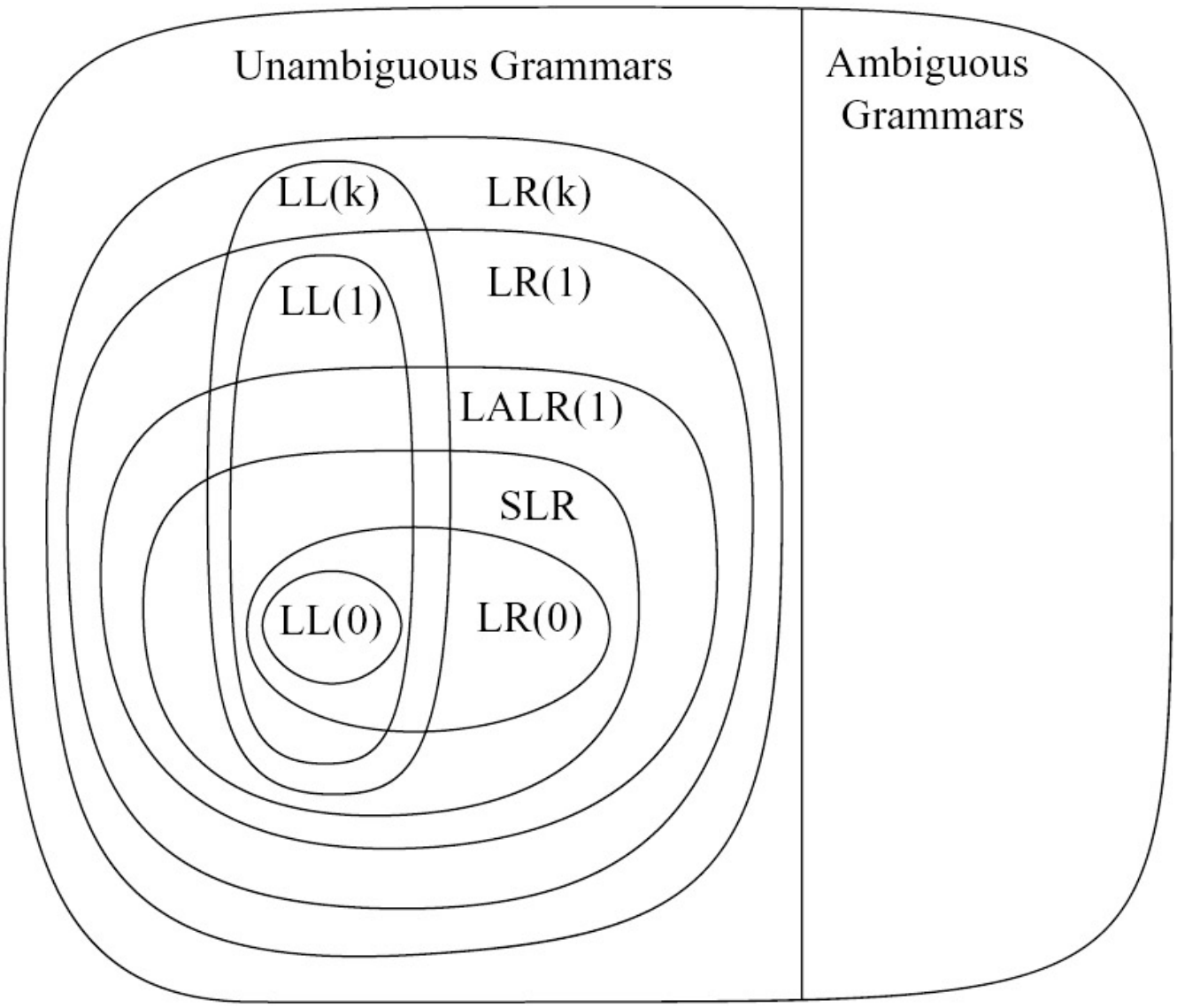
CS502: Compilers & Programming Systems

Bottom-up Parsing &
Interleaving Parsing with Semantic Actions

Zhiyuan Li
Department of Computer Science
Purdue University, USA

PURDUE
UNIVERSITY


Unambiguous Grammars
Ambiguous Grammars
LL(k)
LR(k)
LR(1)
LL(1)
LALR(1)
SLR
LL(0)
LR(0)

