CS 344 – Parsing

Note Title 2/3/2012

Announcements

- May redo 1st HW & resubmit by Monday.
- 2nd HW due in 1 week.
- Midterm – Monday before spring break

(ambiguous CNF)

Other parsing algorithms

CYK is still pretty slow, especially for large programming languages. $O(n^3)$

After it was developed, a lot of work was put into figuring out what grammars could have faster algorithms.

Two big (& useful) classes have $O(n)$ time parsers: LL & LR.

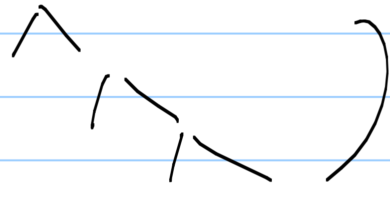
LL & LR grammars

"LL" is left-to-right, leftmost derivation

"LR" is left-to-right, rightmost derivation

- So parser will scan left to right either way.
- LL will make a leftmost derivation

(so right-leaning tree)



LL versus LR

- LL are a bit simpler so we'll start with them
  - Note: LR is a larger class (so more grammars are LR than are LL)
- Both are used in production compilers today

Example: LL parsing

leftmost derivation

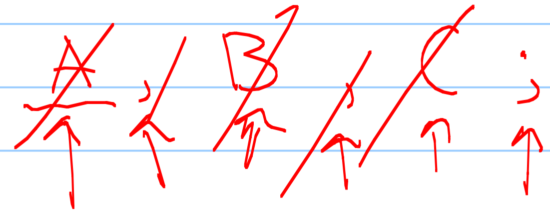
A B

LL(1)

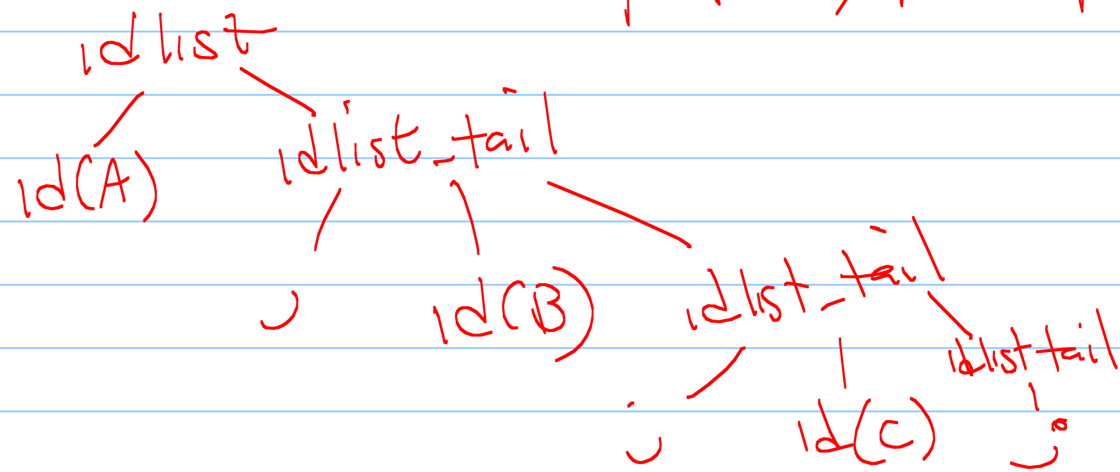
idlist → id idlist_tail

idlist_tail → , id idlist_tail

idlist_tail → ;



Parse tree for A, B, C;



LL(1), LL(2)

LL(k) + LR(k)

When LL or LR is written with (1), (2), etc, it refers to how much look-ahead is allowed.

LL(1) means we can only look 1 token ahead when making our decision of which rule to match

Most commercial ones are LR(1), but exceptions exist, such as ANTLR.

A non LL(1) example: Left recursion

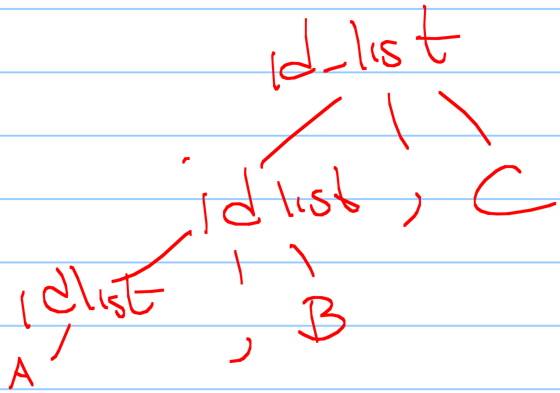
id_list → id
→ id_list , id

LR

Imagine: Scanning left to right, & encounter an id token.

Which parse tree do we build?

