Bottom-up Parsing

Saumya Debray

CSc 453

Oct 20, 2016

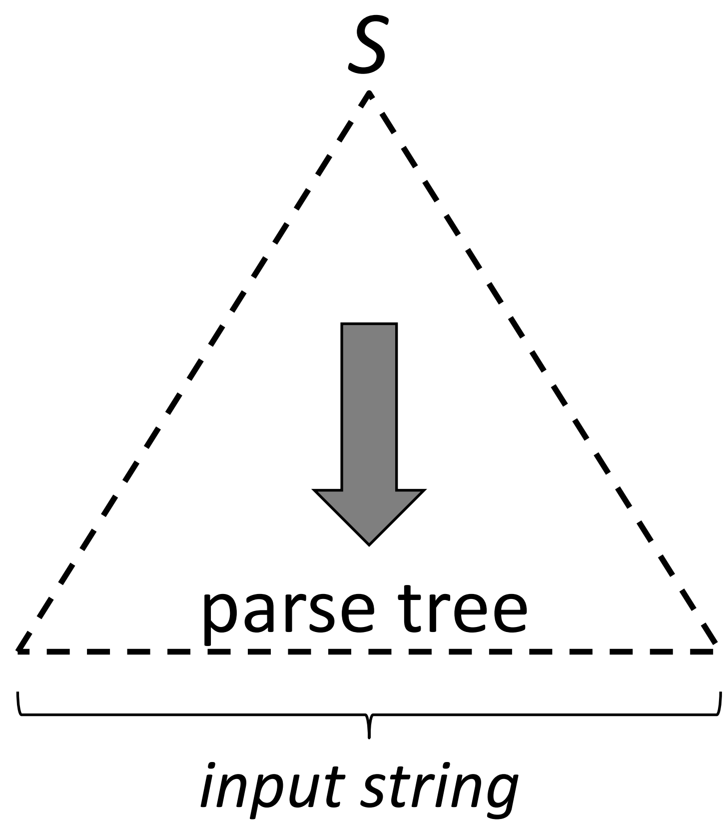
BASIC CONCEPTS

Parsing: Top-down vs. Bottom-up

top-down parsing:

- starts with the start symbol
- identifies a derivation sequence that produces the input string

→ grows the parse tree from the top down

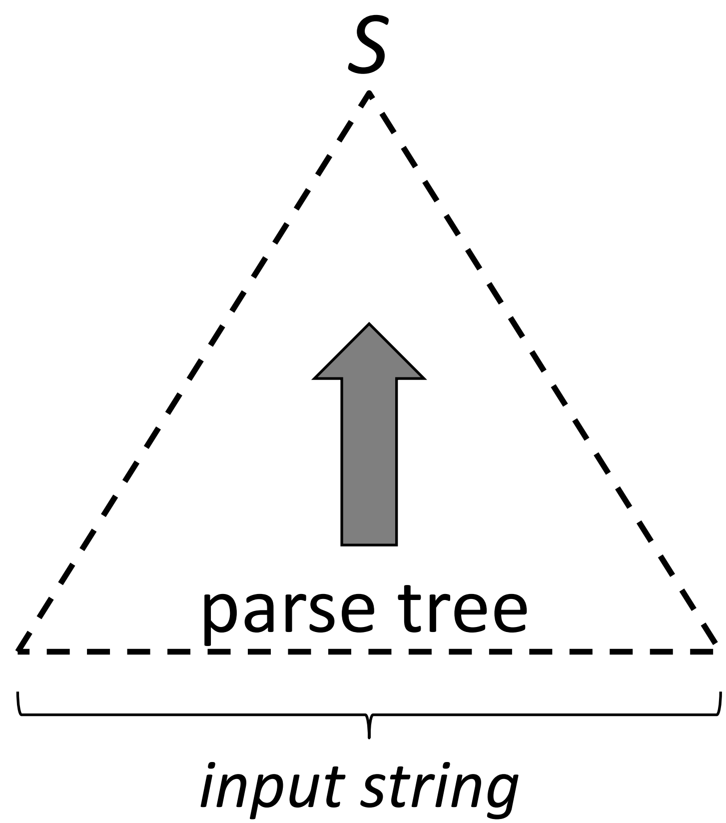


Parsing: Top-down vs. Bottom-up

bottom-up parsing:

- starts with the input tokens
- identifies a "backwards derivation sequence" that produces the start symbol

→ assembles the parse tree from the bottom up



Why use a bottom-up parser?

