RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers
Syntax Analysis
Top-Down Parsing Review
Introduction to Bottom-up Parsing

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

Announcements

- **Midterm on Wednesday, March 8**
  - closed book, closed notes
  - covers everything up to (and including) top-down parsing
  - no recitation on Tuesday, March 8
  - Q&A session second half of this Thursday 3/3 lecture – come prepared!

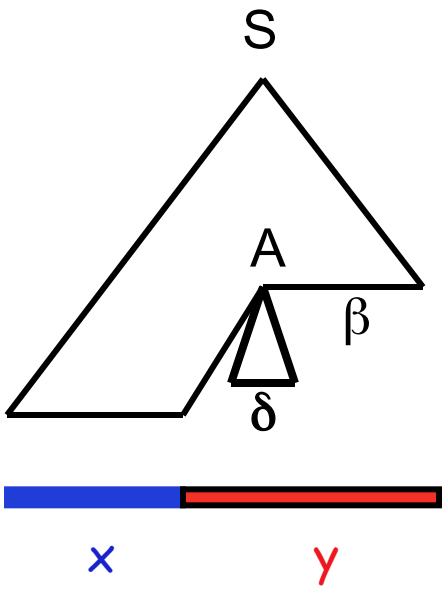
- Homework set 5 released,
  - → last homework before midterm (only one problem)
  - → due this Friday 11:59pm EST, no late submission will be accepted.

Review: Top-down parsers

LL(1), recursive descent

- 1 input symbol lookahead
- construct leftmost deriviation (forwards)
- input: read left-to-right

$$S \Rightarrow^*_{lm} x\, A\, \beta \Rightarrow_{lm} x\, \delta\, \beta \Rightarrow^*_{lm} x\, y$$



Review: Predictive Parsing

Basic idea

Given A → α | β, and some lookahead symbols, the parser should be able to choose between α & β

Review: The FIRST Set

$a \in \text{FIRST}(\alpha)$ *iff* $\alpha \Rightarrow^* a\,\gamma$, for some γ

FIRST sets

For some *rhs* $\alpha \in G$, define FIRST(α) as the set of tokens that appear as the first (terminal) symbol in some string that derives from α

That is, $a \in \text{FIRST}(\alpha)$ *iff* $\alpha \Rightarrow^* a\,\gamma$, for some γ

Review: The FOLLOW Set

For a non-terminal A, define FOLLOW(A) as

> *FOLLOW(A) := the set of terminals that can appear immediately to the right of A in some sentential form.*

Thus, a non-terminal's FOLLOW set specifies the tokens that can legally appear after it; a terminal has no FOLLOW set

Review: The FIRST+ set

The FIRST+ set only corresponds to a certain rule

Define FIRST+(δ) for rule $A \rightarrow \delta$ as

- (FIRST(δ) - { ε }) ∪ FOLLOW(A), if ε ∈ FIRST(δ)
- FIRST(δ), otherwise

Review: First, Follow, and First+ set

For the following grammar:

S ::= a S b | ε

First(aSb) = { a }
First(ε) = { ε }

Follow (S) = { eof, b }

First$^+$(S ::= aSb) = { a }
First$^+$(S ::= ε) = (First (ε) - { ε }) ∪ Follow (S) = { eof, b }

Review: Predictive Parsing

The LL(1) Property

A grammar is *LL(1)* iff $A \rightarrow \alpha$ and $A \rightarrow \beta$ implies

$$\text{FIRST}^{+}(\alpha) \cap \text{FIRST}^{+}(\beta) = \varnothing$$

This would allow the parser to make a correct choice with a lookahead of exactly one symbol !

Review: LL(1) Parser Example

Is the following grammar LL(1)?

S ::= a S b | ε

First(aSb) = { a }
First(ε) = { ε }

First+(S ::= aSb) = { a }
First+(S ::= ε) = (First (ε) - { ε }) ∪ Follow (S) = { eof, b }

LL(1)?

Review: LL(1) Parser Example

Is the following grammar LL(1)?

S ::= a S b | ε

First(aSb) = { a }
First(ε) = { ε }

First$^+$(aSb) = { a }
First$^+$(ε) = (First (ε) - { ε }) ∪ Follow (S) = { eof, b }

LL(1)? YES, since { a } ∩ { eof, b } = ∅

Predictive Parsing

Given a grammar that has the *LL(1)* property

- Problem: NT A needs to be replaced in next derivation step
- Assume $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$, with

  $FIRST^+(\beta_1) \cap FIRST^+(\beta_2) = \varnothing$, $FIRST^+(\beta_1) \cap FIRST^+(\beta_3) = \varnothing$, and

  $FIRST^+(\beta_2) \cap FIRST^+(\beta_3) = \varnothing$ (pair-wise disjoint sets)

```
/* find rule for A */
if (current token ∈ FIRST+ (β1))
    select A → β1
else if (current token ∈ FIRST+(β2))
    select A → β2
else if (current token ∈ FIRST+(β3))
    select A → β3
else
    report an error and return false
```

Grammars with the *LL(1)* property are called *predictive grammars* because the parser can "predict" the correct expansion at each point in the parse.

