Syntax Analysis – Part VI

(LR(1) Parsing)

Yongjun Park

Hanyang University

From Last Time: LR(1) Parsing

- **Get as much as possible out of 1 lookahead symbol parsing table**
- **LR(1) grammar = recognizable by a shift/reduce parser with 1 lookahead**
- **LR(1) parsing uses similar concepts as LR(0)**
  - Parser states = set of items
  - LR(1) item = LR(0) item + lookahead symbol possibly following production
    - LR(0) item: $S \rightarrow . S + E$
    - LR(1) item: $S \rightarrow . S + E \underline{, +}$
    - Lookahead only has impact upon REDUCE operations, apply when lookahead = next input

LR(1) States

- **LR(1) state = set of LR(1) items**
- **LR(1) item = ($X \rightarrow \alpha \,.\, \beta$, y)**
  - Meaning: α already matched at top of the stack, next expect to see β y
- **Shorthand notation**
  - ($X \rightarrow \alpha \,.\, \beta$, {x1, ..., xn})
  - means:
    - ($X \rightarrow \alpha \,.\, \beta$, x1)
    - . . .
    - ($X \rightarrow \alpha \,.\, \beta$, xn)
- **Need to extend closure and goto operations**

$S \rightarrow S \,.\, + E$	+,$
$S \rightarrow S + \,.\, E$	num

LR(1) Closure

- **LR(1) closure operation:**
  - Start with Closure(S) = S
  - For each item in S:
    - $X \rightarrow \alpha \,.\, Y \beta$, z
    - and for each production $Y \rightarrow \gamma$, add the following item to the closure of S: $Y \rightarrow .\, \gamma$, FIRST(βz)
  - Repeat until nothing changes
- **Similar to LR(0) closure, but also keeps track of lookahead symbol**

LR(1) Start State

- **Initial state: start with (S' → . S , $), then apply closure operation**
- **Example: sum grammar**

S' → S $
S → E + S | E
E → num

S' → . S , $

closure

S' → . S , $
S → . E + S , $
S → . E , $
E → . num , +,$

LR(1) Goto Operation

- **LR(1) goto operation = describes transitions between LR(1) states**
- **Algorithm: for a state S and a symbol Y (as before)**
  - If the item [X → α . Y β] is in I, then
  - Goto(I, Y) = Closure([X → α Y . β])

S1

S → E . + S , $
S → E . , $

Goto(S1, ‘+’)

S2

Closure({S → E + . S , $})

