CS 4120
Introduction to Compilers

Andrew Myers
Cornell University

Lecture 7: LR parsing and parser generators

11 Sep 09

Shift-reduce parsing

$$S \rightarrow S + E \mid E$$
$$E \rightarrow \text{number} \mid (S)$$

derivation	stack	input stream	action
(1+2+(3+4))+5 ←		(1+2+(3+4))+5	*shift*
(1+2+(3+4))+5 ←	(	1+2+(3+4))+5	*shift*
(1+2+(3+4))+5 ←	(1	+2+(3+4))+5	*reduce* $E \rightarrow$ num
(*E*+2+(3+4))+5 ←	(E	+2+(3+4))+5	*reduce* $S \rightarrow E$
(S+2+(3+4))+5 ←	(S	+2+(3+4))+5	*shift*
(S+2+(3+4))+5 ←	(S+	2+(3+4))+5	*shift*
(S+2+(3+4))+5 ←	(S+2	+(3+4))+5	*reduce* $E \rightarrow$ num
(S+*E*+(3+4))+5 ←	(S+E	+(3+4))+5	*reduce* $S \rightarrow S+E$
(S+(3+4))+5 ←	(S	+(3+4))+5	*shift*
(S+(3+4))+5 ←	(S+	(3+4))+5	*shift*
(S+(3+4))+5 ←	(S+(	3+4))+5	*shift*
(S+(3+4))+5 ←	(S+(3	+4))+5	*reduce* $E \rightarrow$ num

LR(0) states

- A state is a set of *items* keeping track of progress on possible upcoming reductions
- An *LR(0) item* is a production from the language with a separator "**.**" somewhere in the RHS of the production

state

$$E \rightarrow \textbf{num}\ .$$

item

$$E \rightarrow (\ .\, S\ )$$

- Stuff before "**.**" is already on stack (beginnings of possible γ's to be reduced)
- Stuff after "**.**" : what we might see next
- The prefixes α represented by state itself

LR(k) parsing

- As much power as possible out of parsing table with k look-ahead symbols
- LR(1) grammar = recognizable by a shift/reduce parser with 1 look-ahead.
- LR(1) item = LR(0) item + look-ahead symbols possibly following production

LR(0): $\boxed{S \rightarrow \,.\, S + E}$

LR(1): $\boxed{S \rightarrow \,.\, S + E \qquad +}$

Remaining input will reduce to
S + E + ...

LR(1) state

- LR(1) state = set of LR(1) items
- LR(1) item = LR(0) item + set of look-ahead symbols
- No two items in state have same production + dot configuration

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

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

LR(1) closure

Consider closure of item $\boxed{A \rightarrow \beta \,.\, C\,\delta \quad \lambda}$

Closure formed just as for LR(0) *except*

1. Lookahead symbols include characters following the non-terminal symbol to the right of dot: FIRST(δ)
2. If non-terminal symbol may produce last symbol of production (δ is nullable), lookahead symbols include lookahead symbols of production (λ)

$$
\boxed{\begin{array}{lll}
S' \rightarrow & .\, S & \$ \\
S \rightarrow & .\, E + S & \$ \\
S \rightarrow & .\, E & \$ \\
E \rightarrow & .\, \mathrm{num} & +,\$ \\
E \rightarrow & .\, (\, S\,) & +,\$
\end{array}}
$$

1

2

$$
\begin{array}{l}
S \rightarrow E + S \mid E \\
E \rightarrow \mathrm{num} \mid (\, S\,)
\end{array}
$$

LR(1) construction

$$S \rightarrow E + S \mid E$$
$$E \rightarrow \text{num} \mid (S)$$

1

$S' \rightarrow . S$	\$
$S \rightarrow . E + S$	\$
$S \rightarrow . E$	\$
$E \rightarrow . \text{num}$	+,\$
$E \rightarrow . (S)$	+,\$

