Bottom-Up Parsing

CS 132: Compiler Design

January 31, 2011

Review: LL(k)

A grammar is said to be $LL(1)$ if one can always predict the next grammar production by peeking at one character of input.

A grammar is said to be $LL(k)$ if we can make the prediction based on the next k tokens of the input.

A language is said to be $LL(k)$ if it has an $LL(k)$ grammar (which generates the right strings, but not necessarily the right parse trees)

Massaging Grammars

```
S -> E $
E -> n
   | E - n
```

$\Downarrow$

Eliminate "left recursion"

$\Downarrow$

```
S -> E $
E -> n
   | n - E
```

- ✓ In this case, a non-LL grammar becomes $\mathrm{LL}(2)$.
- ✓ Note: for recursive descent, $\mathrm{LL}(2)$ usually isn't that bad!
- ✓ But, this transformation can change the parse tree.

Massaging Grammars

```
S -> E $
E -> n
   | n - E
```

$\Downarrow$

"Left-factor"

$\Downarrow$

```
S  -> E $
E  -> n E'
E' -> ϵ
    | - E'
```

- ✓ In this case, an $\mathrm{LL}(2)$ grammar becomes $\mathrm{LL}(1)$.
- ✓ Possibly useful if predictions go via a big lookup table.
- ✓ Parse tree is even less pretty.

ALTERNATIVE: EBNF

```
S -> E $
E -> E
   | E - n
```

⇓

```
S  -> E $
E  -> n { - n }
```

or

```
S  -> E $
E  -> n ( - n )*
```

- ✓ *Comparatively easy to handle in recursive descent (Take one* **n***, and then loop to gather as many* **- n***'s as you can.)*
- ✓ *Result is a "list" of subtractions; can turn these into a proper tree.*

A (Contrived) Grammar

Given the grammar:

```
S -> a C E | b D
C -> b c | C d
D -> d | c D e
E -> e
```

Draw the parse trees for the inputs `a b c d e` and `b c d e`.

LR Parsing

```
S -> a C E | b D
C -> b c | C d
D -> d | c D e
E -> e
```

The essence of LR parsing is to build parse trees in a bottom-up fashion:

- ✓ Start with the leaves (`a b c d e` and `b c d e` respectively)
- ✓ Repeatedly combine subtrees into bigger trees using the productions.

Sadly, we can't naïvely "just run productions backwards"

Shift-Reduce Parsing

Stack	Input	Action (Shift/Reduce)

Some Terminology

Assume we have a parse tree.

A *handle* of a parse tree is the leftmost, complete and nontrivial subtree.

Some Terminology

Assume we have a parse tree.

A *handle* of a parse tree is the leftmost, complete and nontrivial subtree.

We *prune* the handle of a parse tree by removing the children/leaves.

From Trees to Production Sequences

```
S -> a C E | b D
C -> b c | C d
D -> d | c D e
E -> e
```

Start with a parse tree (say, for `a b c d e`).

From Trees to Production Sequences

```
S -> a C E | b D
C -> b c | C d
D -> d | c D e
E -> e
```

Start with a parse tree (say, for `a b c d e`).

Repeatedly prune the handle.

From Trees to Production Sequences

```
S -> a C E | b D
C -> b c | C d
D -> d | c D e
E -> e
```

Start with a parse tree (say, for `a b c d e`).

Repeatedly prune the handle.

How can we get a *rightmost* production sequence out of this?

From Trees to Production Sequences

```
S -> a C E | b D
C -> b c | C d
D -> d | c D e
E -> e
```

Start with a parse tree (say, for `a b c d e`).

Repeatedly prune the handle.

How can we get a *rightmost* production sequence out of this?

If only we could find handles in the original string, without precomputing the parse tree…

Viable Prefixes

Assume that $\vec{\alpha}$ and $\vec{\beta}$ are sequences of terminals and non-terminals.

Viable Prefixes

Assume that $\vec{\alpha}$ and $\vec{\beta}$ are sequences of terminals and non-terminals.

We say that $\vec{\alpha}\vec{\beta}$ is a *viable prefix* if there is a sequence $\ldots$ of terminals such that

$$\vec{\alpha}\vec{\beta}\ldots$$

has a parse tree with $\vec{\beta}$ as the handle.

Viable Prefixes

Assume that $\vec{\alpha}$ and $\vec{\beta}$ are sequences of terminals and non-terminals.