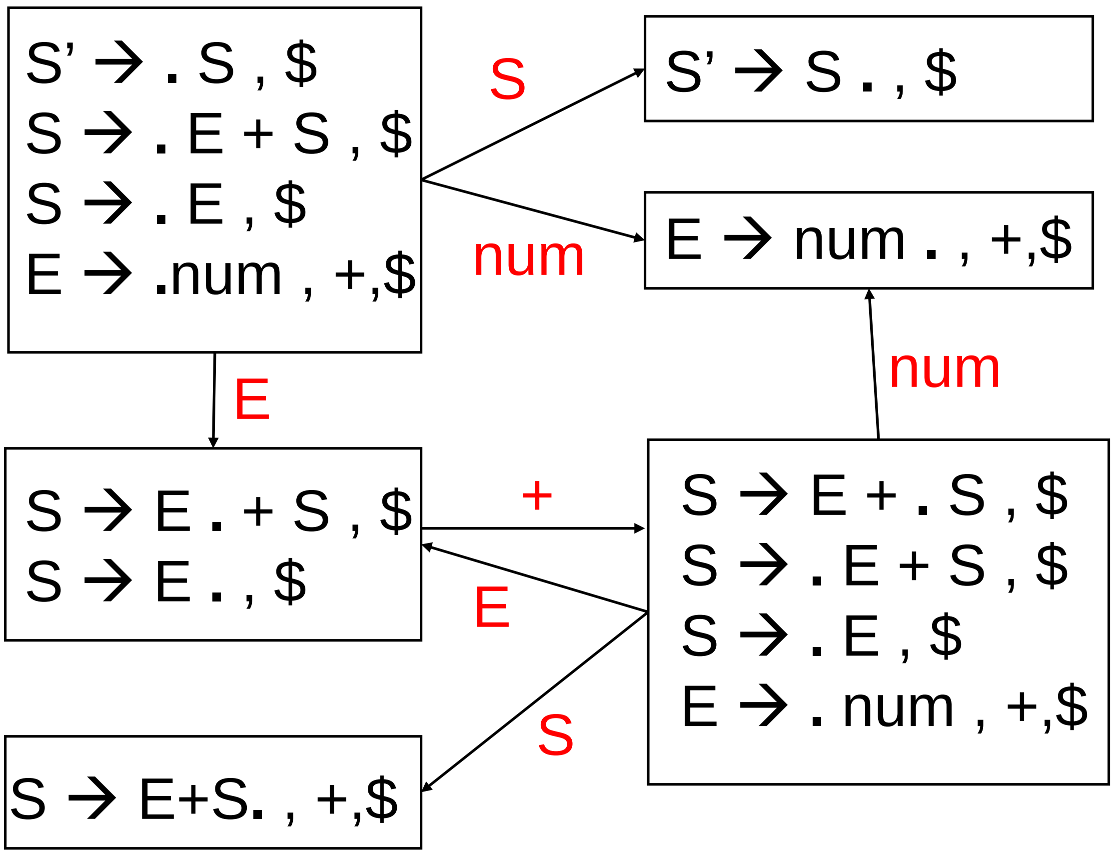
Grammar:
S’ → S$
S → E + S | E
E → num

Class Problem

1. Compute: Closure(I = {S → E + . S , $})
2. Compute: Goto(I, num)
3. Compute: Goto(I, E)

S' → S $
S → E + S | E
E → num

LR(1) DFA Construction



Grammar

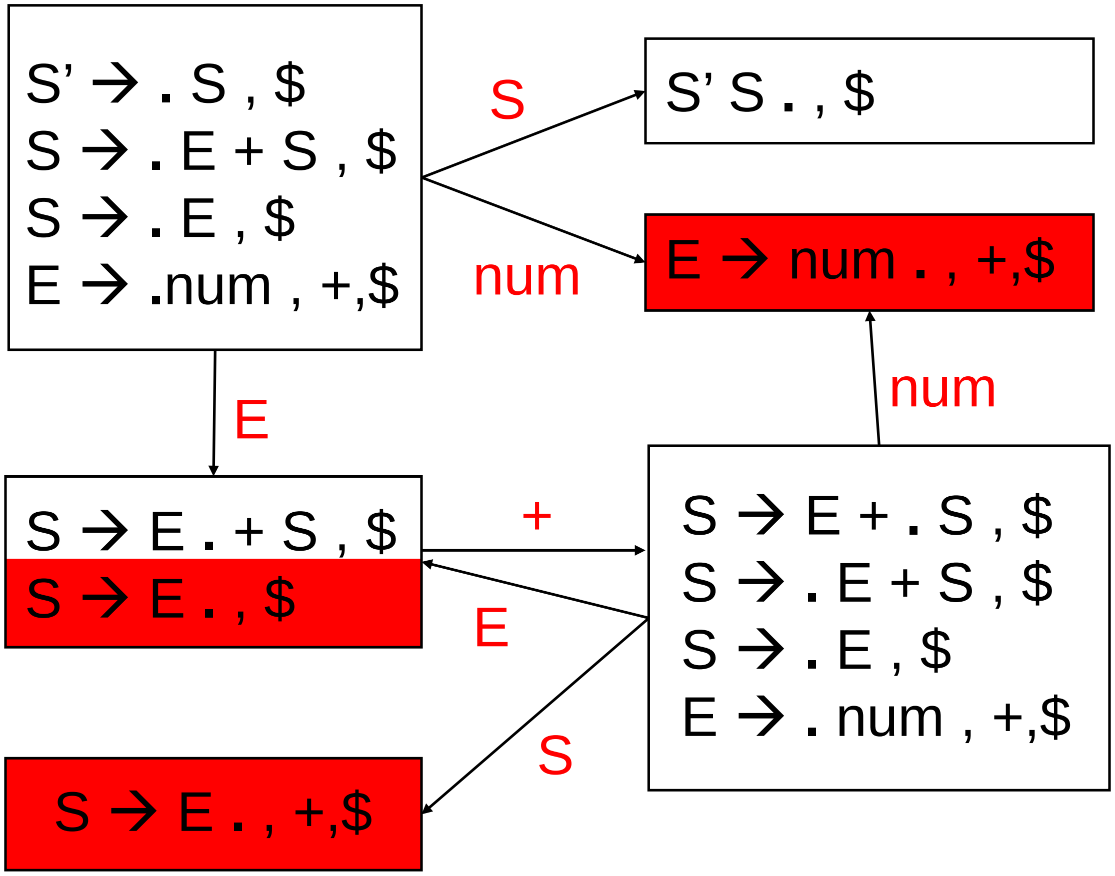
S' → S$

S → E + S | E

E → num

LR(1) Reductions

- Reductions correspond to LR(1) items of the form (X → γ . , y)

