COMP 442 /6421 Compiler Design

Grammars and Parsing

Instructor: Dr. Joey Paquet paquet@cse.concordia.ca
TAs: Vashisht Marhwal vmarhwal97@gmail.com
Hamed Jafarpour hamed.jafarpour@concordia.ca

Assignment 2 :Syntax Analysis (Highlights)

- To achieve assignment #2, there are 2 stages:
  - Transform the grammar into an LL(1) grammar
  - Implement the parser
- **The implementation absolutely cannot start before the grammar has been transformed**.
  - Set of tools to help achieve the transformation
  - Sample usage of these tools.

The Goal of Assignment 2

1. Convert the given CFG to an LL(1) grammar
   a. Use tools to help your transformation procedure
   b. Change EBNF to non-EBNF representation in grammar
   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 predictiveparsing

Obstacles to overcome in Assignment 2

Quick review

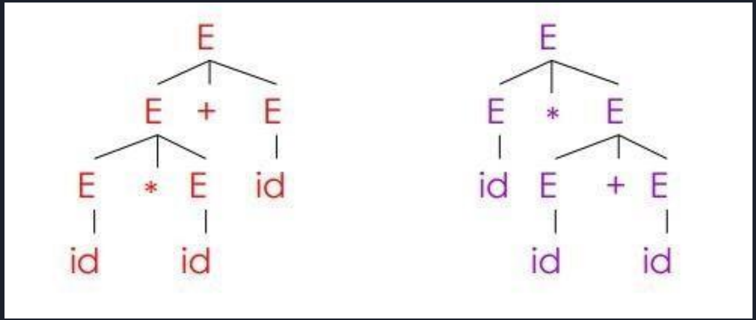
1. Ambiguity
2. Non-deterministic
3. Left recursion

For in-detail theory, see the lecture slide set [syntax analysis: introduction].

Ambiguity Grammar

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

Input string: id * id + id



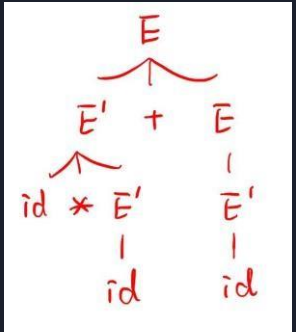
Requirement of the parse tree:

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

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

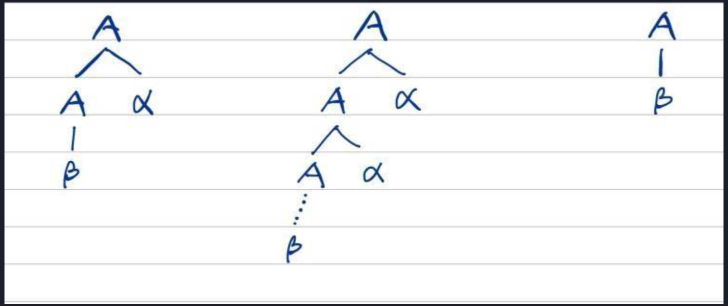
$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$$

Left Recursion

Garmmar: 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



Example: removing EBNF constructs

Assume you was given a grammar as following, with EBNF repetition:

```
commaSeparatedList        -> a {,a}  |  EPSILON
```

You should remove the EBNF repetition and come up with the following grammar:

```
commaSeparatedList        -> a commaSeparatedListTail
                           | EPSILON
commaSeparatedListTail    -> ,a commaSeparatedListTail
                           | EPSILON
```

Example: removing left recursion

After removal of all EBNF format instances, assume you have something like:

```
expr   -> expr + term    | term
term   -> term * factor | factor
factor -> '(' expr ')'   | 'x'
```

Remove left recursions (on `expr` and `term`) using the transformation shown in class:

1- Isolate each set of productions of the form:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \qquad \text{(left-recursive)}$$

$$A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \qquad \text{(non-left-recursive)}$$

2- Introduce a new non-terminal A'

3- Change all the non-recursive productions on A to:

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots$$

4- Remove the left-recursive production on A and substitute:

$$A' \rightarrow \varepsilon \mid \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots \qquad \text{(right-recursive)}$$

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.
- How to install A to CC were described in previous labs

How to come up with the proper grammar?