+ Can handle some grammars that recursive-descent parsers cannot
  – don't need to rewrite the grammar
+ Created using parser generators (e.g., bison)
  – speeds up parser creation

– Provides less control over the parsing process than recursive-descent parsers
  – e.g., error messages are less helpful

Doing derivations backwards

Grammar:

$S \longrightarrow \mathbf{a}AB\mathbf{e}$

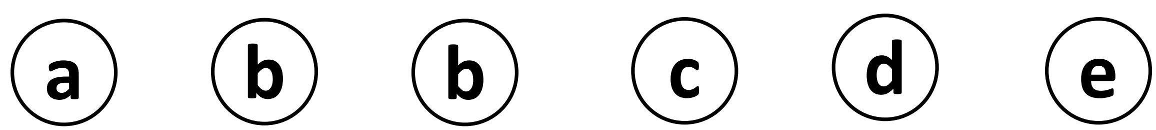
$A \longrightarrow A\mathbf{bc}$

$A \longrightarrow \mathbf{b}$

$B \longrightarrow \mathbf{d}$

Input:

abbcde



Doing derivations backwards

Grammar:

$S \rightarrow \mathbf{a}AB\mathbf{e}$

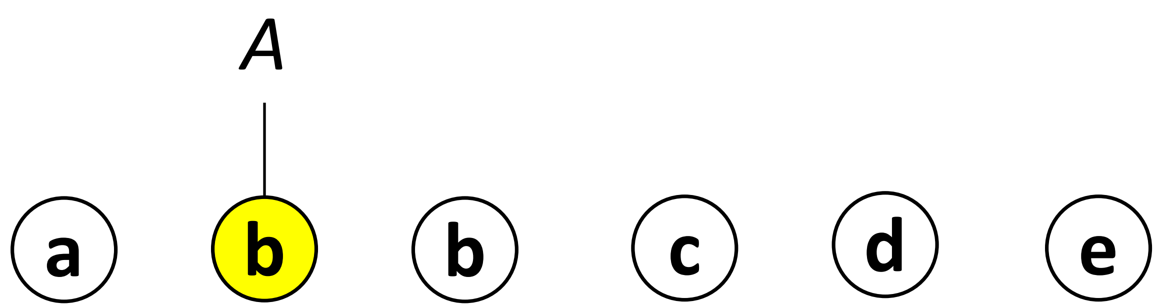
$A \rightarrow A\mathbf{bc}$

$A \rightarrow \mathbf{b}$

$B \rightarrow \mathbf{d}$

Input:

abbcde



Doing derivations backwards

Grammar:

$S \rightarrow \mathbf{a}AB\mathbf{e}$

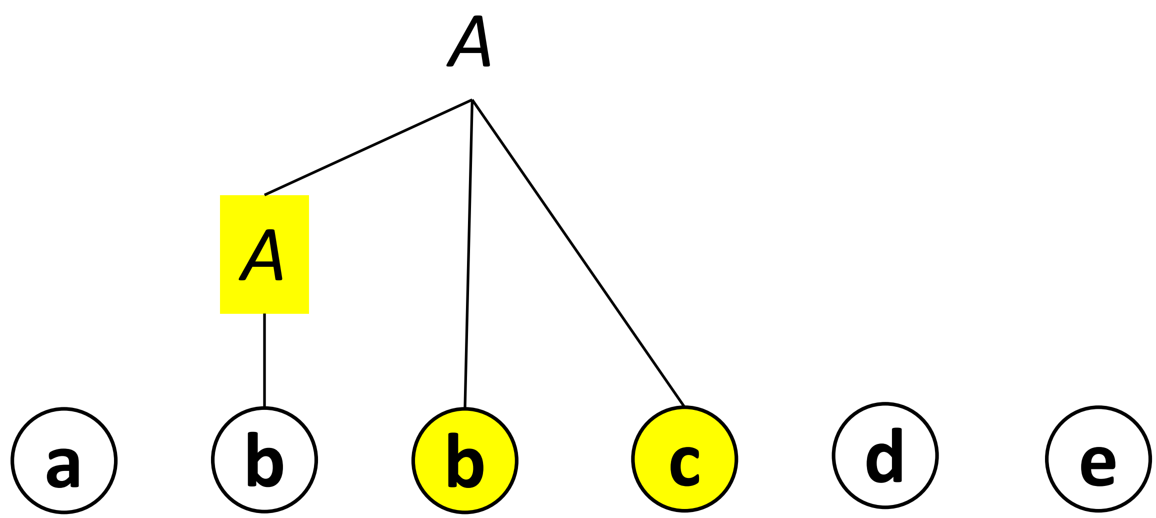
$A \rightarrow A\mathbf{bc}$

$A \rightarrow \mathbf{b}$

$B \rightarrow \mathbf{d}$

Input:

abbcde



Doing derivations backwards

Grammar:

$S \rightarrow$ **a***AB***e**

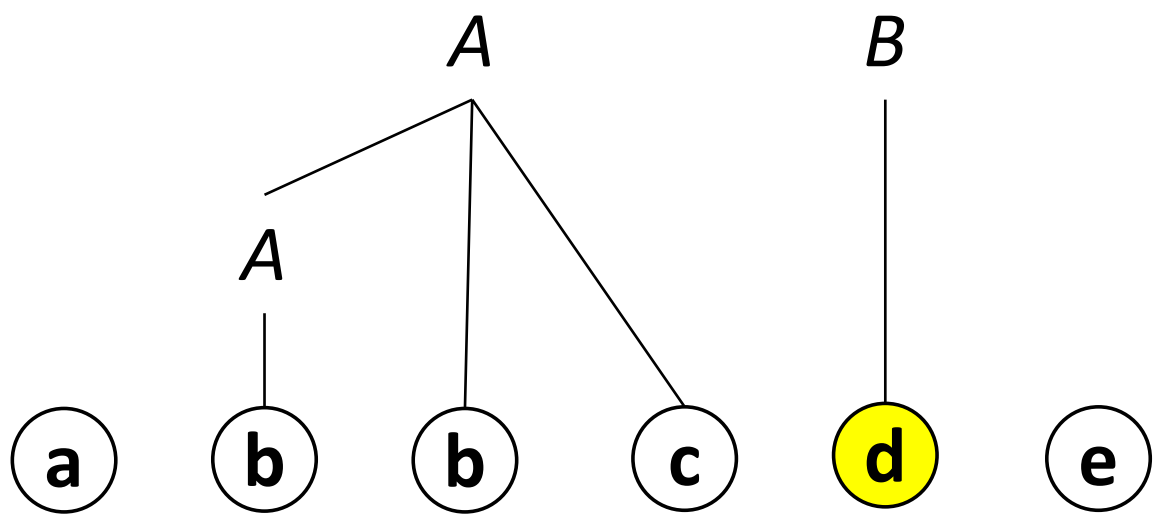
$A \rightarrow$ *A***bc**

$A \rightarrow$ **b**

$B \rightarrow$ **d**

Input:

abbcde



Doing derivations backwards

Grammar:

$S \rightarrow \mathbf{a}A B\mathbf{e}$

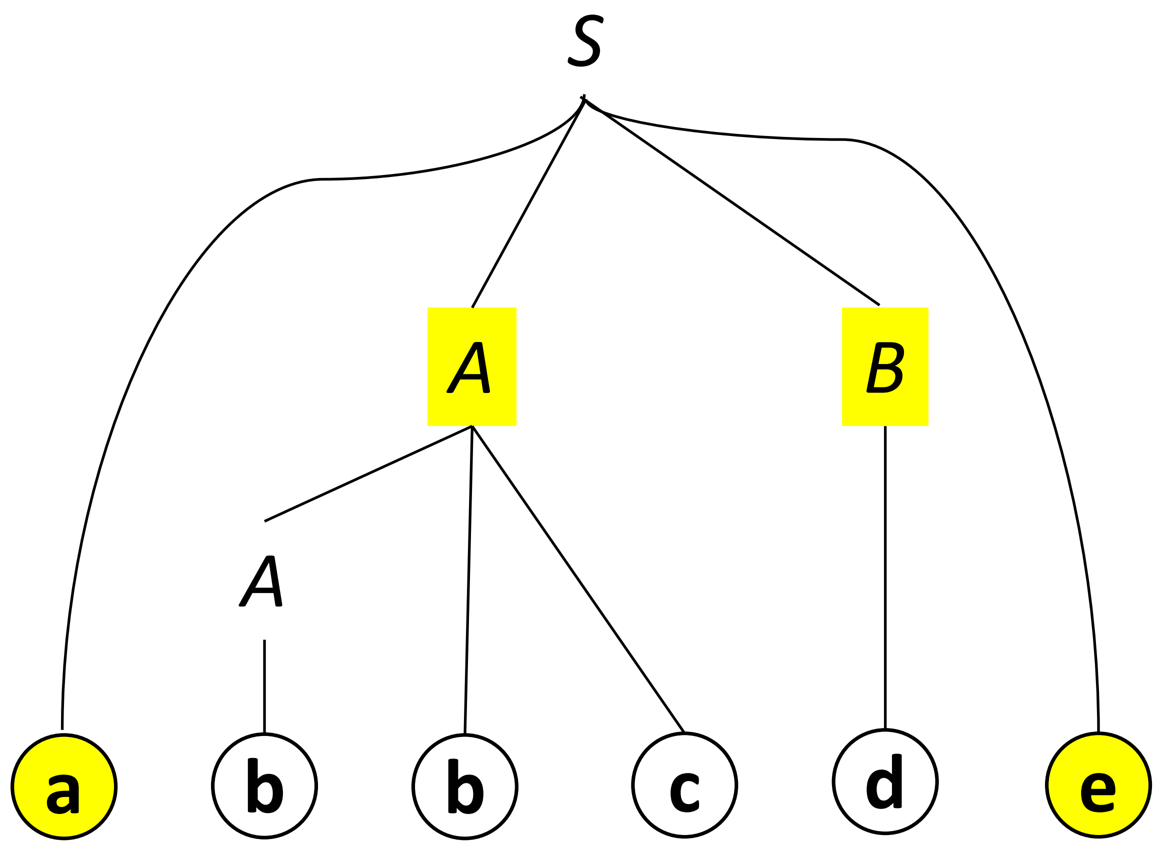
$A \rightarrow A\mathbf{bc}$

$A \rightarrow \mathbf{b}$

$B \rightarrow \mathbf{d}$

Input:

abbcde



Doing derivations backwards

Grammar:

$S \rightarrow$ **a***AB***e**

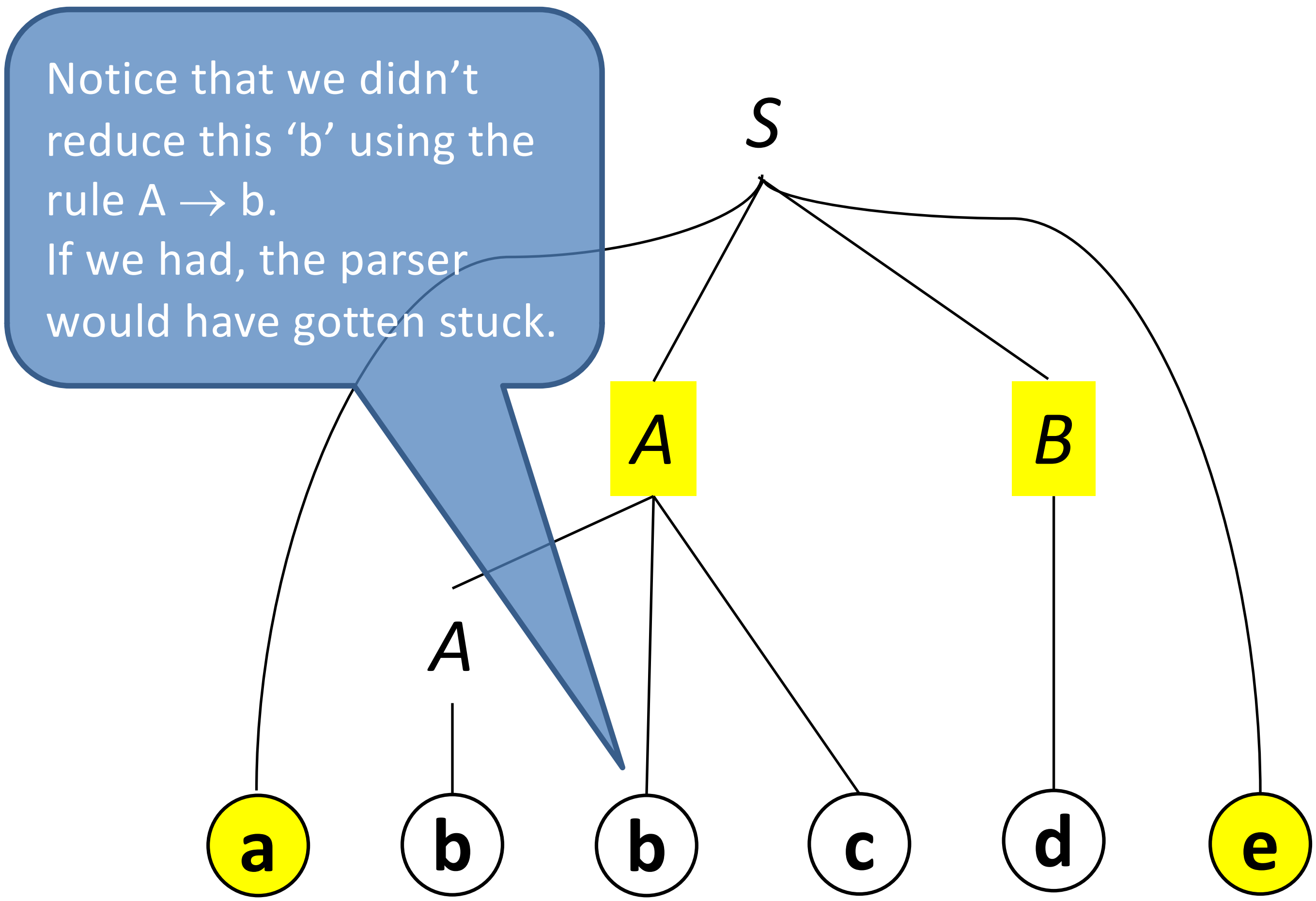
$A \rightarrow$ *A***bc**

$A \rightarrow$ **b**

$B \rightarrow$ **d**

Input:

abbcde



SHIFT-REDUCE PARSING

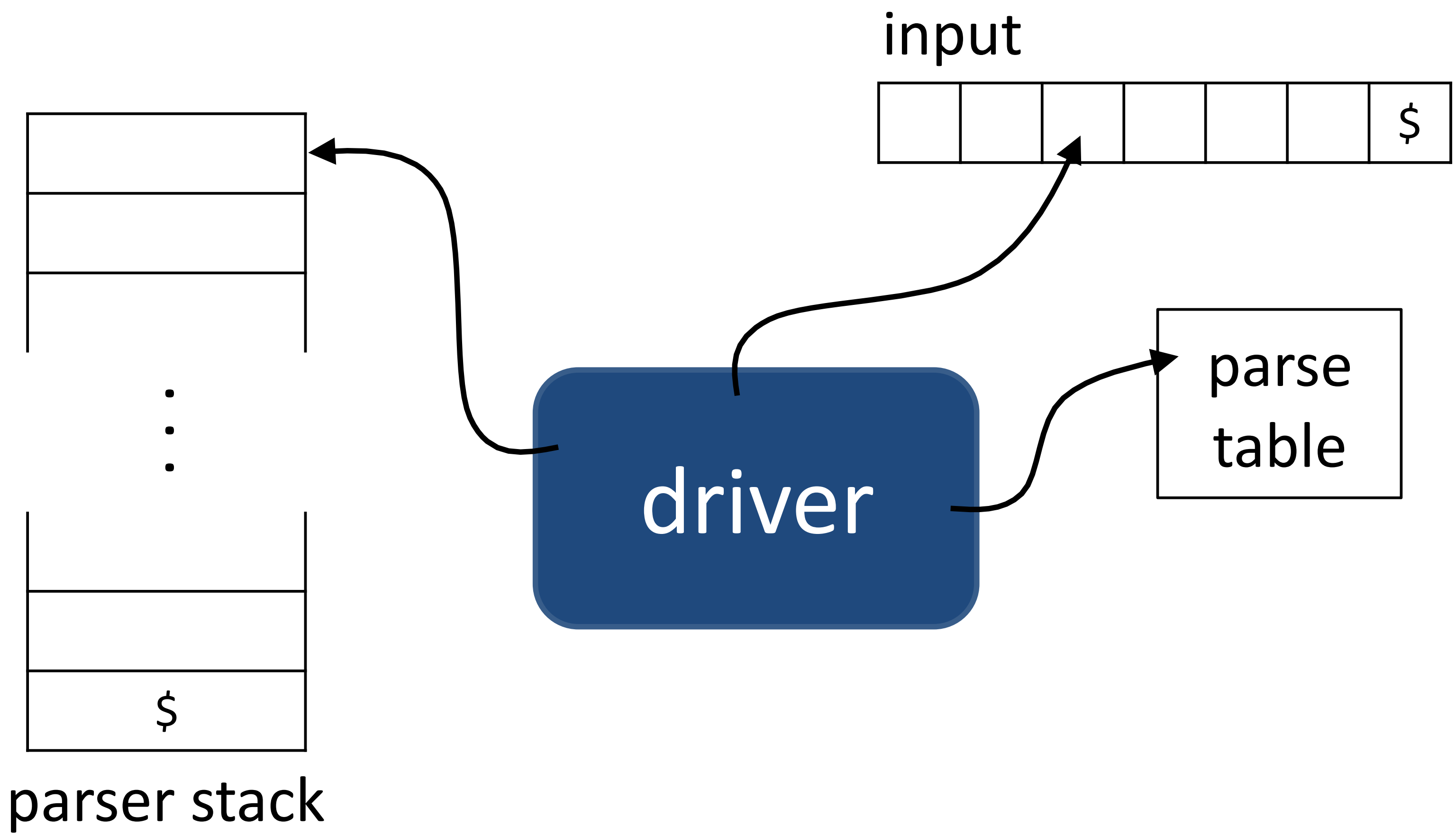
Shift-reduce parsing

- An instance of bottom-up parsing
- *Basic idea*: repeat
    1. in the string being processed, find a substring α such that $A \rightarrow \alpha$ is a production
    2. replace α by *A* (i.e., reverse a derivation step)

  until we get the start symbol.
- *Technical issues*: Figuring out
    1. which substring to replace; and
    2. which production to reduce with.