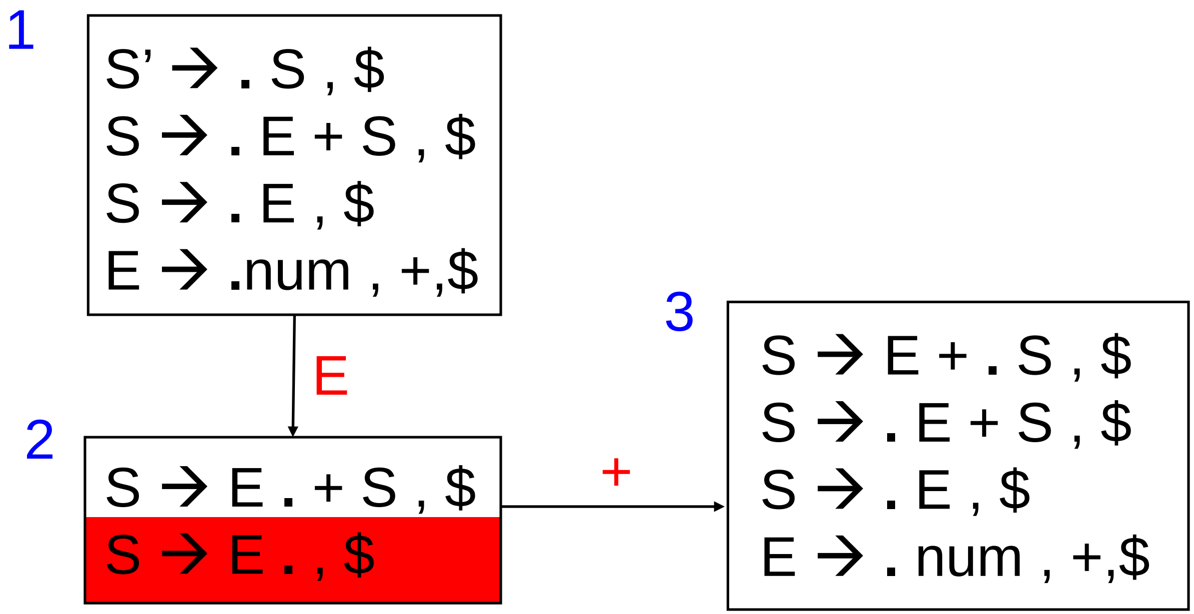


<u>Grammar</u>
S' → S$
S → E + S | E
E → num

LR(1) Parsing Table Construction

- **Same as construction of LR(0), except for reductions**
- **For a transition S → S' on terminal x:**
  - Table[S,x] += Shift(S')
- **For a transition S → S' on non-terminal N:**
  - Table[S,N] += Goto(S')
- **If I contains {(X → γ . , y)} then:**
  - Table[I,y] += Reduce(X → γ)

