Ch 5 Syntax Analysis – Part II

(Quick Look at Using Bison Top-Down Parsers)

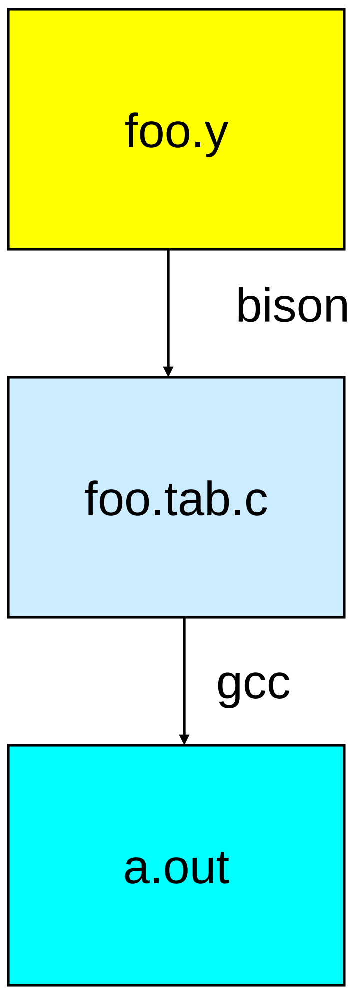
Yongjun Park

Hanyang University

Reading/Announcements

- **Reading: Section 4.4 (top-down parsing)**
- **Working example posted on webpage**
  - Converts expressions with infix notation to expression with prefix notation
  - Running the example
    - bison –d example.y
      - Creates example.tab.c and example.tab.h
    - flex example.l
      - Creates lex.yy.c
    - g++ example.tab.c lex.yy.c –lfl
      - g++ required here since user code uses C++ (new,<<)
    - a.out < ex_input.txt

Bison Overview



yyparse() is
main routine

Format of .y file
(same structure as lex file)

declarations

%%

rules

%%

support code

Declarations Section

- **User types: As in flex, these are in a section bracketed by “%{“ and “%}”**
- **Tokens – terminal symbols of the grammar**
  - %token terminal1 terminal2 ...
    - Values for tokens assigned sequentially after all ASCII characters
  - or %token terminal1 val1 terminal2 val2 ...
- **<u>Tip</u> – Use ‘-d’ option in bison to get foo.tab.h that contains the token definitions that can be included in the flex file**

Declarations (2)

- **Start symbol**
  - %start non-terminal
- **Associativity – (left, right or none)**
  - %left TK_PLUS
  - %right TK_EXPONENT
  - %nonassoc TK_LESSTHAN
- **Precedence**
  - Order of the directives specifies precedence
  - %prec changes the precedence of a rule

Declarations (3)

- **Attribute values – information associated with all terminal/non-terminal symbols – passed from the lexer**
  - %union {
    - int ival;
    - char *name;
    - double dval;
  - }
  - Becomes YYSTYPE
- **Symbol attributes – types of non-terminals**
  - %type<union_entry>non_terminal
  - Example: %type<ival>IntNumber

Values Used by yyparse()

- **Error function**
  - yyerror(char *s);
- **Last token value**
  - yylval of type YYSTYPE (%union decl)
- **Setting yylval in flex**
  - [a-z]   {yylval.ival = yytext[0] – 'a'; return TK_NAME;}
- **Then, yylval is available in bison**
  - But in a strange way

Rules Section

- **Every name appearing that has not been declared is a non-terminal**
- **Productions**
  - non-terminal : first_production | second_production | ... ;
  - ε production has the form
    - non-terminal : ;
    - Thus you can say, foo: production1 | /* nothing*/ ;
  - Adding actions
    - non-terminal : RHS {action routine} ;
    - Action called before LHS is pushed on parse stack

Attribute Values (aka $ vars)

- **Each terminal/non-terminal has one**
- **Denoted by $n where n is its rank in the rule starting by 1**
  - $$ = LHS
  - $1 = first symbol of the RHS
  - $2 = second symbol, etc.
  - Note, semantic actions have values too!!!
    - A: B {...} C {...} ;
    - C’s value is denoted by $3

