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

SLR grammars

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

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

\[
\begin{align*}
\text{LR}(0): & \quad S \rightarrow S + E \\
\text{LR}(1): & \quad S \rightarrow S + E \quad +, S
\end{align*}
\]
**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: $\text{FIRST}(\delta)$
2. If non-terminal symbol may be last symbol in production ($\delta$ is nullable), look-ahead symbol includes look-ahead symbols of production.

**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.

**LR(1) example**

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

**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

**Classification of Grammars**

**How are parsers written?**
- Automatic parser generators: yacc, bison, CUP
- Accept LALR(1) grammar specification
- plus: declarations of precedence, associativity
### Associativity

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

\[ E \rightarrow E + E \mid \text{num} \mid (E) \]

What happens if we run this grammar through LALR construction?

### Conflict!

\[ E \rightarrow E + E \mid \text{num} \mid (E) \]

\[ E \rightarrow E + E, + \]
\[ E \rightarrow \text{E} . + E \}

1+2+3

\[ \text{shift/reduce conflict} \]

### Grammar in CUP

non terminal E; terminal PLUS, LPAREN...

precedence left PLUS;

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

\[ E ::= E \ \text{PLUS} \ E \]
\[ \text{LPAREN} \ E \ \text{RPAREN} \]
\[ \text{NUMBER} ; \]

### Precedence

- Also can handle operator precedence

\[ E \rightarrow E + E \mid T \]
\[ T \rightarrow T \times T \mid \text{num} \mid (E) \]

\[ E \rightarrow E + E \mid E \times E \]
\[ \mid \text{num} \mid (E) \]

### Conflicts w/o precedence

\[ E \rightarrow E + E \mid E \times E \]
\[ \mid \text{num} \mid (E) \]

### Predecence in CUP

precedence left PLUS;
precedence left TIMES; // TIMES > PLUS

\[ E ::= E \ \text{PLUS} \ E \mid E \ \text{TIMES} \ E \mid \ldots \]

Rule: in conflict, choose \textit{reduce} if production symbol higher precedence than shifted symbol; choose \textit{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

Compiler.compile
Parser.parse
Lexer.getToken
Lexer
Parser
InputStream.read

Semantics Analysis

Source code

lexical analysis

lexical errors

tokens

parsing

syntax errors

abstract syntax tree

semantic analysis

semantic errors

valid programs: decorated AST

Grammar production

non_terminal ::= expr1 PLUS expr2

Result = new Add(expr1, expr2);

Semantics action

Building the AST bottom-up

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

- 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

Compiler.compile
Parser.parse
Lexer.getToken
Lexer
Parser
InputStream.read

Semantics Analysis

Source code

lexical analysis

lexical errors

tokens

parsing

syntax errors

abstract syntax tree

semantic analysis

semantic errors

valid programs: decorated AST

Grammar production

non_terminal ::= expr1 PLUS expr2

Result = new Add(expr1, expr2);

Semantics action

Building the AST bottom-up

- Semantic actions are attached to grammar statements
- E.g. CUP: Java statement attached to each production
- 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
Do we need an AST?
- Old-style compilers: semantic actions generate code during parsing!
- Especially for stack machine:

\[
\text{expr ::= expr1 PLUS expr2}
\]

\[
\{ \text{: emitCode(add1); :} \}
\]

Problems:
- limits language features
- bad code!

Actions in S-R parser

non terminal Expr expr; ...

\[
E \rightarrow \text{expr} \mid (E)
\]

Parser stack stores value of each non-terminal

\[
(1 + 2) + 3
\]

\[
(1) + 2 + 3 \quad \text{RESULT=new Num(1)}
\]

\[
(2) + 3 \quad \text{RESULT=new Num(2)}
\]

\[
(3) \quad \text{RESULT=new Add(e1,e2)}
\]

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

\[
S \rightarrow E \mid R
\]

\[
E \rightarrow \text{num | } (E)
\]

\[
E \rightarrow + E \mid - E
\]

\[
E \rightarrow E * E
\]

\[
E \rightarrow E / E
\]

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

\[
E \rightarrow \text{expr} \mid + \text{expr} \mid - \text{expr}
\]

abstract class Expr {
  \[
  \text{abstract Add extends Expr \{ Expr left, right; \}}
  \]
  \[
  \text{abstract Mult extends Expr \{ Expr left, right; \}}
  \]
  \[
  \text{// or: class BinExpr extends Expr \{ Oper o; Expr l, r; \}}
  \]
  \[
  \text{class Negate extends Expr \{ Expr e; \}}
  \]

Creating the AST

non terminal Expr expr; ...

\[
\{ \text{RESULT = new BinaryExpr(plus, e1, e2); :} \}
\]

\[
\{ \text{RESULT = new BinaryExpr(times, e1, e2); :} \}
\]

\[
\{ \text{RESULT = new UnaryExpr(negate, e); :} \}
\]

\[
\{ \text{RESULT = e; :} \}
\]
Another Example

\[
\text{expr ::= num \mid (expr) \mid expr + expr \mid id}\\
\text{stmt ::= expr \mid if (expr) stmt \mid if (expr) stmt else stmt \mid id = expr}\\
\]

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 
  }
}