

Syntax Analysis – Part II

Quick Look at Using Bison

Top-Down Parsers

EECS 483 – Lecture 5

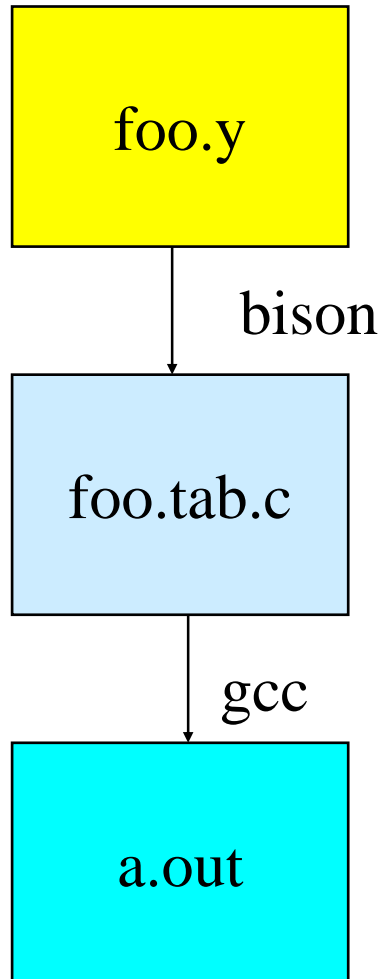
University of Michigan

Wednesday, September 20, 2006

Reading/Announcements

- ❖ Reading: Section 4.4 (top-down parsing)
- ❖ Working example posted on webpage
 - » Converts 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 ...
- ❖ Tip – 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
 - » $\$\$ = \text{LHS}$
 - » $\$1 = \text{first symbol of the RHS}$
 - » $\$2 = \text{second symbol, etc.}$
 - » Note, semantic actions have values too!!!
 - $A: B \{ \dots \} C \{ \dots \} ;$
 - 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

$S \rightarrow \text{if } (E) S$

$S \rightarrow \text{if } (E) S \text{ else } S$

$S \rightarrow \text{other}$

Anything wrong with this grammar?

Grammar for Closest-if Rule

- ❖ Want to rule out: $\text{if (E) if (E) S else S}$
- ❖ Impose that unmatched “if” statements occur only on the “else” clauses
 - » statement \rightarrow matched | unmatched
 - » matched \rightarrow if (E) matched else matched | other
 - » unmatched \rightarrow if (E) statement | if (E) matched else unmatched

Parsing Top-Down

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

$$S \rightarrow E + S \mid E$$

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

Partly-derived String	Lookahead	parsed part unparsed part
$\rightarrow E + S$	((1+2+(3+4))+5
$\rightarrow (S) + S$	1	(1+2+(3+4))+5
$\rightarrow (E+S)+S$	1	(1+2+(3+4))+5
$\rightarrow (1+S)+S$	2	(1+2+(3+4))+5
$\rightarrow (1+E+S)+S$	2	(1+2+(3+4))+5
$\rightarrow (1+2+S)+S$	2	(1+2+(3+4))+5
$\rightarrow (1+2+E)+S$	((1+2+(3+4))+5
$\rightarrow (1+2+(S))+S$	3	(1+2+(3+4))+5
$\rightarrow (1+2+(E+S))+S$	3	(1+2+(3+4))+5
$\rightarrow \dots$		

Problem with Top-Down Parsing

Want to decide which production to apply based on next symbol

$S \rightarrow E + S \mid E$

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

Ex1: “(1)”

$S \rightarrow E \rightarrow (S) \rightarrow (E) \rightarrow (1)$

Ex2: “(1)+2”

$S \rightarrow \underline{E+S} \rightarrow (S)+S \rightarrow (E)+S$
 $\rightarrow (1)+E \rightarrow (1)+2$

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

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

- ❖ 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

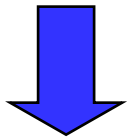
Making a Grammar LL(1)

