CS412/413

Introduction to
Compilers and Translators
Cornell University – Spring ’00

Lecture 7: Parser Generators and
Abstract Syntax Trees

Sum grammar?

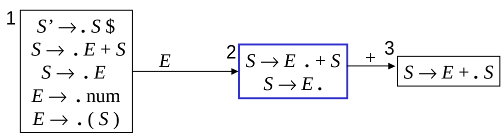
$$S \rightarrow S + E \mid E$$
$$E \rightarrow \text{num} \mid (S)$$

- This is LR(0)
- Right-associative version isn’t:

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

LR(0) construction

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



What to do on +?

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

SLR grammars

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

- Idea: Only add reduce action to table if look-ahead symbol is in the *FOLLOW* set of the non-terminal being reduced
- Eliminates some conflicts
- *FOLLOW*(S) = { $,) }
- Many language grammars are SLR

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

LR(1) parsing

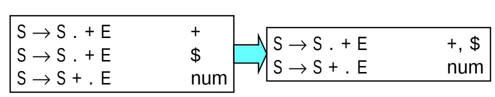
- Gets as much power as possible out of 1 look-ahead symbol
- LR(1) grammar = recognizable by a shift/reduce parser with 1 look-ahead.
- LR(1) items keep track of look-ahead symbols expected to follow this production

LR(0): $S \rightarrow . S + E$

LR(1): $S \rightarrow . S + E$ +,$

LR(1) state

- LR(1) state is a 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



LR(1) closure

Consider $A \rightarrow \beta \,.\, C\, \delta$. Closure formed just as for LR(0) *except*

1. Look-ahead symbol includes characters following the non-terminal symbol to the right of dot : FIRST(δ)
2. If non-terminal symbol may be last symbol in production (δ is nullable), look-ahead symbol includes look-ahead symbols of production

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

1 2

$S \rightarrow E + S \mid E$
$E \rightarrow$ num $\mid (\, S\,)$

LR(1) DFA construction

- Given LR(1) state, for each symbol (terminal or non-terminal) following a dot, construct a state with dot shifted across symbol, perform closure

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

$\xrightarrow{E}$

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

$S \rightarrow E + S \mid E$
$E \rightarrow$ num $\mid (\, S\,)$

LR(1) example

$S \rightarrow E + S \mid E$
$E \rightarrow$ num $\mid (\, S\,)$

1

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

$\xrightarrow{E}$ 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 look-ahead
  - Results in smaller parser tables -- works extremely well in practice

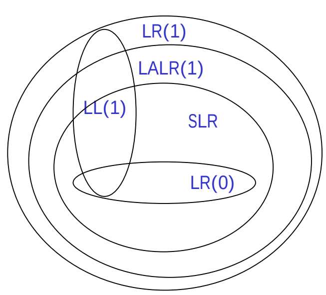
$S \rightarrow id\ .$	+
$S \rightarrow E\, .$	\$

+

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

= ?

Classification of Grammars



How are parsers written?

- Automatic parser generators: yacc, bison, CUP
- Accept LALR(1) grammar specification
- *plus:* declarations of precedence, associativity

Associativity

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

What happens if we run this grammar through LALR construction?

Conflict!

$$E \to E + E \mid \text{num} \mid (E)$$

$E \to E + E\,.$	$+$
$E \to E\,. + E$	$+,\$$

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

$$E \to E + E \mid T$$
$$T \to T \times T \mid \text{num} \mid (E)$$
$$\Downarrow$$
$$E \to E + E \mid E \times E$$
$$\mid \text{num} \mid (E)$$

Conflicts w/o precedence

$$E \to E + E \mid E \times E$$
$$\mid \text{num} \mid (E)$$

$E \to E\,. + E$	*any*
$E \to E \times E\,.$	$+$

$E \to E + E\,.$	$\times$
$E \to E\,. \times E$	*any*

Predecence in CUP

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

$E \to E\,. + E$	*any*
$E \to E \times E\,.$	$+$

$E \to E + E\,.$	$\times$
$E \to E\,. \times E$	*any*

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

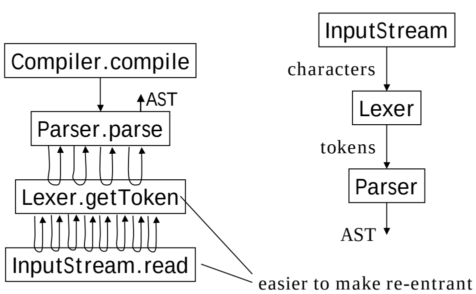
Summary

- Look-ahead information makes SLR(1), LALR(1), LR(1) grammars expressive
- Automatic parser generators support LALR(1)
- Precedence, associativity declarations simplify grammar writing
- Can we use parsers for programs other than compilers?

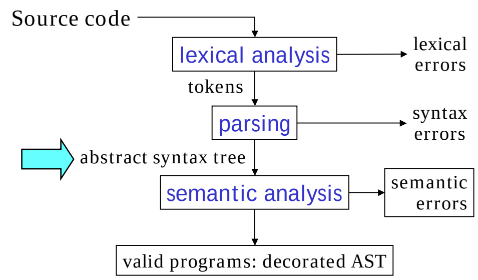
Compiler 'main program'

```
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



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

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); :}
```

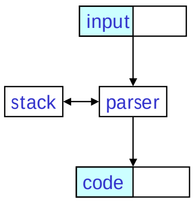
grammar production

semantic action

- *Semantic action* executed when parser reduces a production
- Variable RESULT is *value* of non-terminal symbol being reduced (in yacc: $$)

Do we need an AST?

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




```
expr ::= expr:e1 PLUS expr:e2
    {: emitCode(add); :}
```

- Problems:
  - limits language features
  - bad code!

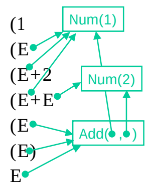
Actions in S-R parser

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

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

- 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

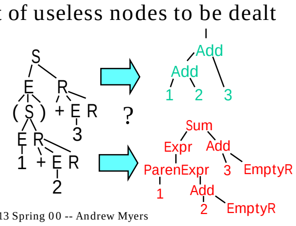


How not to design an AST

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

$$\boxed{\begin{array}{l} S \rightarrow E\ R \\ R \rightarrow \varepsilon \mid + E\ R \\ E \rightarrow \text{num} \mid (S) \end{array}}$$

(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

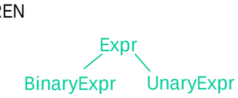
$$E \rightarrow E + E \mid E * E \mid -E \mid (E)$$

```
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; …
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; :}
```

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



plus, times, negate: Oper

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)…}
```


Top-down

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

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


Structuring Semantic Analysis

- Semantic analysis (type checking) is a recursive walk over AST structure (sometimes >1)
- Idea: add typeCheck method to every AST node

```
abstract class Expr {
  Type typecheck() throws SemanticError;
     // Return the type of this node or
     // throw an exception
```


Implementing typeCheck

- “An addition operation has type int if both of its operands have type int; otherwise it is illegal”

```
class Add {
   Expr e1, e2;
   Type typeCheck() throws SemanticError {
     if (e1.typeCheck() == Int &&
        e2.typeCheck() == Int)
       return Int;
     else throw new SemanticError(
       “operands to + have wrong type”);
   }
```

walk tree recursively