Parsers that capitalize on the *LL(1)* property are called *predictive parsers*.

One kind of predictive parser is the *recursive descent* parser. *The other is a table-driven parser table-driven parser.*

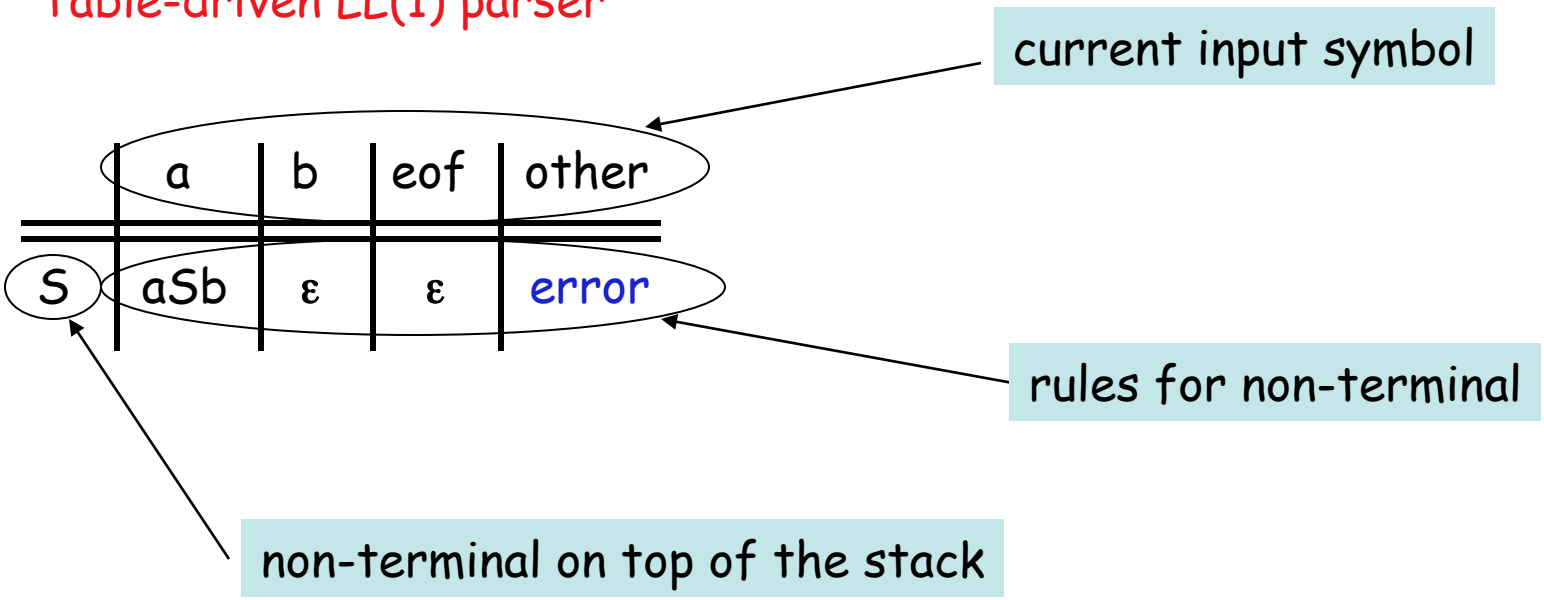
LL(1) Parser Example

Table-driven LL(1) parser

	a	b	eof	other
S	aSb	ε	ε	error

LL(1) Parser Example

Table-driven LL(1) parser



Building Top Down Parsers

Building the complete table

- Need a row for every *NT* & a column for every *T*
- Need an algorithm to build the table

Filling in TABLE[X,y], X ∈ *NT*, y ∈ *T*

- entry is the rule *X ::= β*, if y ∈ FIRST+(*β*)
- entry is error otherwise

If any entry is defined multiple times, *G* is not *LL(1)*

This is the *LL(1)* table construction algorithm

LL(1) Skeleton Parser

```
token ← next_token()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and token = EOF then
     break & report success
  else if TOS is a terminal then
     if TOS matches token then
        pop Stack                    // recognized TOS
        token ← next_token()
     else report error looking for TOS
  else                               // TOS is a non-terminal
     if TABLE[TOS,token] is A→ B1B2…Bk then
        pop Stack                    // get rid of A
        push Bk, Bk-1, …, B1         // in that order
     else report error expanding TOS
  TOS ← top of Stack
```

exit on success

LL(1) Parser Example

Table-driven LL(1) parser

	a	b	eof	other
S	aSb	ε	ε	error

How to parse input a a a b b b ?

