LECTURE 9

Computing an LL(1) Parsing Table

REVIEW

Last lecture, we parsed an input string using our recursive descent parser. We then discussed how we could create an equivalent table-driven parser.

We ended the lecture with a discussion of how we might begin to build a parsing table.

Today, we’ll create the parsing table from scratch and use it to parse an input string.

LL(1) GRAMMARS

- A grammar whose parsing table has no multiply-defined entries is an LL(1) grammar.
- Uses one input symbol of look-ahead at each step to make a parsing decision.
- No ambiguous or left-recursive grammar can be LL(1).

PARSING TABLE

The basic outline for creating a parsing table from a LL(1) grammar is the following:

- Compute the First sets of the non-terminals.
- Compute the Follow sets of the non-terminals.
- For each production N → ω,
  - Add N → ω to M[N, t] for each t in First(ω).
  - If First(ω) contains ϵ, add N → ω to M[N, t] for each t in Follow(N).
- All undefined entries represent a parsing error.

FIRST SETS

Let’s calculate our First sets. The rules are:

- If X is a terminal symbol, First(X) = X.
- If X is ϵ, add ϵ to First(X).
- If X is a non-terminal, look at all productions where X is on left-hand side. Each production will be of the form: X → $Y_1Y_2 \ldots Y_k$ where Y is a nonterminal or terminal. Then:
  - Put First(Y_1) - ϵ in First(X).
  - If ϵ is in First(Y_1), then put First(Y_2) - ϵ in First(X).
  - If ϵ is in First(Y_2), then put First(Y_3) - ϵ in First(X).
  - …
  - If ϵ is in Y_1, Y_2, …, Y_k, then add ϵ to First(X).

FIRST SETS

The very first thing we'll do is compute the First set of each of our non-terminals.

program → *expr*
expr → *term expr_tail*
expr_tail → + *term expr_tail* | ϵ
term → *factor term_tail*
term_tail → * *factor term_tail* | ϵ
factor → (*expr*) | int

First(factor) = ?

FIRST SETS

The very first thing we'll do is compute the First set of each of our non-terminals.

program → *expr*
expr → *term expr_tail*
expr_tail → + *term expr_tail* | ϵ
term → *factor term_tail*
term_tail → * *factor term_tail* | ϵ
factor → (*expr*) | int

First(factor) = {'(', int}

FIRST SETS

The very first thing we’ll do is compute the First set of each of our non-terminals.

program → *expr*
expr → *term expr_tail*
expr_tail → + *term expr_tail* | ϵ
term → *factor term_tail*
term_tail → * *factor term_tail* | ϵ
factor → (*expr*) | int

First(term) = ?

First(factor) = {‘(‘, int}

FIRST SETS

The very first thing we’ll do is compute the First set of each of our non-terminals.

program → *expr*

expr → *term expr_tail*

expr_tail → + *term expr_tail* | ϵ

term → *factor term_tail*

term_tail → * *factor term_tail* | ϵ

factor → (*expr*) | int

First(term) = {‘(‘, int}

First(factor) = {‘(‘, int}

FIRST SETS

The very first thing we’ll do is compute the First set of each of our non-terminals.

program → *expr*

expr → *term expr_tail*

expr_tail → + *term expr_tail* | ϵ

term → *factor term_tail*

term_tail → * *factor term_tail* | ϵ

factor → (*expr*) | int

First(expr) = {‘(‘, int}

First(term) = {‘(‘, int}

First(factor) = {‘(‘, int}

FIRST SETS

The very first thing we’ll do is compute the First set of each of our non-terminals.

program → *expr*	First(program) = {‘(‘, int}
expr → *term expr_tail*	First(expr) = {‘(‘, int}
expr_tail → + *term expr_tail* | ϵ	
term → *factor term_tail*	First(term) = {‘(‘, int}
term_tail → * *factor term_tail* | ϵ	
factor → (*expr*) | int	First(factor) = {‘(‘, int}

FIRST SETS

The very first thing we’ll do is compute the First set of each of our non-terminals.

program → *expr*
expr → *term expr_tail*
expr_tail → + *term expr_tail* | ϵ
term → *factor term_tail*
term_tail → * *factor term_tail* | ϵ
factor → (*expr*) | int

First(program) = {‘(‘, int}
First(expr) = {‘(‘, int}
First(expr_tail) = {‘+’, ϵ}
First(term) = {‘(‘, int}