LR(1) Parsing Table Example



Grammar
S' → S$
S → E + S | E
E → num

Fragment of the parsing table

	+	$	E
1			g2
2	s3	S→E	

Class Problem

Compute the LR(1) DFA for the following grammar

- E → E + T | T
- T → TF | F
- F → F* | a | b

LALR(1) Grammars

- **Problem with LR(1): too many states**
- **LALR(1) parsing (aka LookAhead LR)**
  - Constructs LR(1) DFA and then merge any 2 LR(1) states whose items are identical except lookahead
  - Results in smaller parser tables
  - Theoretically less powerful than LR(1)

$$\boxed{\begin{array}{l} S \rightarrow id\ .\ ,\ + \\ S \rightarrow E\ .\ ,\ \$ \end{array}} + \boxed{\begin{array}{l} S \rightarrow id\ .\ ,\ \$ \\ S \rightarrow E\ .\ ,\ + \end{array}} = \quad ??$$

- **LALR(1) grammar = a grammar whose LALR(1) parsing table has no conflicts**

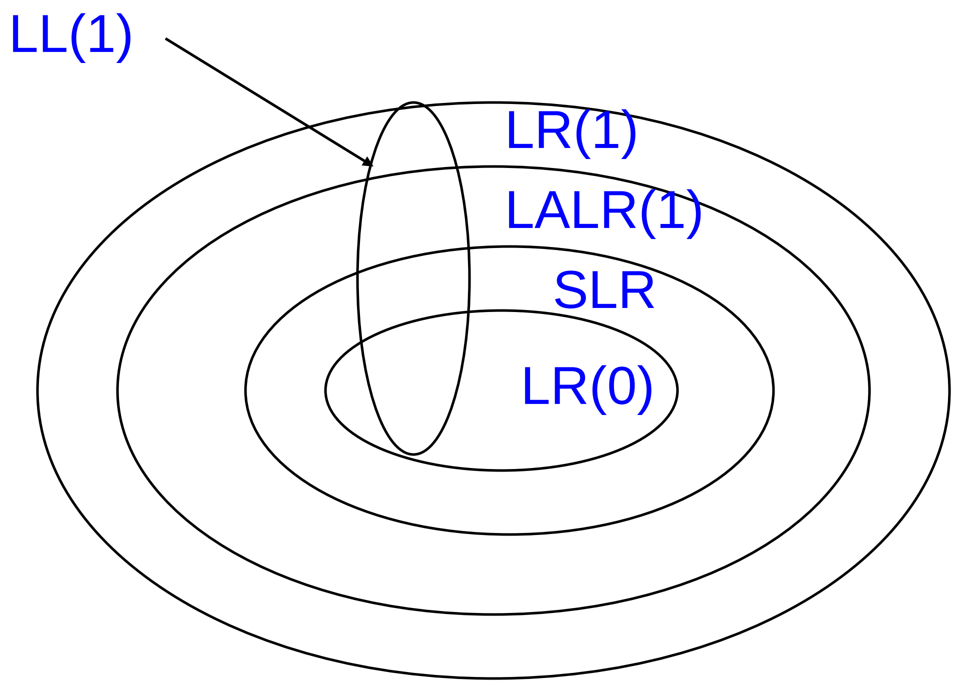
LALR Parsers

- **LALR(1)**
  - Generally same number of states as SLR (much less than LR(1))
  - But, with same lookahead capability of LR(1) (much better than SLR)
  - Example: Pascal programming language
    - In SLR, several hundred states
    - In LR(1), several thousand states

LL/LR Grammar Summary

- **LL parsing tables**
  - Non-terminals x terminals → productions
  - Computed using FIRST/FOLLOW
- **LR parsing tables**
  - LR states x terminals → {shift/reduce}
  - LR states x non-terminals → goto
  - Computed using closure/goto operations on LR states
- **A grammar is:**
  - LL(1) if its LL(1) parsing table has no conflicts
  - same for LR(0), SLR, LALR(1), LR(1)