A, B, C

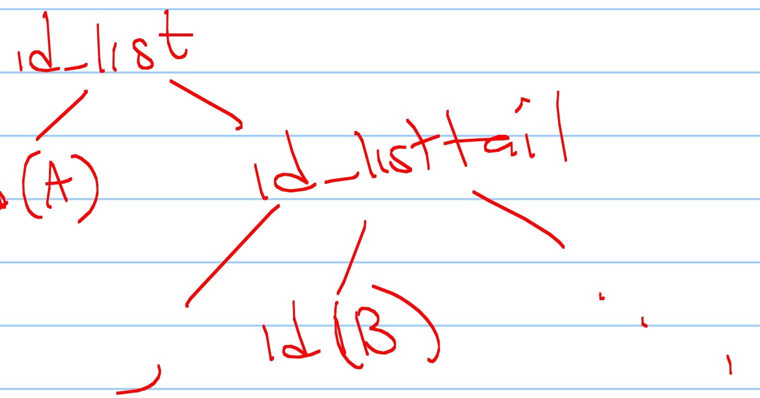


Making the grammar LL(1):

* id_list $\longrightarrow$ id id_list_tail

id_list_tail $\longrightarrow$, id id_list_tail | ε

A, B, C



→ A B C

Another non-LL(0) example: common prefixes

stmt → id := expr
stmt → id (arguement_list)

So when next token is an id, don't know which rule to use.

Fix?

LL(0)

stmt → id stmt_tail

stmt_tail → := expr
→ (arguement_list)

A := B + C

stmt
id(A   stmt_tail

Some grammars are non-LL:

- Eliminating left recursion and common prefixes is a very mechanical procedure which can be applied to any grammar.
- However, might not work! There are examples of inherently non-LL grammars.
- In these cases generally add some heuristic to deal with odd cases

Example: non-LL language

$$\text{stmt} \rightarrow \text{if condition then\_clause else\_clause}$$

$$\text{then\_clause} \rightarrow \text{then stmt}$$

$$\text{else\_clause} \rightarrow \text{else stmt} \mid \varepsilon$$

What syntax?

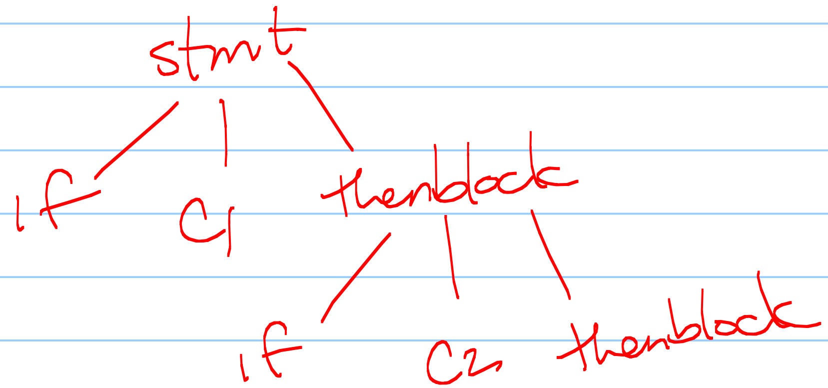
if

then

else

Ex: if C_1 then if C_2 then S_1 else S_2

Parse tree:



if C_1
then if C_2
then S_1
else

no possible LL grammar for if statements

Back to LL-parsing

We have seen mostly top-down parsing.

Start with S_0, the start token, + try to construct the tree based on the next input.

Bottom-up parsing starts at the leaves (here, the tokens), + tries to build the tree upward.

Continues scanning + shifting tokens onto a forest, then builds up when it finds a valid production.

