Syntax Analysis

Chapters 3 and 4

Syntax Analysis Tasks

- Recognize syntactic structure in token stream
- Produce parse tree that represents the structure of input program

Motivation from Regular Expression Macros

- Regular expression macros for summation expressions

```
digits = [0-9]+
sum    = ({digits} "+")* {digits}
```

  - Works for input `10+2+30`

- How about:

```
digits = [0-9]+
sum    = ({expr} "+")* {expr}
expr   = {digits} | "(" {sum} ")"
```

  - Describes input `10+(2+30)`
  - But is not a regular language anymore

Context-free Grammar Motivation

- Functionality
  - Similar to regular expressions plus recursion

- Top-level alternation

```
expr = a b (c | d) e
```

  - Needs to be translated to

```
aux  = c | d
expr = a b aux e
```

  - Which is short-hand for

```
aux  = c
aux  = d
expr = a b aux e
```

Context-free Grammar Motivation, cont'd

- Recursion instead of Kleene closure

```
expr = (a b c)*
```

  - Becomes

```
expr = a b c expr
expr = ε
```

Context-free Grammar

- Terminal symbols (tokens)
- Non-terminal symbols (higher-level syntactic constructs)
- Start symbol (one of the non-terminals)
- Rules of the form

  ```
  N -> X*
  ```

  - Where `N` is a non-terminal and `X` is a (non-)terminal

CFG Syntax Variants

- Definition operator: `->` vs. `=` vs. `::=`
- Identification of non-terminals: capitalized vs. triangular brackets
- Identification of terminals: in quotes vs. all uppercase

- BNF (Backus-Naur Form), developed for Algol-60

```
<stmt> ::= 'if' <expr> 'then' <stmt> | ...
```

- EBNF (Extended Backus-Naur Form)
  - `[ … ]` for optional parts
  - `{ … }` for repetition zero or more times

- Scheme Reports (R6RS): ellipses (…) for repetition zero or more times

Motivating Example in Grammar Notation

```
digit =  0
   ...
digit =  9

digits = digit
digits = digit digits

sum =  expr "+" sum
sum =  expr

expr = digits
expr = "(" sum ")"
```

Recursion

- Right-recursive grammar

```
N -> t
N -> t N
```

- Left-recursive grammar

```
N -> t
N -> N t
```

Example Grammar

```
S -> S ; S
S -> id := E
S -> print ( L )

E -> id
E -> num
E -> E + E
E -> ( S , E )

L -> E
L -> L , E
```

Example: Variants for Lists of Expressions

- Left-recursive

```
L -> E
L -> L , E
```

- Right-recursive

```
L -> E
L -> E , L
```

- Ambiguous

```
L -> E
L -> L , L
```

Styles of Grammar Rules

- Left-recursive
  - Preferable for left-associative operators
  - Doesn't work with top-down parsers
    - E.g., handwritten recursive-descent parsers
  - Cost: $O(n)$

- Right-recursive
  - Works with either top-down or bottom-up parsers
  - Cost: $O(n)$

- Ambiguous
  - Result in multiple possible parse trees, which is undesirable
  - E.g., supported by Earley parser and GLR parser
  - Cost: $O(n^3)$

Grammar Classes and Parser Types

- LL(k): **L**eft-to-right, **L**eft-most derivations, **k** tokens lookahead
- LR(k): **L**eft-to-right, **R**ight-most derivations, **k** tokens lookahead

- Parser Technologies
  - LL(0): simplest top-down parser
  - LL(1): typical handwritten recursive-descent parser
  - LL(k): generated by top-down parser generator (ANTL, JavaCC, …)