Classification of Grammars



not to scale ☺

LR(k) $\subseteq$ LR(k+1)
LL(k) $\subseteq$ LL(k+0)

LL(k) $\subseteq$ LR(k)
LR(0) $\subseteq$ SLR
LALR(1) $\subseteq$ LR(1)

Automate the Parsing Process

- **Can automate:**
  - The construction of LR parsing tables
  - The construction of shift-reduce parsers based on these parsing tables
- **LALR(1) parser generators**
  - yacc, bison
  - Not much difference compared to LR(1) in practice
  - Smaller parsing tables than LR(1)
  - Augment LALR(1) grammar specification with declarations of precedence, associativity
  - Output: LALR(1) parser program

Associativity

S → S + E | E
E → num

→

E → E + E
E → num

What happens if we run this grammar through LALR construction?

E → E + E
E → num

$\boxed{\begin{array}{l} E \rightarrow E + E\,.\,,\ + \\ E \rightarrow E\,.\,+ E\ ,\ +,\$ \end{array}} \xrightarrow{+}$

1 + 2 + 3

↑

shift/reduce
conflict

shift: 1+ (2+3)
reduce: (1+2)+3

Associativity (2)

- **If an operator is left associative**
  - Assign a slightly higher value to its precedence if it is on the parse stack than if it is in the input stream
  - Since stack precedence is higher, reduce will take priority (which is correct for left associative)
- **If operator is right associative**
  - Assign a slightly higher value if it is in the input stream
  - Since input stream is higher, shift will take priority (which is correct for right associative)

Precedence

E → E + E | T
T → T x T | num | (E)

E → E + E | E x E | num | (E)

What happens if we run this grammar through LALR construction?

E → E . + E , ...
E → E x E . , +

E → E + E . , x
E → E . x E, ...