First(factor) = {‘(‘, int}

FIRST SETS

The very first thing we’ll do is compute the First set of each of our non-terminals.

program → *expr*

expr → *term expr_tail*

expr_tail → + *term expr_tail* | ϵ

term → *factor term_tail*

term_tail → * *factor term_tail* | ϵ

factor → (*expr*) | int

First(program) = {‘(‘, int}

First(expr) = {‘(‘, int}

First(expr_tail) = {‘+’, ϵ}

First(term) = {‘(‘, int}

First(term_tail) = {‘*’, ϵ}

First(factor) = {‘(‘, int}

FOLLOW SETS

Now, we'll look at the Follow sets. The rules are:

- If N is the starting non-terminal, put $ in Follow(N).
- If X → αN, where α is some string of non-terminals and/or terminals, put Follow(X) in Follow(N).
- If X → $\alpha N \beta$ where α,β are some string of non-terminals and/or terminals, put First(β) in Follow(N). If First(β) includes ϵ, then put Follow(X) in Follow(N).

FOLLOW SETS

Now let's calculate the Follow sets for our non-terminals.

program → *expr*
expr → *term expr_tail*
expr_tail → + *term expr_tail* | ϵ
term → *factor term_tail*
term_tail → * *factor term_tail* | ϵ
factor → (*expr*) | int

Follow(program) = {$}

FOLLOW SETS

Now let’s calculate the Follow sets for our non-terminals.

program → *expr*
expr → *term expr_tail*
expr_tail → + *term expr_tail* | ϵ
term → *factor term_tail*
term_tail → * *factor term_tail* | ϵ
factor → (*expr*) | int

Follow(program) = {$}
Follow(expr) = {$, ‘)’}

FOLLOW SETS

Now let's calculate the Follow sets for our non-terminals.

program → *expr*
expr → *term expr_tail*
expr_tail → + *term expr_tail* | ϵ
term → *factor term_tail*
term_tail → * *factor term_tail* | ϵ
factor → (*expr*) | int

Follow(program) = {$}
Follow(expr) = {$, ')'}
Follow(expr_tail) = {$, ')'}

FOLLOW SETS

Now let’s calculate the Follow sets for our non-terminals.

program → *expr*
expr → *term expr_tail*
expr_tail → + *term expr_tail* | ϵ
term → *factor term_tail*
term_tail → * *factor term_tail* | ϵ
factor → (*expr*) | int

Follow(program) = {$}
Follow(expr) = {$, ‘)’}
Follow(expr_tail) = {$, ‘)’}
Follow(term) = {‘+’, $, ‘)’}

FOLLOW SETS

Now let’s calculate the Follow sets for our non-terminals.

program → *expr*
expr → *term expr_tail*
expr_tail → + *term expr_tail* | ϵ
term → *factor term_tail*
term_tail → * *factor term_tail* | ϵ
factor → (*expr*) | int

Follow(program) = {$}
Follow(expr) = {$, ‘)’}
Follow(expr_tail) = {$, ‘)’}
Follow(term) = {‘+’, $, ‘)’}
Follow(term_tail) = {‘+’, $, ‘)’}

FOLLOW SETS

Now let’s calculate the Follow sets for our non-terminals.

program → *expr*
expr → *term expr_tail*
expr_tail → + *term expr_tail* | ϵ
term → *factor term_tail*
term_tail → * *factor term_tail* | ϵ
factor → (*expr*) | int

Follow(program) = {$}
Follow(expr) = {$, ‘)’}
Follow(expr_tail) = {$, ‘)’}
Follow(term) = {‘+’, $, ‘)’}
Follow(term_tail) = {‘+’, $, ‘)’}
Follow(factor) = {‘*’, ‘+’, $, ‘)’}

FIRST AND FOLLOW SETS

So, now we have our First and Follow sets. From here, we can construct our parsing table.

program → *expr*	First(program) = {‘(‘, int}	Follow(program) = {$}
expr → *term expr_tail*	First(expr) = {‘(‘, int}	Follow(expr) = {$, ‘)’}
expr_tail → + *term expr_tail* | *ϵ*	First(expr_tail) = {‘+’, *ϵ*}	Follow(expr_tail) = {$, ‘)’}
term → *factor term_tail*	First(term) = {‘(‘, int}	Follow(term) = {‘+’, $, ‘)’}
term_tail → * *factor term_tail* | ϵ	First(term_tail) = {‘*’, *ϵ*}	Follow(term_tail) = {‘+’, $, ‘)’}
factor → (*expr*) | int	First(factor) = {‘(‘, int}	Follow(factor) = {‘*’, ‘+’, $, ‘)’}