$1 \xrightarrow{E} 2$

2

$S \rightarrow E . + S$	\$
$S \rightarrow E .$	\$

Know what to do if:
- reduce look-aheads distinct
- not to right of any dot

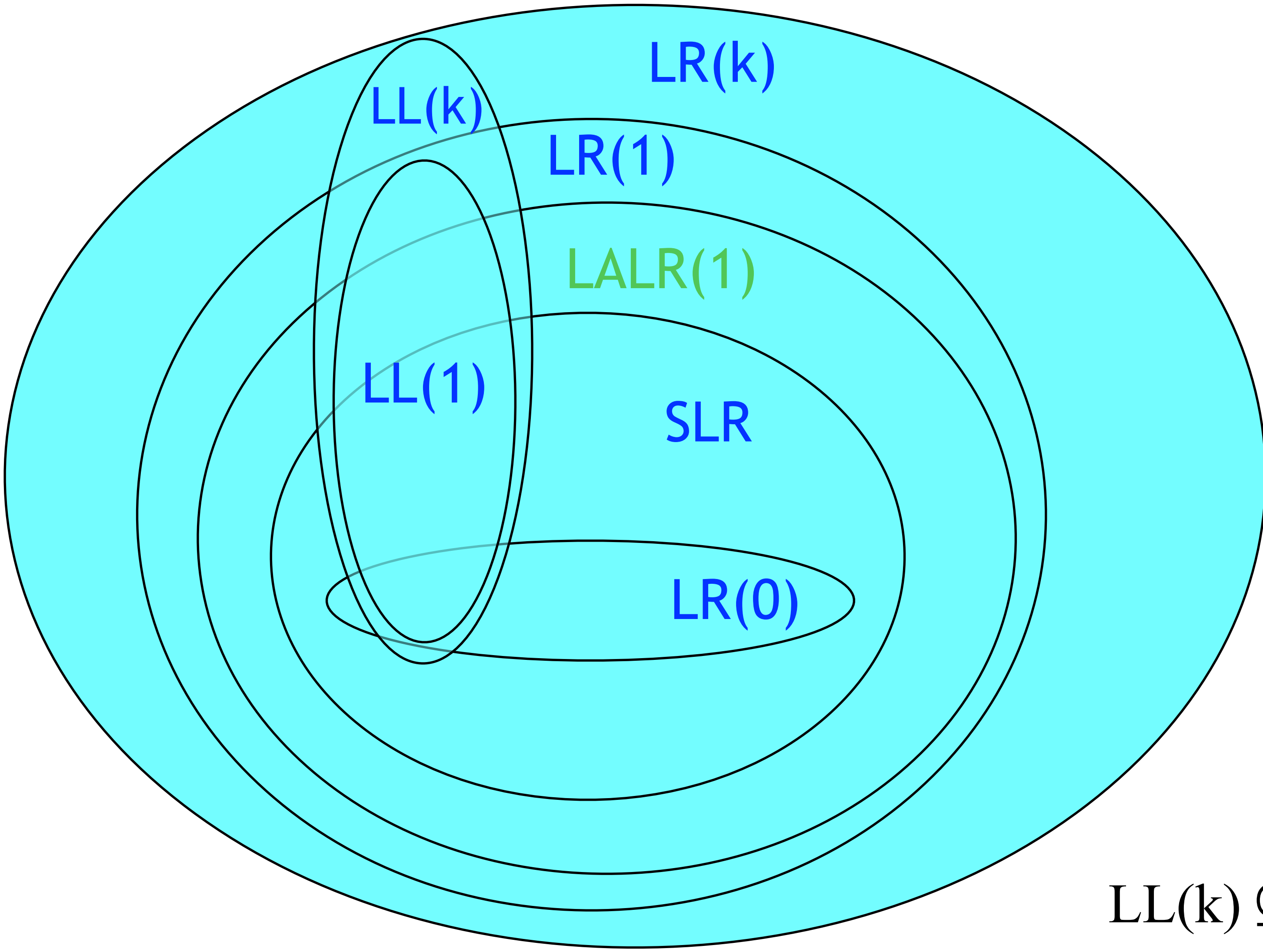
	+	\$	E
1			2
2	s3	$S \rightarrow E$	

LALR grammars

- Problem with LR(1): too many states
- LALR(1) (Look-Ahead LR)
  - Merge any two LR(1) states whose items are identical except for look-ahead
  - Results in smaller parser tables—works extremely well in practice
  - The usual technology for automatic parser generators

$$\boxed{\begin{array}{ll} S \rightarrow id\,. & + \\ S \rightarrow E\,. & \$ \end{array}} + \boxed{\begin{array}{ll} S \rightarrow id\,. & \$ \\ S \rightarrow E\,. & + \end{array}} = \;?$$

Classification of Grammars



$LL(k) \subseteq LR(k)$

How are parsers written?