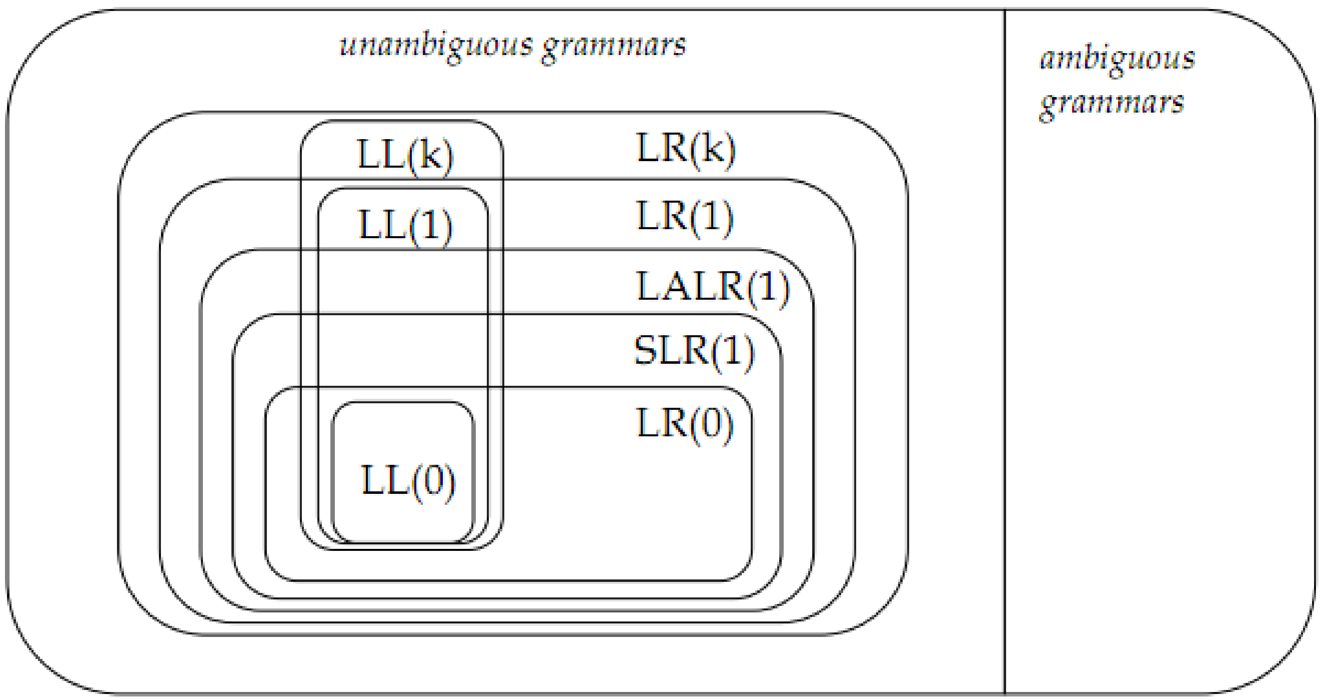
  - LR(0): simplest bottom-up parser
  - SLR: simplest bottom-up parser with 1 token lookahead
  - LALR(1): using bottom-up parser generator (yacc, bison, JavaCUP, MLYacc, …)
  - LR(1): bottom-up parser with 1 token lookahead

Hierarchy of Grammar Classes

- Found on Stackoverflow, similar to Textbook, p. 68:

LL(1) versus LR(k)

A picture is worth a thousand words:



Concrete Parse Trees

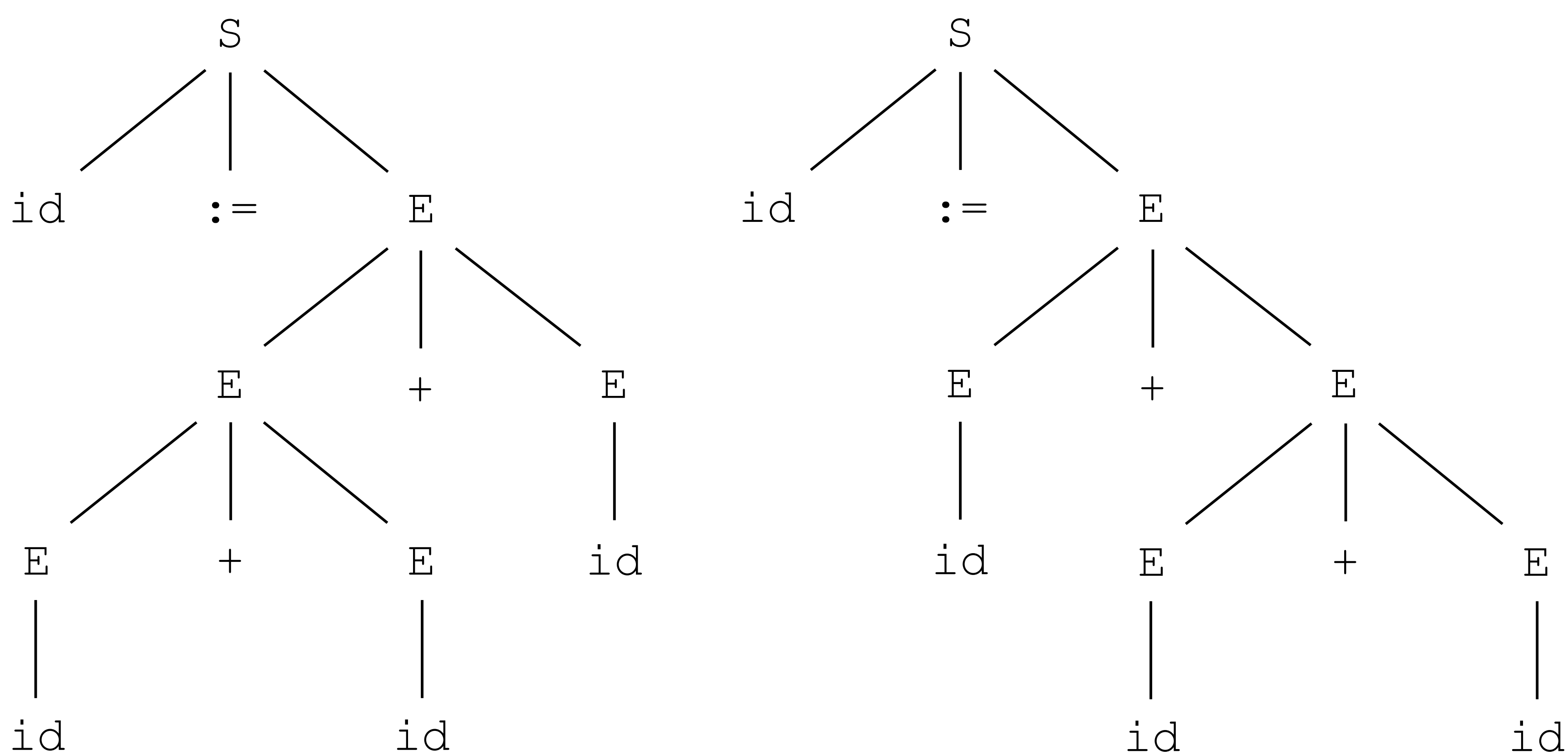
- Aka. concrete syntax tree or derivation tree
- Represent syntactic structure of input
- Also summarizes how the input was parsed

