CS 315 – Programming Languages
Parsing

CFG (context-free grammar)

- Parser uses CFG rules to perform a *derivation* and create a *parse tree*

For any CFG, we can construct a parser that runs in $O(n^3)$ time, where n is the input size (# of tokens).

We are interested in more scalable parsers, but these parsers cannot work with arbitrary CFGs

- LL parsers work with LL-grammars
  - LL: **L**eft-to-right input order, **L**eftmost derivation
  - These are *top-down* parsers
- LR parsers work with LR-grammars
  - LR: **L**eft-to-right input order, **R**ightmost derivation (in reverse)
  - These are *bottom-up* parsers

Top-down: the parse tree is constructed from top (root first) to bottom (leaves last)
Bottom-up: the parse tree is constructed from bottom (leaves first) to top (root last)

The class of LR grammars is larger than the class of LL grammars, i.e., more CFG grammars are in the former class.

Let us see an example. Consider the following grammar, which is both an LL and an LR grammar:

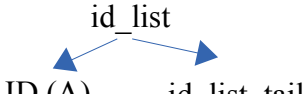
1. id_list → ID id_list_tail
2. id_list_tail → ',' ID id_list_tail
3. id_list_tail → ';'

Also consider the input: A, B, C;

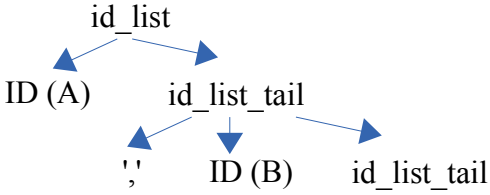
<u>Top-down parsing</u>

1) id_list

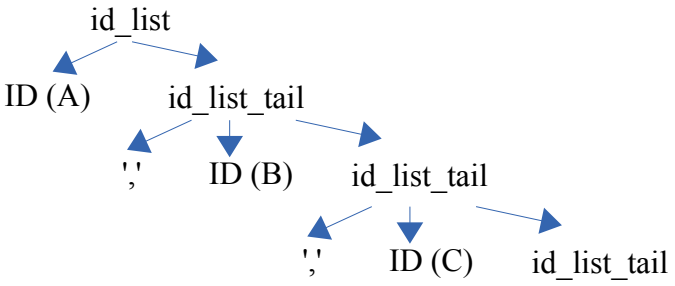
2)



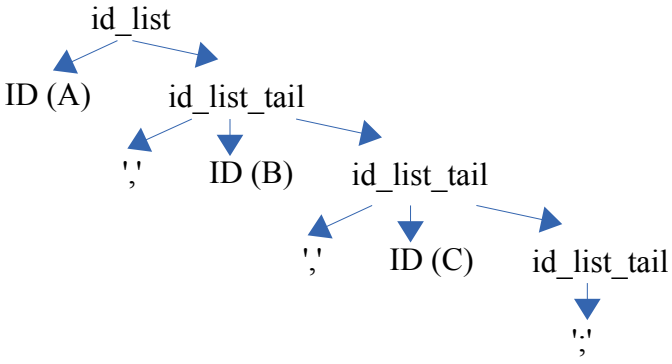
3)



4)



5)



Bottom-up parsing

Either push the next token from the input into the rightmost position in the queue
or
Reduce a set of tokens from the rightmost end of the queue using one of the rules

0) shift

1) ID(A) shift

2) ID(A) ',' shift

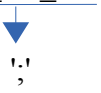
3) ID(A) ',' ID(B) shift

4) ID(A) ',' ID(B) ',' shift

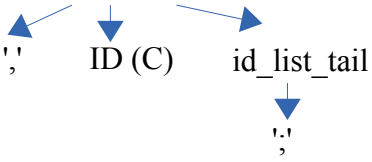
5) ID(A) ',' ID(B) ',' ID(C) shift

6) ID(A) ',' ID(B) ',' ID(C) ';' reduce 3

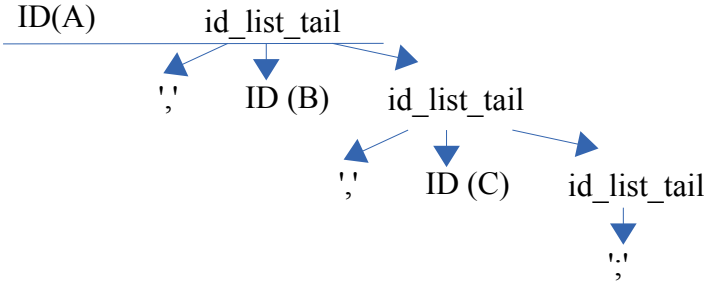
7) ID(A) ',' ID(B) ',' ID(C) id_list_tail reduce 2