$S \rightarrow E + S$

$S \rightarrow E$

$E \rightarrow \text{num}$

$E \rightarrow (S)$



$S \rightarrow ES'$

$S' \rightarrow \epsilon$

$S' \rightarrow +S$

$E \rightarrow \text{num}$

$E \rightarrow (S)$

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

$S \rightarrow ES'$	$S' \rightarrow \varepsilon \mid +S$	$E \rightarrow \text{num} \mid (S)$
Partly-derived String	Lookahead	parsed part unparsed part
$\rightarrow ES'$	((1+2+(3+4))+5
$\rightarrow (S)S'$	1	(1+2+(3+4))+5
$\rightarrow (ES')S'$	1	(1+2+(3+4))+5
$\rightarrow (1S')S'$	+	(1+2+(3+4))+5
$\rightarrow (1+ES')S'$	2	(1+2+(3+4))+5
$\rightarrow (1+2S')S'$	+	(1+2+(3+4))+5
$\rightarrow (1+2+S)S'$	((1+2+(3+4))+5
$\rightarrow (1+2+ES')S'$	((1+2+(3+4))+5
$\rightarrow (1+2+(S)S')S'$	3	(1+2+(3+4))+5
$\rightarrow (1+2+(ES')S')S'$	3	(1+2+(3+4))+5
$\rightarrow (1+2+(3S')S')S'$	+	(1+2+(3+4))+5
$\rightarrow (1+2+(3+E)S')S'$	4	(1+2+(3+4))+5
$\rightarrow \dots$		

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 \rightarrow productions

Parsing with Table

$S \rightarrow ES'$

$S' \rightarrow \varepsilon \mid +S$

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

Partly-derived String

Lookahead

parsed part **unparsed part**

$\rightarrow ES'$

(

(1+2+(3+4))+5

$\rightarrow (S)S'$

1

(1+2+(3+4))+5

$\rightarrow (ES')S'$

1

(1+2+(3+4))+5

$\rightarrow (1S')S'$

+

(1+2+(3+4))+5

$\rightarrow (1+ES')S'$

2

(1+2+(3+4))+5

$\rightarrow (1+2S')S'$

+

(1+2+(3+4))+5

	num	+	()	\$
S	$\rightarrow ES'$		$\rightarrow ES'$		
S'		$\rightarrow +S$		$\rightarrow \varepsilon$	$\rightarrow \varepsilon$
E	$\rightarrow \text{num}$		$\rightarrow (S)$		

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	$\rightarrow ES'$		$\rightarrow ES'$		
S'		$\rightarrow +S$		$\rightarrow \epsilon$	$\rightarrow \epsilon$
E	$\rightarrow \text{num}$		$\rightarrow (S)$		

Recursive-Descent Parser

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

lookahead token

	num	+	()	\$
S	$\rightarrow ES'$		$\rightarrow ES'$		
S'		$\rightarrow +S$		$\rightarrow \epsilon$	$\rightarrow \epsilon$
E	$\rightarrow \text{num}$		$\rightarrow (S)$		

Recursive-Descent Parser (2)

