RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers

Syntax Analysis
Part 4

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

Announcements

Roadmap for the remainder of the course

- Fourth homework:
  Due Monday, March 28
- Project #2 – Bottom-up parser and compiler
  Will be posted next week, due April 13 (tentative)
- Project #3 – Peephole optimizer for ILOC
  Will be posted April 13, due May 2 (tentative)
- Second midterm on Wednesday, April 6 (60 minutes in class)
- Final exam on May 10 (60 minutes at assigned location)
- At least 3 more homeworks

Bottom-up Parsing (Syntax Analysis)

EAC Chapters 3.4

Reminder: Top-down parsers

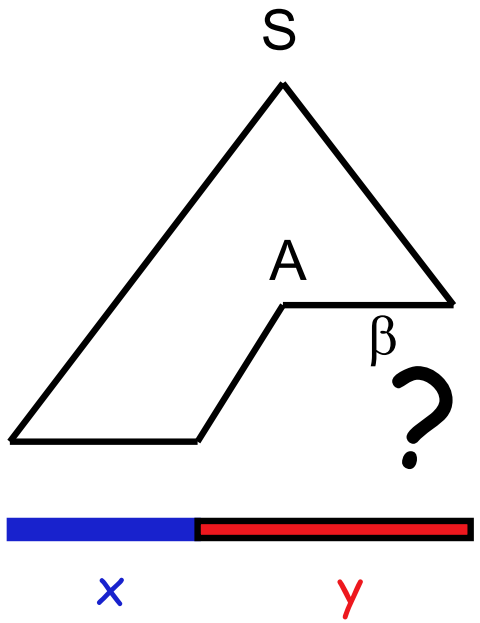
LL(1), recursive descent

1 input symbol lookahead

construct leftmost deriviation (forwards)

input: read left-to-right

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



? Means that we don't know yet this part of the parse tree

Parsing Techniques: Bottom-up parsers

LR(1), operator precedence

1 input symbol lookahead

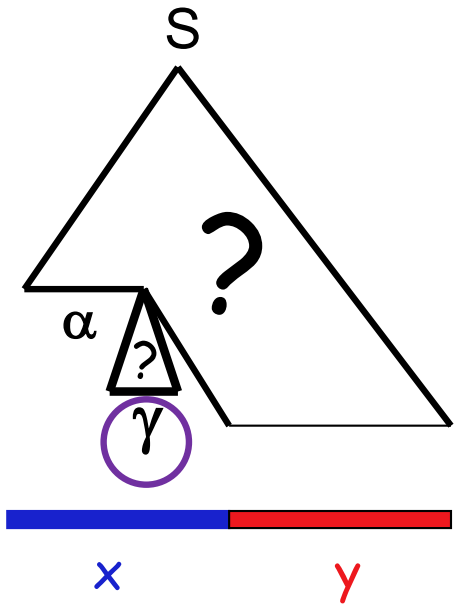
construct rightmost deriviation (backwards)

input: read left-to-right

handle

rule B ::= γ

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



? Means that we don't know yet this part of the parse tree

Parsing Techniques: Bottom-up parsers

LR(1), operator precedence

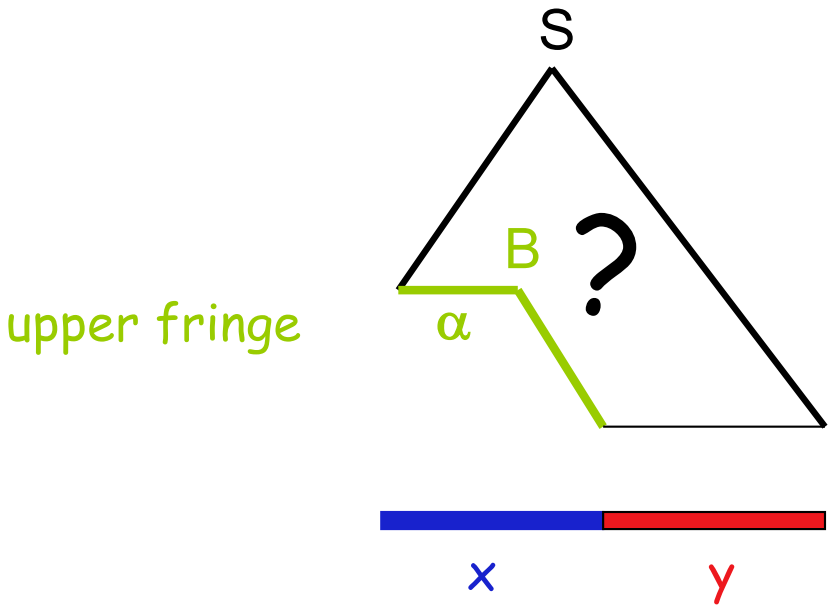
1 input symbol lookahead

construct rightmost deriviation (backwards)

input: read left-to-right

rule B ::= γ

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



? Means that we don't know yet this part of the parse tree

LR(1) Parser Example

Is the following grammar LL(1), L(2), or LR(1)?