Example .y File – Partial Calculator

```
%union {
        int       value;
        char      *symbol;
}
%type<value> exp term factor
%type<symbol> ident

...

exp : exp '+' term {$$ = $1 + $3; };
              /* Note, $1 and $3 are ints here */
factor : ident {$$ = lookup(symbolTable, $1); };
              /* Note, $1 is a char* here */
```

Conflicts

- **Bison reports the number of shift/reduce and reduce/reduce conflicts found**
- **Shift/reduce conflicts**
  - Occurs when there are 2 possible parses for an input string, one parse completes a rule (reduce) and one does not (shift)
  - Example
    - e: 'X' | e '+' e ;\
    - "X+X+X" has 2 possible parses "(X+X)+X" or "X+(X+X)"

Conflicts (2)

- **Reduce/reduce conflict occurs when the same token could complete 2 different rules**
  - Example
    - prog : proga | progb ;
    - proga : ‘X’ ;
    - progb : ‘X’ ;
    - “X” can either be a proga or progb
  - Ambiguous grammar!!

Ambiguity Review: Class Problem

S → if (E) S
S → if (E) S else S
S → other

Anything wrong with this grammar?

Grammar for Closest-if Rule

- **Want to rule out: if (E) <u>if (E) S</u> else S**
- **Impose that unmatched "if" statements occur only on the "else" clauses**
  - statement → matched | unmatched
  - matched → if (E) matched else matched |
    other
  - unmatched → if (E) statement |
    if (E) matched else unmatched

Parsing Top-Down

Goal: construct a leftmost derivation of string while reading in sequential token stream

S → E + S | E
E → num | (S)

Partly-derived String	Lookahead	parsed part unparsed part
→E + S	(	(1+2+(3+4))+5
→(S) + S	1	(1+2+(3+4))+5
→(E+S)+S	1	(1+2+(3+4))+5
→(1+S)+S	2	(1+2+(3+4))+5
→(1+E+S)+S	2	(1+2+(3+4))+5
→(1+2+S)+S	2	(1+2+(3+4))+5
→(1+2+E)+S	(	(1+2+(3+4))+5
→(1+2+(S))+S	3	(1+2+(3+4))+5
→(1+2+(E+S))+S	3	(1+2+(3+4))+5

→ …

Problem with Top-Down Parsing

Want to decide which production to apply based on next symbol

S → E + S | E
E → num | (S)

Ex1: "(1)"
Ex2: "(1)+2"

S → E → (S) → (E) → (1)
S → E+S → (S)+S → (E)+S
→ (1)+E → (1)+2

How did you know to pick E+S in Ex2, if you picked E followed by (S), you couldn't parse it?

Grammar is Problem

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

- **This grammar cannot be parsed top-down with only a single look-ahead symbol!**
- **Not LL(1) = Left-to-right scanning, Left-most derivation, 1 look-ahead symbol**
- **Is it LL(k) for some k?**
- **If yes, then can rewrite grammar to allow top-down parsing: create LL(1) grammar for same language**

Making a Grammar LL(1)

S → E + S
S → E
E → num
E → (S)

- Problem: Can't decide which S production to apply until we see the symbol after the first expression

↓

S → ES'
S' → ε
S' → +S
E → num
E → (S)

- Left-factoring: Factor common S prefix, add new non-terminal S' at decision point. S' derives (+S)*

- Also: Convert left recursion to right recursion

Parsing with New Grammar

S → ES' S' → ε | +S   E → num | (S)

Partly-derived String	Lookahead	parsed part unparsed part
→ES'	(	(1+2+(3+4))+5
→(S)S'	1	(1+2+(3+4))+5
→(ES')S'	1	(1+2+(3+4))+5
→(1S')S'	+	(1+2+(3+4))+5
→(1+ES')S'	2	(1+2+(3+4))+5
→(1+2S')S'	+	(1+2+(3+4))+5
→(1+2+S)S'	(	(1+2+(3+4))+5
→(1+2+ES')S'	(	(1+2+(3+4))+5
→(1+2+(S)S')S'	3	(1+2+(3+4))+5
→(1+2+(ES')S')S'	3	(1+2+(3+4))+5
→(1+2+(3S')S')S'	+	(1+2+(3+4))+5
→(1+2+(3+E)S')S'	4	(1+2+(3+4))+5

→ ...

Class Problem

Are the following grammars LL(1)?