Describe action as sequence of states

(PDA stack content, remaining input, next action)

PDA stack content: [X, … Z], where Z is the TOS
next actions: rule or next input+pop or error or accept

LL(1) Parser Example

Table-driven LL(1) parser

	a	b	eof	other
S	aSb	ε	ε	error

([eof, S], aaabbb, S -> aSb) ⇒

LL(1) Parser Example

Table-driven LL(1) parser

	a	b	eof	other
S	aSb	ε	ε	error

([eof, S], aaabbb, S -> aSb) ⇒
([eof, b, S, a], aaabbb, next input+pop) ⇒

LL(1) Parser Example

Table-driven LL(1) parser

	a	b	eof	other
S	aSb	ε	ε	error

([eof, S], aaabbb, S -> aSb) ⇒
([eof, b, S, a], aaabbb, next input+pop) ⇒
([eof, b, S], aabbb, S -> aSb) ⇒

LL(1) Parser Example

Table-driven LL(1) parser

	a	b	eof	other
S	aSb	ε	ε	error

([eof, S], aaabbb, S -> aSb) ⇒
([eof, b, S, a], aaabbb, next input+pop) ⇒
([eof, b, S], aabbb, S -> aSb) ⇒
([eof, b, b, S, a], aabbb, next input+pop) ⇒

LL(1) Parser Example

Table-driven LL(1) parser

	a	b	eof	other
S	aSb	ε	ε	error

([eof, S], aaabbb, S -> aSb) ⇒
([eof, b, S, a], aaabbb, next input+pop) ⇒
([eof, b, S], aabbb, S -> aSb) ⇒
([eof, b, b, S, a], aabbb, next input+pop) ⇒
([eof, b, b, S], abbb, S -> aSb) ⇒

LL(1) Parser Example

Table-driven LL(1) parser

	a	b	eof	other
S	aSb	ε	ε	error

([eof, S], aaabbb, S -> aSb) ⇒
([eof, b, S, a], aaabbb, next input+pop) ⇒
([eof, b, S], aabbb, S -> aSb) ⇒
([eof, b, b, S, a], aabbb, next input+pop) ⇒
([eof, b, b, S], abbb, S -> aSb) ⇒
([eof, b, b, b, S, a], abbb, next input+pop) ⇒

LL(1) Parser Example

Table-driven LL(1) parser

	a	b	eof	other
S	aSb	ε	ε	error

([eof, S], aaabbb, S -> aSb) ⇒
([eof, b, S, a], aaabbb, next input+pop) ⇒
([eof, b, S], aabbb, S -> aSb) ⇒
([eof, b, b, S, a], aabbb, next input+pop) ⇒
([eof, b, b, S], abbb, S -> aSb) ⇒
([eof, b, b, b, S, a], abbb, next input+pop) ⇒
([eof, b, b, b, S], bbb, S -> ε) ⇒

LL(1) Parser Example

Table-driven LL(1) parser

	a	b	eof	other
S	aSb	ε	ε	error

([eof, S], aaabbb, S -> aSb) ⇒
([eof, b, S, a], aaabbb, next input+pop) ⇒
([eof, b, S], aabbb, S -> aSb) ⇒
([eof, b, b, S, a], aabbb, next input+pop) ⇒
([eof, b, b, S], abbb, S -> aSb) ⇒
([eof, b, b, b, S, a], abbb, next input+pop) ⇒
([eof, b, b, b, S], bbb, S -> ε) ⇒
([eof, b, b, b], bbb, next input+pop) ⇒ ([eof, b, b], bb, next input+pop) ⇒
([eof, b], b, next input+pop) ⇒ ([eof], eof, accept)

LL(1) Parser Example

Table-driven LL(1) parser

	a	b	eof	other
S	aSb	ε	ε	error

([eof, S], aaabbb, S -> aSb) ⇒
([eof, b, S, a], aaabbb, next input+pop) ⇒
([eof, b, S], aabbb, S -> aSb) ⇒
([eof, b, b, S, a], aabbb, next input+pop) ⇒
([eof, b, b, S], abbb, S -> aSb) ⇒
([eof, b, b, b, S, a], abbb, next input+pop) ⇒
([eof, b, b, b, S], bbb, S -> ε) ⇒
([eof, b, b, b], bbb, next input+pop) ⇒ ([eof, b, b], bb, next input+pop) ⇒
([eof, b], b, next input+pop) ⇒ ([eof], eof, accept)

LL(1) Parser Example

Recursive descent LL(1) parser

	a	b	eof	other
S	aSb	ε	ε	error

1. Every NT is associated with a parsing procedure.

2. The parsing procedure for A ε NT, proc A, is responsible to parse and consume any string that can be derived from A ; it may recursively call other parsing procedures.