Bottom-Up Parsing

- With top-down parsing, it is often difficult to **predict** which production rule to apply such that the nonterminal being considered can eventually derive a string of terminals matching the input.
- With bottom-up parsing, the compiler construct the right-most derivation backward, from the input towards the start nonterminal, as if the parsing-tree is constructed bottom-up.
- Parsing actions:
  - **Shift:** scan the next terminal. This is performed when there is no new subtree ready be be built.

Hence the compiler needs to fetch one more more terminals (as the leaves in the parsing tree) in order to proceed.

  - **reduce:** insert a new internal node when all its children have been build.

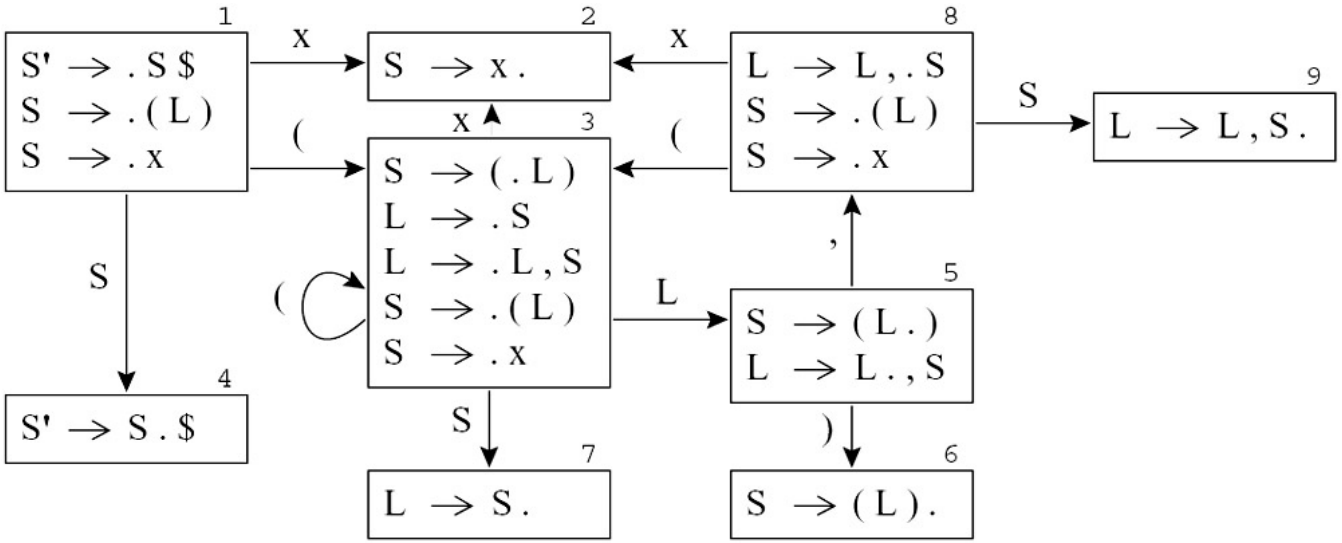
- Within the compiler, A *parsing-stack* is used during the parsing. The parsing stack can be thought as storing the prefix of the right-most sentential form. (The postfix is the rest of the sentence to be scanned.)

  A shift causes the parser to move into a new state and push that state number to the parsing stack.

  A reduce causes the parser to pop out a number of

states off the parsing stack. These states correspond to the grammar symbols in the right-hand side of a production rule. The left-hand side nonterminal symbol (as the result of the reduction) is checked against the newly exposed state number on top of the stack, using the GOTO table. The GOTO table tells the parser what the next state should be.

