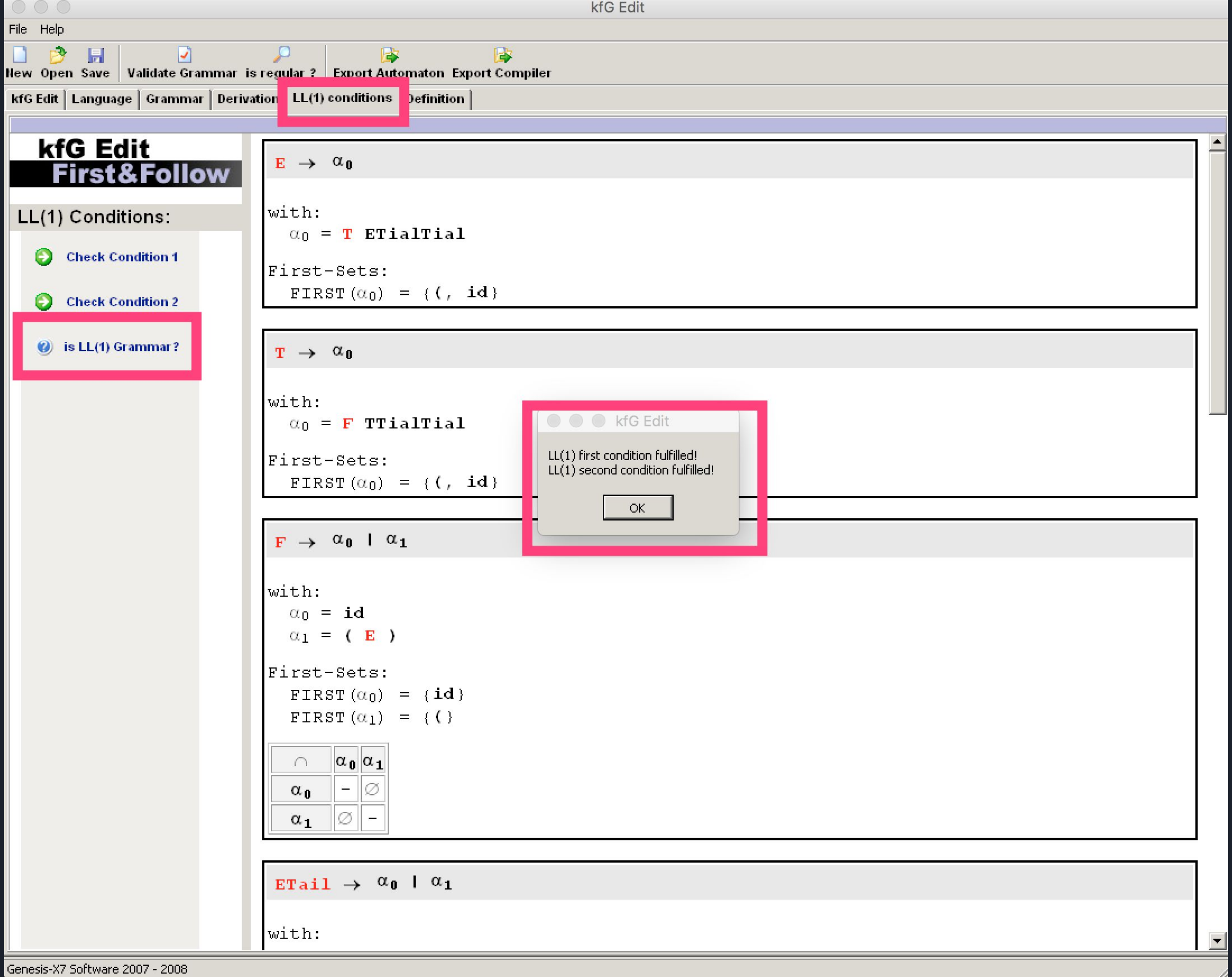
```
E -> T E''
T -> F T''
F -> ( E )
  | id
E' -> + T
    | - T
T' -> * F
    | / F
E'' -> E' E''
     | ?
T'' -> T' T''
     | ?
```

result from the tool



```
E -> T ETailTail
T -> F TTailTail
F -> ( E )
  | id
ETail -> + T
       | - T
TTail -> * F
       | / F
ETailTail -> ETail ETailTail
           | EPSILON
TTailTail -> TTail TTailTail
           | EPSILON
```

after modification, adapted to AtoCC



$E \rightarrow \alpha_0$

with:
α_0 = T ETialTial

First-Sets:
FIRST(α_0) = {(, id}

$T \rightarrow \alpha_0$

with:
α_0 = F TTialTial

First-Sets:
FIRST(α_0) = {(, id}

kfG Edit

LL(1) first condition fulfilled!
LL(1) second condition fulfilled!