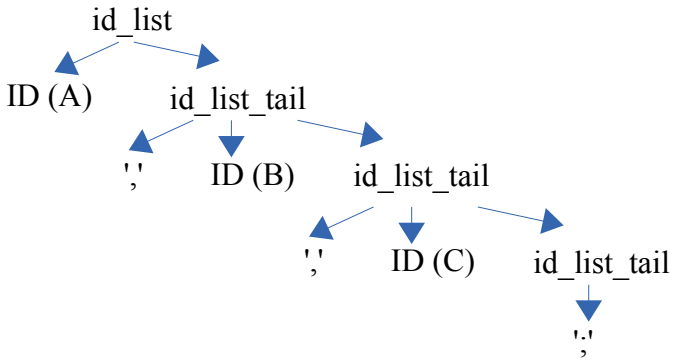
8) ID(A) ',' ID(B) id_list_tail reduce 2



9) ID(A) id_list_tail reduce 1



10)



A few things to note:

- We have not learned how to programmatically create a parse tree, given an LL or LR grammar.
- For now, we are doing the construction using our intuition. Sometimes there are multiple choices, if you pick the wrong one, you cannot complete the parse tree creation.
- In the upcoming lectures we will learn programmatically creating a parse tree given an LL grammar. The same for LR grammars is left for a compilers course.
- You might have noticed that our bottom-up parse above resulted in pushing the entire input first, and then reducing it. This is undesirable and is a consequence of the right recursion in the grammar. As we will see, converting it into left recursion will improve the bottom-up parse, but making the grammar a non-LL grammar. → left recursion is good for LR parsing.

LL-parser: Also called predictive parser

- It predicts what to see next in the input based on the current partial derivation and uses the appropriate rule to expand the leftmost nonterminal

→ look at the next char in the input, and use the CFG accordingly.

LR-parser: Also called the shift-reduce parser

- It pushes tokens into a queue until it sees a complete RHS of a production rule at the end of the queue, which is then reduced to the LHS of the rule. Sometimes there is a choice to be made between pushing and reducing (pushing now may make it possible to reduce later) → Example of this???

→ shift until you see a fit that looks like one of the productions in your CFG.

LL-parser: from top to bottom, we get a leftmost derivation
LR-parser: From bottom to top, we get a rightmost derivation

There are different classes of LR parsers, such as (from more powerful to less powerful):

- LR(k)
- LALR(k)
- SLR

} This is how you waste my time

LL parsers can also be generalized as LL(k) parsers. Here *k* is the number of tokens in the input you can use for looking ahead before you make a decision.

Consider the following grammar for the problem of identifier lists we used before. The grammar is changed to use left recursion, which makes it a non-LL grammar, but it is still an LR grammar. Note that we have still not learned what makes a grammar an LL-grammar.

1. id_list → id_list_prefix ';'
2. id_list_prefix → id_list_prefix ',' ID
3. id_list_prefix → ID

→ recalls the LHS!