3. The parser is invoked by calling proc S for start symbol S.

LL(1) Parser Example

Recursive descent LL(1) parser

	a	b	eof	other
S	aSb	ε	ε	error

```
main ( ) {
  token = next_token();
  if (S ( ) and token = eof )
     print "accept"
  else
     print "error";
}
```

```
bool S ( ) {
   switch token {
      case a: token = next_token();
              S();
              if token = b
                 {token = next_token(); return true;}
              else
                 return false;
              break;
      case b, case eof: return true; break;
     default: return false;
   }
}
```

LL(1) Parser Example

Recursive descent LL(1) parser

	a	b	eof	other
S	aSb	ε	ε	error

```
main ( ) {
  token = next_token();
  if (S ( ) and token = eof )
     print "accept"
  else
     print "error";
}
```

How to parse input a a a b b b ?

```
bool S ( ) {
   switch token {
      case a: token = next_token();
              S();
              if token = b
                 {token = next_token(); return true;}
              else
                 return false;
              break;
      case b, case eof: return true; break;
     default: return false;
   }
}
```

Recursive Descent (homework 5)

Program ::= Stmtlist **.**
Stmtlist ::= Stmt NextStmt
NextStmt := **;** Stmtlist | **epsilon**
Stmt ::= Assign | Print
Assign ::= **ID** = Expr
Print ::= **!** **ID**
Expr ::= + Expr Expr |
- Expr Expr |
* Expr Expr |
ID |
ICONST

Example:
a=2;b=3;c=+3*ab;!c.

Write a recursive descent, single pass ILOC code generator.
- only single char IDs or ICONST; no blanks; single line program
- use call to "next_register()" to get a fresh, virtual register
- use call to offset(ID) to get memory location relative to base addr 1024
- use call to value(ICONST) to get integer value of constant
- hint: recursive procedures return virtual register in which computation result will be stored at runtime, or "null".

Recursive Descent (Summary)

- Build FIRST (and FOLLOW) sets
- Massage grammar to have *LL(1)* condition
  - Remove left recursion
  - Left factor it (→will talk about this within the next few slides)
- Define a procedure for each non-terminal
  - Implement a case for each right-hand side
  - Call procedures as needed for non-terminals
- Add extra code, as needed
  - Perform context-sensitive checking
  - Build an IR (e.g., simple code generation)

Can we automate this process?

Left Factoring

What if my grammar does not have the LL(1) property?

⇒ Sometimes, we can transform the grammar

The Algorithm

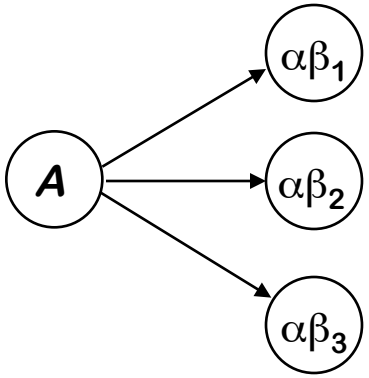
∀ $A \in NT$,
  find the longest prefix α *that occurs in two or more right-hand sides of A*
  if $\alpha \neq \varepsilon$ *then replace all of the A productions,*
    $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$,
  with
    $A \rightarrow \alpha Z \mid \gamma$
    $Z \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
  where Z is a new element of NT

Repeat until no common prefixes remain

Left Factoring

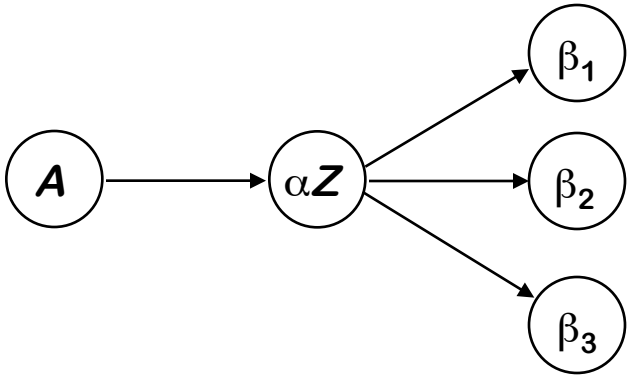
A graphical explanation for the same idea

$A \rightarrow \alpha\beta_1$
$\quad | \ \alpha\beta_2$
$\quad | \ \alpha\beta_3$



becomes …

$A \rightarrow \alpha\ Z$
$Z \rightarrow \beta_1$
$\quad | \ \beta_2$
$\quad | \ \beta_3$



Left Factoring (An example)

Consider the following fragment of the expression grammar

Factor	→	Identifier
	|	Identifier [*ExprList*]
	|	Identifier (*ExprList*)