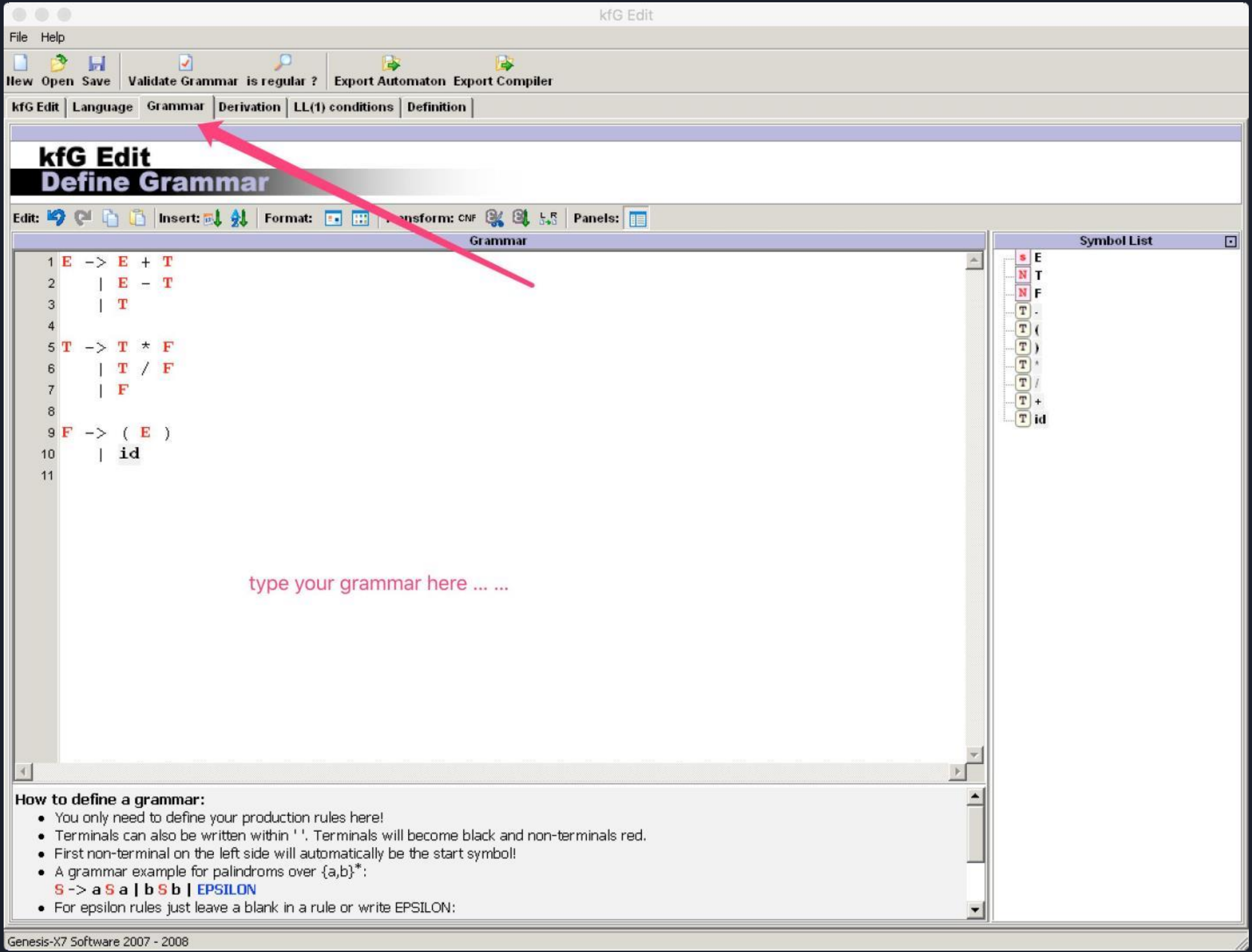
- You receive the initial grammar in EBNF in assignment 2 description already
- You need to remove the EBNF since AtoCC kfgEdit cannot understand this form
- Perform left factoring (if necessary)
- Remove left recursion (if exist, unfortunately, they exist in the given grammar)