Let us do a bottom-up derivation for this grammar, again using the input:

A, B, C ;

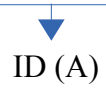
0) shift

1) ID(A) reduce 3

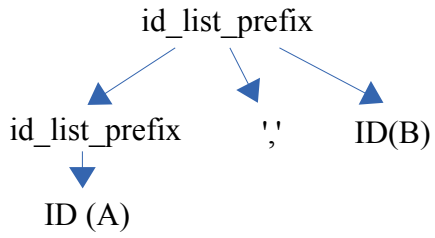
2) id_list_prefix shift

3) id_list_prefix ',' shift

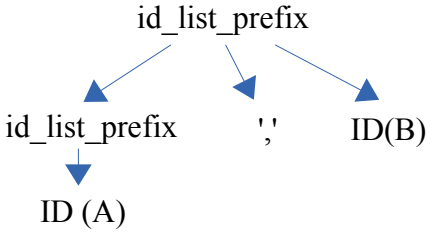
4) id_list_prefix ',' ID (B) reduce 2



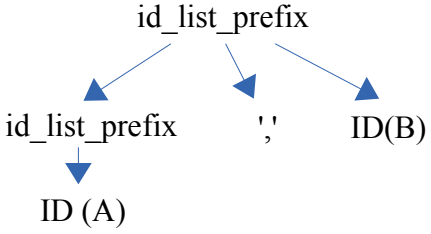
5) shift



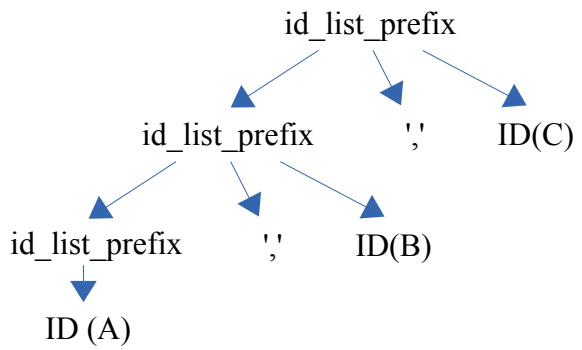
6) ',' shift



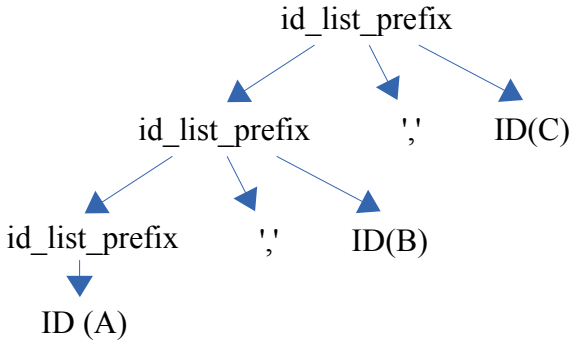
7) ',' ID(C) reduce 2



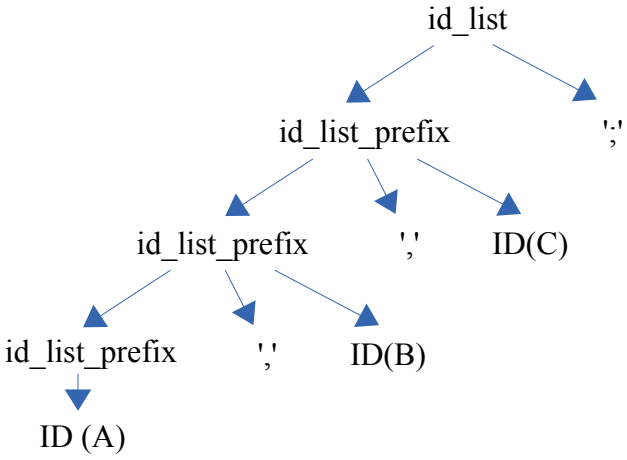
8) shift



9) ';' reduce 1



10)



LL Grammars

Consider our famous CFG for arithmetic expressions:

1. expr → expr add_op term
2. expr → term
3. term → term mult_op factor
4. term → factor
5. factor → ID
6. factor → NUM
7. factor → '(' expr ')'
8. add_op → '+'
9. add_op → '-'
10. mul_op → '*'
11. mul_op → '/'

This grammar creates problems for LL(1) parsing:

- Assume we have *expr* as our leftmost nonterminal in our current partial derivation and we have ID as our current input token. Which rule do we apply? Is it rule 1 or rule 2? Note that both rule 1 and rule 2 can expand to eventually produce an ID as the first thing.
- Left recursion: We can expand rule 1 as many times as we want, without impacting the next character from the input that can be consumed.