CONSTRUCTING AN LL(1) PARSING TABLE

1. *program* → *expr*
2. *expr* → *term expr_tail*
3. *expr_tail* → + *term expr_tail*
4. *expr_tail* → ϵ
5. *term* → *factor term_tail*
6. *term_tail* → * *factor term_tail*
7. *term_tail* → ϵ
8. *factor* → (*expr*)
9. *factor* → int

First(program) = {‘(‘, int}
First(expr) = {‘(‘, int}
First(expr_tail) = {‘+’, ϵ}
First(term) = {‘(‘, int}
First(term_tail) = {‘*’, ϵ}
First(factor) = {‘(‘, int}

Follow(program) = {$}
Follow(expr) = {$, ‘)’}
Follow(expr_tail) = {$, ‘)’}
Follow(term) = {‘+’, $, ‘)’}
Follow(term_tail) = {‘+’, $, ‘)’}
Follow(factor) = {‘*’, ‘+’, $, ‘)’}

Let’s start with program, production number 1. What is First(expr)?

N	(	int	*	+	)	$
program						
expr						
expr_tail						
term						
term_tail						
factor						

CONSTRUCTING AN LL(1) PARSING TABLE

1. *program* → *expr*
2. *expr* → *term expr_tail*
3. *expr_tail* → + *term expr_tail*
4. *expr_tail* → ϵ
5. *term* → *factor term_tail*
6. *term_tail* → * *factor term_tail*
7. *term_tail* → ϵ
8. *factor* → (*expr*)
9. *factor* → int

First(program) = {‘(‘, int}
First(expr) = {‘(‘, int}
First(expr_tail) = {‘+’, ϵ}
First(term) = {‘(‘, int}
First(term_tail) = {‘*’, ϵ}
First(factor) = {‘(‘, int}

Follow(program) = {$}
Follow(expr) = {$, ‘)’}
Follow(expr_tail) = {$, ‘)’}
Follow(term) = {‘+’, $, ‘)’}
Follow(term_tail) = {‘+’, $, ‘)’}
Follow(factor) = {‘*’, ‘+’, $, ‘)’}

N	**(**	**int**	*	**+**	**)**	**$**
program	(1)	(1)				
expr						
expr_tail						
term						
term_tail						
factor						

Note: You are not looking at First() of the left side of the production. You are calculating First() of the right side!

CONSTRUCTING AN LL(1) PARSING TABLE

1. *program* → *expr*
2. *expr* → *term expr_tail*
3. *expr_tail* → + *term expr_tail*
4. *expr_tail* → ϵ
5. *term* → *factor term_tail*
6. *term_tail* → * *factor term_tail*
7. *term_tail* → ϵ
8. *factor* → (*expr*)
9. *factor* → int

First(program) = {‘(‘, int}
First(expr) = {‘(‘, int}
First(expr_tail) = {‘+’, ϵ}
First(term) = {‘(‘, int}
First(term_tail) = {‘*’, ϵ}
First(factor) = {‘(‘, int}

Follow(program) = {$}
Follow(expr) = {$, ‘)’}
Follow(expr_tail) = {$, ‘)’}
Follow(term) = {‘+’, $, ‘)’}
Follow(term_tail) = {‘+’, $, ‘)’}
Follow(factor) = {‘*’, ‘+’, $, ‘)’}

N	(	int	*	+	)	$
program	(1)	(1)				
expr	(2)	(2)				
expr_tail						
term						
term_tail						
factor						

CONSTRUCTING AN LL(1) PARSING TABLE

1. *program* → *expr*
2. *expr* → *term expr_tail*
3. *expr_tail* → + *term expr_tail*
4. *expr_tail* → ϵ
5. *term* → *factor term_tail*
6. *term_tail* → * *factor term_tail*
7. *term_tail* → ϵ
8. *factor* → (*expr*)
9. *factor* → int

First(program) = {‘(‘, int}
First(expr) = {‘(‘, int}
First(expr_tail) = {‘+’, ϵ}
First(term) = {‘(‘, int}
First(term_tail) = {‘*’, ϵ}
First(factor) = {‘(‘, int}