$S \rightarrow Abc \mid aAcb$
$A \rightarrow b \mid c \mid \varepsilon$

$S \rightarrow aAS \mid b$
$A \rightarrow a \mid bSA$

Predictive Parsing

- **LL(1) grammar:**
  - For a given non-terminal, the lookahead symbol uniquely determines the production to apply
  - Top-down parsing = predictive parsing
  - Driven by predictive parsing table of
    - non-terminals x terminals → productions

Parsing with Table

S → ES' S' → ε | +S E → num | (S)

Partly-derived String	Lookahead	parsed part unparsed part
→ES'	(	(1+2+(3+4))+5
→(S)S'	1	(1+2+(3+4))+5
→(ES')S'	1	(1+2+(3+4))+5
→(1S')S'	+	(1+2+(3+4))+5
→(1+ES')S'	2	(1+2+(3+4))+5
→(1+2S')S'	+	(1+2+(3+4))+5

	num	+	(	)	$
S	→ ES'		→ ES'		
S'		→ +S		→ ε	→ ε
E	→ num		→ (S)		

($ is a special "endmarker" to indicate the end of file.)

How to Implement This?

- Table can be converted easily into a recursive descent parser

- 3 procedures: parse_S(), parse_S'(), and parse_E()

	num	+	(	)	$
S	→ ES'		→ ES'		
S'		→ +S		→ ε	→ ε
E	→ num		→ (S)		

Recursive-Descent Parser

lookahead token

```
void parse_S() {
    switch (token) {
            case num: parse_E(); parse_S'(); return;
            case '(': parse_E(); parse_S'(); return;
            default: ParseError();
    }
}
```

	num	+	(	)	$
S	→ ES'		→ ES'		
S'		→ +S		→ ε	→ ε
E	→ num		→ (S)		

Recursive-Descent Parser (2)

```
void parse_S'() {
    switch (token) {
        case '+': token = input.read(); parse_S(); return;
        case ')': return;
        case EOF: return;
        default: ParseError();
    }
}
```

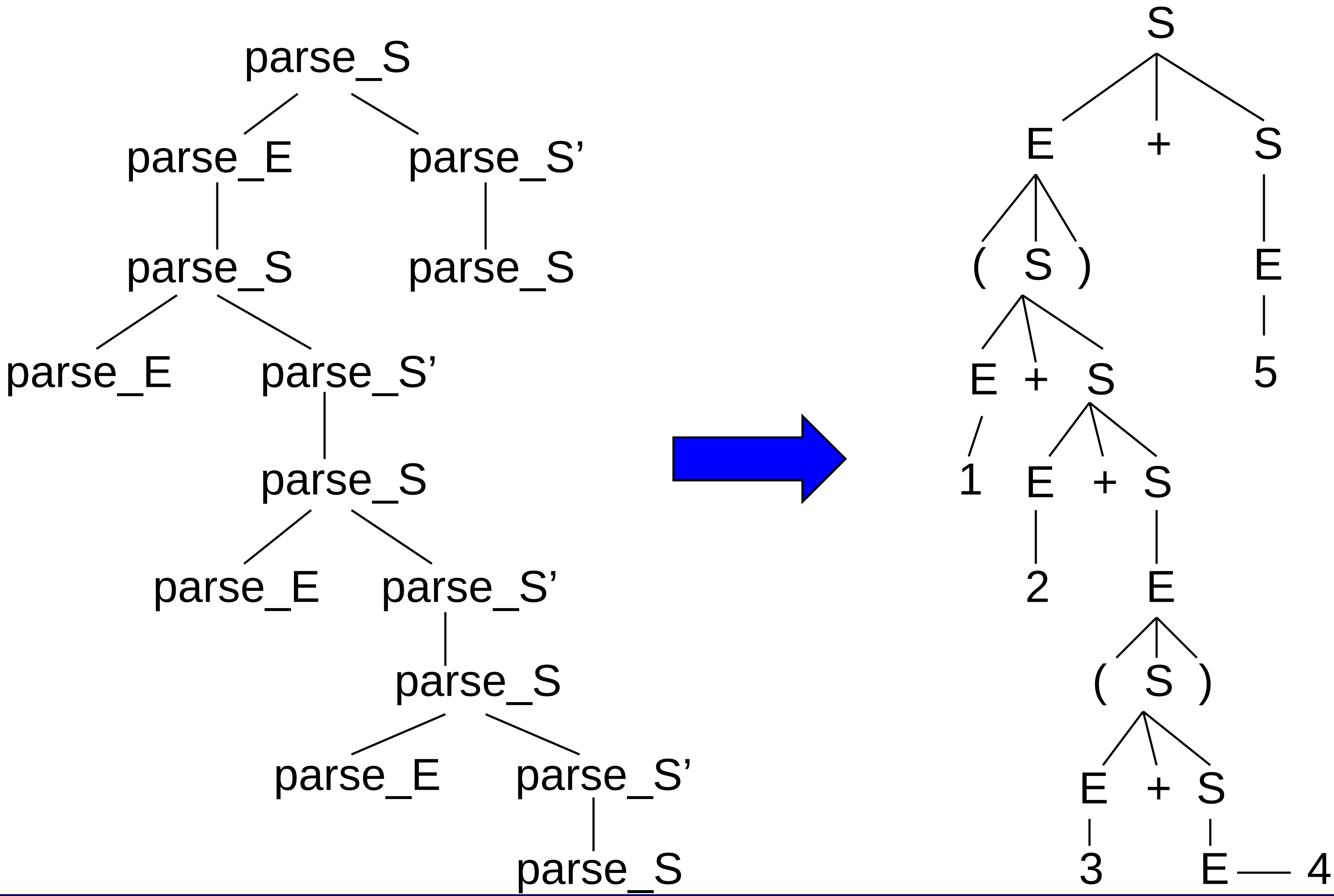
	num	+	(	)	$
S	→ ES'		→ ES'		
S'		→ +S		→ ε	→ ε
E	→ num		→ (S)		