- Leaves are terminal symbols (tokens)
- Interior nodes are non-terminal symbols
- The root is the start symbol

- Reading leaves left-to-right gives input

Possible Parse Trees for Ambiguous Grammar

- **Input:** `id := id + id + id`



Derivations

- Start with start symbol of grammar
- In each step replace a non-terminal by RHS of grammar rule

- E.g.:

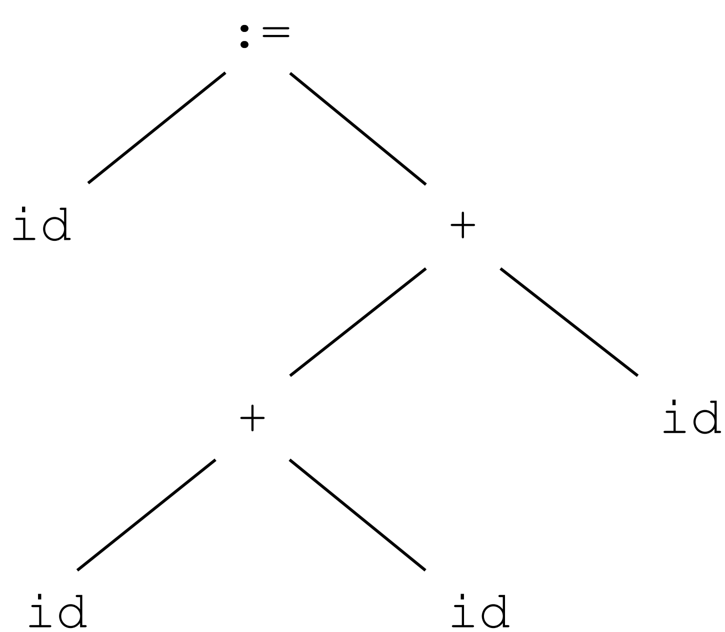
```
S
S ; S
S ; id := E
id := E ; id := E
id := num ; id := E
…
```

Use of Derivations

- Used to define language described by grammar
  - Language is set of all possible strings that can be derived from start symbol
- Left-most derivations
  - In each step expand the left-most non-terminal by RHS of grammar rule
  - Corresponds to top-down (L**L**) parsing
- Right-most derivations
  - In each step expand the right-most non-terminal by RHS of grammar rule
  - Corresponds to bottom-up (L**R**) parsing
- Concrete parse tree shows summary of derivation steps

Abstract Parse Trees

- Aka. abstract syntax tree or just parse tree
- Represent syntactic structure of input
- Does not summarize how the input was parsed anymore