We say that $\vec{\alpha}\vec{\beta}$ is a *viable prefix* if there is a sequence $\ldots$ of terminals such that

$$\vec{\alpha}\vec{\beta}\ldots$$

has a parse tree with $\vec{\beta}$ as the handle.

- ✓ What are some viable prefixes for trees whose root is `E`? `C`? `D`? `S`?

```
S -> a C E | b D
C -> b c | C d
D -> d | c D e
E -> e
```

Why Should You Care?

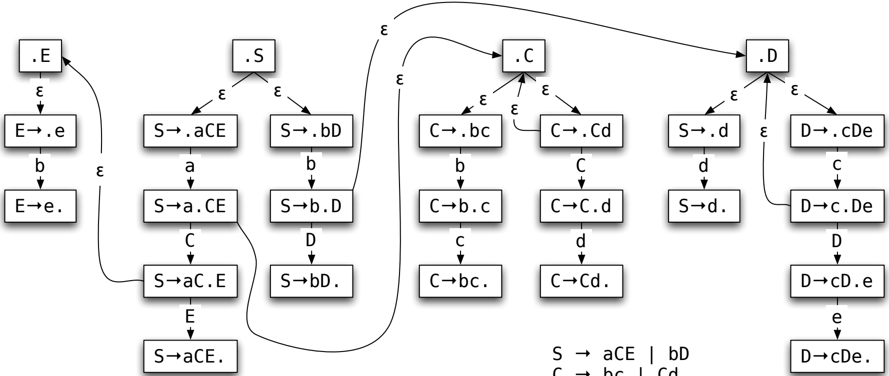
Why Should You Care?

- ✓ The viable prefixes of *every* context-free grammar are *regular*!
- ✓ That is, a finite state machine can recognize viable prefixes. It can *even* tell you what the handle is.

Why Should You Care?

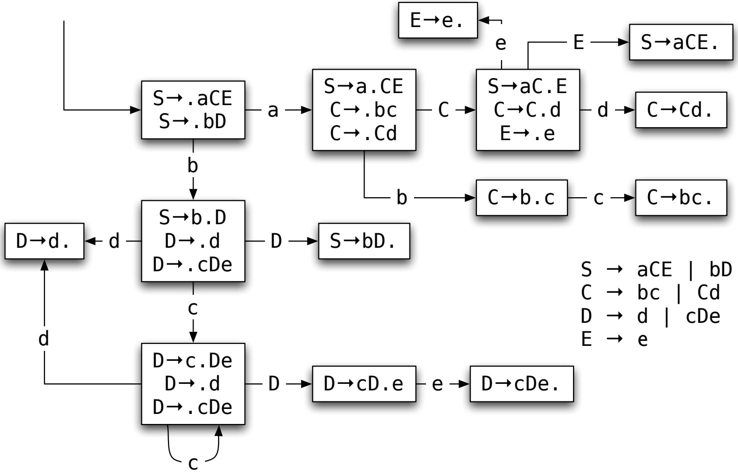
- ✓ The viable prefixes of *every* context-free grammar are *regular*!
- ✓ That is, a finite state machine can recognize viable prefixes.
  It can even tell you what the handle is.
- ✓ Every parseable string (by definition) has a viable prefix.
- ✓ Some grammars have a property that no viable prefix extends another.
  These are called $\mathrm{LR}(0)$ grammars.
- ✓ If so, every derivable string starts with a *unique* viable prefix.
  This gives us a parsing algorithm for $\mathrm{LR}(0)$ gramamrs.

Parsing Automaton: NFA



S → aCE | bD
C → bc | Cd
D → d | cDe
E → e

Building a Parsing Automaton



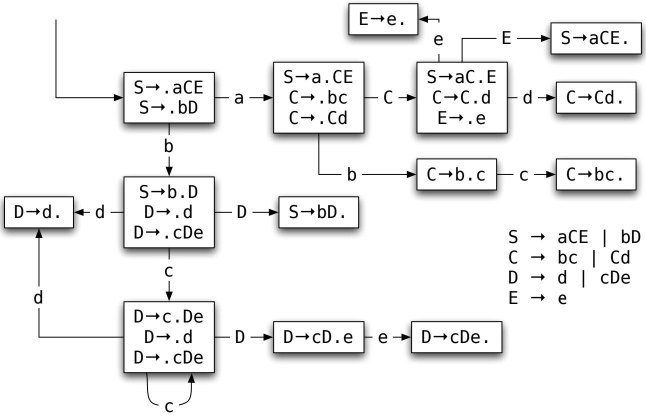
Using the Parsing Automaton

Parse `b c d e` and `a b c d e` using the parsing automaton.

