Compiler Design

Fatemeh Deldar

Isfahan University of Technology

1402-1403

LALR(1) Grammar

حالات	action				goto		
	a	b	d	$	E	T	S
0	s2		s3,10		1	4,9	13
1	r2			r2			
2		s5		r6			
3,10	s7		s3,10		6,11	4,9	
4,9		r5		r5			
5				r3			
6,11		s8,12					
7		r6					
8,12				r4			
13				accept			

پشته	رشته ورودی
0	dab$
0d3,10	ab$
0d3,10a7	b$
0d3,10a7	b$
0d3,10T	b$
0d3,10T4,9	b$
0d3,10T4,9	b$
0d3,10E	b$
0d3,10E6,11	b$
0d3,10E6,11b8,12	$
0E	$
0E1	$
0S	$
0S13 (accept)	$

1- $A \rightarrow S$
2- $S \rightarrow E$
3- $S \rightarrow ab$
4- $E \rightarrow dEb$
5- $E \rightarrow T$
6- $T \rightarrow a$

LALR(1) Grammar

Exercise 4.7.4: Show that the following grammar

$$\begin{array}{lcl} S & \rightarrow & A\ a \mid b\ A\ c \mid d\ c \mid b\ d\ a \\ A & \rightarrow & d \end{array}$$

is LALR(1) but not SLR(1).

Ambiguous Grammars

- *Every ambiguous grammar fails to be LR*
- *Every LR grammar is not ambiguous*
- **However, certain types of ambiguous grammars are quite useful in the specification and implementation of languages**
  - **We can specify dis-ambiguating rules that allow only one parse tree for each sentence**
- ***Compilers usually use precedence and associativity to resolve conflicts***

Ambiguous Grammars

- **Precedence and Associativity to Resolve Conflicts**
- **Example** $E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$
- ***This grammar is ambiguous because it does not specify the associativity or precedence of the operators + and ****
- This grammar is equivalent to the above grammar, which is not ambiguous

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

- **There are two reasons why we might prefer to use the ambiguous grammar**
  - We can easily change the associativity and precedence of the operators without disturbing the productions or the number of states in the resulting parser
  - The parser for the ambiguous grammar will not waste time reducing by single productions

Ambiguous Grammar

- Assuming + and * are left associative and * takes precedence over +

STATE	ACTION						GOTO
	id	$+$	$*$	(	)	\$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

I_0: $E' \to \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \textbf{id}$

I_1: $E' \to E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_2: $E \to (\cdot E)$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \textbf{id}$

I_3: $E \to \textbf{id} \cdot$

I_4: $E \to E + \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \textbf{id}$

I_5: $E \to E * \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \textbf{id}$

I_6: $E \to (E \cdot)$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_7: $E \to E + E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_8: $E \to E * E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_9: $E \to (E) \cdot$

Ambiguous Grammars

- **The "Dangling-Else" Ambiguity**

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{other}
\end{array}
$$

$$
\begin{array}{lcl}
S' & \rightarrow & S \\
S & \rightarrow & i\ S\ e\ S \ |\ i\ S\ |\ a
\end{array}
$$

$$
\begin{array}{ll}
I_0: & S' \rightarrow \cdot S \\
 & S \rightarrow \cdot iSeS \\
 & S \rightarrow \cdot iS \\
 & S \rightarrow \cdot a \\
I_1: & S' \rightarrow S\cdot \\
I_2: & S \rightarrow i\cdot SeS \\
 & S \rightarrow i\cdot S \\
 & S \rightarrow \cdot iSeS \\
 & S \rightarrow \cdot iS \\
 & S \rightarrow \cdot a
\end{array}
\qquad
\begin{array}{ll}
I_3: & S \rightarrow a\cdot \\
I_4: & S \rightarrow iS\cdot eS \\
 & S \rightarrow iS\cdot \\
I_5: & S \rightarrow iSe\cdot S \\
 & S \rightarrow \cdot iSeS \\
 & S \rightarrow \cdot iS \\
 & S \rightarrow \cdot a \\
I_6: & S \rightarrow iSeS\cdot
\end{array}
$$

- ***We should shift else, because it is "associated" with the previous then***

Ambiguous Grammars

- **The "Dangling-Else" Ambiguity**

STATE	ACTION				GOTO
	i	e	a	$\$$	S
0	s2		s3		1
1				acc	
2	s2		s3		4
3		r3		r3	
4		s5		r2	
5	s2		s3		6
6		r1		r1	

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		$i\,i\,a\,e\,a\,\$$	shift
(2)	0 2	i	$i\,a\,e\,a\,\$$	shift
(3)	0 2 2	$i\,i$	$a\,e\,a\,\$$	shift
(4)	0 2 2 3	$i\,i\,a$	$e\,a\,\$$	shift
(5)	0 2 2 4	$i\,i\,S$	$e\,a\,\$$	reduce by $S \to a$
(6)	0 2 2 4 5	$i\,i\,S\,e$	$a\,\$$	shift
(7)	0 2 2 4 5 3	$i\,i\,S\,e\,a$	$\$$	reduce by $S \to a$
(8)	0 2 2 4 5 6	$i\,i\,S\,e\,S$	$\$$	reduce by $S \to iSeS$
(9)	0 2 4	$i\,S$	$\$$	reduce by $S \to iS$
(10)	0 1	S	$\$$	accept

Ambiguous Grammars

Exercise 4.8.1: The following is an ambiguous grammar for expressions with n binary, infix operators, at n different levels of precedence:

$$E \rightarrow E\ \theta_1\ E \mid E\ \theta_2\ E \mid \cdots \mid E\ \theta_n\ E \mid (\ E\ ) \mid \mathbf{id}$$

a) As a function of n, what are the SLR sets of items?

b) How would you resolve the conflicts in the SLR items so that all operators are left associative, and θ_n takes precedence over θ_{n-1}, which takes precedence over θ_{n-2}, and so on?

c) Show the SLR parsing table that results from your decisions in part (b).