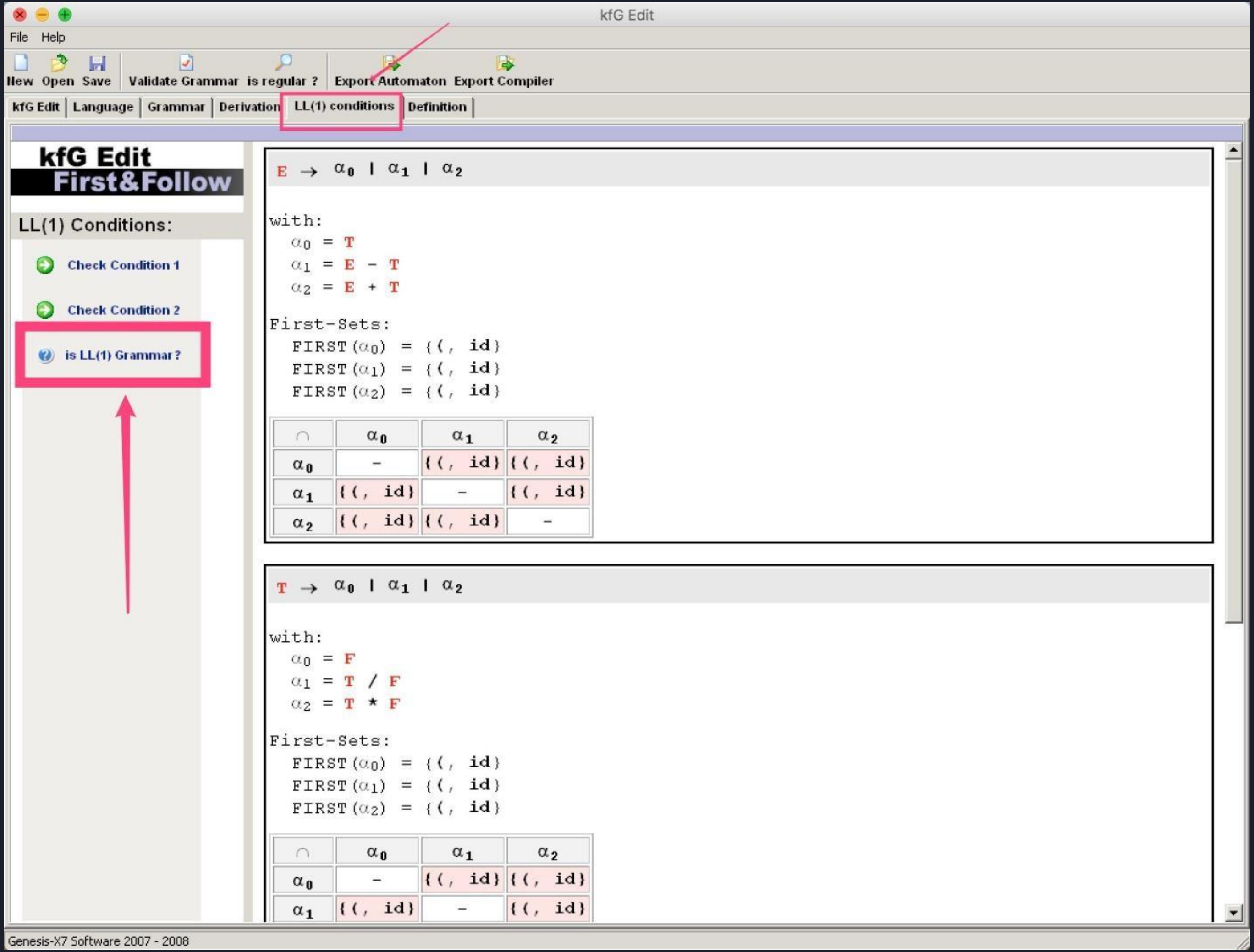
It is strongly suggested that every time you make a single transformation step, that you use AtoCC to check whether your transformation broke the grammar or not.