S ::= a b | a b c

Is the following grammar LR(1) or even LR(0)?

S ::= a S b | c

Basic idea:

shift symbols from input onto the stack until top of the stack is a RHS of a rule; if so, "apply" rule backwards (***reduce***) by replacing top of the stack by the LHS non-terminal.

Challenge: When to shift, and when to reduce

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

This all follows from the definitions

Revisit previous example

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

LR(0) items and LR(0) cannonical collection

S0: {[*Goal* → • *a A B e*]}

S1: {[*Goal* → *a* • *A B e*], [*A* → • *A b c*], [*A* → • *b*]}

S2: {[*Goal* → *a A* • *B e*], [*A* → *A* • *b c*], [B→ • *d*] }

S3: {[*A*→ *b* •] }

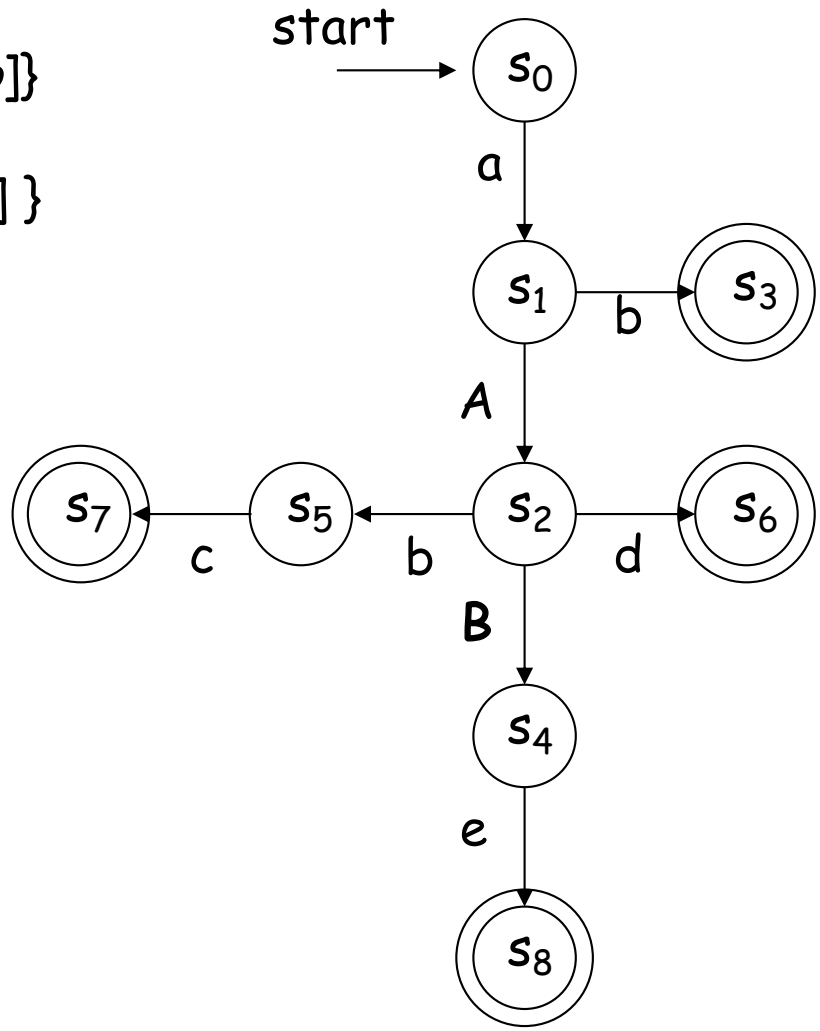
S4: {[*Goal* → *a A B* • *e*] }

S5: {[*A* → *A b* • *c*] }

S6: {[B→ *d* •] }

S7: {[*A* → *A b c* •] }

S8: {[*Goal* → *a A B e* •] }



Another Example (a very busy slide)

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 $\langle A_i \rightarrow \beta_i , k_i \rangle$ 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 // bottom of stack marker
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