FIRST(rhs_1) = { Identifier }
FIRST(rhs_2) = { Identifier }
FIRST(rhs_3) = { Identifier }

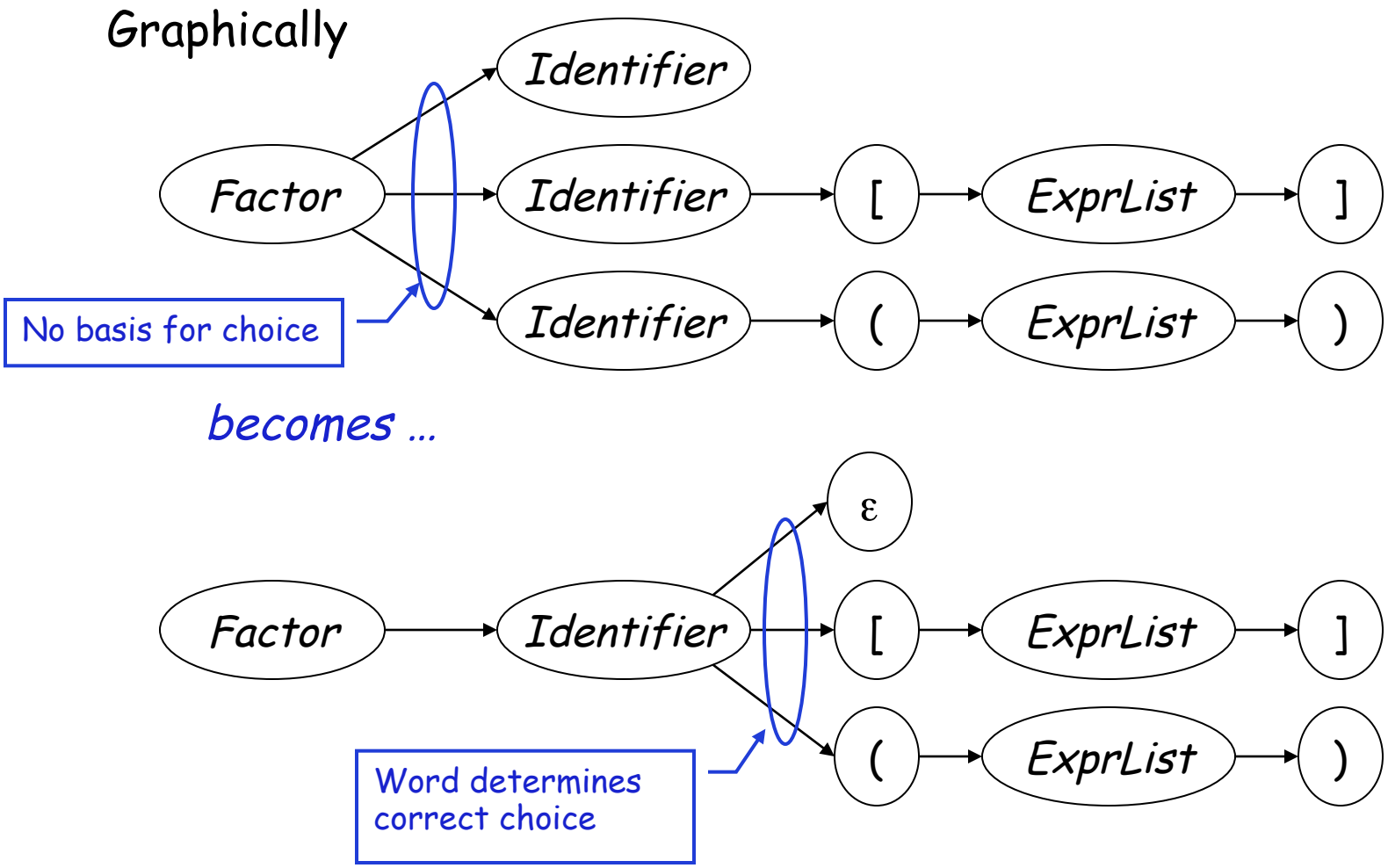
After left factoring, it becomes

Factor	→	Identifier *Arguments*
Arguments	→	[*ExprList*]
	|	(*ExprList*)
	|	ε

FIRST(rhs_1) = { Identifier }
FIRST(rhs_2) = { [}
FIRST(rhs_3) = { (}
FIRST(rhs_4) = FOLLOW(*Factor*)
⇒ It has the *LL(1)* property

This form has the same syntax, with the *LL(1)* property

Left Factoring (An example)



LL(1) Grammars

Question

By *eliminating left recursion* and *left factoring*, can we transform an arbitrary CFG to a form where it meets the *LL(1)* condition? (and can be parsed predictively with a single token lookahead?)

Answer

Given a CFG that doesn't meet the *LL(1)* condition, it is undecidable whether or not an equivalent *LL(1)* grammar exists.