- Automatic parser generators: yacc, bison, CUP
- Accept LALR(1) grammar specification
  - plus: declarations of precedence, associativity
  - output: LR parser code (inc. parsing table)
- Some parser generators accept LL(1), e.g. javacc – less powerful, or LL(k), e.g. ANTLR
- Rest of this lecture: how to use parser generators
- Can we use parsers for programs other than compilers?

Associativity

$$S \rightarrow S + E \mid E$$
$$E \rightarrow \text{num} \mid (\, S \,)$$

$$\Downarrow$$

$$E \rightarrow E + E \mid \text{num} \mid (\, E \,)$$

What happens if we run this grammar through LALR construction?

Conflict!

$$E \rightarrow E + E \;\mid\; \text{num} \;\mid\; (\,E\,)$$

$$\boxed{\begin{array}{ll} E \rightarrow E + E\,. & + \\ E \rightarrow E\,. + E & +, \$ \end{array}}$$

shift/reduce conflict

1+2+3
^

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

Grammar in CUP

non terminal E; terminal PLUS, LPAREN...
precedence left PLUS;

"When shifting + conflicts with reducing a production containing +, choose reduce"

```
E ::= E  PLUS  E
    |   LPAREN  E  RPAREN
    |   NUMBER ;
```

Precedence

- Also can handle operator precedence

$$E \rightarrow E + E \;\mid\; T$$

$$T \rightarrow T \times T \;\mid\; \mathsf{num} \;\mid\; (\,E\,)$$

$$\Downarrow$$

$$E \rightarrow E + E \;\mid\; E \times E$$

$$\mid\; \mathsf{num} \;\mid\; (\,E\,)$$

Conflicts w/o precedence

$$E \rightarrow E + E \;\mid\; E \times E$$
$$\mid\; \text{num} \;\mid\; (\, E \,)$$

$$\boxed{\begin{array}{l} E \rightarrow E \,\bullet + E \ldots \\ E \rightarrow E \times E \,\bullet\; + \end{array}}$$

$$\boxed{\begin{array}{l} E \rightarrow E + E \,\bullet\; \times \\ E \rightarrow E \,\bullet \times E \ldots \end{array}}$$

Predecence in CUP

```
precedence left PLUS;
precedence left TIMES; // TIMES > PLUS
E ::= E PLUS E  |  E TIMES E  |  ...
```

$$E \rightarrow E \,.\, + E \ldots$$
$$E \rightarrow E \times E \,.\, +$$

$$E \rightarrow E + E \,.\, \times$$
$$E \rightarrow E \,.\, \times E \ldots$$

Rule: in conflict, choose **reduce** if production symbol higher precedence than shifted symbol; choose **shift** if vice-versa