Bottom-up parsing

idlist → id idlist_tail

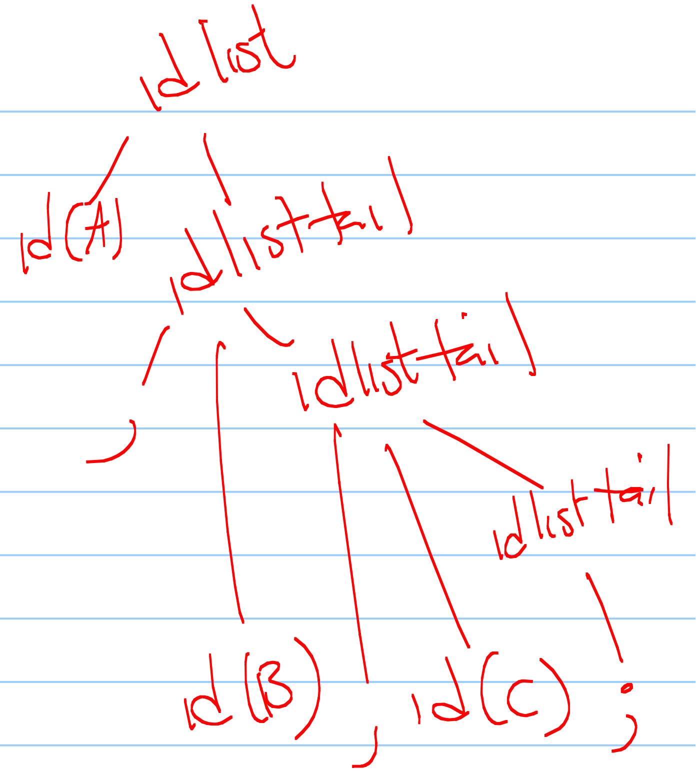
idlist_tail → , id idlist_tail

idlist_tail → ;

Ex: A, B, C;

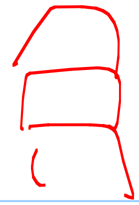
Bottom up parsing:

(is this left-most or rightmost?)



leftmost derivation

Shift-reduce:



- Bottom up parsers are also called shift-reduce:
  - shift token onto ~~forest~~ stack,
  - when a rule is recognized, reduce to left-hand side
- Problem with last example: must shift all tokens onto the forest before reducing. What could happen in a large program?
  overflow your stack
- Sometimes unavoidable. However, sometimes other options...

Bottom-up parsings: another example

id_list → id_list_prefix ;

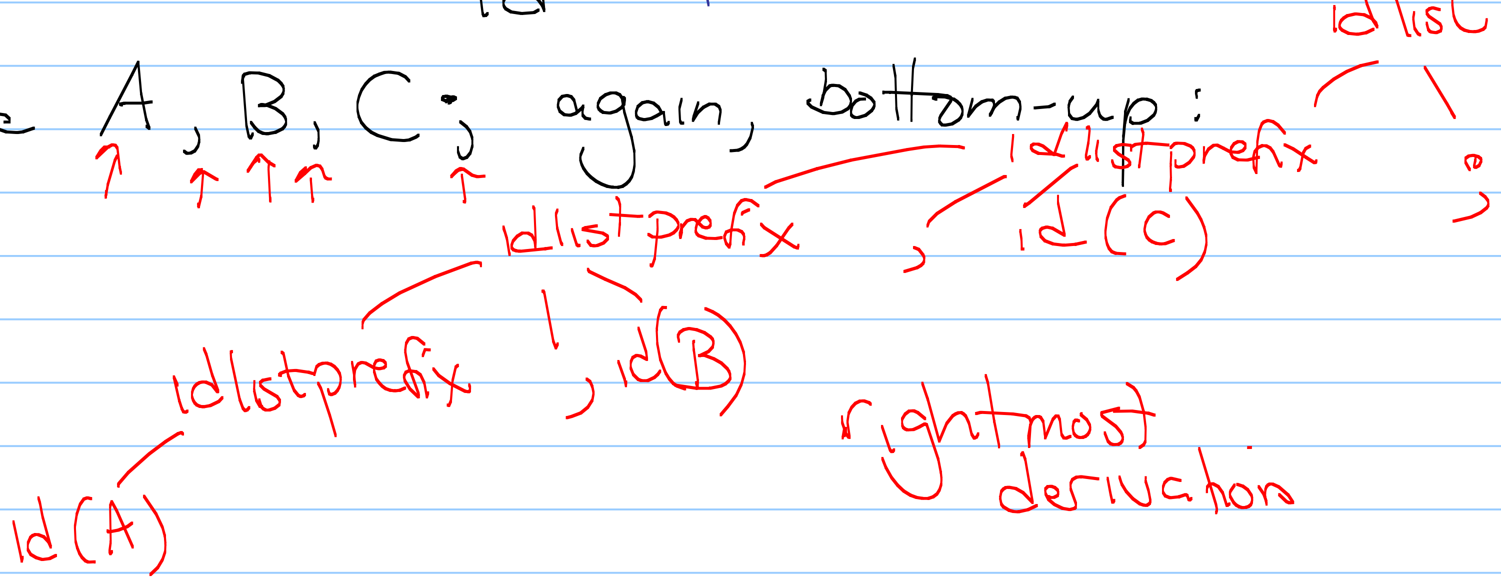
left recursion

id_list_prefix → id_list_prefix , id

→ id

id

Parse A , B , C ; again, bottom-up:



rightmost derivation

Bottom-up parsing: some notes

- The previous example cannot be parsed top-down. (left recursion)
- Note that it also is not an LL grammar, although the language is LL!
- There is a distinction between a language & a grammar. Remember, any language can be generated by an infinite number of grammars.