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.

Example

--- How to use AtoCC for verification


kfG Edit
File Help
New Open Save Validate Grammar is regular ? Export Automaton Export Compiler
kfG Edit | Language | Grammar | Derivation | LL(1) conditions | Definition
kfG Edit
Define Grammar
Edit: Insert: Format: Transform: CNF Panels:
Grammar
E -> E + T
| E - T
| T
T -> T * F
| T / F
| F
F -> (E)
| id
type your grammar here ... ...
Symbol List
S E
N T
N F
T -
T (
T)
T *
T /
T +
T id
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:
Genesis-X7 Software 2007 - 2008


kfG Edit
File Help
New Open Save Validate Grammar is regular ? Export Automaton Export Compiler
kfG Edit | Language | Grammar | Derivation | LL(1) conditions | Definition
kfG Edit
First&Follow
LL(1) Conditions:
Check Condition 1
Check Condition 2
is LL(1) Grammar ?
E → α0 | α1 | α2
with:
α0 = T
α1 = E – T
α2 = E + T
First-Sets:
FIRST(α0) = {(, id}
FIRST(α1) = {(, id}
FIRST(α2) = {(, id}
∩ α0 α1 α2
α0 – {(, id} {(, id}
α1 {(, id} – {(, id}
α2 {(, id} {(, id} –
T → α0 | α1 | α2
with:
α0 = F
α1 = T / F
α2 = T * F
First-Sets:
FIRST(α0) = {(, id}
FIRST(α1) = {(, id}
FIRST(α2) = {(, id}
∩ α0 α1 α2
α0 – {(, id} {(, id}
α1 {(, id} – {(, id}
Genesis-X7 Software 2007 - 2008

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

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}
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**}

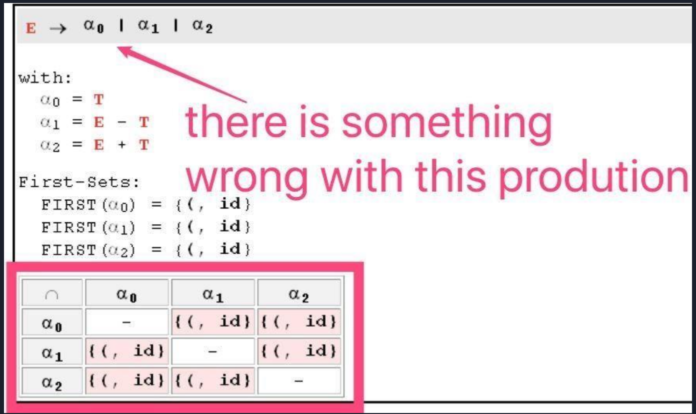
FIRST(α_1) = {**(**}

$\cap$	α_0	α_1
α_0	-	$\emptyset$
α_1	$\emptyset$	-

go to the very end of the page

LL(1) first condition not fulfilled!

What you should do?