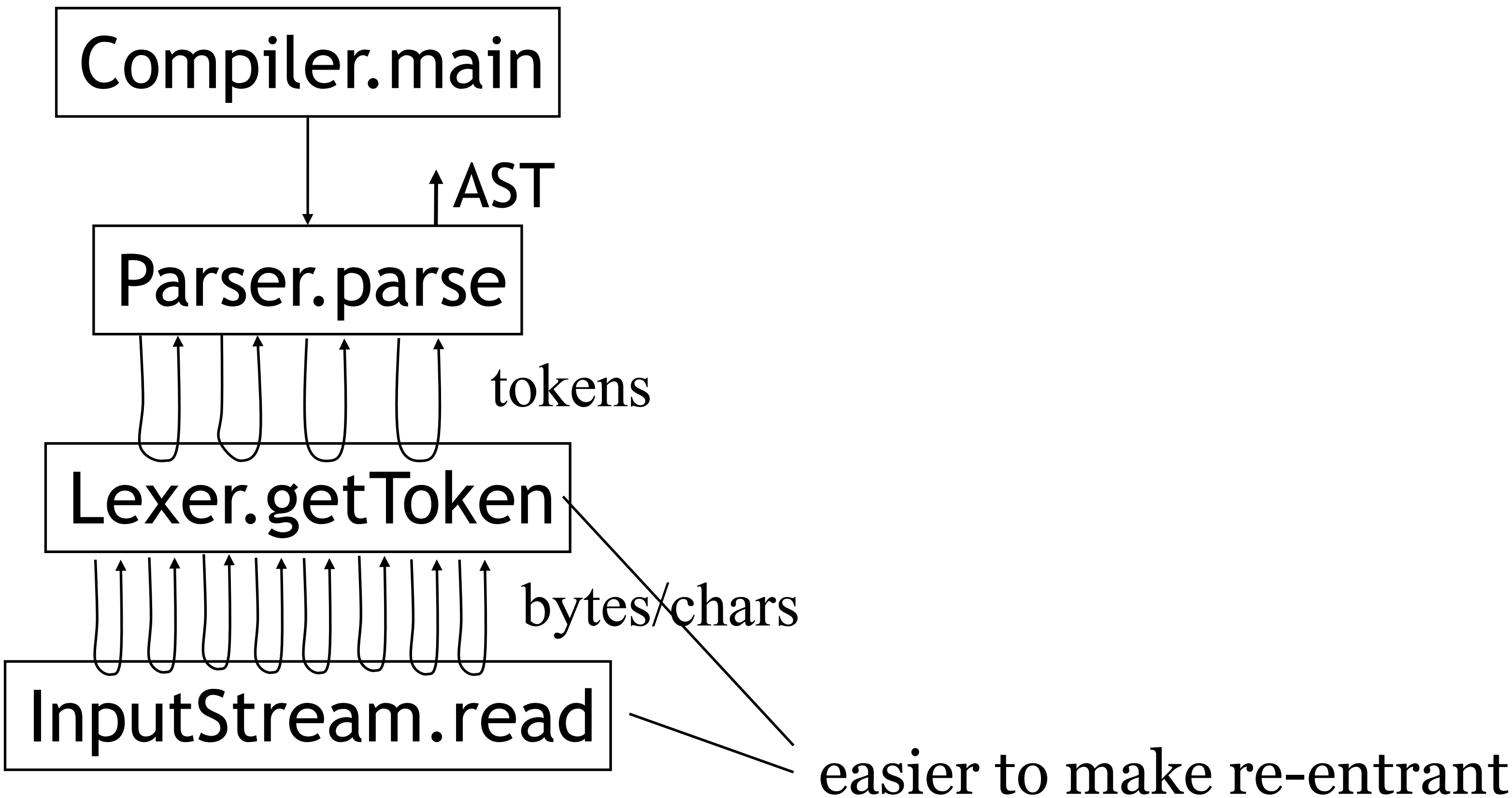
Summary

- Look-ahead information makes SLR(1), LALR(1), LR(1) grammars expressive
- Automatic parser generators support LALR(1)
- Precedence, associativity declarations simplify grammar writing
- Easiest and best way to read structured human-readable input