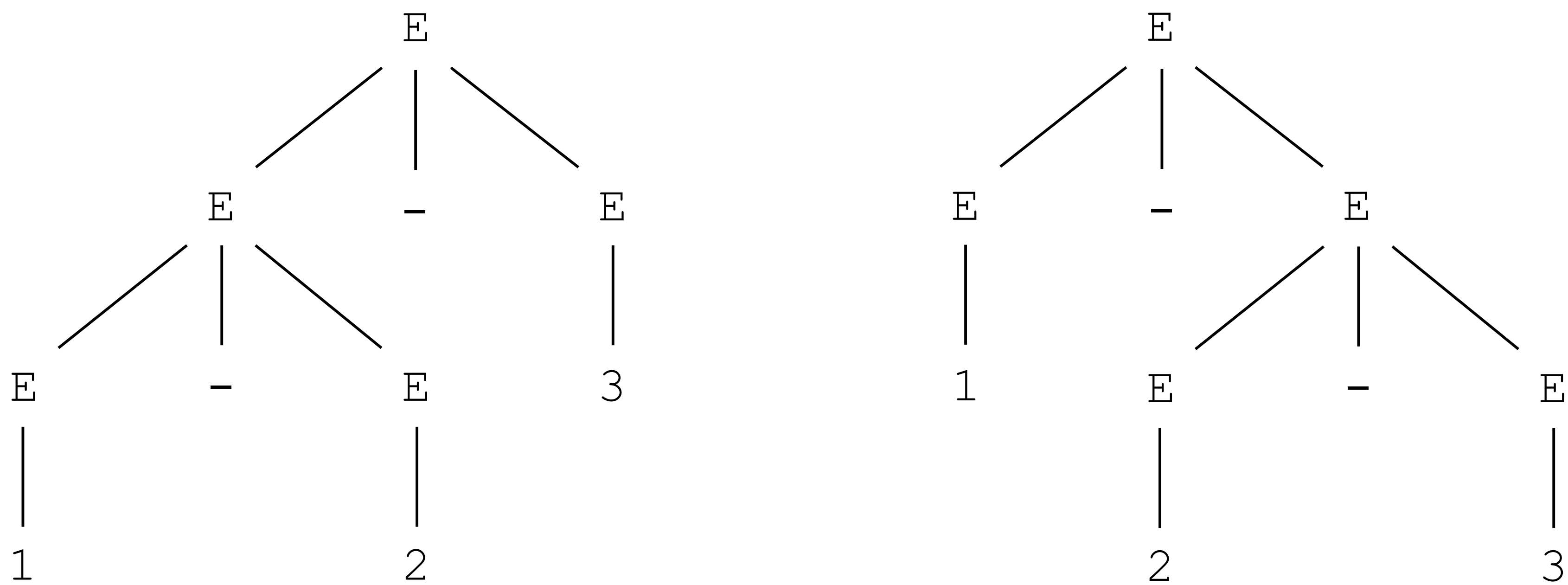
Ambiguous Expression Grammar

- Convenient and compact to write

```
E -> id
E -> num
E -> E * E
E -> E / E
E -> E + E
E -> E - E
E -> ( E )
```