Recursive-Descent Parser (3)

```
void parse_E() {
    switch (token) {
            case number: token = input.read(); return;
            case '(': token = input.read(); parse_S();
                        if (token != ')') ParseError();
                        token = input.read(); return;
            default: ParseError();
    }
}
```

	num	+	(	)	$
S	→ ES'		→ ES'		
S'		→ +S		→ ε	→ ε
E	→ num		→ (S)		

Call Tree = Parse Tree



How to Construct Parsing Tables?

Needed: Algorithm for automatically generating a predictive parse table from a grammar

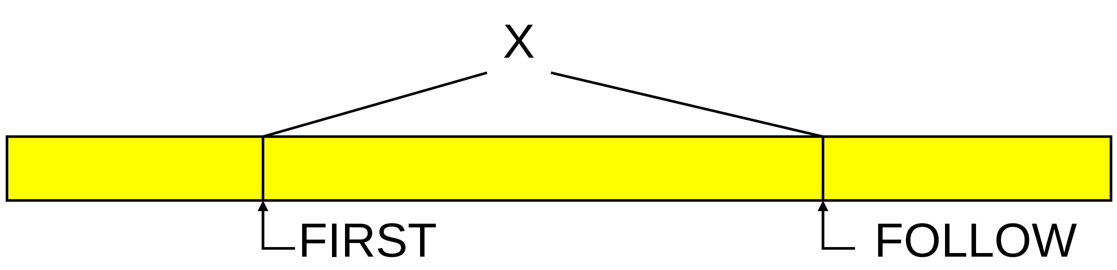
S → ES'
S' → ε | +S
E → number | (S)

??

	num	+	(	)	$
S	ES'		ES'		
S'		+S		ε	ε
E	num		(S)		

Constructing Parse Tables

- **Can construct predictive parser if:**
  - For every non-terminal, every lookahead symbol can be handled by at most 1 production
- **FIRST(β) for an arbitrary string of terminals and non-terminals β is:**
  - Set of symbols that might begin the fully expanded version of β
- **FOLLOW(X) for a non-terminal X is:**
  - Set of symbols that might follow the derivation of X in the input stream



Parse Table Entries

- Consider a production X → β
- Add → β to the X row for each symbol in FIRST(β)
- If β can derive ε (β is nullable), add → β for each symbol in FOLLOW(X)
- Grammar is LL(1) if no conflicting entries

S → ES'
S' → ε | +S
E → number | (S)

	num	+	(	)	$
S	ES'		ES'		
S'		+S		ε	ε
E	num		(S)		