LR(1) parsing
- As much power as possible out of 1 look-ahead symbol parsing table
- 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):
\[ S \rightarrow \ast S + E \]

LR(1):
\[ S \rightarrow \ast S + E + \]

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

LR(1) closure
Consider \[ A \rightarrow \beta \cdot C \delta \lambda \]
Closure formed just as for LR(0) except
1. Look-ahead symbols include characters following the non-terminal symbol to the right of dot: FIRST(\(\delta\))
2. If non-terminal symbol may produce last symbol of production (\(\delta\) is nullable), look-ahead symbols include look-ahead symbols of production (\(\lambda\))

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
Reductions unambiguous if:
look-aheads are disjoint, not to right of any dot in state

1. \[ S \rightarrow \ast E + S \]
2. \[ S \rightarrow \ast E + S \]
3. \[ S \rightarrow \ast E + S \]
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
  - Usual technology for automatic parser generators

\[
S \rightarrow \text{id } \cdot \ + \ \cdot \\
S \rightarrow \text{E } \cdot \ + \\
S \rightarrow \text{E } \cdot \ , \ \cdot \\
\]

Classification of Grammars

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
- Rest of this lecture: how to use parser generators
- Can we use parsers for programs other than compilers?

Associativity

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

What happens if we run this grammar through LALR construction?

Conflict!

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

Grammar in CUP

- non terminal E; terminal PLUS, LPAREN...
- precedence left PLUS;
- "When shifting + conflicts with reducing a production containing +, choose reduce"
Precedence

- Also can handle operator precedence

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

Conflicts w/o precedence

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

Precedence in CUP

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

\[
E ::= E \ PLUS \ E \mid E \ TIMES \ E \mid ... \\
E \rightarrow E + E ... \\
E \rightarrow E \times E + \\
E \rightarrow E + E \times E ... \\
E \rightarrow E \times E +
\]

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’

```java
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

```plaintext
Compiler.compile \rightarrow AST
Parser.parse \rightarrow AST
Lexer.getToken \rightarrow AST
InputStream.read \rightarrow easier to make re-entrant
```
Semantic Analysis

Source code → lexical analysis → tokens → parsing → abstract syntax tree → semantic analysis → semantic errors → valid programs: decorated AST

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
- bad code!

AST

- Abstract Syntax Tree 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