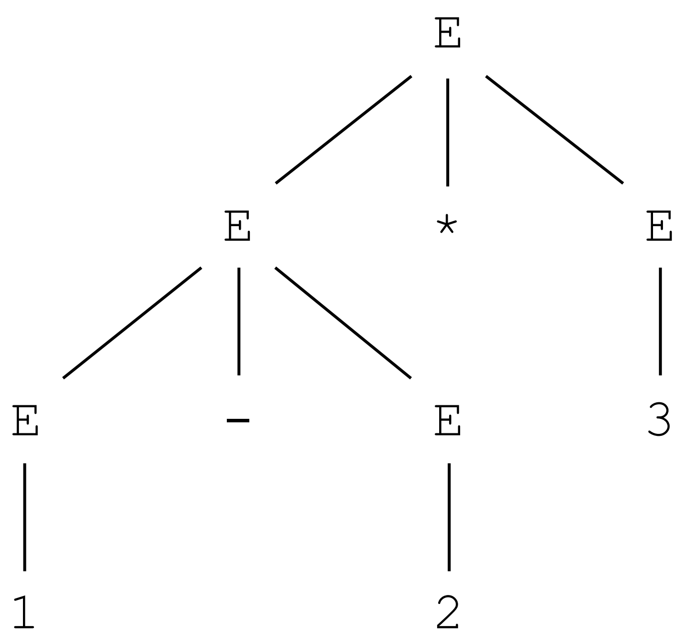
Possible Parse Trees

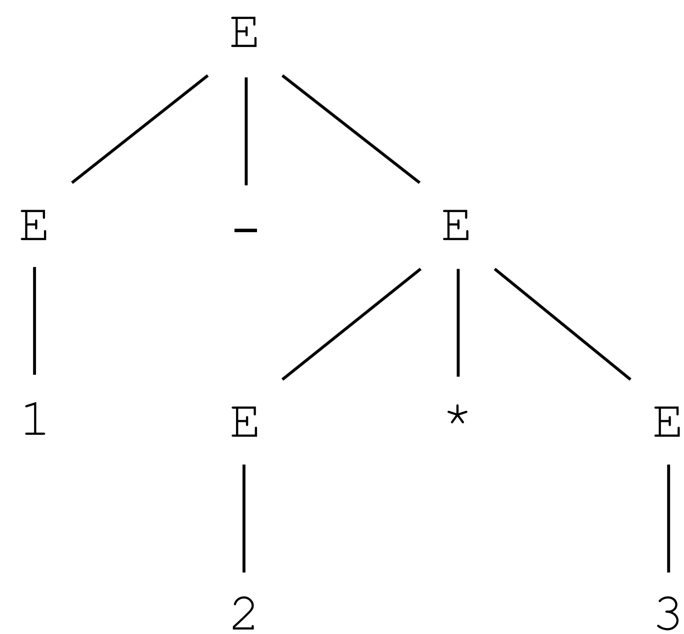
- **Input:** `1-2-3`



Possible Parse Trees, cont'd

- **Input:** `1-2*3`





Desired Parse Trees

- Higher-precedence operators should be below lower-precedence operators
  - Right parse tree for `1-2*3`
- For operators with equal precedence, the parse tree should reflect the associativity of the operator
  - Left parse tree for `1-2-3`, since it’s meaning is `(1-2)-3`
- Parantheses do not need to be represented in the parse tree
  - Implicit in the tree structure

Unambiguous Expression Grammar

```
E -> E + T
E -> E - T
E -> T

T -> T * F
T -> T / F
T -> F

F -> id
F -> num
F -> ( E )
```

- Lower-precedence operators are at the top
- Left recursion corresponds to left-associative operators
- Necessary for handwritten parser
- C, C#, and Java have 15 precedence levels, C++ has 17

Ambiguous Grammar for If-Statement

```
S -> if E then S else S
S -> if E then S
S -> other
```

- Ambiguous for two nested if-statements (aka. a dangling else)

```
if E1 then
   if E2 then
      S1
   else
      S2
```

- Does `else` belong to inner `if` or outer `if`?

Unambiguous Grammar for If-Statement

```
S -> M
S -> U

M -> if E then M else M
M -> other

U -> if E then S
U -> if E then M else U
```

- `M`: matched statement (each if comes with an else)
- `U`: unmatched statement
- The `then` part is always matched, therefore each `else` goes with the inner `if`
- LR parsers can automatically repair the ambiguous grammar for if-statements

JavaCUP Precedence Directives

```
precedence nonassoc EQ, NEQ;
precedence left     PLUS, MINUS;
precedence left     TIMES, DIV;
precedence right    EXP;
precedence left     UMINUS;

exp ::= INT
    |   exp PLUS exp
    |   exp MINUS exp
    |   exp TIMES exp
    |   MINUS exp %prec UMINUS
    ;
```

Associativity for Comparison Operators

- In Tiger, comparison operators are non-associative
  - Instead of the math expression `x<y<z`
  - Programmers have to write: `x<y and y<z`

- In Java, consecutive comparison operators can result in a type error
  - The result of `x<y` is of type `boolean`
  - Which is not a legal argument type for `<`

- In C and C++, comparison operators are left-associative
  - The expression `x<y<z`
  - Compares the result of `x<y` with `z`

Associativity for Assignment

- In the C family, the assignment operator is right-associative
- The value of an assignment expression is the RHS value
- I.e., `x=y=z` is equivalent to `x=(y=z)`
- It assigns `z` to `y` and then the result (which is `z`) to `x`

- In Tiger, assignments cannot be used inside expressions
- I.e., the assignment operator is non-associative

Building Parse Trees in JavaCUP

```
terminal      Integer   INT;
terminal      PLUS;
non terminal Absyn.Exp Exp;

precedence left PLUS;

Exp ::= INT:e
        {: RESULT = new IntExp(eleft, e.intValue()); :}
     |  Exp:l PLUS:o Exp:r
        {: RESULT = new OpExp(oleft,l,OpExp.PLUS,r); :}
     ;
```

Building Parse Trees in JavaCUP, cont'd

- Not every grammar rule needs a constructor call

```
Exp ::= LPAREN SeqExp:e RPAREN
        {: RESULT = e; :}
     ;
```

  - Where `SeqExp` is a sequence of expressions separated by commas

- When constructing lists, an empty list might be represented as `null`

```
XList ::=
          {: RESULT = null; :}
       |  Xs:l
          {: RESULT = l; :}
       ;
```

  - Where `Xs` is a list of one or more `X`'s maybe with a separator

LR Parse Engine: Push-Down Automaton

- Stack
- DFA
  - Applied to the stack
  - Edges labeled with (non-)terminals

- Actions
  - s_n shift and goto state n
  - r_k reduce by rule k
  - g_n goto state n (always following r_k)
  - a accept (shift EOF)
  - \- error