Follow(program) = {$}
Follow(expr) = {$, ‘)’}
Follow(expr_tail) = {$, ‘)’}
Follow(term) = {‘+’, $, ‘)’}
Follow(term_tail) = {‘+’, $, ‘)’}
Follow(factor) = {‘*’, ‘+’, $, ‘)’}

N	(	int	*	+	)	$
program	(1)	(1)				
expr	(2)	(2)				
expr_tail				(3)	(4)	(4)
term						
term_tail						
factor						

CONSTRUCTING AN LL(1) PARSING TABLE

1. *program* → *expr*
2. *expr* → *term expr_tail*
3. *expr_tail* → + *term expr_tail*
4. *expr_tail* → ϵ
5. *term* → *factor term_tail*
6. *term_tail* → * *factor term_tail*
7. *term_tail* → ϵ
8. *factor* → (*expr*)
9. *factor* → int

First(program) = {‘(‘, int}
First(expr) = {‘(‘, int}
First(expr_tail) = {‘+’, ϵ}
First(term) = {‘(‘, int}
First(term_tail) = {‘*’, ϵ}
First(factor) = {‘(‘, int}

Follow(program) = {$}
Follow(expr) = {$, ‘)’}
Follow(expr_tail) = {$, ‘)’}
Follow(term) = {‘+’, $, ‘)’}
Follow(term_tail) = {‘+’, $, ‘)’}
Follow(factor) = {‘*’, ‘+’, $, ‘)’}

N	(	int	*	+	)	$
program	(1)	(1)				
expr	(2)	(2)				
expr_tail				(3)	(4)	(4)
term	(5)	(5)				
term_tail						
factor						

CONSTRUCTING AN LL(1) PARSING TABLE

1. *program* → *expr*
2. *expr* → *term expr_tail*
3. *expr_tail* → + *term expr_tail*
4. *expr_tail* → ϵ
5. *term* → *factor term_tail*
6. *term_tail* → * *factor term_tail*
7. *term_tail* → ϵ
8. *factor* → (expr)
9. *factor* → int

First(program) = {‘(‘, int}
First(expr) = {‘(‘, int}
First(expr_tail) = {‘+’, ϵ}
First(term) = {‘(‘, int}
First(term_tail) = {‘*’, ϵ}
First(factor) = {‘(‘, int}

Follow(program) = {$}
Follow(expr) = {$, ‘)’}
Follow(expr_tail) = {$, ‘)’}
Follow(term) = {‘+’, $, ‘)’}
Follow(term_tail) = {‘+’, $, ‘)’}
Follow(factor) = {‘*’, ‘+’, $, ‘)’}

N	**(**	**int**	*	**+**	**)**	**$**
program	(1)	(1)				
expr	(2)	(2)				
expr_tail				(3)	(4)	(4)
term	(5)	(5)				
term_tail			(6)	(7)	(7)	(7)
factor						

CONSTRUCTING AN LL(1) PARSING TABLE

1. *program* → *expr*
2. *expr* → *term expr_tail*
3. *expr_tail* → + *term expr_tail*
4. *expr_tail* → ϵ
5. *term* → *factor term_tail*
6. *term_tail* → * *factor term_tail*
7. *term_tail* → ϵ
8. *factor* → (*expr*)
9. *factor* → int

First(program) = {‘(‘, int}
First(expr) = {‘(‘, int}
First(expr_tail) = {‘+’, ϵ}
First(term) = {‘(‘, int}
First(term_tail) = {‘*’, ϵ}
First(factor) = {‘(‘, int}

Follow(program) = {$}
Follow(expr) = {$, ‘)’}
Follow(expr_tail) = {$, ‘)’}
Follow(term) = {‘+’, $, ‘)’}
Follow(term_tail) = {‘+’, $, ‘)’}
Follow(factor) = {‘*’, ‘+’, $, ‘)’}

N	(	**int**	*	**+**	)	**$**
program	(1)	(1)				
expr	(2)	(2)				
expr_tail				(3)	(4)	(4)
term	(5)	(5)				
term_tail			(6)	(7)	(7)	(7)
factor	(8)	(9)				