Language that cannot be LL(k)

Example

$\{a^n 0\, b^n \mid n \geq 1\} \cup \{a^n 1\, b^{2n} \mid n \geq 1\}$ has no *LL(k)* grammar

$G \rightarrow \underline{a}A\underline{b}$
$\quad \mid \underline{a}B\underline{bb}$
$A \rightarrow \underline{a}A\underline{b}$
$\quad \mid \underline{0}$
$B \rightarrow \underline{a}B\underline{bb}$
$\quad \mid \underline{1}$

Problem: need an unbounded number of a characters before you can determine whether you are in the A group or the B group.

Parsing (Syntax Analysis)

EAC Chapters 3.4

Parsing Techniques: Bottom-up parsers

LR(1), operator precedence

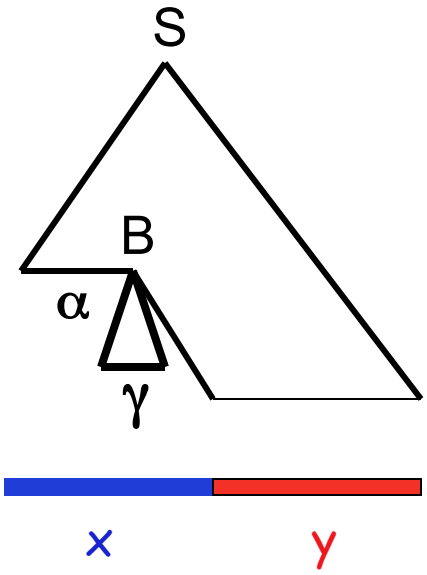
1 input symbol lookahead

construct rightmost deriviation (backwards)

input: read left-to-right

rule B ::= γ

$$S \Rightarrow^*_{rm} \alpha B y \Rightarrow_{rm} \alpha \gamma y \Rightarrow^*_{rm} x y$$



Parsing Techniques: Bottom-up parsers

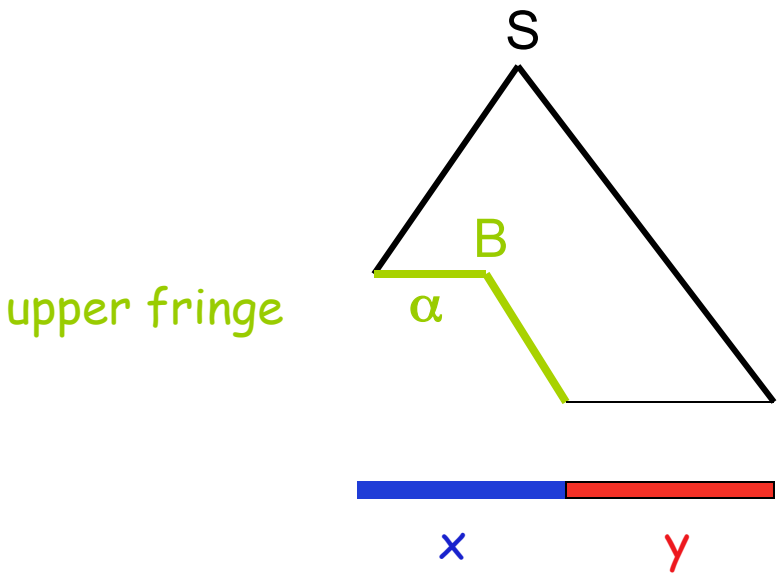
LR(1), operator precedence

1 input symbol lookahead

construct rightmost deriviation (backwards)

input: read left-to-right

$$S \Rightarrow^*_{rm} \alpha\, B\, y \Rightarrow_{rm} \alpha\, \gamma\, y \Rightarrow^*_{rm} x\, y$$



Finding Reductions

Consider the simple grammar

1	*Goal*	→	a *A B* e
2	*A*	→	*A* b c
3		|	b
4	*B*	→	d

And the input string abbcde

Sentential Form	*Next Reduction*	
	Prod'n	*Pos'n*
abbcde	3	2

The trick is scanning the input and finding the next reduction
The mechanism for doing this must be efficient

Finding Reductions

Consider the simple grammar

1	*Goal*	→	a *A B* e
2	*A*	→	*A* b c
3		|	b
4	*B*	→	d

And the input string abbcde

Sentential Form	*Next Reduction*	
	Prod'n	*Pos'n*
abbcde	3	2
a *A* bcde	2	4

The trick is scanning the input and finding the next reduction
The mechanism for doing this must be efficient

Finding Reductions

Consider the simple grammar

1	*Goal*	→	a *A B* e
2	*A*	→	*A* b c
3		|	b
4	*B*	→	d

And the input string abbcde

Sentential Form	*Next Reduction* *Prod'n*	*Pos'n*
abbcde	3	2
a *A* bcde	2	4
a *A* de	4	3