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

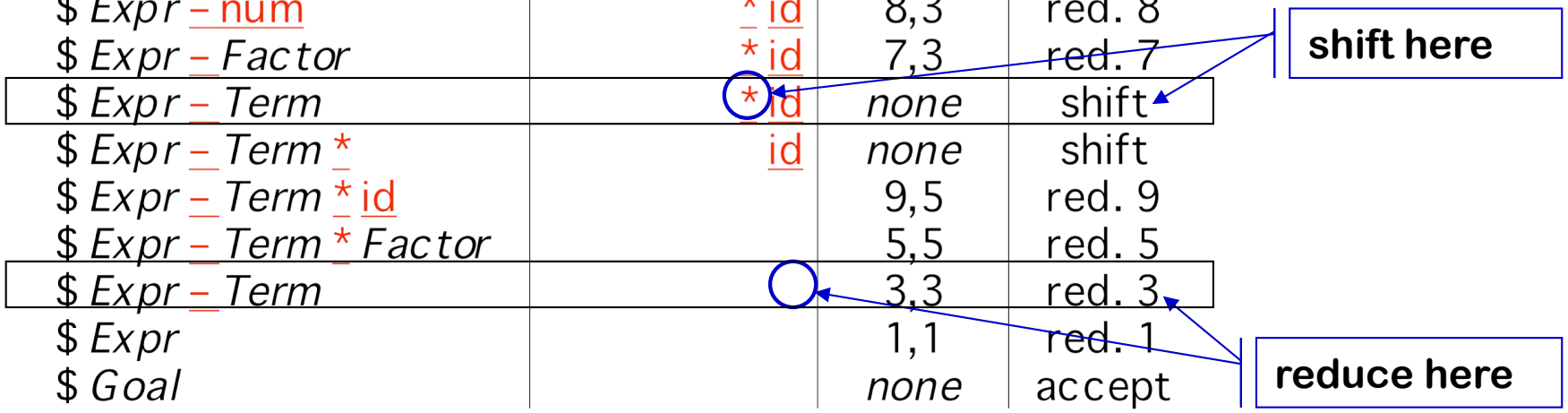
Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id	*none*	shift
$ *Expr* –	num * id	*none*	shift
$ *Expr* – num	* id	8,3	red. 8
$ *Expr* – *Factor*	* id	7,3	red. 7
$ *Expr* – *Term*	* id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

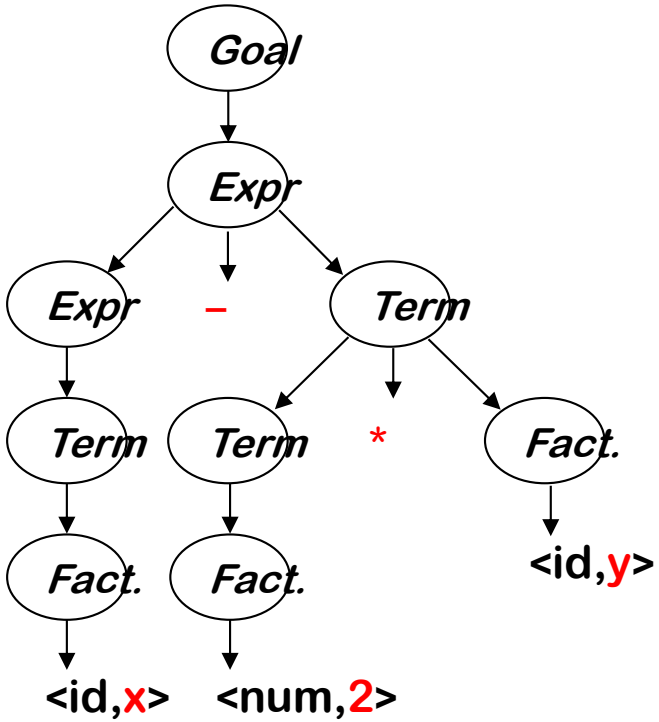
Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id	*none*	shift
$ *Expr* –	num * id	*none*	shift
$ *Expr* – num	* id	8,3	red. 8
$ *Expr* – *Factor*	* id	7,3	red. 7
$ *Expr* – *Term*	* id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



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



LR(1) Skeleton Parser

```
stack.push(INVALID); stack.push(s0);
not_found = true;
token = scanner.next_token();
do while (not_found) {
        s = stack.top();
        if ( ACTION[s,token] == "reduce A→β" ) then {
                stack.popnum(2*|β|); // pop 2*|β| symbols
        s = stack.top();
        stack.push(A);
        stack.push(GOTO[s,A]);
        }
        else if ( ACTION[s,token] == "shift si" ) then {
                stack.push(token); stack.push(si);
                token ← scanner.next_token();
        }
        else if ( ACTION[s,token] == "accept"
                                & token == EOF )
                then not_found = false;
        else report a syntax error and recover;
}
report success;
```

The skeleton parser

- uses ACTION & GOTO tables
- does |*words*| shifts
- does |derivation| reductions
- does 1 accept

Next class

More Bottom-up LR(1) parsing

Error Recovery