$F \rightarrow \alpha_0 \mid \alpha_1$

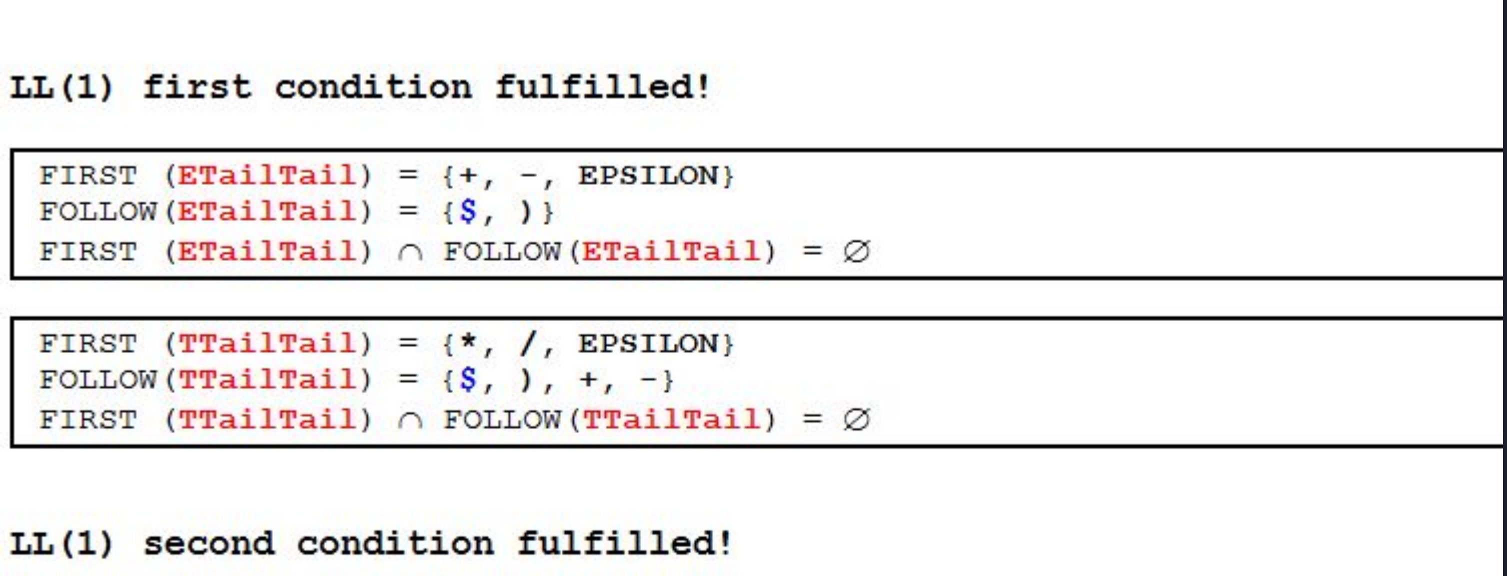
with:
α_0 = id
α_1 = (E)

First-Sets:
FIRST(α_0) = {id}
FIRST(α_1) = {(}

$\cap$	α_0	α_1
α_0	–	∅
α_1	∅	–

$ETail \rightarrow \alpha_0 \mid \alpha_1$

with:



LL(1) first condition fulfilled!

```
FIRST (ETailTail) = {+, -, EPSILON}
FOLLOW(ETailTail) = {$, )}
FIRST (ETailTail) ∩ FOLLOW(ETailTail) = ∅
```

```
FIRST (TTailTail) = {*, /, EPSILON}
FOLLOW(TTailTail) = {$, ), +, -}
FIRST (TTailTail) ∩ FOLLOW(TTailTail) = ∅
```

LL(1) second condition fulfilled!

Tool

If you plan to use the table-driven approach, you will need a parse table. Of course you can generate your own parse table, or put a proper grammar into a tool and it will give you the table.

We propose an online tool to do that:
http://hackingoff.com/compilers/ll-1-parser-generator

note: you need to use EPSILON to repersent ε

Thanks