The trick is scanning the input and finding the next reduction
The mechanism for doing this must be efficient

Finding Reductions

Consider the simple grammar

1	*Goal*	→	a *A B* e
2	*A*	→	*A* b c
3		|	b
4	*B*	→	d

And the input string abbcde

Sentential Form	*Next Reduction*	
	Prod'n	*Pos'n*
abbcde	3	2
a *A* bcde	2	4
a *A* de	4	3
a *A B* e	1	4

The trick is scanning the input and finding the next reduction
The mechanism for doing this must be efficient

Finding Reductions

Consider the simple grammar

1	*Goal*	→	a *A B* e
2	*A*	→	*A* b c
3		|	b
4	*B*	→	d

And the input string abbcde

Sentential Form	*Next Reduction*	
	Prod'n	*Pos'n*
abbcde	3	2
a *A* bcde	2	4
a *A* de	4	3
a *A B* e	1	4
Goal	—	—

The trick is scanning the input and finding the next reduction
The mechanism for doing this must be efficient

Finding Reductions (Handles)

The parser must find a substring β of the tree's frontier that

matches some production $A \rightarrow \beta$ that occurs as one step in the rightmost derivation

Informally, we call this substring β a *handle*

Formally,

A *handle* of a right-sentential form γ is a pair $\langle A \rightarrow \beta, k \rangle$ where $A \rightarrow \beta \in P$ and k is the position in γ of β's rightmost symbol.

If $\langle A \rightarrow \beta, k \rangle$ is a handle, then replacing β at k with A produces the right sentential form from which γ is derived in the rightmost derivation.

Because γ is a right-sentential form, the substring to the right of a handle contains only terminal symbols

$\Rightarrow$ the parser doesn't need to scan past the handle (only lookahead)

$\Rightarrow$ The right end of the handle will be on top of the stack, not within the stack. Need lookahead to determine whether we reached the handle.

Finding Reductions (Handles)

Critical Insight *(Theorem?)*

If G is unambiguous, then every right-sentential form has a unique handle.

If we can find those handles, we can build a derivation !

Sketch of Proof:

1. *G* is unambiguous ⇒ rightmost derivation is unique
2. ⇒ a unique production $A \rightarrow \beta$ applied to derive γ_i from γ_{i-1}
3. ⇒ a unique position *k* at which $A \rightarrow \beta$ is applied
4. ⇒ a unique handle $\langle A \rightarrow \beta, k \rangle$

This all follows from the definitions

Example

1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	number
9		|	id
10		|	(*Expr*)

The expression grammar

Prod'n.	*Sentential Form*	*Handle*
—	*Goal*	—
1	*Expr*	1,1
3	*Expr* - *Term*	3,3
5	*Expr* - *Term* * *Factor*	5,5
9	*Expr* - *Term* * <id,y>	9,5
7	*Expr* - *Factor* * <id,y>	7,3
8	*Expr* - <num,2> * <id,y>	8,3
4	*Term* - <num,2> * <id,y>	4,1
7	*Factor* - <num,2> * <id,y>	7,1
9	<id,x> - <num,2> * <id,y>	9,1

Handles for rightmost derivation of x - 2 * y

Handle-pruning, Bottom-up Parsers

The process of discovering a handle & reducing it to the appropriate left-hand side is called *handle pruning*

Handle pruning forms the basis for a bottom-up parsing method

To construct a rightmost derivation

$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow w$

Apply the following simple algorithm

for $i \leftarrow n$ *to 1 by -1*

Find the handle $\langle A_i \rightarrow \beta_i , \mathbf{k}_i \rangle$ in γ_i

Replace β_i *with* A_i *to generate* γ_{i-1}

This takes *2n* steps

Handle-pruning, Bottom-up Parsers

One implementation technique is the *shift-reduce parser*

```
push INVALID
token ← next_token( )
repeat until (top of stack = Goal and token = EOF)
    if the top of the stack is a handle A→β
        then     // reduce β to A
            pop |β| symbols off the stack
            push A onto the stack
        else if (token ≠ EOF)
            then // shift
                push token
                token ← next_token( )
            else     // need to shift but out of input
            report an error
```

How do errors show up?