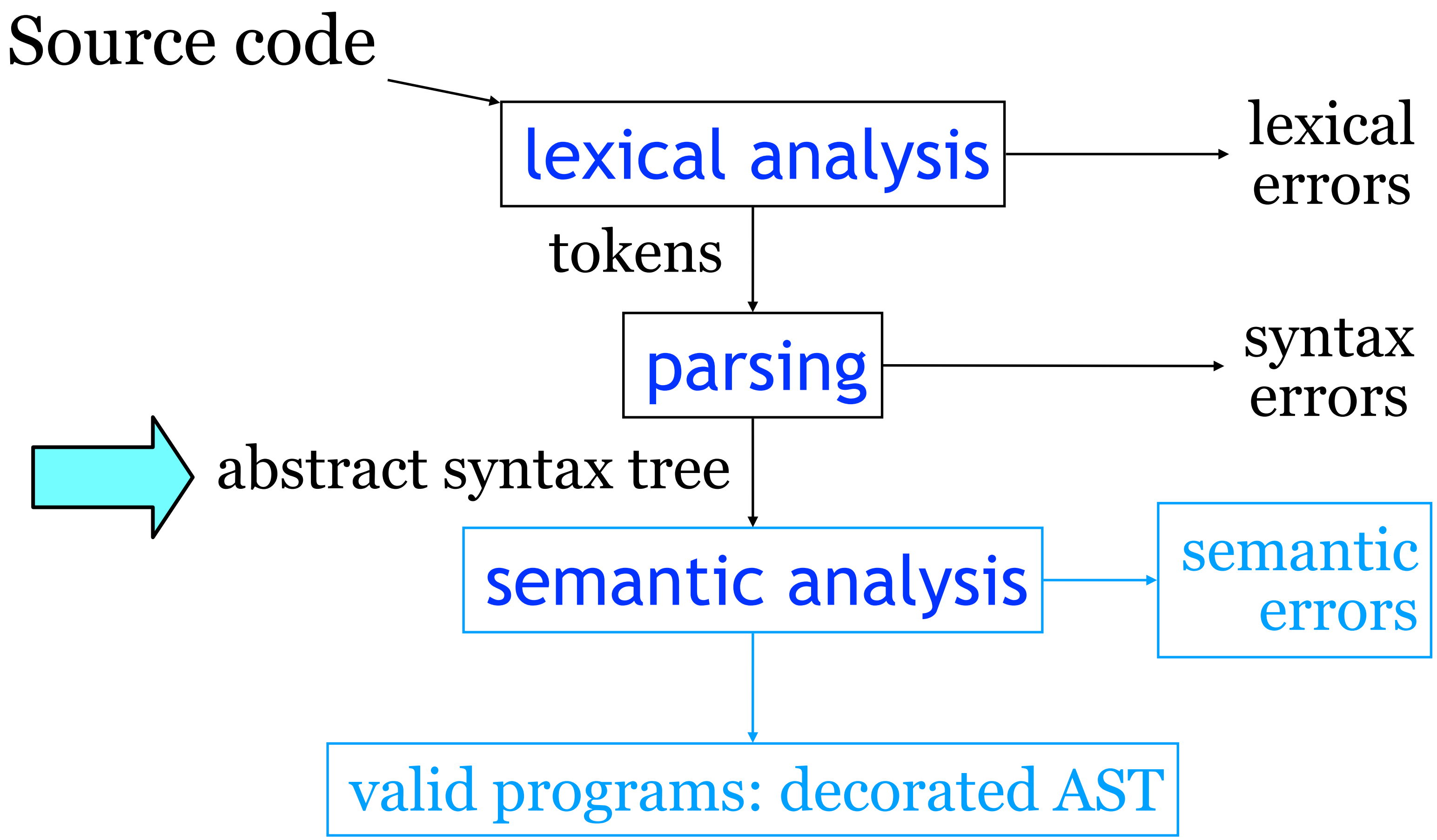
Compiler 'main program'

```
class Compiler {
    void compile() throws CompileError {
        Lexer l = new Lexer(input);
        Parser p = new Parser(l);
        AST tree = p.parse();
            // calls l.getToken() to read tokens
        if (typeCheck(tree))
            IR = genIntermediateCode(tree);
        IR.emitCode();
    }
}
```

Thread of Control

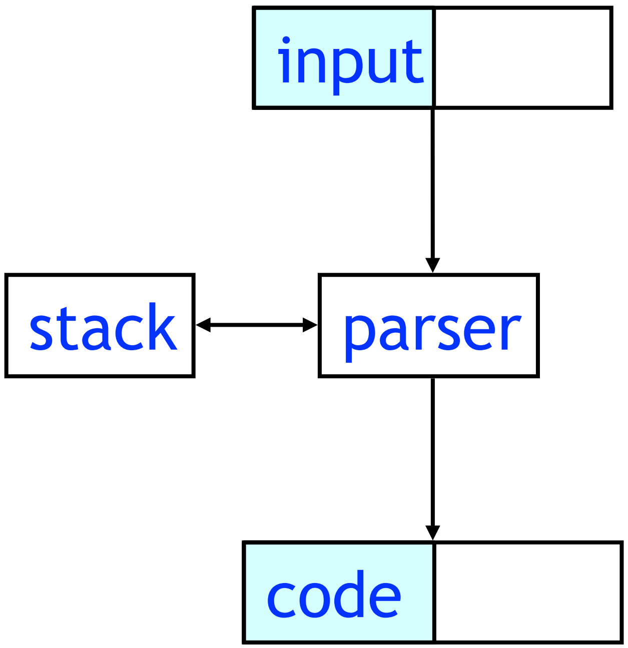


Semantic Analysis



Do we need an AST?