Error Recovery in LR Parsing

- An LR parser will detect an error when it consults the parsing **action table** and finds an error entry
  - **All empty entries in the action table are error entries**
- A **canonical LR parser (LR(1) parser)** will never make even a single reduction before announcing an error
- **The SLR and LALR parsers may make several reductions before announcing an error**
- But, all LR parsers (LR(1), LALR and SLR parsers) will never shift an erroneous input symbol onto the stack

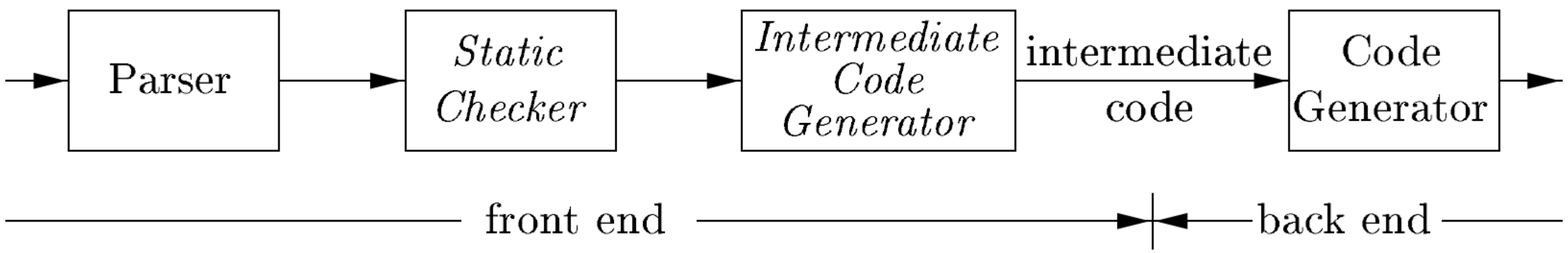
Panic Mode Error Recovery in LR Parsing

- In LR parsing, we can implement panic-mode error recovery as follows:
  - Scan down the stack until a **state $\boldsymbol{s}$** with a goto on a particular **non-terminal $\boldsymbol{A}$** is found
  - Discard zero or more input symbols until a **symbol $\boldsymbol{a}$** is found that can legitimately follow $\boldsymbol{A}$
  - The **symbol $\boldsymbol{a}$** is simply in $\boldsymbol{FOLLOW(A)}$, but this may not work for all situations
  - The parser stacks the **non-terminal $\boldsymbol{A}$** and the state $\boldsymbol{goto[s, A]}$, and it resumes the normal parsing

Intermediate-Code Generation

Intermediate-Code Generation

- $m * n$ compilers can be built by writing just m front ends and n back ends



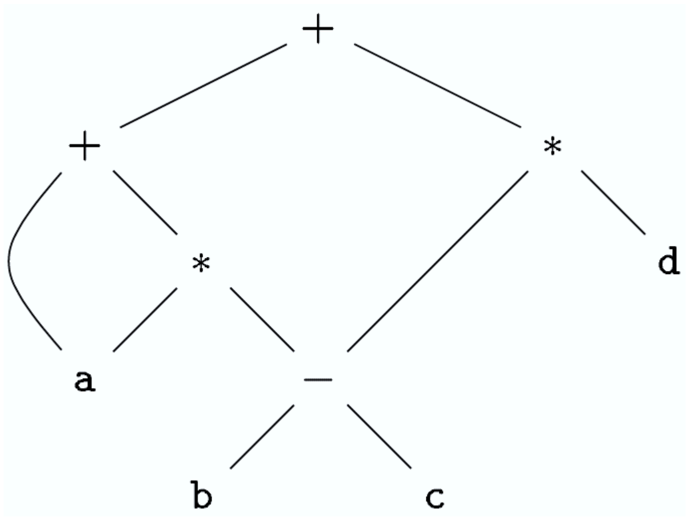
- An intermediate representation may either be an actual language
- C is a programming language, yet it is often used as an intermediate form because **it is flexible**, **it compiles into efficient machine code**, and **its compilers are widely available**
  - The original C++ compiler consisted of a front end that generated C, treating a C compiler as a back end

Directed Acyclic Graph

- A directed acyclic graph (DAG) for an expression identifies the common subexpressions of the expression

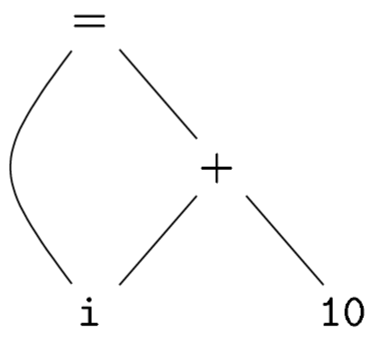
- DAGs can be constructed by using the same techniques that construct syntax trees
  - **A DAG has leaves corresponding to atomic operands and interior nodes corresponding to operators**
  - **A node N in a DAG has more than one parent if N represents a common subexpression**

- **Example**
  - $a + a * (b - c) + (b - c) * d$



The Value-Number Method for Constructing DAGs

- Often, the nodes of a syntax tree or DAG are stored in an array of records



(a) DAG

1	**id**		to entry for `i`
2	**num**	10	
3	+	1 | 2	
4	=	1 | 3	
5	...		

(b) Array.

- In this array, we refer to nodes by giving the **integer index of the record for that node within the array**, which called the **value number** for the node or for the expression represented by the node