- **failure to find a handle**
- **hitting EOF & needing to shift (final else clause)**

Either generates an error

Figure 3.7 in EAC

Back to x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id		

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id		

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id	*none*	shift
$ *Expr* –	num * id	*none*	shift
$ *Expr* – num	* id		

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id	*none*	shift
$ *Expr* –	num * id	*none*	shift
$ *Expr* – num	* id	8,3	red. 8
$ *Expr* – *Factor*	* id	7,3	red. 7
$ *Expr* – *Term*	* id		

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id	*none*	shift
$ *Expr* –	num * id	*none*	shift
$ *Expr* – num	* id	8,3	red. 8
$ *Expr* – *Factor*	* id	7,3	red. 7
$ *Expr* – *Term*	* id	*none*	shift
$ *Expr* – *Term* *	id	*none*	shift
$ *Expr* – *Term* * id			

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x – 2 * y

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id	*none*	shift
$ *Expr* –	num * id	*none*	shift
$ *Expr* – num	* id	8,3	red. 8
$ *Expr* – *Factor*	* id	7,3	red. 7
$ *Expr* – *Term*	* id	*none*	shift
$ *Expr* – *Term* *	id	*none*	shift
$ *Expr* – *Term* * id		9,5	red. 9
$ *Expr* – *Term* * *Factor*		5,5	red. 5
$ *Expr* – *Term*		3,3	red. 3
$ *Expr*		1,1	red. 1
$ *Goal*		*none*	accept

5 shifts +
9 reduces +
1 accept

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x – 2 * y

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id	*none*	shift
$ *Expr* –	num * id	*none*	shift
$ *Expr* – num	* id	8,3	red. 8
$ *Expr* – *Factor*	* id	7,3	red. 7
$ *Expr* – *Term*	* id	*none*	shift
$ *Expr* – *Term* *	id	*none*	shift
$ *Expr* – *Term* * id		9,5	red. 9
$ *Expr* – *Term* * *Factor*		5,5	red. 5
$ *Expr* – *Term*		3,3	red. 3
$ *Expr*		1,1	red. 1
$ *Goal*		*none*	accept

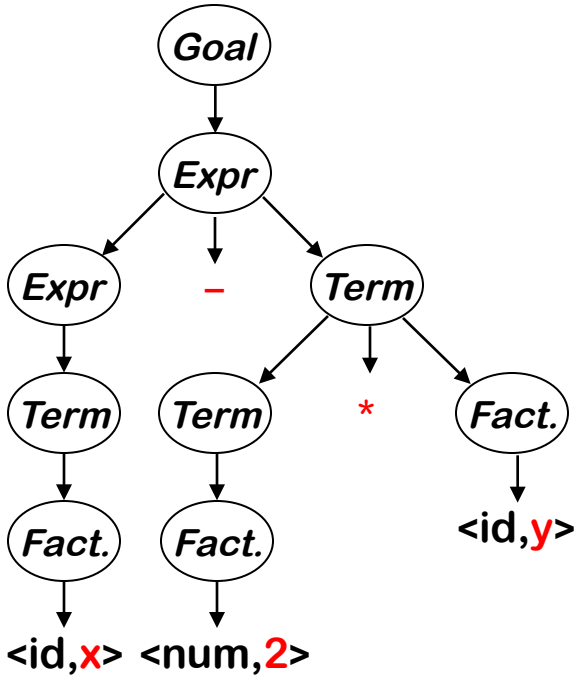
shift here

reduce here

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x - 2 * y

Stack	Input	Action
$	id – num * id	shift
$ id	– num * id	red. 9
$ *Factor*	– num * id	red. 7
$ *Term*	– num * id	red. 4
$ *Expr*	– num * id	shift
$ *Expr* –	num * id	shift
$ *Expr* – num	* id	red. 8
$ *Expr* – *Factor*	* id	red. 7
$ *Expr* – *Term*	* id	shift
$ *Expr* – *Term* *	id	shift
$ *Expr* – *Term* * id		red. 9
$ *Expr* – *Term* * *Factor*		red. 5
$ *Expr* – *Term*		red. 3
$ *Expr*		red. 1
$ *Goal*		accept



Shift-reduce Parsing

Shift reduce parsers are easily built and easily understood

A shift-reduce parser has just four actions

- *Shift* — next word is shifted onto the stack
- *Reduce* — right end of handle is at top of stack
  - Locate left end of handle within the stack
  - Pop handle off stack & push appropriate *lhs*
- *Accept* — stop parsing & report success
- *Error* — call an error reporting/recovery routine

Accept & Error are simple

Shift is just a push and a call to the scanner

Reduce takes |*rhs*| pops & 1 push

If handle-finding requires state, put it in the stack ⇒ 2x work

Handle finding is key
- handle is on stack
- finite set of handles

⇒ use a DFA !

Announcements

- **Midterm on Wednesday, March 8**
  - closed book, closed notes
  - covers everything up to (and including) top-down parsing
  - no recitation on Tuesday, March 8
  - Q&A session second half of this Thursday 3/3 lecture – come prepared!

- Homework set 5 released,
  - → last homework before midterm (only one problem)
  - → due this Friday 11:59pm EST, no late submission will be accepted.

Next class

More Syntax Analysis (bottom-up)

Read EaC: Chapter 3.4