- We want to be able to determine the parsing action without having to look down the parsing stack. Instead, we want to determine by looking at the top of the stack (and perhaps the terminals next in the input).
- Hence, we store **states**, instead of grammar symbols in the stack. A state encodes a part of the parsing history. Bottom-up parsing based on states follows the *LR parsing method*.
- For LR(k) parsers, the next parsing action is determined by the top-of-stack state and the next k token.



This example shows how a deterministic LR(0) parser can be built. The parsing action (shift or reduce) can be determined w/o looking at the next token. However, to determine the next state, one still needs to look at the next token.

In an LR(0) parser, a state represents a set of LR(0) items. Each item describes one of the possible "con-gurations". A period "." separates the item into a prefix (history) and a postfix (a possible future).

-- Each state contains one or more initial items and the closure of such initial items

-- The state-transition diagram is implemented as a parsing table

An LR(k) parsing table has two parts:
(1) The action table
(2) the goto table.

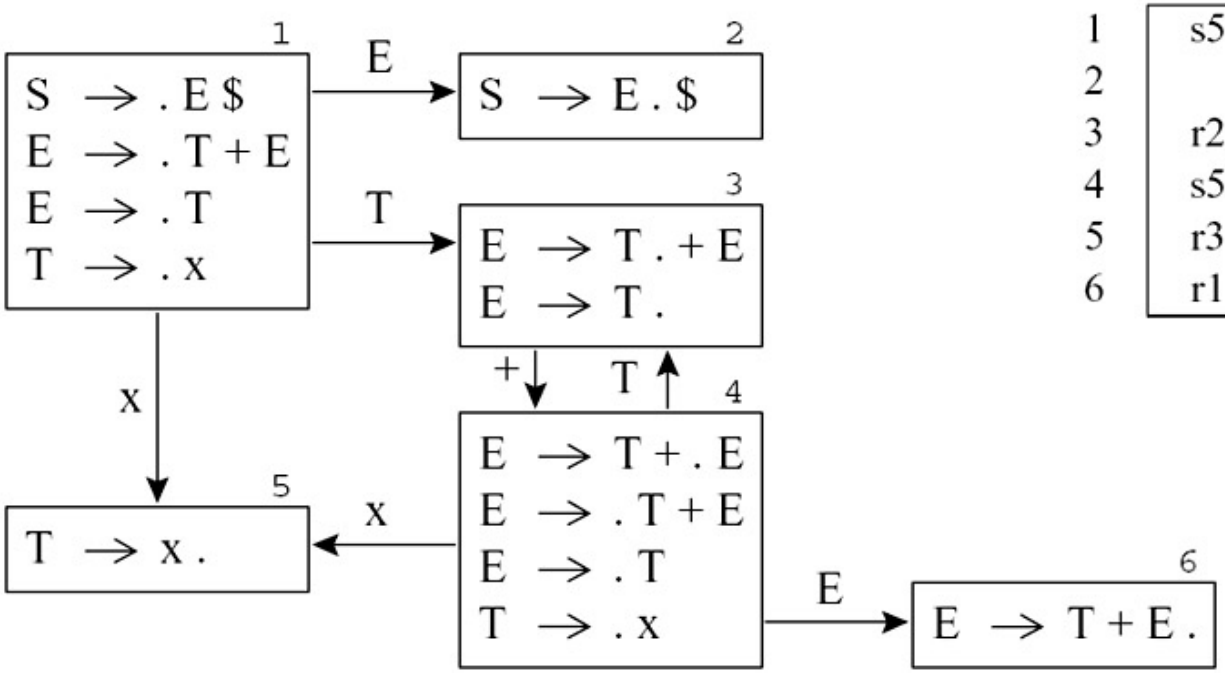
The goto table checks the top-of-stack state against the newly reduced nonterminal and determines what should be the next state to push into the parsing stack.