CONSTRUCTING AN LL(1) PARSING TABLE

1. *program* → *expr*
2. *expr* → *term expr_tail*
3. *expr_tail* → + *term expr_tail*
4. *expr_tail* → ϵ
5. *term* → *factor term_tail*
6. *term_tail* → * *factor term_tail*
7. *term_tail* → ϵ
8. *factor* → (*expr*)
9. *factor* → int

First(program) = {‘(‘, int}
First(expr) = {‘(‘, int}
First(expr_tail) = {‘+’, ϵ}
First(term) = {‘(‘, int}
First(term_tail) = {‘*’, ϵ}
First(factor) = {‘(‘, int}

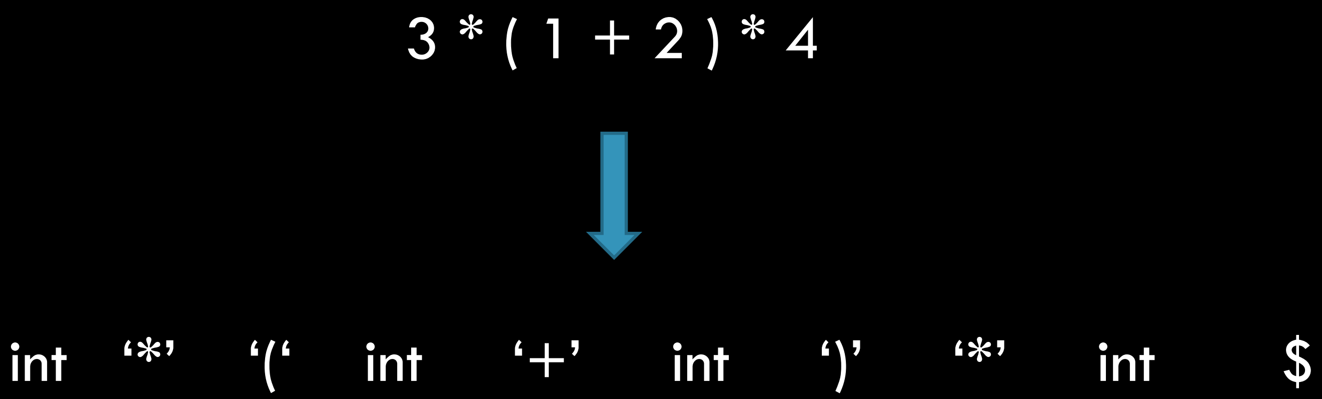
Follow(program) = {$}
Follow(expr) = {$, ‘)’}
Follow(expr_tail) = {$, ‘)’}
Follow(term) = {‘+’, $, ‘)’}
Follow(term_tail) = {‘+’, $, ‘)’}
Follow(factor) = {‘*’, ‘+’, $, ‘)’}

N	(	**int**	*	**+**	)	**$**
program	(1)	(1)	-	-	-	-
expr	(2)	(2)	-	-	-	-
expr_tail	-	-	-	(3)	(4)	(4)
term	(5)	(5)	-	-	-	-
term_tail	-	-	(6)	(7)	(7)	(7)
factor	(8)	(9)	-	-	-	-

PARSING A STRING

Let’s pick a sample string from our grammar to parse.

1. *program* → *expr*
2. *expr* → *term expr_tail*
3. *expr_tail* → + *term expr_tail*
4. *expr_tail* → ϵ
5. *term* → *factor term_tail*
6. *term_tail* → * *factor term_tail*
7. *term_tail* → ϵ
8. *factor* → (*expr*)
9. *factor* → int



3 * (1 + 2) * 4

int ‘*’ ‘(‘ int ‘+’ int ‘)’ ‘*’ int $

PARSING A STRING

1. program → expr
2. expr → term expr_tail
3. expr_tail → + term expr_tail
4. expr_tail → ϵ
5. term → factor term_tail
6. term_tail → * factor term_tail
7. term_tail → ϵ
8. factor → (expr)
9. factor → int

N	(	int	*	+	)	$
program	(1)	(1)	-	-	-	-
expr	(2)	(2)	-	-	-	-
expr_tail	-	-	-	(3)	(4)	(4)
term	(5)	(5)	-	-	-	-
term_tail	-	-	(6)	(7)	(7)	(7)
factor	(8)	(9)	-	-	-	-