Stack	**Input**	**Action** (Shift/Reduce)

LR(0)

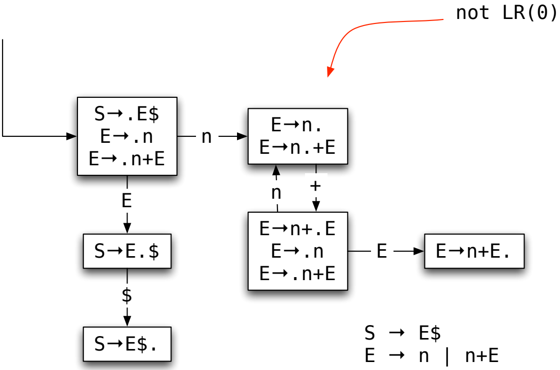
A grammar is LR(0) if the parsing automaton (as described) gives unambiguous instructions.



BTW, why *doesn't anyone ever consider* LL(0) grammars?

A Non-LR(0) Example

Sadly, few useful grammars are LR(0). For example,



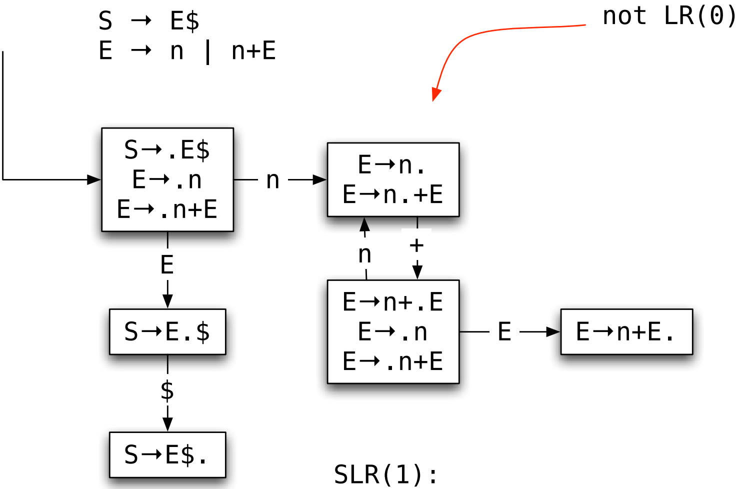

```
S → E$
E → n | n+E
```

- ✓ Show a parse tree for an input starting with **n**, where **n** is the handle.
- ✓ Show a parse tree for an input starting with **n**, where **n** isn't the handle.

Thus, in practice we need to peek past the handle.

SLR(1)

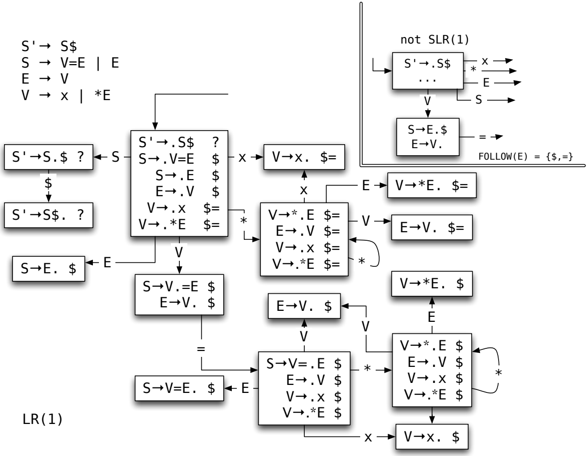
Reduce using a rule $B \to \vec{\beta}$ only if the next input token is in `FOLLOW(B)`.



SLR(1):
FOLLOW(E) = {$}

LR(k)

Add lookahead tokens to the LR(0) items in the NFA.



Also, LALR(k): Merge similar states ("lossy compression")

Dangling Else

```
S -> if E then S else S
   | if E then S
   | ...
```

How to parse

```
if a then if b then s1 else s2
```

Dangling Else

```
S -> if E then S else S
   | if E then S
   | ...
```

How to parse

```
if a then if b then s1 else s2
```

No ambiguous grammar is LR(k), so you'd get a shift-reduce conflict

```
S -> if E then S . else S    <any>
S -> if E then S .           else
```

Solutions

Hack the grammar

```
S -> M
S -> U

M -> if E then M else M
M -> ...

U -> if E then S
U -> if E then M else U
```

Solutions

Hack the grammar

```
S -> M
S -> U

M -> if E then M else M
M -> ...

U -> if E then S
U -> if E then M else U
```

Hack the language

```
S -> if E then S else S fi
   | if E then S fi
   | ...
```

Solutions

Hack the grammar

```
S -> M
S -> U

M -> if E then M else M
M -> ...

U -> if E then S
U -> if E then M else U
```

Hack the language

```
S -> if E then S else S fi
   | if E then S fi
   | ...
```

Hack the parser:
Whenever you reach this "conflicting" state, just go ahead and shift.