- Old-style compilers: semantic actions generate code during parsing!
- Especially for stack machine:




```
expr ::= expr PLUS expr
                {: emitCode(add); :}
```

Problems:

- hard to maintain
- limits language features (e.g., recursion)
- bad code!

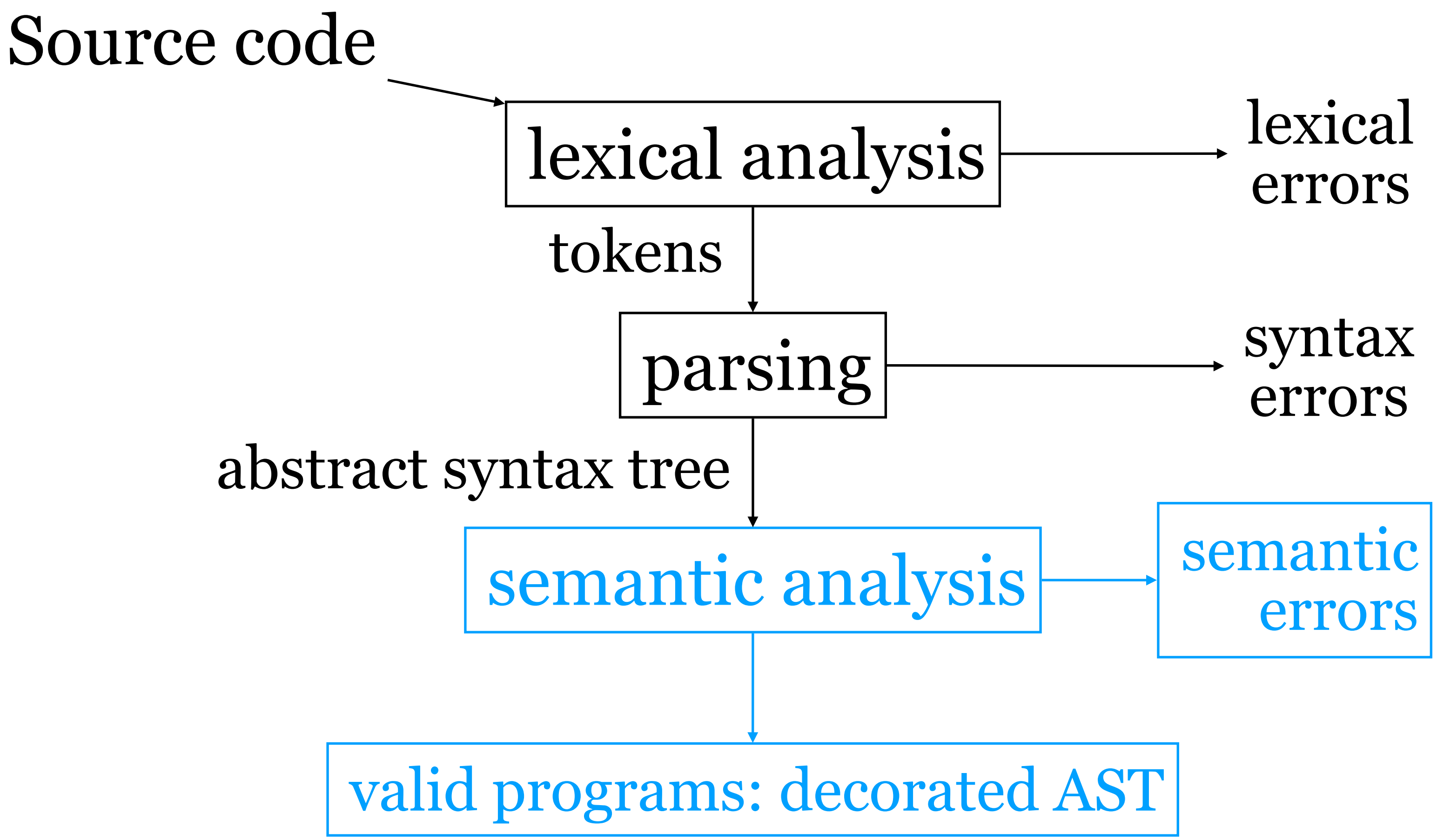
AST

- **A**bstract **S**yntax **T**ree is a tree representation of the program. Used for
  - semantic analysis (type checking)
  - some optimization (*e.g.* constant folding)
  - intermediate code generation (sometimes intermediate code = AST with somewhat different set of nodes)
- Compiler phases = recursive tree traversals
- Object-oriented languages convenient for defining AST nodes

Outline

- Abstract syntax trees
- Type checking
- Symbol tables
- Using symbol tables for analysis

Semantic Analysis



Building the AST bottom-up