An LR(k) parsing table is usually much larger than LL(k) parsing table for the same CFG, because the number of rows equal to the number of possible states, instead of the number of non-terminals. In practice, it is better to let parser-generators to generate LR(k) parsing tables.

But again, we need to understand how LR(k) parsing is performed in order to write a suitable grammar.

It is important to keep k as small as possible.

LR(0) parsing is limited in its power. A simple way to enhance LR(0) parsing is to perform
SLR parsing, namely simple LR.



	x	+	$	*E*	*T*
1	s5			g2	g3
2			a		
3	r2	s4,r2	r2		
4	s5			g6	g3
5	r3	r3	r3		
6	r1	r1	r1		

	x	+	$	*E*	*T*
1	s5			g2	g3
2			a		
3		s4	r2		
4	s5			g6	g3
5		r3	r3		
6			r1		

Algorithm (Constructing a SLR(1) Parsing Table)

Given: an augmented grammar G.

Output: 1) A set of states, each associated with a set of items; 2) An action table and a goto table.

Making the Action Table

1. If state S_1 transits to S_2 on $token\ x$, then Action[S_1, x] = (shift,S_2).

2. If state S_1 contains a completed item [C $\longrightarrow \beta\bullet$], then

for every $x \in$ FOLLOW(C), Action[S_1, x] is (reduce,n),

where n is the index of the production rule $C \longrightarrow \beta$ in the grammar.

(For people who forget what FOLLOW set is, we have some notes in the end to review that.)

3. If S_1 contains the completed item $Z \longrightarrow E\bullet$, where Z is the new start nonterminal and E is the old start nonterminal, then Action[S_1,\$] = accept.

Making the Goto Table.

An Example of an SLR Table

Consider the following grammar:

```
S  --> R (rule 1)
R  --> id[R] (rule 2)
     | id (rule 3)
```


Using the method to be discussed later, we can build the parsing table as follows.

	ACTION				GOTO
	id	[	]	$	R
0	s2				1
1				Accept	
2		s3	r3	r3	
3	s2				4
4			s5		
5			r2	r2	

In the above $ means the end of the sentence.

We go through an example sentence, say `id[id[id]]`

Example of Constructing an SLR(1) Parsing Table

Using the previous example gramma.

```
S  --> R          (rule 1)
R  --> id[R]      (rule 2)
     | id         (rule 3)
```

```
I0:  S ->  .R          R  ---> I1
     R  -> .id[R]      id ---> I2
     R  -> .id
```

```
I1:  S -> R.

I2:  R  -> id.[R]                    [  ---> I3
     R  -> id.

I3:  R  -> id[.R]                    R  ---> I4
     R  -> .id[R],                   id ---> I2
     R  -> .id,

I4:  R  -> id[R.]                    ]  ---> I5

I6:  R  -> id[R].
```

To decide whether I2 presents a shift-reduce conflict, we compute the FOLLOW set of R, i.e. what tokens can possibly follow R in any sentential forms. The answer is FOLLOW(R) $= \{$ \$,] $\}$. There is no conflict.

We show how this leads to the parsing table presented earlier.

- LR(1) parsing is even more powerful than SLR. LR(1) parsing is based on LR(1) items.
- To reduce the memory requirement for storing the parsing table, many LR(1) states can be merged w/o causing parsing conflicts. The YACC (or bison) tool

uses LALR(1) parsing tables.

- Given a CFG, if an LALR(1) parsing table can be constructed w/o parsing conflicts, then the CFG is called an LALR(1) grammar.
- Rarely can one write a large LALR(1) grammar correctly in one attempt. YACC may complain about parsing conflicts, and the grammar writer then needs to (1) specify how to force a parsing action, or (2) modify the grammar so it becomes LALR(1).
- This is the main reason for understanding the construction of LALR(1) parsing table, even if one does not plan to write a parser-generator.
- LR(k) items are constructed by carrying along k

lookahead.

- We will carefully look at how to determine the single look-ahead token in LR(1) states construction.