LR Parse Actions

- Conceptually
  - Shift moves a token from the input on top of the stack
  - Reduce(+goto) replaces the RHS of a rule on top of the stack by the LHS
- Actual implementation
  - The stack contains DFA state numbers, not tokens or non-terminals
  - Shift and goto put state numbers onto the stack

Parse Trace Example, Conceptually

Stack	**Input**	**Action**
	a := 7 $	shift
id	:= 7 $	shift
id :=	7 $	shift
id := num	$	reduce E -> num
id := E	$	reduce S -> id := E
S	$	accept

Grammar

S -> id := E
E -> id
E -> num

Parse Trace Example, with States

Stack	**Input**	**Action**
$_1$	a := 7 $	shift 4
$_1$ id $_4$	:= 7 $	shift 6
$_1$ id $_4$:= $_6$	7 $	shift 10
$_1$ id $_4$:= $_6$ num $_{10}$	$	reduce by rule 5 (E->num)
$_1$ id $_4$:= $_6$ E $_{11}$	$	reduce by rule 2 (S->id:=E)
$_1$ S $_2$	$	accept

Grammar

2: S -> id := E
4: E -> id
5: E -> num

Parse Table

	id	num	print	;	,	+	:=	(	)	$	S	E	L
1	s4		s7								g2		
2				s3						a			
3	s4		s7								g5		
4							s6						
5				r1	r1					r1			
6	s20	s10						s8				g11	
7						…							
8						…							
9						…							
10				r5	r5	r5			r5	r5			
11				r2	r2	s16				r2			
…						…							

Parse Table Notes

- The right three columns are the goto-table, the rest is the parse table
- $ indicates EOF

- This table is for an LALR(1) or LR(1) parser
- In state on top of stack, parser looks up action based on input token

- An LR(0) parser needs to decide whether to shift or reduce based on the stack content alone before looking at the input token
- An LR(0) parse table has the same reduce action in each column

Parse Conflict

- If the table construction algorithm results in two action in the same table slot, that's a parse conflict

- Shift-Reduce conflict: CUP resolves it in favor or shift
- Reduce-Reduce conflict: CUP resolves it based on order of rules

- The automatic conflict resolution sometimes works for S/R conflicts

CUP Output for S/R Conflict

```
state 17:          shift/reduce conflict
                   (shift ELSE, reduce 4)

    stm ::= IF ID THEN stm .
    stm ::= IF ID THEN stm . ELSE stm

    ELSE           shift 19
    .              reduce by rule 4
```

- Grammar rule with dot (or `(*)`) is a parser configuration
  - Everything left of dot is on stack, everything right of dot is in input
  - Java CUP also show possible lookahead tokens after parser configurations

Parse Actions

- Shift(n)
  - Eat one token
  - Shift state n onto the stack
- Reduce(k)
  - Pop n states off the stack, where n is number of symbols on RHS of rule k
  - In state on top of stack, look up X (LHS of rule k) in goto table to find goto(m)
  - Push state m onto the stack
- Accept
  - Stop, report success
- Error
  - Report error
  - Go into error recovery mode

Reduce/Reduce Conflict

```
precedence left      OR;
precedence left      AND;
precedence nonassoc EQUAL;
precedence left      PLUS;

stm ::= ID ASSIGN ae
     |  ID ASSIGN be;
be  ::= be OR be
     |  be AND be
     |  ae EQUAL ae
     |  ID;
ae  ::= ae PLUS ae
     |  ID;
```

R/R conflict

Reduce/Reduce Conflict, cont'd

```
state 5:          R/R conflict
                  between rule 6 and 4
                  on EOF

     be ::= ID .
     ae ::= ID .

     PLUS         R6
     AND          R4
     OR           R4
     EQUAL        R6
     EOF          R4
     .            error
```

<- conflict resolved in favor of rule 4 (`be::=ID`)

Solution: Push Decision to Semantic Analysis

```
stm ::= ID ASSIGN E;

E   ::= E OR E
     |  E AND E
     |  E EQUAL E
     |  E PLUS E
     |  ID;
```

- Allow incorrect expressions in parser, such as

```
x := a + b and c
```

Solution: Push Decision to Scanner

- The following C++ statement is ambiguous:

```
C x(a, b, c);
```

- If `a`, `b`, and `c` are types, this is a forward declaration of function `x`
- If they are variables, this is a constructor call to initialize object `x`

- Older versions of g++ required the scanner to distinguish between

```
IDENT
TYPEIDENT
LABELIDENT
```

- Needs type info from semantic analysis to be fed back to scanner
- Also requires lookahead on token stream if type info not available

Shift/Reduce Conflicts for Operators

- Use precedence declarations for operators

```
precedence left      PLUS;
precedence right     ASSIGN;
precedence nonassoc EQUAL;
```