- Semantic actions are attached to grammar statements
- E.g. CUP: Java statement attached to each production

```
non terminal Expr expr; ...
expr ::= expr:e1 PLUS expr:e2
    {: RESULT = new Add(e1,e2); :}
```

grammar production

semantic action

- *Semantic action* executed when parser reduces a production
- Variable RESULT is *value* of non-terminal symbol being reduced (in yacc: $$)
- AST is built bottom-up along with parsing

Actions in S-R parser

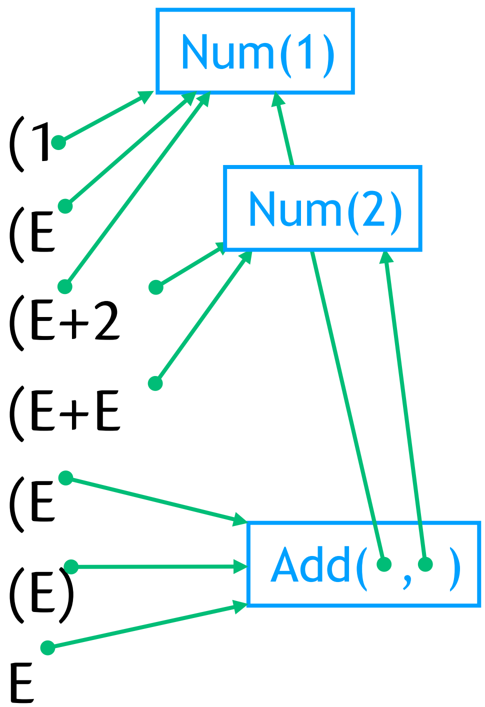
$$E \rightarrow \mathbf{num} \mid (\, E \,) \mid E + E$$

non terminal Expr expr; …

expr ::= expr:e1 PLUS expr:e2

{: RESULT = new Add(e1,e2); :}

- Parser stack stores value of each non-terminal



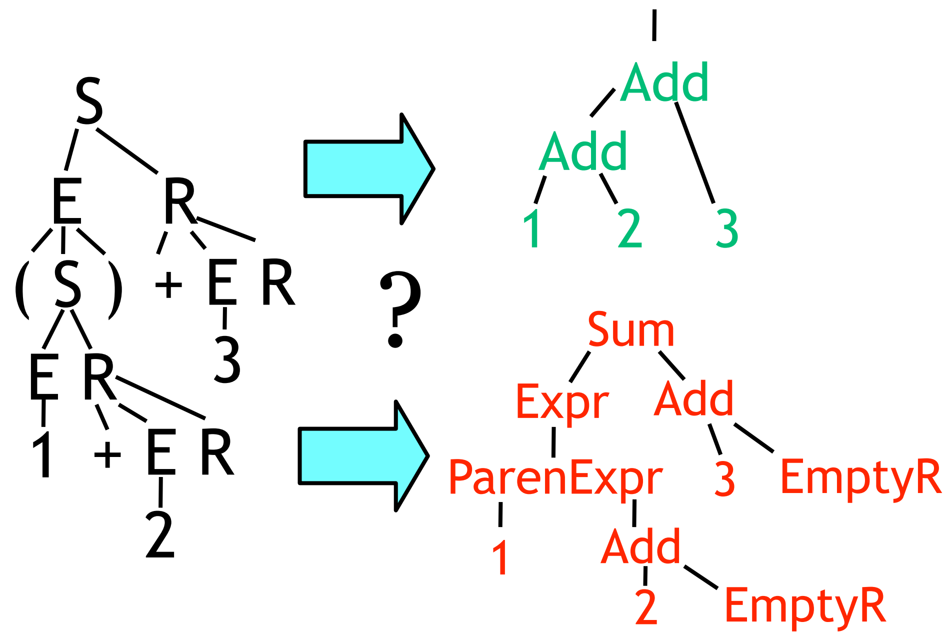
	(1 + 2) + 3	
(1	+2)+3	
(E	+2)+3	RESULT=new Num(1)
(E+2	)+3	
(E+E	)+3	RESULT=new Num(2)
(E	)+3	RESULT=new Add(e1,e2)
(E)	+3	
E	+3	RESULT=e