- There exist examples of grammars which generate SLR parsing conflicts, but which can be parsed by $canonical$ LR(1) parsers without parsing conflicts.
- Consider this example [Aho, Sethi, Ullman88]:

$$
\begin{aligned}
S' &\longrightarrow \mathsf{S} \\
S &\longrightarrow L = R \\
S &\longrightarrow R \\
L &\longrightarrow *R \\
L &\longrightarrow \mathsf{id} \\
R &\longrightarrow \mathsf{L}
\end{aligned}
$$

Construction of the LR(1) Parsing Table

Given: an augmented grammar G.

Output: 1) A set of states, each associated with a set of items; 2) An action table and a goto table.

Making the Action Table

1. If state S_1 transits to S_2 on $token\ x$, then Action[S_1, x] = (shift,S_2).

2. If state S_1 contains a completed item [A $\longrightarrow \beta\bullet$, t], then Action[S_1, x] is (reduce,n), where n is the index of the production rule $A \longrightarrow \beta$ in the grammar.

3. If S_1 contains the completed item $S' \longrightarrow S\bullet$, where S' is the new start nonterminal and S the old

start nonterminal, then

Action[S_1,eof] $=$ accept.

The Goto Table is built in a similar way to the Action Table.

Constructing an LALR(1) Table

1. Identifying the $core$ of each state

2. If two states have the same core, then they are merged into one state. Two items of the same $core$ are merged into one, taking the union of their lookahead sets.

3. Modify the state transitions:

If S_1 is merged into T_1, S_2 is merged into T_2, and $S_1 \longrightarrow^X S_2$, then make $T_1 \longrightarrow^X T_2$.

4. Update the action table.

It is easy to prove that if $s_1 \longrightarrow^X s_2$ and $s_3 \longrightarrow s_4$, s_1, s_3 are merged into T_1, then s_2, s_4 must be merged into T_2. The reason: s_2 and s_4 must have the same

cores.

No new shift-reduce conflicts may result from the merger.

But new reduce-reduce conflicts may result.

Upper Bounds on the Number of States

SLR $2^{\#ofLR(0)items}$

SLR is simply LALR(1) with each lookahead set replaced by the FOLLOW set of the left-hand side nonterminal. The state construction is more straightforward, because no look-aheads are traced.

Canonical LR(1) $2^{\#ofLR(0)items} \times 2^{\#ofterminals}$

LALR(1) – same as SLR.

Exercise 1

$$
\begin{aligned}
< expr > \longrightarrow \quad & < expr > + < term > \\
& | < term > \\
< term > \longrightarrow \quad & < term > * < factor > \\
& | < factor > \\
< factor > \longrightarrow \; & id \mid const \mid (< expr >)
\end{aligned}
$$

Exercise 2

$$
\begin{aligned}
< factor > \longrightarrow \; & ID \mid const \mid (< expr >) \\
& \mid ID(< expr\_list >) \\
& \mid ID.ID
\end{aligned}
$$

Abstract Syntax Tree (AST)

- Each program unit (a function, a subroutine, or a method) can be represented by a list of executable statements.
- The operations within each executable statement can be represented by a tree whose root represents the statement itself.
- Each $leaf$ in a tree is a scalar operand which may be a scalar ID (represented by a pointer to the symbol table) or a constant (represented by the constant value or a pointer to the constant pool, which is a part of the symbol table).

- The children of a node representing a control statement may be a pointer to a list of executable statements.
  - Expressions (incl. ALU ops, references to scalars, array elements, structure fields and so on.)
  - IF statements
  - Loops
  - Function calls
  - (Others)

Semantic Actions

The compiler generates code according to the programming language's *semantic rules*, such as

- *data types* (and hence *operation types*).
- execution order.

Generally speaking, semantic actions are tasks performed by the compiler to determine semantic meanings of each syntax constructs in the program.
In modern compilers, part of the semantic actions are interleaved with the parsing steps. These semantic actions do two things:

- collect attributes of identifiers and store them in a *symbol table*.
- construct the AST and linking the IDs to the symbol table.