- Use `%prec` if no appropriate operators are available

```
precedence left      UMINUS;

E ::= MINUS E %prec UMINUS;
```

Shift/Reduce Conflict in Tiger

```
Exp ::= Var
     |  ID LBRACK Exp RBRACK OF Exp
     ;

Var ::= ID
     |  Var LBRACK Exp RBRACK
     ;
```

- Solution: unroll recursion for Var
  - `ID` before `LBRACK` should not be reduced to `Var`
  - Postpones parse decision between array creation and array reference expressions until after `RBRACK`

Parse Conflicts for Declarations in Tiger

```
Decls ::=
       |  Decls Decl ;
Decl  ::= VarDecl
       |  FunDecls
       |  TypeDecls ;
```

- In `Decls`, parse tree is built with `Absyn.DecList` nodes
- In `FunDecls` and `TypeDecls`, tree is built using the `next` field in `Absyn.FunctionDec` and `Absyn.TypeDec`, respectively
- Structure grammar to allow correct constructor calls
- Avoid R/R. Goal: S/R that is automatically resolved in favor of shift

Shift/Reduce Conflict for Java Subset

```
Stm      ::= VarDec
           |  Assign ;
VarDec   ::= Type ID SEM ;
Assign   ::= LVal EQ Exp SEM ;
Type     ::= QualName
           |  BuiltinType ;
LVal     ::= QualName
           |  LVal LBRACK Exp RBACK
           |  LVal DOT ID ;
QualName ::= ID
           |  QualName DOT ID ;
```

Shift-Reduce Conflict for Java Subset, cont'd

- A qualified name is something like `java.util.Hashtable`
- An l-value is something like `x.y.z` on the LHS of an assignment (or in an expression)

- Different tree nodes result in different semantic processing

- Goal: give preference to `DOT` in `QualName`

- Solution: modify grammar or use precedence declarations

Error Reporting and Error Recovery

- All empty entries in parse table result in “Syntax error”
- By default, the parser stops after the first error

- Goal: recover from errors and continue parsing to find more errors

- Maybe produce specific errors for common problems

Error Recovery

- Use grammar rules with special `error` token

- Error recovery after reporting parse error:
  - Pop stack until there is a shift on `error`
  - Shift `error` token
  - Discard input until we are in a state with a non-error action
  - Resume parsing

- Often easiest if there is a synchronizing token following `error`
- Too many error recovery rules can lead to S/R and R/R parse conflicts

Error Recovery Example

```
E  -> ID
E  -> E + E
E  -> ( E )
E  -> ( error )

Es -> E
Es -> Es ; E
Es -> error ; E
```

Error Recovery Example, cont'd

- Given the input
  `( x + + y )`

- Parser recognizes error, when `(ID+` is on the stack and `+` in the input
- Parse reports syntax error
- `ID+` is deleted from stack, since with `(` on stack, we can shift `error`
- The error token is shifted onto the stack
- `+ID` is removed from input, since `(error` must be followed by `)`
- Parser continues parsing normally

Grammar Rules for Error Reporting

```
Decl ::= Type ID LBRACK INT RBRACK
         ASSIGN LBRACE Exp RBRACE SEM
         {: ... :}

      |  Type ID LBRACK INT RBRACK
         ASSIGN LBRACE RBRACE SEM
         {:
            error("empty array initializer");
            RESULT = null;
         :}
```

Global Error Repair

- Example

  `let type a := intArray[10] of 0;`

- Solutions
  - Error production: delete `type … 0`
  - Local repair: replace `:=` with `=`
  - Global repair: replace `type` with `var`

Burke-Fisher Error Repair

- Consider possible single-token insertions/deletions/substitutions in the last K tokens (e.g., K=15)
- Use the repair that gets us the farthest, preferably at least R tokens (e.g., R=4)

Burke-Fisher Error Repair, cont’d

- Backing up K tokens to consider repairs requires more complex data structure
  - Old stack: parse stack for stack content more than K tokens in the past
  - Token queue of K tokens
  - Current stack: parse stack including those K tokens
- On error, current stack is tossed, queue is reprocessed

- Cost:
  - For window size K and N tokens: K + N*K + N*K
  - K deletions, N*K insertions, and N*K substitutions

ML-Yacc

- Semantic actions for insertions

```
%value ID     ("bogus")
%value INT    (1)
%value STRING ("")
```

- Programmer-specified substitutions

```
%change EQ -> ASSIGN
     |  ASSIGN -> EQ
     |  SEM ELSE -> ELSE
     |  -> IN INT END
```

LR Parser Technologies

- So far, we have been assuming one token lookahead
- Grammar categories and parser technologies
  - LR(0): parse table construction and parser don't use lookahead
  - SLR: uses LR(0) parse table construction but only reduces if lookahead is in FOLLOW set (see LL parsing for FOLLOW set), parser uses lookahead
  - LALR(1): parser uses compressed LR(1) parse tables
  - LR(1): parse table construction and parser use one token lookahead