How not to design an AST

- Introduce a tree node for every node in parse tree
  - not very abstract
  - creates a lot of useless nodes to be dealt with later

$$
\begin{aligned}
S &\rightarrow E\ R \\
R &\rightarrow \varepsilon \mid +\ E\ R \\
E &\rightarrow \mathrm{num} \mid (\ S\ )
\end{aligned}
$$

(1 + 2) + 3



How not to design the AST, part II

- Simple(minded) approach: have one class AST_node
- E.g. need information for if, while, +, *, ID, NUM

```
class AST_node {
    int node_type;
    AST_node[ ] children;
    String name; int value; ...etc...
}
```

- Problem: must have fields for every different kind of node with attributes
- Not extensible, Java type checking no help

Using class hierarchy

- Can use subclassing to solve problem
  - write *abstract* class for each "interesting" non-terminal in grammar
  - write non-abstract subclass for (almost) every prod'n

$$E \rightarrow E + E \mid E * E \mid \text{-}E \mid (E)$$

```
abstract class Expr { … } // E
class Add extends Expr { Expr left, right; … }
class Mult extends Expr { Expr left, right; … }
// or: class BinExpr extends Expr { Oper o; Expr l, r;  }
class Negate extends Expr { Expr e; …}
```

Creating the AST

```
non terminal Expr expr; ...
```

"RESULT has type Expr in all semantic actions for expr"

```
expr ::= expr:e1 PLUS expr:e2
    {: RESULT = new BinaryExpr(plus, e1, e2); :}
| expr:e1 TIMES expr:e2
    {: RESULT = new BinaryExpr(times, e1, e2); :}
| MINUS expr:e
    {: RESULT = new UnaryExpr(negate, e); :}
| LPAREN expr:e RPAREN
    {: RESULT = e; :}
```

plus, times, negate: Oper

Expr

BinaryExpr UnaryExpr

Another Example

expr ::= *num* | (*expr*) | *expr* + *expr* | **id**

stmt ::= *expr* ; | *if* (*expr*) *stmt* |
if (expr) *stmt* else *stmt* | **id** = *expr* ; | ;

```
abstract class Expr { … }
class Num extends Expr { Num(int value) … }
class Add extends Expr { Add(Expr e1, Expr e2) … }
class Id extends Expr { Id(String name) … }
abstract class Stmt { … }
class If extends Stmt { If(Expr cond, Stmt s1, Stmt s2) }
class EmptyStmt extends Stmt { EmptyStmt() … }
class Assign extends Stmt { Assign(String id, Expr e)…}
```

And…top-down

- parse_*X* method for each non-terminal X
- Return type is abstract class for X

```
Stmt parseStmt() {
  switch (next_token) {
    case IF: consume(IF); consume(LPAREN);
     Expr e = parseExpr;
     consume(RPAREN);
     Stmt s2, s1 = parseStmt();
     if (next_token == ELSE) { consume(ELSE);
                               s2 = parseStmt(); }
     else s2 = new EmptyStmt();
     return new IfStmt(e, s1, s2); }
    case ID: …
```