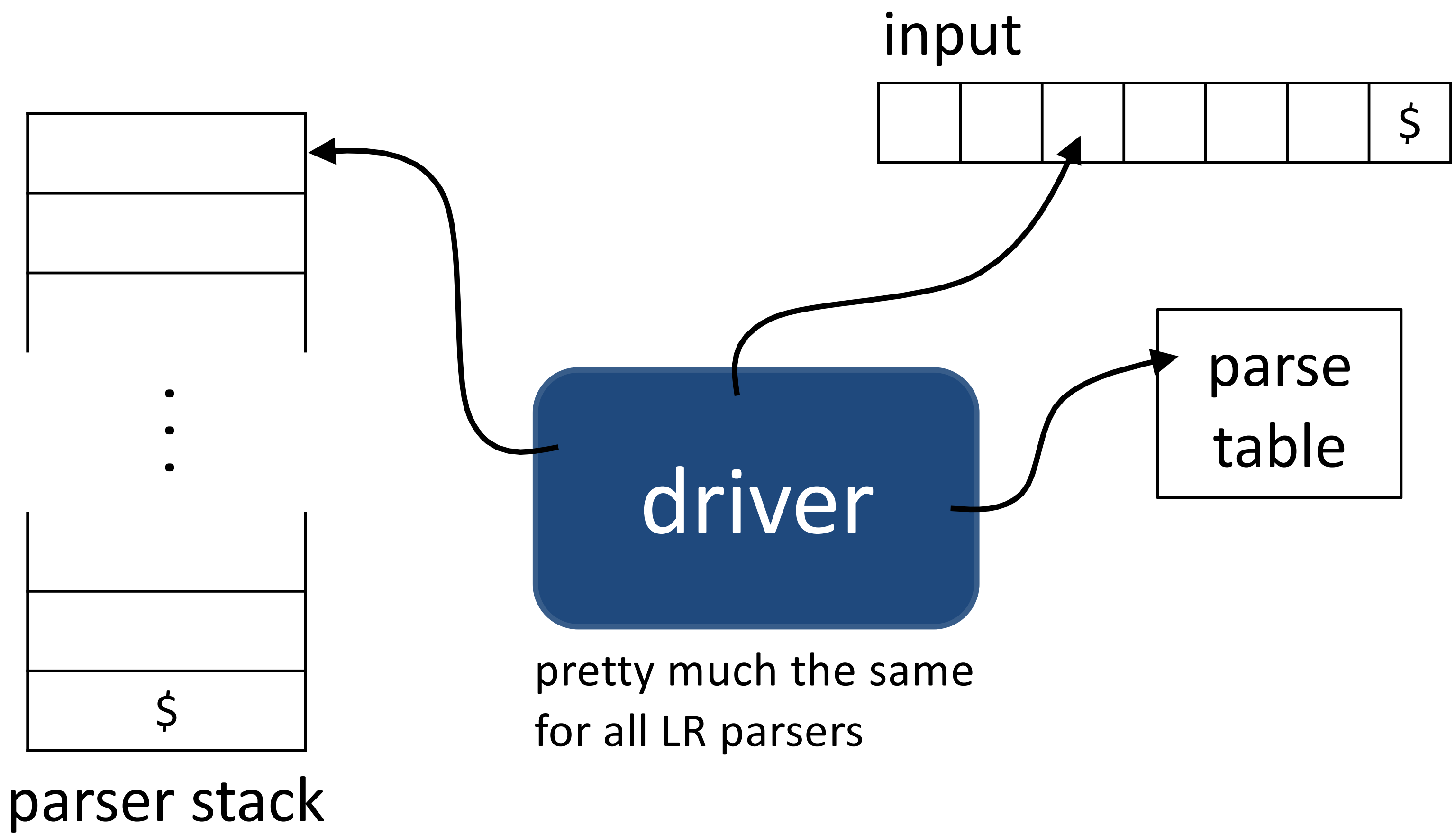
encoded in the *parse table*

LR parsing*



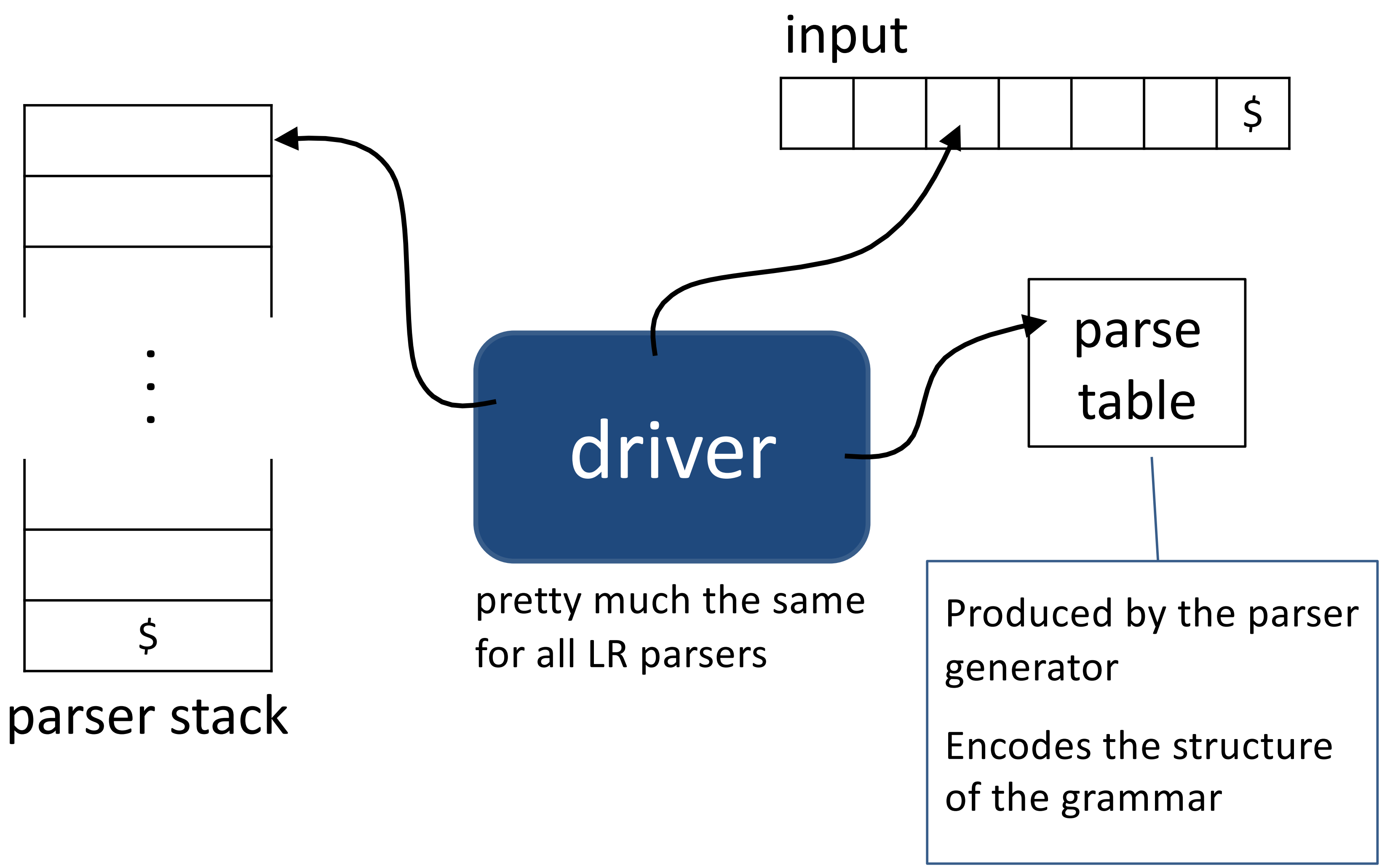
* LR parsing: a particular type of shift-reduce parsing

LR parsing



pretty much the same
for all LR parsers

LR parsing


input
$
parse
table
driver
$
parser stack
pretty much the same
for all LR parsers
Produced by the parser
generator
Encodes the structure
of the grammar

LR parsing

- *Data Structures*:
  - a stack, its bottom marked by ‘**$**’. Initially empty.
  - the input string, its right end marked by ‘**$**’
- *Actions*:

  repeat

  1. *Shift* some (≥ 0) symbols from the input string onto the stack, until parse table says to reduce.
  2. *Reduce* β to the LHS of the appropriate production.

  until ready to accept.

  - *Acceptance*: when input is empty and stack contains only the start symbol.

Example

Stack (→)	***Input***	***Action****
$	**abbcde$**	shift
$a	**bbcde$**	shift
$ab	**bcde$**	reduce: *A* → **b**
$a*A*	**bcde$**	shift
$a*A***b**	**cde$**	shift
$a*A***bc**	**de$**	reduce: *A* → *A***bc**
$a*A*	**de$**	shift
$a*A***d**	**e$**	reduce: *B* → **d**
$a*AB*	**e$**	shift
$a*AB***e**	**$**	reduce: *S* → **a***AB***e**
$*S*	**$**	accept

Grammar :

