CS 344 - LL and LR grammars

Announcements

- HW 3 - due Sunday by midnight

- Next HW up this weekend
Other parsing algorithms

CYK is still pretty slow, especially for large programs.

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

Two big (and useful) classes have \(O(n)\) time parsers: \(LL\) and \(LR\).
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

\[ \text{idlist} \rightarrow \text{id} \ \text{idlist}_\text{tail} \]

\[ \text{idlist}_\text{tail} \rightarrow \text{id} \ \text{idlist}_\text{tail} \]

\[ \text{idlist}_\text{tail} \rightarrow . \]

Parse tree for "A, B, C, j"

```
                        idlist
                     /     |
                    id(A)  idlist_tail
                          /    |
                         id(B) idlist_tail
```
id_list
  id_list
    id(A)  id_list_tail
  id_list
    id(A)  id_list_tail
      ,  id(B)  id_list_tail
    ,  id(B)  id_list_tail
      ,  id(C)  id_list_tail
    ,  id(C)  id_list_tail
  ;
\(\text{LL}(k) \oplus \text{LR}(k)\)

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

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

Most commercial ones are \(\text{LR}(1)\), but exceptions exist (such as ANTLR).
A non LL(1) example: Left recursion

\[ \text{id-list} \rightarrow \text{id} \rightarrow \text{id-list, id} \]

Left recursion is bad in LL

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

Which parse tree do we build?

\[ \text{id(A), id(B), id(C)} \]

\[ \text{(this is LL(2))} \]
Making the grammar LL(1):

\[ id\text{-list} \rightarrow id \ id\text{-list}\text{-tail} \]

\[ id\text{-list}\text{-tail} \rightarrow id \ id\text{-list}\text{-tail} \]

\[ \rightarrow \epsilon \]

\[ \text{\(L'(A)\)} \]

\[ id\text{-list}\text{-tail} \]
Another non-LL(0) example: common prefixes

\[
\begin{align*}
\text{stmt} & \rightarrow \text{id} := \text{expr} \\
\text{stmt} & \rightarrow \text{id} (\text{argument-list})
\end{align*}
\]

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

Fix?

\[
\begin{align*}
\text{stmt} & \rightarrow \text{id} \text{ stmt-tail} \\
\text{stmt-tail} & \rightarrow : = \text{expr} \\
& \rightarrow (\quad)
\end{align*}
\]
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 (or use CYK)
Example: non-LL language: optional else

\[
\text{stmt} \rightarrow \text{condition then-clause else-clause}
\]

\[
\text{then-clause} \rightarrow \text{then stmt}
\]

\[
\text{else-clause} \rightarrow \text{else stmt}
\]

\[
\text{What syntax?} \quad \text{if-else statement} \quad \text{(PASCAL)}
\]
Ex: if $C_1$ then if $C_2$ then $S_1$ else $S_2$

Parse tree:
Back to LL-parsing

We have seen mostly top-down parsing.

Start with so, the start token, and try to construct the tree based on the next input.

Also called predictive parsing - matches the rule based on current token/state plus the next input.
LR grammars

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

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

Never predicts - when it recognizes right hand side of a rule simplifies to left hand side.
Bottom-up parsing (LR parsing)

idlist → id  idlist_tail

idlist_tail → id  idlist_tail
idlist_tail → \$

Ex.

Input: id(A), id(B), id(C)

idlist_tail
Shift-reduce:

- Bottom up parsers are also called shift-reduce:
  - Shift token onto stack (in a forest)
  - 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?

  - Sometimes unavoidable. However, sometimes other options...
Bottom-up parsing: another example

\[ id\_list \rightarrow id\_list\_prefix \]

\[ id\_list\_prefix \rightarrow id\_list\_prefix \circ id \]

Parse A, B, C; again, bottom-up:
id(A)

id_list_prefix
  |
  id(A)

id_list_prefix, id(B)
  |
  id(A)

id_list_prefix
  |
  id(A)

id_list_prefix, id(B)
  |
  id(A)

id_list_prefix
  |
  id(A)

id_list_prefix, id(B)
  |
  id(A)

id_list_prefix, id(C)
  |
  id(A)

id_list_prefix, id(B)
  |
  id(A)

id_list_prefix
  |
  id(A)

id_list_prefix, id(B)
  |
  id(A)

id_list_prefix
  |
  id(A)

id_list_prefix, id(B)
  |
  id(A)

id_list_prefix
  |
  id(A)

never put all tokens on stack!
much better in terms of space.
Bottom-up parsing: some notes

- The previous example cannot be parsed top-down. Why? *left recursion!*

- Note that it also is not an LL grammar, although the language is LL.

- There is a distinction between a language and a grammar. Remember, any language can be generated by an infinite number of grammars.
LR grammars: An old example

\[
\begin{align*}
expr & \rightarrow \text{term} \mid \text{expr add-op term} \\
term & \rightarrow \text{factor} \mid \text{term mul-op factor} \\
factor & \rightarrow \text{id} \mid \text{number} \mid - \text{factor} \mid (\text{expr}) \\
\text{add-op} & \rightarrow + \mid - \\
\text{mul-op} & \rightarrow * \mid \div
\end{align*}
\]

Diagram:

```
  expr
    \mid term
      \mid expr add-op term
  \mid term
    \mid factor
      \mid term mul-op factor
  \mid factor
    \mid id
    \mid number
    \mid - factor
    \mid (expr)
  add-op
    \mid +
    \mid -
  mul-op
    \mid *
    \mid /
  5
    \mid +
    \mid ( expr 
      \mid term
        \mid term
          \mid factor
            \mid add-op
              \mid factor
                \mid mul-op
                  \mid factor
                    \mid num(2)
```
This grammar is not LL!

- If we get an id as input when expecting an expr, no way to choose between the 2 possible productions.

- It suffers from the common prefix issue we saw before.

(We can fix this → )
Another LL example:

```
expr  →  term  term_tail

term_tail  →  add_op  term  term_tail
           →  ε

term  →  factor  factor_tail

factor_tail  →  mult_op  factor  factor_tail
             →  ε

factor  →  ( expr ) | id | number

add_op  →  + | -

mult_op  →  * | /
```
expr
  └── term
    └── term_tail
      └── factor
          └── add_op
              └── num(5) + num(3)

ll(1)
  └── (not ll(0))

factor
  └── term
    └── term_tail
      └── factor
          └── factor_tail
            └── mult_op
                └── factor
                    └── factor_tail
                        └── num(2)
                            └── 3
Now can add this as part of a simple calculator language:

```
program → stmt_list $\$
stmtlist → Stmt stmt_list
      → ε
stmt → id := expr
      → read id
      → write expr
```

end of file
Program: What does it do?

read A
read B

\[ \text{sum} := \text{expr} \]

write sum

write \( \text{sum} / 2 \)
LL(1)