There are two conditions to ensure that a grammar is LL(1):

1. If two rules have the same LHS, then the set of input characters for which these rules are applicable (called the PREDICT set) must be **non-overlapping**. We will more formally define the PREDICT set a little later.
2. There can be no left recursion.

Now consider the following modified CFG, which is LL(1), for expressions:

1. stmt → expr$
2. expr → term term_tail
3. term_tail → add_op term term_tail
4. term_tail → ε
5. term → factor factor_tail
6. factor_tail → mul_op factor factor_tail
7. factor_tail → ε
8. factor → '(' expr ')'
9. factor → ID
10. factor → NUM
11. add_op → '+'
12. add_op → '-'
13. mul_op → '*'
14. mul_op → '/'

Consider term_tail with '+' in the input:

- R3 is possible, as add_opt will eventually expand into '+'
- R4 is not possible, as only input characters it is applicable for are $ and ')'

How did we determine the possible input characters for which R4 can be applied?

- If we expand R4, we get ε, which means empty string
- So what input character can be consumed if we apply R4?
- It is the input character that can follow *term_tail* in the grammar
- We search the RHS of the rules to find *term_tail*
- We find it in two places:
  - R2, where *term_tail* appears as the last thing
    - So we cannot find, right away, what may come after it
    - But we know that whatever can come after *expr* (which is the LHS of R2) can come after *term_tail* as well
    - So we search what can come after *expr*
    - We search the RHS of the rules to find *expr*
    - We find it in two places
      - R1, where only $ can come after it
      - R8, where only ')' can come after it
  - R3, where the *term_tail* again appears as the last thing
  - However, LHS is *term_tail* as well, so no new information is learned

As a result, R4 is only applicable for are $ and ')'. $ and ')' form the PREDICT set of rule 4. Again we will define this more formally, a little later.

<u>There are two ways to construct a top-down parser</u>

1. A hand coded *recursive descent parser*
   For each non-terminal (LHS), write a procedure
   - If there are more than one possible RHS, use the PREDICT set and the current input token to decide which alternative rule to apply
   - For the rule being applied
     - consume input for terminals
     - call other procedures for non-terminals

   See the book for complete examples. At this point you should be able to see why left recursion creates a problem.
2. *Table-driven* parsers
   Create a table, which, for each non-terminal/token pair, specifies the production rule to use. This table is usually auto-generated, and can be used to drive a simple parsing procedure.

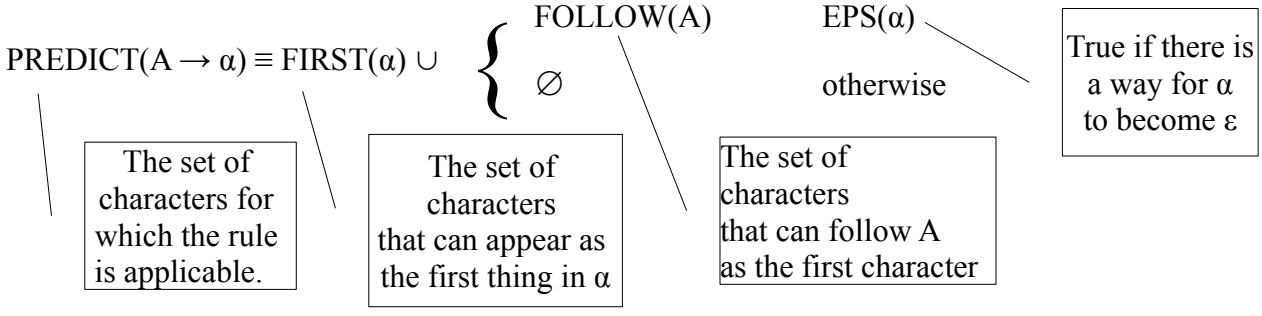
	ID	NUM	(	)	+	-	*	/	$
stmt	1	1	1						
expr	2	2	2						
term_tail				4	3	3			4
term	5	5	5						
factor_tail				7	7	7	6	6	7
factor	9	10	8						
add_op					11	12			
mul_op							13	14	