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

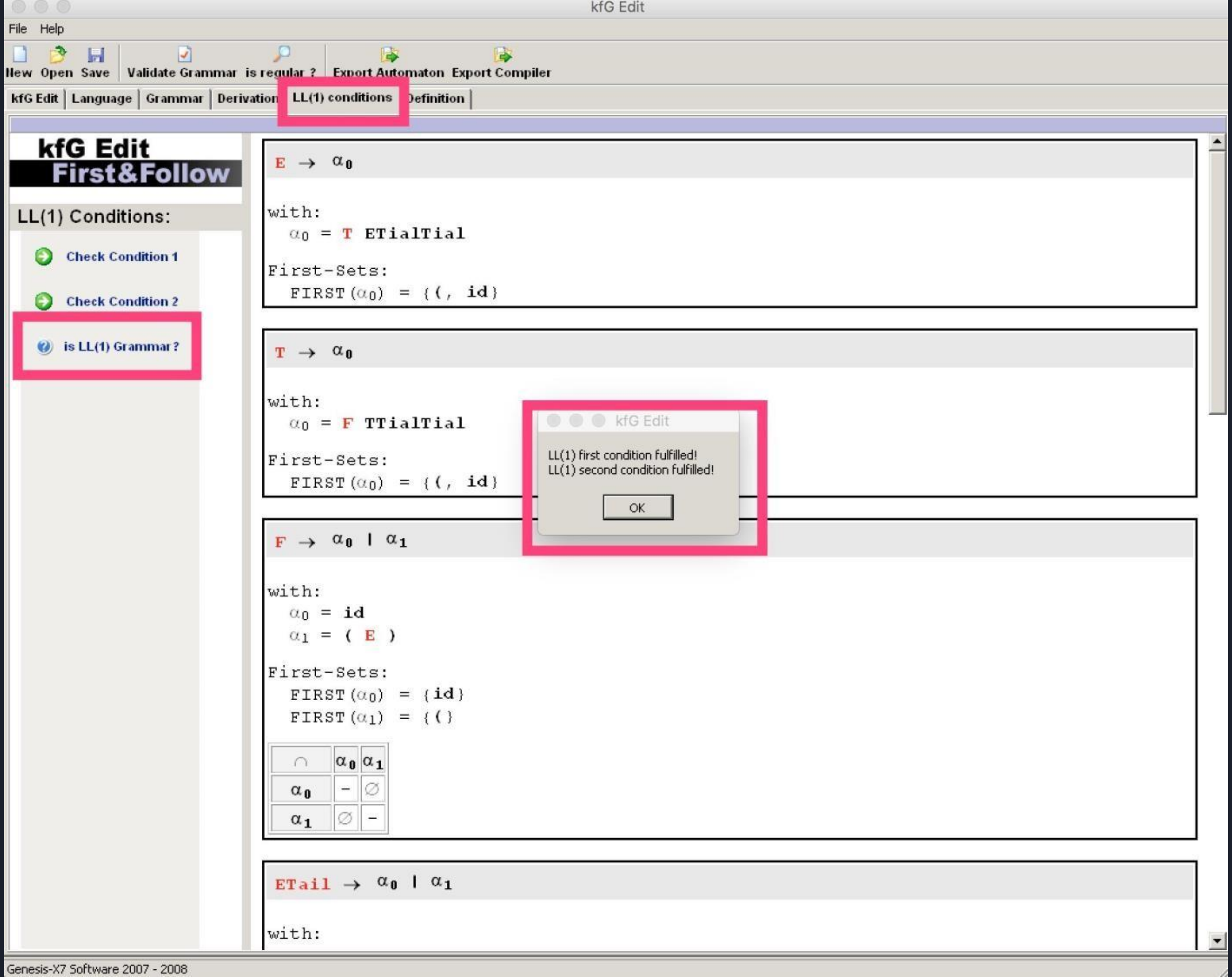
Note: Don't try to solve more than one production at a time. When you solve one production's error, use the tool to check to make sure you are not bringing new errors.

```
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


kfG Edit
File Help
New Open Save Validate Grammar is regular ? Export Automaton Export Compiler
kfG Edit Language Grammar Derivation LL(1) conditions Definition
kfG Edit
First&Follow
LL(1) Conditions:
Check Condition 1
Check Condition 2
is LL(1) Grammar ?
E → α0
with:
α0 = T ETialTial
First-Sets:
FIRST(α0) = {(, id}
T → α0
with:
α0 = F TTialTial
First-Sets:
FIRST(α0) = {(, id}
kfG Edit
LL(1) first condition fulfilled!