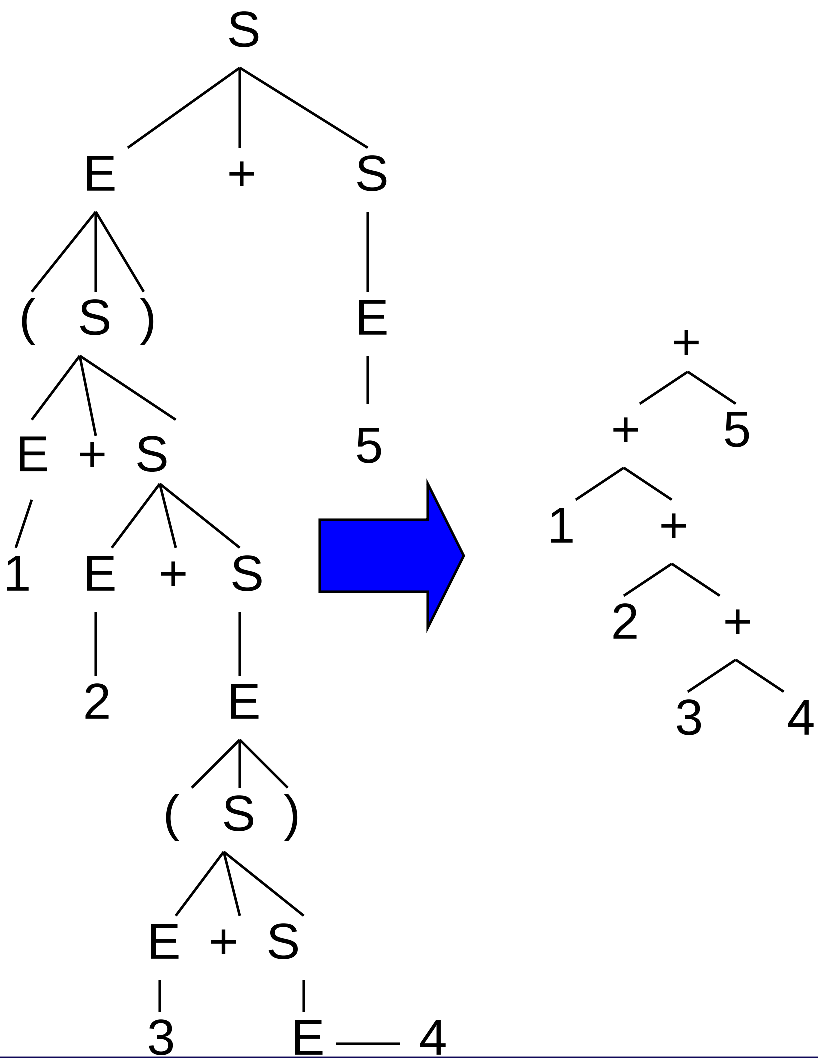
Shift/reduce conflict results

Precedence: attach precedence indicators to terminals
Shift/reduce conflict resolved by:

1. If precedence of the input token is greater than the last terminal on parse stack, favor shift over reduce
2. If the precedence of the input token is less than or equal to the last terminal on the parse stack, favor reduce over shift

Abstract Syntax Tree (AST) - Review

- **Derivation = sequence of applied productions**
  - S → E+S → 1+S → 1+E →1+2
- **Parse tree = graph representation of a derivation**
  - Doesn't capture the order of applying the productions
- **AST discards unnecessary information from the parse tree**



Implicit AST Construction

- **LL/LR parsing techniques implicitly build AST**
- **The parse tree is captured in the derivation**
  - LL parsing: AST represented by applied productions
  - LR parsing: AST represented by applied reductions
- **We want to explicitly construct the AST during the parsing phase**

AST Construction - LL

LL parsing: extend procedures for non-terminals

S → ES'
S' → ε | +S
E → num | (S)

```
void parse_S() {
    switch (token) {
        case num: case '(':
            parse_E();
            parse_S'();
            return;
        default:
            ParseError();
    }
}
```

→

```
Expr parse_S() {
    switch (token) {
        case num: case '(':
            Expr left = parse_E();
            Expr right = parse_S'();
            if (right == NULL) return left
            else return new Add(left,right);
        default:
            ParseError();
    }
}
```