Example input: 3 * (x + y)

input	**Production**	
NUM(3)	stmt	R1
NUM(3)	expr $	R2
NUM(3)	term term_tail $	R5
NUM(3)	factor factor_tail term_tail $	R10
NUM(3)	NUM factor_tail term_tail $	Eat NUM
*	factor_tail term_tail $	R6
*	mul_op factor factor_tail term_tail $	R13
*	* factor factor_tail term_tail $	Eat *
(	factor factor_tail term_tail $	R8
(	'(' expr ')' factor_tail term_tail $	Eat (
ID(x)	expr ')' factor_tail term_tail $	R2
ID(x)	term term_tail ')' factor_tail term_tail $	R5
ID(x)	factor factor_tail term_tail ')' factor_tail term_tail $	R9
ID(x)	ID factor_tail term_tail ')' factor_tail term_tail $	Eat ID
+	factor_tail term_tail ')' factor_tail term_tail $	R7
+	term_tail ')' factor_tail term_tail $	R3
+	add_op term term_tail ')' factor_tail term_tail $	R11
+	'+' term term_tail ')' factor_tail term_tail $	Eat +
ID(y)	term term_tail ')' factor_tail term_tail $	R5
ID(y)	factor factor_tail term_tail ')' factor_tail term_tail $	R9
ID(y)	ID factor_tail term_tail ')' factor_tail term_tail $	Eat ID
)	factor_tail term_tail ')' factor_tail term_tail $	R7
)	term_tail ')' factor_tail term_tail $	R4
)	')' factor_tail term_tail $	Eat)
$	factor_tail term_tail $	R7
$	term_tail $	R4
$	$	Eat $

PREDICT Sets

How to find the set of characters for which the rule is applicable.



FIRST(α) ≡ {c: α =>* c β } EPS(α) ≡ [α =>* ε] FOLLOW(A) ≡ {c: S =>+ α A c β }

Here A is a non-terminal, α and β are sentimental forms, which contains a mixture of terminals and non-terminals. =>* means 0 or more productions and =>+ means one or more productions. There are simple algorithms to find the PREDICT set programmatically (see the book).

PREDICT(1) ≡ { '(', ID, NUM }
PREDICT(2) ≡ { '(', ID, NUM }
PREDICT(3) ≡ { '+', '-' }
PREDICT(4) ≡ { '$', ')' }
FOLLOW(term_tail) ≡ FOLLOW(expr) ≡ { $, ')' }
PREDICT(5) ≡ { '(', ID, NUM }
PREDICT(6) ≡ { '*', '/' }
PREDICT(7) ≡ { '+', '-', $, ')' }
FOLLOW(factor_tail) ≡ FOLLOW(term)

$$\equiv \text{FIRST(term\_tail)} \cup \begin{cases} \text{FOLLOW(term\_tail)} & \text{EPS(term\_tail)} \\ \varnothing & \text{otherwise} \end{cases}$$

≡ {'+', '-'} ∪FOLLOW(term_tail)
≡ {'+', '-'} ∪{$, ')'}
≡ {'+', '-', $, ')'}

Writing an LL(1) Grammar

Problems:

1. Left recursion

   A =>+ A α

   E.g.: id_list → id_list_prefix ';'
   id_list_prefix → id_list_prefix ',' ID
   id_list_prefix → ID

2. Common prefixes

   A → α β
   A → α ɣ

   E.g.: stmt → ID ":-" expr
   stmt → ID '(' arguments ')'

Techniques:

1. Eliminate left recursion
   A → A α_1 | … | A α_n
   A → β_1 | … | β_m
   becomes
   A → β_1 A' | … | β_m A'
   A' → α_1 A' | … | α_n A' | ε

   id_list → id_list_body ';'
   id_list_body → ID id_list_tail
   id_list_tail → ',' ID id_list_tail | ε

2. Eliminate common prefixes by left factoring

   $A \rightarrow \alpha \beta$
   $A \rightarrow \alpha \gamma$

   becomes

   $A \rightarrow \alpha$ tail
   tail $\rightarrow \beta \mid \gamma$

   stmt → ID stmt_tail
   stmt_tail → ":-" expr
   stmt_tail → '(' arguments ')'