- Most bottom-up parser generators build LALR(1) parsers
  - If two parse states are only distinguished by lookahead, they are merged
  - LR(1) parse tables are very large for little additional benefit
  - LALR(1) parsers can have R/R conflicts that an LR(1) parser would not have

LL-Parsing

- Recursive Descent Parsing

```
void S() {
  switch (tok) {
    case IF:
      eat(IF);
      E();
      eat(THEN);
      S();
      eat(ELSE);
      S();
      break;
    case BEGIN:
      ...
    default: error();
  }
}
```

Structure of Recursive-Descent Parser

- Code structure corresponds to grammar structure
  - Concatenation in grammar corresponds to straight-line code
  - Alternation in grammar corresponds to if-then-else or switch statement
- Assumes one token lookahead
  - The variable `tok` contains the lookahead before calling any parse function
  - Each parse function reads lookahead before returning
  - The function `eat()` checks lookahead against argument and gets next token
- Parse tree construction
  - Each parse function returns a parse tree
  - Parse tree constructors are called in return statements of parse functions

Problems with LL Parsing

- Left recursion

```
X -> X y
   |  z
```

  - Solution: restructure grammar to make it right-recusive

- Common left factors

```
X -> a Y
   |  a Z
```

  - Solution: restructure grammar to remove common left factors
  - Similar to left-factoring in math: translating `aY+aZ` into `a(Y+Z)`

Translating Grammar into Code

- Parser must choose alternatives based on lookahead
- Rules starting with non-terminals

```
X -> Y
  |  Z
```

  - We need to know the sets of terminals with which `Y` and `Z` can start
- Empty RHSs

```
X ->
  |  Y
```

  - We need to know the sets of terminals with which `Y` can start
  - And the sets of terminals that can follow an `X` after the empty RHS

- We need to compute FIRST and FOLLOW sets for each rule

Nullable, FIRST, and FOLLOW

- Nullable(`X`)
  - Can `X` derive the empty string?
- FIRST(`X`)
  - The set of terminals that can begin strings derived from `X`
- FOLLOW(`X`)
  - The set of terminals that can follow `X` in any derivation
  - If `X` is followed by end of file, include `$` in FOLLOW set
- Generalize Nullable, FIRST, and FOLLOW for entire grammar RHSs

Construction of Predictive Parser

- Enter production `X -> ` γ
  - In row `X`, column `T` for each `T` ∈ FIRST(γ)

- If γ is Nullable, enter production `X -> ` γ
  - In row `X`, column `T` for each `T` ∈ FOLLOW(`X`)

- Also include column for `$` (end of file)
  - In case any FOLLOW set includes `$`

- If a table slot contains multiple productions, the grammar is not LL(1)
- LL parse table can be translated into recursive-descent code or for use in table-driven parser

LL Parsing Example

- Grammar

```
Z -> d
   |  X Y Z
Y ->
   |  c
X -> Y
   |  a
```

- Nullable, FIRST, and FOLLOW

	Nullable	FIRST	FOLLOW
X	yes	a c	a c d
Y	yes	c	a c d
Z	no	a c d	$

LL Parsing Example, cont'd

- LL parse table

	a	c	d	$
X	X -> a X -> Y	X -> Y	X -> Y	
Y	Y ->	Y -> Y -> c	Y ->	
Z	Z -> XYZ	Z -> XYZ	Z -> d Z -> XYZ	

Grammar is not LL(1)

Single-Pass vs. Multi-Pass Compiler

- Multi-pass compiler
  - Main calls parser
  - Parser calls lexical analyzer as needed and returns a parse tree
  - Main calls semantic analysis, optimizers, code generation

- Single-pass compiler
  - Main calls parser
  - Parser drives the entire compilation
- Example from single-pass compiler

```
FunDec ::= Type ID Params
           {: add ID and parameters into symtab :}
           LBRACE StmtList RBRACE
           {: type-check body, generate code :} ;
```

Summary

- Parse recognizes context-free syntactic structures in token stream
- We use context-free grammar for describing syntax
- Bottom-up parsing
  - Parser generator translates grammar into table-driven parser
  - Parse engine is a push-down automaton, which consists of a DFA and a stack
  - Parser generators: yacc, bison, JavaCUP, ML-Yacc
  - Grammar classes: LR(0), SLR, LALR(1), LR(1), LR(k)
- Top-down parsing
  - Handwritten recursive-descent parser
  - Predictive parser (table-driven) or code generated by LL(k) parser generator
  - Parser generators: ANTLR, JavaCC
  - Grammar classes: LL(0), LL(1), LL(k)