Ambiguous Expression Grammars

```
E -> E + E | E - E
   | E * E | E / E
   | ( E ) | n
```

Ambiguous Expression Grammars

```
E -> E + E | E - E
   | E * E | E / E
   | ( E ) | n
```

All sorts of shift-reduce conflicts:

```
E -> E . + E     <any>
E -> E * E .     +
```

```
E -> E . * E     <any>
E -> E + E .     *
```

```
E -> E . + E     <any>
E -> E + E .     +
```

Solutions

Hack the grammar

```
E -> E + T | E - T | T     E -> T (+ T | - T)*
T -> T * F | T / F | F     T -> F (* F | / F)*
F -> ( E ) | n             F -> ( E ) | n
```

Solutions

Hack the grammar

```
E -> E + T | E - T | T     E -> T (+ T | - T)*
T -> T * F | T / F | F     T -> F (* F | / F)*
F -> ( E ) | n             F -> ( E ) | n
```

Hack the language

```
E -> + E E | * E E | ...
```

Solutions

Hack the grammar

```
E -> E + T | E - T | T     E -> T (+ T | - T)*
T -> T * F | T / F | F     T -> F (* F | / F)*
F -> ( E ) | n             F -> ( E ) | n
```

Hack the language

```
E -> + E E | * E E | ...
```

Hack the parser to shift or reduce as desired.

Solutions

Hack the grammar

```
E -> E + T | E - T | T     E -> T (+ T | - T)*
T -> T * F | T / F | F     T -> F (* F | / F)*
F -> ( E ) | n             F -> ( E ) | n
```

Hack the language

```
E -> + E E | * E E | ...
```

Hack the parser to shift or reduce as desired. YACC support:

```
%left PLUS MINUS
%left STAR SLASH
```

Solutions

Hack the grammar

```
E -> E + T | E - T | T     E -> T (+ T | - T)*
T -> T * F | T / F | F     T -> F (* F | / F)*
F -> ( E ) | n             F -> ( E ) | n
```

Hack the language

```
E -> + E E | * E E | ...
```

Hack the parser to shift or reduce as desired. YACC support:

```
%left PLUS MINUS
%left STAR SLASH
```

✓ Terminals listed in increasing precedence

Solutions

Hack the grammar

```
E -> E + T | E - T | T      E -> T (+ T | - T)*
T -> T * F | T / F | F      T -> F (* F | / F)*
F -> ( E ) | n              F -> ( E ) | n
```

Hack the language

```
E -> + E E | * E E | ...
```

Hack the parser to shift or reduce as desired. YACC support:

```
%left PLUS MINUS
%left STAR SLASH
```

- ✓ Terminals listed in increasing precedence
- ✓ Precedence of a rule is that of its last terminal

Solutions

Hack the grammar

```
E -> E + T | E - T | T      E -> T (+ T | - T)*
T -> T * F | T / F | F      T -> F (* F | / F)*
F -> ( E ) | n              F -> ( E ) | n
```

Hack the language

```
E -> + E E | * E E | ...
```

Hack the parser to shift or reduce as desired. YACC support:

```
%left PLUS MINUS
%left STAR SLASH
```

- ✓ Terminals listed in increasing precedence
- ✓ Precedence of a rule is that of its last terminal
- ✓ Reduce if rule precedence greater than next token, or if same and rule precedence is left-associative. Otherwise shift.

Unary minus

We want these rules to be in increasing precedence:

```
exp: exp MINUS exp
   | exp TIMES exp
   | MINUS exp
```

But YACC would give the same precedence to the first and third rules.

Solution: Imaginary Tokens + Override Rule Precedence

```
%left PLUS MINUS
%left STAR SLASH
%left UNARY_MINUS

exp: exp MINUS exp
   | exp TIMES exp
   | MINUS exp      %prec UNARY_MINUS
```

(The lexer never produces a `UNARY_MINUS` token!)