After parsing, additional semantic actions are performed, including

- Type checking
- Allocate memory for variables

Semantic Actions in Bottom-up Parsing

- The method of interleaving semantic actions with parsing steps is called *Syntax-Directed Translation*.
- The compiler performs semantic actions to propagate semantic attributes.
- *Semantic records* are used store semantic attributes which are needed to build the symbol table and the AST.
- We now study how semantic actions are performed in bottom-up parsing.

- In bottom-up parsing, semantic actions are always performed when a $reduction$ is performed.
- A $semantic\ stack$ is maintained during parsing to temporarily store the semantic records. Each semantic record in the stack corresponds to a state in the parsing stack (therefore it matches a grammar symbol).

Consider the following example:

$< var\_decl > \longrightarrow$ int $< init\_id\_list >$;
float $< init\_id\_list >$;

$< init\_id\_list > \longrightarrow$ $< init\_id >$
$| < init\_id\_list >, < init\_id >$

$< init\_id > \longrightarrow$ ID = const
| ID

Note that here we changed $< init\_id > \longrightarrow ID =$ $< relop\_expr >$ *to* $< init\_id > \longrightarrow ID = const$*, for simpler illustration.*

Let us draw the parsing tree for "int a = 100, b, c;" and let us mark the order of the reductions.

In our example, each ID is annotated by its value if

initialized.

Each nonterminal, $< init\_id >$ or $< init\_id\_list >$ is annotated by a $list$ of IDs.)

Each ID is represented by a node which has ID.lexeme and ID.value.

Now let's see how the $semantic\ stack$ is maintained during LR(1) parsing.

Finally, let's consider how to write the semantic actions for each grammar rule.

$< var\_decl > \longrightarrow$ int $< init\_id\_list >$;

{For each ID in $< init\_id\_list >$.list,
if an entry exists in symtab at current
$scope\text{-}level$, report an error.
If ID.value is defined but is not of integer
type, report an error.
Otherwise, $new\_entry =$
$insert(ID.lexeme,\ symtab)$,
$new\_entry \rightarrow type :=$ int,
$new\_entry \rightarrow value :=$ ID.value. }

float $< init\_id\_list >$;

$< init\_id\_list > \longrightarrow$ $< init\_id >$

$\{< init\_id\_list >.\text{list} := < init\_id >.\text{list}.\}$
$|< init\_id\_list >, < init\_id >$

{Append $< init\_id >$.list to
$< init\_id\_list >$.list. }

$< init\_id > \longrightarrow$ ID = const

$\{< init\_id >.\text{list} :=$
make_node(ID.lexeme, value(const)); }
| ID
$\{< init\_id >.\text{list} :=$
make_node(ID.lexeme, undefined); }

- In parser-generator tools, such as YACC, a compiler writer embeds semantic actions in the production rules.
- A YACC-generated parser automatically maintains a semantic stack.
- The compiler writer, however, must define the type, YYSTYPE, of the semantic records. (Its default type is integer in yacc.)
- Normally, YYSTYPE is defined as a pointer to a structure (which stores the semantic attributes.) (For a detailed example, see YACC Example 3 on the class web site.)

- In the YACC source code, within each semantic action code segment, the semantic record associated with each $right\text{-}hand\ side$ grammar symbol is referred to as \$1, \$2, and so on. These are of the type YYSTYPE.
- The semantic record of the left-hand side nonterminal is referred to as \$\$. (Also of type YYSTYPE.)
- It is the compiler writer's responsibility to specify, in the semantic actions, how to assign values to \$\$.
- When the parser $shifts$ a $token$, the $yylval$ of the token is pushed to the stack automatically.
- The compiler writer must declare the global variable

$yylval$ to be of the YYSTYPE type.

- The compiler writer is responsible to assign a semantic record to $yylval$ in the lex source code, as a semantic action performed when a token is recognized.