CS344 – LR parsing

Note Title 2/8/2012

Announcements

- Office hours today – 3-4
- For HW3, not just wanting pattern matching description

```
gcc file.lex -lfl
```

(man gcc)

- HW3 due Monday by 11:59pm

LL parsing

Left-to-right, leftmost derivation.

Usually top-down.

Things that prevent it:

- left recursion
- common prefixes

But not all languages are LL!

Example: LL parsing

leftmost derivation

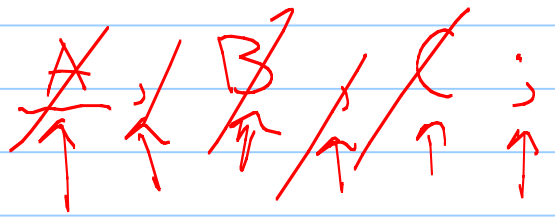
A B

LL(0)

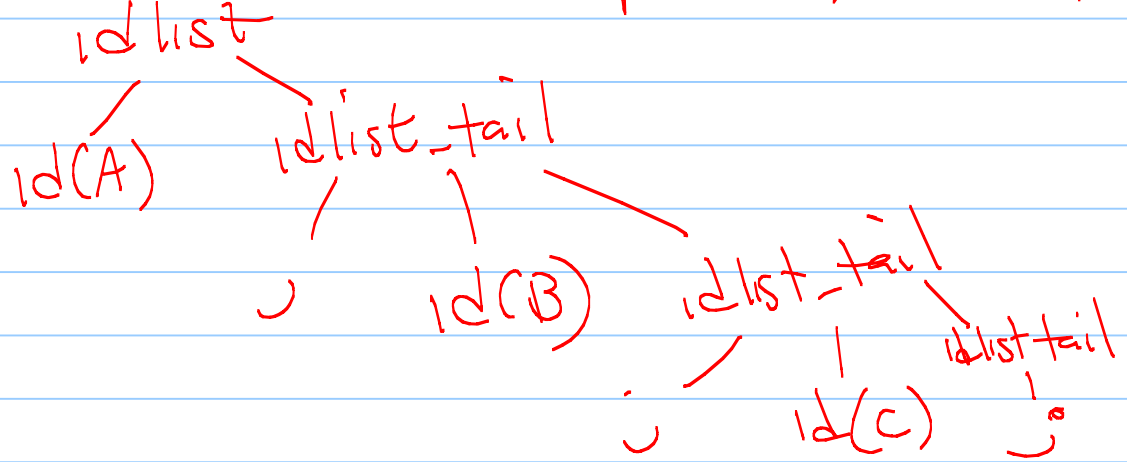
idlist → id idlist_tail

idlist_tail → , id idlist_tail

idlist_tail → ;



Parse tree for A, B, C;



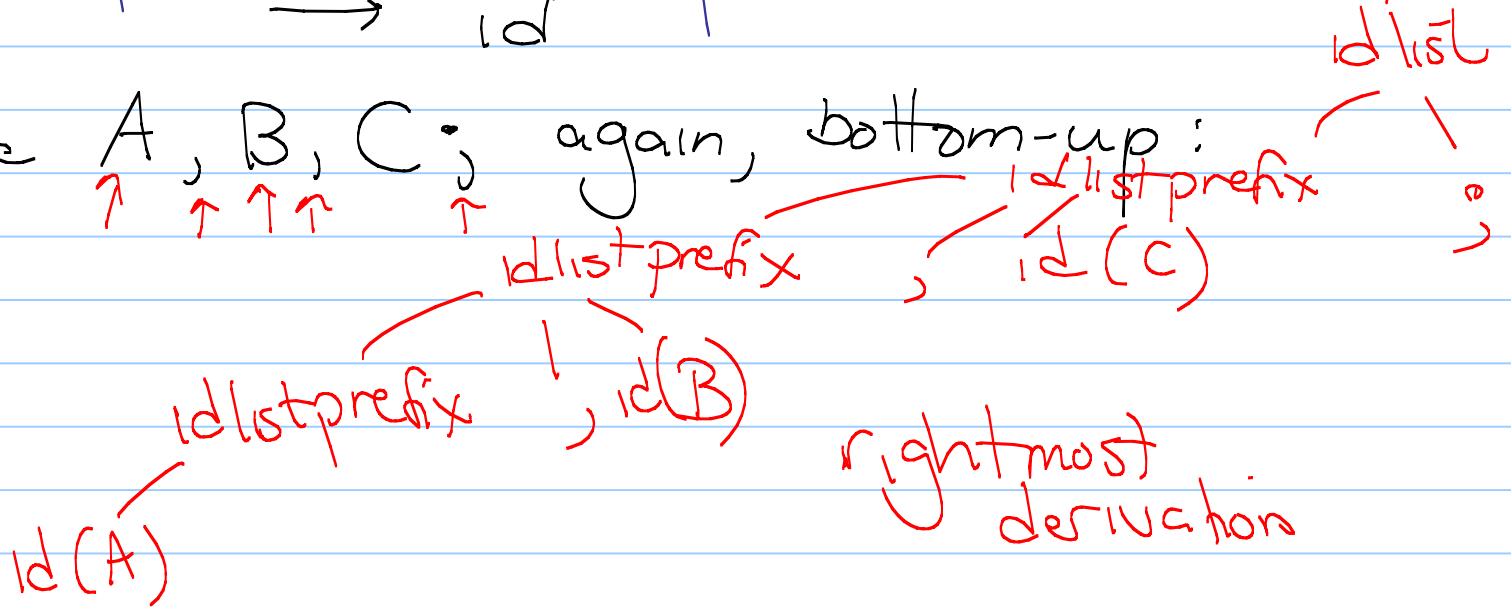
Bottom-up parsings: another example

id_list → id_list_prefix ;

id_list_prefix → id_list_prefix , id ← left recursion

→ 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.