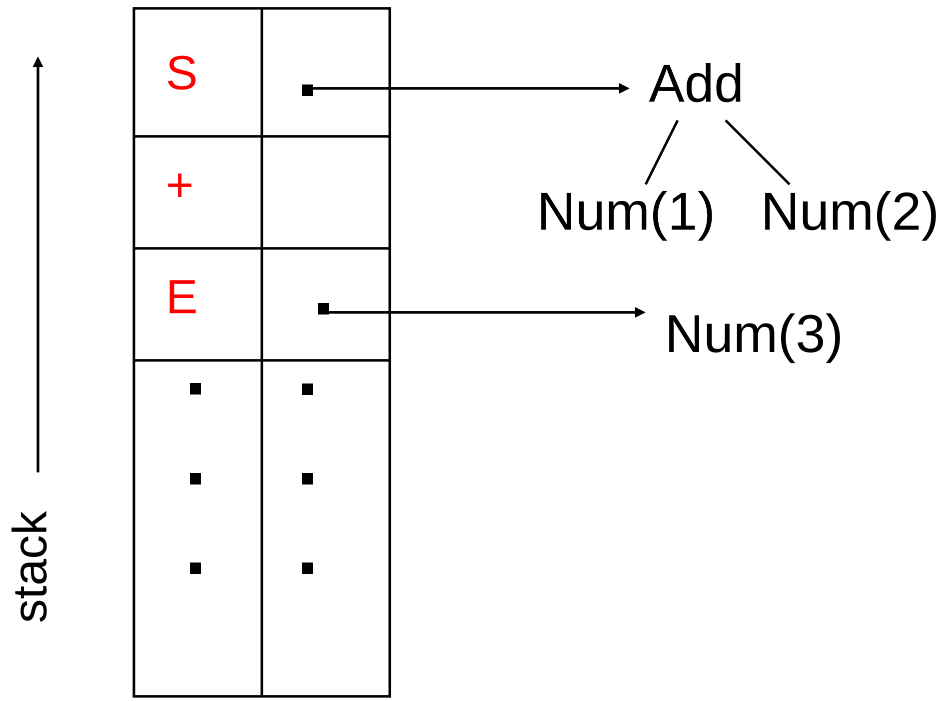
AST Construction - LR

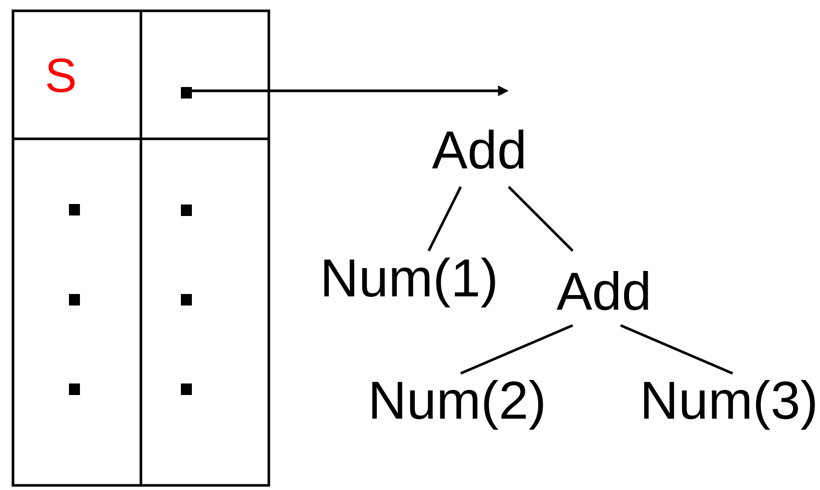
- **We again need to add code for explicit AST construction**
- **AST construction mechanism**
  - Store parts of the tree on the stack
  - For each nonterminal symbol X on stack, also store the sub-tree rooted at X on stack
  - Whenever the parser performs a reduce operation for a production $X \rightarrow \gamma$, create an AST node for X

AST Construction for LR - Example

S → E + S | S
E → num | (S)

input string: “1 + 2 + 3”





Before reduction: S → E + S

After reduction: S → E + S

Problems

- **Unstructured code: mixing parsing code with AST construction code**
- **Automatic parser generators**
  - The generated parser needs to contain AST construction code
  - How to construct a customized AST data structure using an automatic parser generator?
- **May want to perform other actions concurrently with parsing phase**
  - E.g., semantic checks
  - This can reduce the number of compiler passes

Syntax-Directed Definition

- **Solution: Syntax-directed definition**
  - Extends each grammar production with an associated semantic action (code):
    - S → E + S {action}
  - The parser generator adds these actions into the generated parser
  - Each action is executed when the corresponding production is reduced

Semantic Actions

- **Actions = C code (for bison/yacc)**
- **The actions access the parser stack**
  - Parser generators extend the stack of symbols with entries for user-defined structures (e.g., parse trees)
- **The action code should be able to refer to the grammar symbols in the productions**
  - Need to refer to multiple occurrences of the same non-terminal symbol, distinguish RHS vs LHS occurrence
    - E → E + E
  - Use dollar variables in yacc/bison ($$, $1, $2, etc.)
    - expr ::= expr PLUS expr {$$ = $1 + $3;}

Building the AST

- **Use semantic actions to build the AST**
- **AST is built bottom-up along with parsing**

Recall: User-defined type for objects on the stack (%union)

```
expr ::= NUM                  {$$ = new Num($1.val); }
expr ::= expr PLUS expr       {$$ = new Add($1, $3); }
expr ::= expr MULT expr       {$$ = new Mul($1, $3); }
expr ::= LPAR expr RPAR       {$$ = $2; }
```