LL(1) second condition fulfilled!
OK
F → α0 | α1
with:
α0 = id
α1 = (E)
First-Sets:
FIRST(α0) = {id}
FIRST(α1) = {(}
ETail → α0 | α1
with:
Genesis-X7 Software 2007 - 2008

LL(1) first condition fulfilled!

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

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

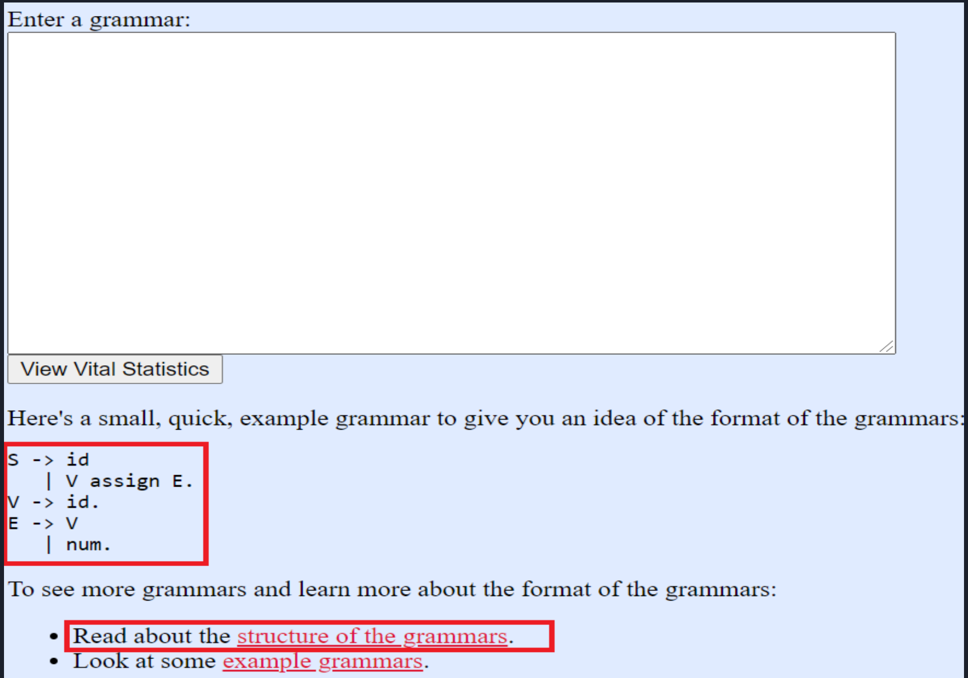
LL(1) second condition fulfilled!

Example

--- How to use University of Calgary Tool

Understand the format of University Calgary grammar tool

- Go to the link – (https://smlweb.cpsc.ucalgary.ca/start.html)
- Enter the grammar in below format.



Enter a grammar:

View Vital Statistics

Here's a small, quick, example grammar to give you an idea of the format of the grammars:

```
S -> id
   | V assign E.
V -> id.
E -> V
   | num.
```

To see more grammars and learn more about the format of the grammars:

- Read about the structure of the grammars.
- Look at some example grammars.

Use Previous Example

Enter a grammar:

```
E -> E + T
     | E minus T
     | T.

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

F -> ( E )
      | id.
```

View Vital Statistics

View statistics

Grammar

```
E → E + T
  | E minus T
  | T.
T → T * F
  | T / F
  | F.
F → ( E )
  | id.
```

Some sentences generated by this grammar: {id, (id), id / id, id + id, id * id, id, id, id + id * id, id, id / id * id, id, id + id / id, id, id * id * id, id, id minus id * id, id, id minus id

- All nonterminals are reachable and realizable.
- There are no nullable nonterminals.
- The endable nonterminals are: F E T.
- No cycles.

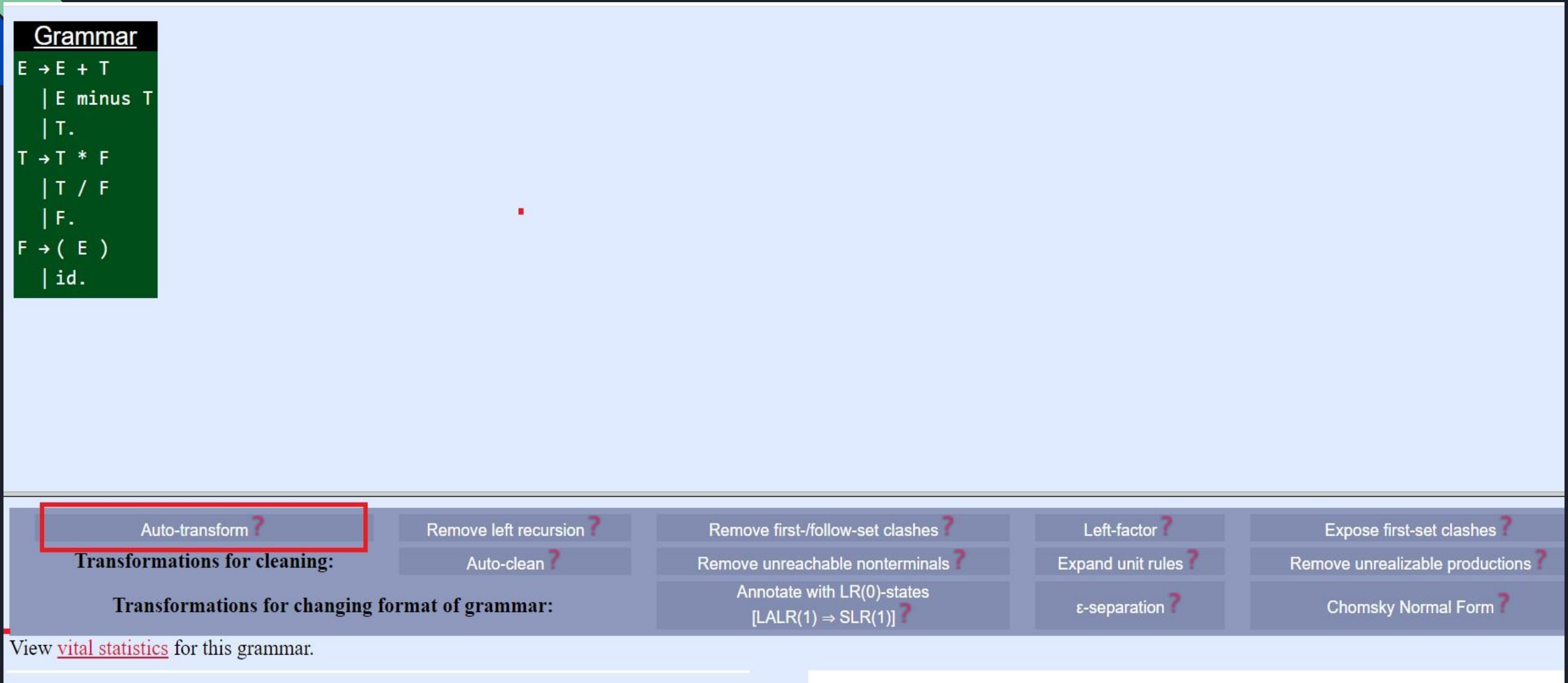
nonterminal	first set	follow set	nullable	endable
E	(id	+ minus)	no	yes
T	(id	* / + minus)	no	yes
F	(id	* / + minus)	no	yes

The grammar is not LL(1) because:

- E is left recursive.
- T is left recursive.

- attempt to transform the grammar (to LL(1))

Transforming to LL(1) Grammar



Transforming to LL(1) Grammar

Old Grammar

Grammar

```
E → E + T
  | E minus T
  | T.
T → T * F
  | T / F
  | F.
F → ( E )
  | id.
```

Grammar

$E \rightarrow T\ E_1\ .$

$E_1 \rightarrow +\ T\ E_1$

$\quad |\ minus\ T\ E_1$

$\quad |\ .$

$T \rightarrow F\ T_1\ .$

$T_1 \rightarrow *\ F\ T_1$

$\quad |\ /\ F\ T_1$

$\quad |\ .$

$F \rightarrow (\ E\ )$

$\quad |\ id\ .$

Performed the 'removing left recursion' transformation.

The grammar is now LL(1).

Auto-transform ?	Remove left recursion ?	Remove first-/follow-set clashes ?	Left-factor ?
Transformations for cleaning:	Auto-clean ?	Remove unreachable nonterminals ?	Expand unit rules ?
Transformations for changing format of grammar:		Annotate with LR(0)-states [LALR(1) ⇒ SLR(1)] ?	ε-separation ?

View vital statistics for this grammar.

Grammar is LL(1)

Grammar