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

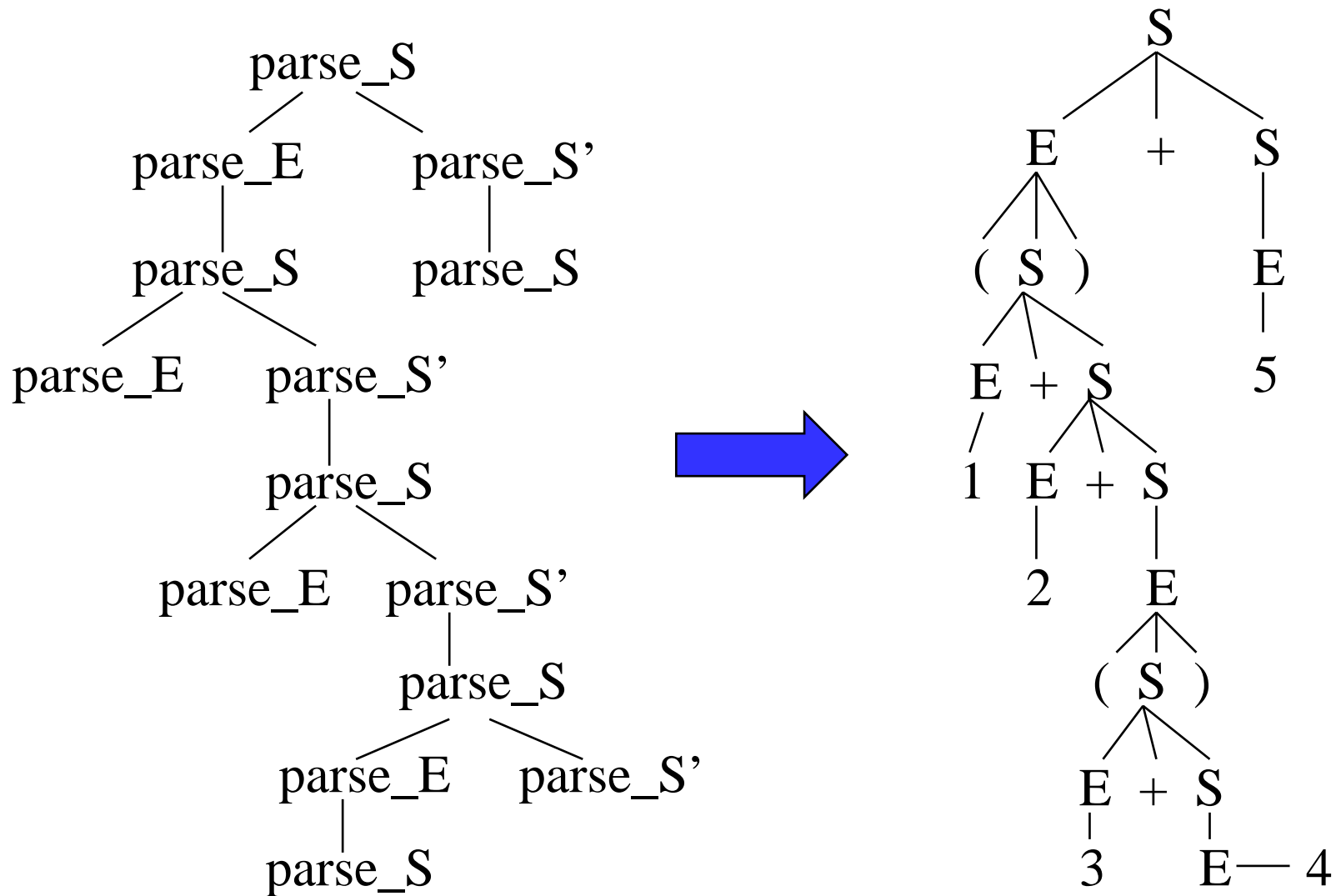
	num	+	()	\$
S	$\rightarrow ES'$		$\rightarrow ES'$		
S'		$\rightarrow +S$		$\rightarrow \epsilon$	$\rightarrow \epsilon$
E	$\rightarrow \text{num}$		$\rightarrow (S)$		

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

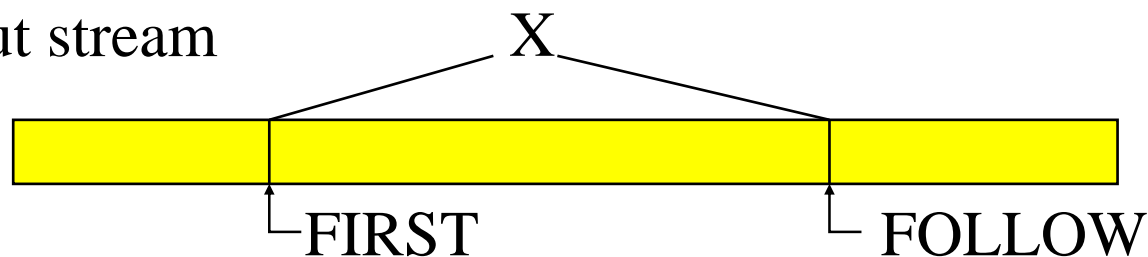
$S \rightarrow ES'$
 $S' \rightarrow \varepsilon \mid +S$
 $E \rightarrow \text{number} \mid (S)$



	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
- ❖ $FIRST(\beta)$ 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 $X \rightarrow \beta$
- ❖ Add $\rightarrow \beta$ to the X row for each symbol in $\text{FIRST}(\beta)$
- ❖ If β can derive ϵ (β is nullable), add $\rightarrow \beta$ for each symbol in $\text{FOLLOW}(X)$
- ❖ Grammar is LL(1) if no conflicting entries

$S \rightarrow ES'$

$S' \rightarrow \epsilon \mid +S$

$E \rightarrow \text{number} \mid (S)$

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