Stack (bottom → top)	Input tokens	Production Used
$ program	int * (int + int) * int $	
$ expr	int * (int + int) * int $	(1)
$ expr_tail term	int * (int + int) * int $	(2)
$ expr_tail term_tail factor	int * (int + int) * int $	(5)
$ expr_tail term_tail int	int * (int + int) * int $	(9)
$ expr_tail term_tail	* (int + int) * int $	
$ expr_tail term_tail factor *	* (int + int) * int $	(6)
$ expr_tail term_tail factor	(int + int) * int $	

PARSING A STRING

1. *program* → *expr*
2. *expr* → *term expr_tail*
3. *expr_tail* → *+ term expr_tail*
4. *expr_tail* → *ϵ*
5. *term* → *factor term_tail*
6. *term_tail* → ** factor term_tail*
7. *term_tail* → ϵ
8. *factor* → (*expr*)
9. *factor* → int

N	(	int	*	+	)	$
program	(1)	(1)	-	-	-	-
expr	(2)	(2)	-	-	-	-
expr_tail	-	-	-	(3)	(4)	(4)
term	(5)	(5)	-	-	-	-
term_tail	-	-	(6)	(7)	(7)	(7)
factor	(8)	(9)	-	-	-	-

Stack (bottom → top)	Input tokens	Production Used
$ expr_tail term_tail factor	(int + int) * int $	
$ expr_tail term_tail) expr (	(int + int) * int $	(8)
$ expr_tail term_tail) expr	int + int) * int $	
$ expr_tail term_tail) expr_tail term	int + int) * int $	(2)
$ expr_tail term_tail) expr_tail term_tail factor	int + int) * int $	(5)
$ expr_tail term_tail) expr_tail term_tail int	int + int) * int $	(9)
$ expr_tail term_tail) expr_tail term_tail	+ int) * int $	
$ expr_tail term_tail) expr_tail	+ int) * int $	(7)
$ expr_tail term_tail) expr_tail term ‘+’	+ int) * int $	(3)

PARSING A STRING

1. *program* → *expr*
2. *expr* → *term expr_tail*
3. *expr_tail* → + *term expr_tail*
4. *expr_tail* → ϵ
5. *term* → *factor term_tail*
6. *term_tail* → * *factor term_tail*
7. *term_tail* → ϵ
8. *factor* → (*expr*)
9. *factor* → int

N	(	int	*	+	)	$
program	(1)	(1)	-	-	-	-
expr	(2)	(2)	-	-	-	-
expr_tail	-	-	-	(3)	(4)	(4)
term	(5)	(5)	-	-	-	-
term_tail	-	-	(6)	(7)	(7)	(7)
factor	(8)	(9)	-	-	-	-

Stack (bottom → top)	Input tokens	Production Used
$ expr_tail term_tail) expr_tail term '+'	+ int) * int $	(3)
$ expr_tail term_tail) expr_tail term	int) * int $	
$ expr_tail term_tail) expr_tail term_tail factor	int) * int $	(5)
$ expr_tail term_tail) expr_tail term_tail int	int) * int $	(9)
$ expr_tail term_tail) expr_tail term_tail	) * int $	
$ expr_tail term_tail) expr_tail	) * int $	(7)
$ expr_tail term_tail)	) * int $	(4)
$ expr_tail term_tail	* int $	

PARSING A STRING

1. *program* → *expr*
2. *expr* → *term expr_tail*
3. *expr_tail* → + *term expr_tail*
4. *expr_tail* → *ϵ*
5. *term* → *factor term_tail*
6. *term_tail* → * *factor term_tail*
7. *term_tail* → ϵ
8. *factor* → (*expr*)
9. *factor* → int

N	(	int	*	+	)	$
program	(1)	(1)	-	-	-	-
expr	(2)	(2)	-	-	-	-
expr_tail	-	-	-	(3)	(4)	(4)
term	(5)	(5)	-	-	-	-
term_tail	-	-	(6)	(7)	(7)	(7)
factor	(8)	(9)	-	-	-	-

Stack (bottom → top)	Input tokens	Production Used
$ expr_tail term_tail	* int $	
$ expr_tail term_tail factor *	* int $	(6)
$ expr_tail term_tail factor	int $	
$ expr_tail term_tail int	int $	(9)
$ expr_tail term_tail	$	
$ expr_tail	$	(7)
$	$	(4)
Accept!		

NEXT LECTURE

Review of Lexical and Syntax Analysis, then we’ll begin Semantic Analysis.