```
E → T E1 .
E1 → + T E1
   | minus T E1
   | .
T → F T1 .
T1 → * F T1
   | / F T1
   | .
F → ( E )
   | id .
```

Some sentences generated by this grammar: {i
id * id, id * id / id / id, id / id * id / id, id * id /

- All nonterminals are reachable and realizable.
- The nullable nonterminals are: E_1 T_1.
- The endable nonterminals are: T_1 F E E_1 T.
- No cycles.

nonterminal	first set	follow set	nullable	endable
E	(id	)	no	yes
E_1	+ minus	)	yes	yes
T	(id	) + minus	no	yes
T_1	* /	) + minus	yes	yes
F	(id	) * / + minus	no	yes

The grammar is LL(1).

First Set and Follow Set

example 1 in format of university of calgary tool :

E -> T E'.

E' ->

| + T E'.

T -> F T'.

T' ->

|* F T'.

F -> 0

| 1

| (E).

Note: Here, E` -> represents the epsilon

Grammar

```
E → T E'.
E' →
    | + T E'.
T → F T'.
T' →
    | * F T'.
F → 0
    | 1
    | ( E ).
```

Some sentences generated by this grammar: {1, 0, 0 + 0, 0 + * 1 * 1, 0 * 0 * 0}

```
FIRST(E)   = {0,1,(}
FIRST(E')  = {+, ε}
FIRST(T)   = {0,1,(}
FIRST(T')  = {*, ε}
FIRST(F)   = {0,1,(}
```

```
FOLLOW(E)  = {$,)}
FOLLOW(E') = {$,)}
FOLLOW(T)  = {+,$,)}
FOLLOW(T') = {+,$,)}
FOLLOW(F)  = {*,+,$,)}
```

- All nonterminals are reachable and realizable.
- The nullable nonterminals are: E' T'.
- The endable nonterminals are: T' F E E' T.
- No cycles.

nonterminal	first set	follow set	nullable	endable
E	0 1 (	)	no	yes
E'	+	)	yes	yes
T	0 1 (	) +	no	yes
T'	*	) +	yes	yes
F	0 1 (	) * +	no	yes

The grammar is LL(1).

Generate LL(1) Parsing Table

- All nonterminals are reachable and realizable.
- The nullable nonterminals are: E' T'.
- The endable nonterminals are: T' F E E' T.
- No cycles.

nonterminal	first set	follow set	nullable	endable
E	0 1 (	)	no	yes
E'	+	)	yes	yes
T	0 1 (	) +	no	yes
T'	*	) +	yes	yes
F	0 1 (	) * +	no	yes

The grammar is LL(1).

- attempt to transform the grammar (to LL(1))
- generate LL(1) parsing table

Generate LL(1) Parsing Table

Grammar

```
E → T E'.
E' →
   | + T E'.
T → F T'.
T' →
   | * F T'.
F → 0
   | 1
   | ( E ).
```

	$	)	(	1	0	*	+
E			E → T E'	E → T E'	E → T E'		
E'	E' → &epsilon	E' → &epsilon					E' → + T E'
T			T → F T'	T → F T'	T → F T'		
T'	T' → &epsilon	T' → &epsilon				T' → * F T'	T' → &epsilon
F			F → (E)	F → 1	F → 0		

Return home to enter a new grammar.

First Set and Follow Set

example 2:

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

Note: covert the grammar into corresponding format before using the tool

First Set and Follow Set

example 3:

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

First Set and Follow Set

example 4:

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

Tool given in Assignment handout

- In assignment 2 ZIP file refer the read me file.
- Follow the steps given in read me file
- Remove the ambiguity from the grammar
- Convert the grammar to the university of Calgary grammar tool or A to CC format
- Analyze your grammar .

Thanks