LR grammars : An old example

expr → term | expr add_op term

term → factor | term multop factor

factor → id | number | - factor | (expr)

add_op → + | -

multop → * | /

(What does LR stand for?)

↑ (right most derivation (so left tree))

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. (also left recursion)

(See sec. 2.3.1 for an example LL grammar that is equivalent.)

Building an LL parsers

2 options:

1. Recursive descent parser: code whose subroutines correspond to non-terminals

   (like case statement version of scanner)

2. LL parse table which is used by a driver program (like bison).

   (like flex)

Comparing the 2 options

- Both recursive descent and parse tables are used in practice
- Generally, recursive descent code is hand written for smaller languages (or when tool is unavailable)
- Exceptions exist, however.
  Example: gcc

Neither LL or LR:

stmt → if condition then_clause else_clause
| other_stmt

then_clause → then stmt
else_clause → else stmt
| ε

This one is inherently ambiguous.

Solution? add rule to force uniformity

Another way: avoid ambiguity

Use an explicit end:

stmt → IF condition then_clause else_clause END
| other_stmt

then_clause → then stmt_list

else_clause → else stmt_list | ε

(Python – indent)

Bottom-up parsing

- Uses a stack to push tokens onto.
  Why?

Constant time!
Good use of space

returns tokens in order I want

left to right

right most

LR version of calculator

```
program    ⟶ stmt_list $$
stmt_list  ⟶ stmt_list stmt | stmt
stmt       ⟶ id := expr | read ID
           | write expr
expr       ⟶ term | expr add_op term
term       ⟶ factor | term mult_op factor
factor     ⟶ (expr) | id | number
add_op     ⟶ + | -
mult_op    ⟶ * | /
```

Example to parse:

```
read A
read B
sum := A + B
write sum
write sum / 2
```

Beginning: know we start with program.

So candidates are:

program → . stmt_list $$

→ . stmt

what else! (take closure)

Big picture: LR parsing

- Keep track of states it has traversed by pushing them into the parse stack along with symbols.

- When top of stack (input + states) says we need to reduce using a rule $A \rightarrow \alpha$:
  - pop len(α) items off stack
  - push A on stack

LR: Main issue

- Need to deal with conflicts.

  Might have 2 possible rules we match, one of which calls for a shift (or push onto stack) & the other a reduce ($A \to \alpha$).

- An LR(0) parser works only when no such conflicts exist.

- Any language which can be parsed bottom up has an LR(0) grammar — but not practical.

Other LR strategies

- SLR (or simple LR) parsers look at future inputs.
  Will only reduce $A \to \alpha$ if the next tokens could follow α in the grammar.

- LALR (look ahead LR) parsers improve on SLR by using local, state-specific look ahead.

- LALR is most common in practice.

Syntax Errors

When parsing a program, will often detect syntax errors where tokens do not form valid states.

What should we do?

Find closest rule that does match.

"Recover" → continue parsing.

Generating good error messages

Most compilers do not just halt;
this would mean code past the
first error is ignored.
However, it is beneficial to detect as
many errors as possible.

So how to continue after an error?

Approaches

1. Panic mode: define a small set of "safe symbols".
   Ex: In C++, use semicolon
   In Python: /n

When error occurs, compiler deletes back to the last safe symbol

(Ever notice that errors often point to the line before or after the actual error?)

(2) Phase-level recovery:
refinement of panic mode with different safe symbols in different states.

Ex: expression →)
Statement → ;

(3) Context specific look-ahead:
improves on (2) by checking various contexts in which the production might appear in the parse tree.

Beyond Parsing (Ch. 4)

Need rules to connect the productions to actual concepts.

Ex:

$$
\begin{aligned}
E &\rightarrow E + T \\
E &\rightarrow E - T \\
E &\rightarrow T \\
T &\rightarrow T * F \\
T &\rightarrow T / F \\
T &\rightarrow F \\
F &\rightarrow -F \\
F &\rightarrow (E) \\
F &\rightarrow \text{const}
\end{aligned}
$$

(LR(0) or LL(0) or neither?)

not LL — left recursion!

This grammar generates all well-formed constant expressions.

However, says nothing about their meaning.

Need to tie these to basic rules for the processor.

These can be specified in machine language (or assembly), or some other existing code on the machine.

Example: Associate a value with each nonterminal (& assume const is given by scanner).

$E \to E + T$: $E_1 \to E_2 + T$

$E_1.val := sum(E_2.val, T.val)$

$E \to T$: $E.val := T.val$

$F \to const$: $F.val := const.val$