S → **a***AB***e**

A → *A***bc** | **b**

B → **d**

* from parse table

PARSER GENERATORS

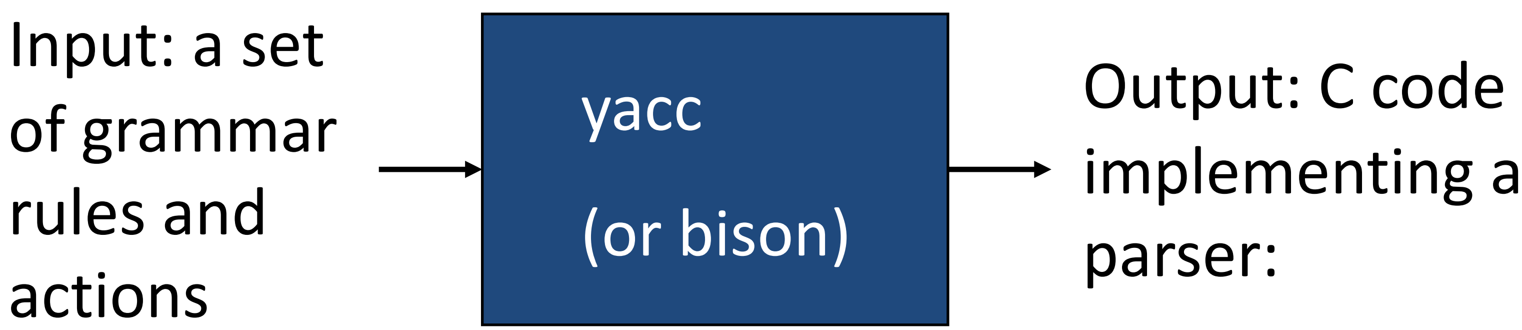
Parser generators

- Constructing LR parsers by hand is painful
  – large no. of parser states

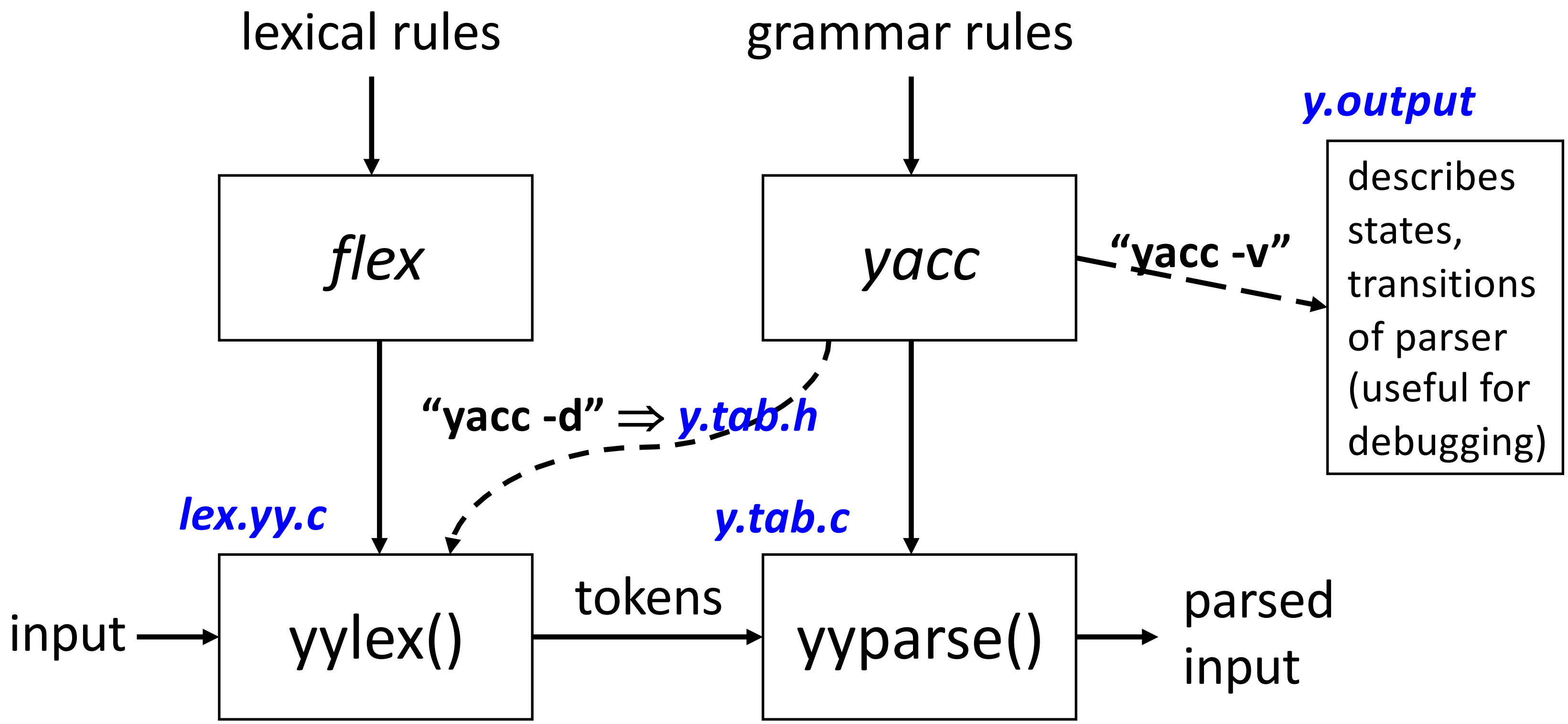
- Parser generators (e.g., yacc, bison) take a specification of a grammar and write out the C code for the parser
  – convenient
  – debugging can be tedious

Parser generators

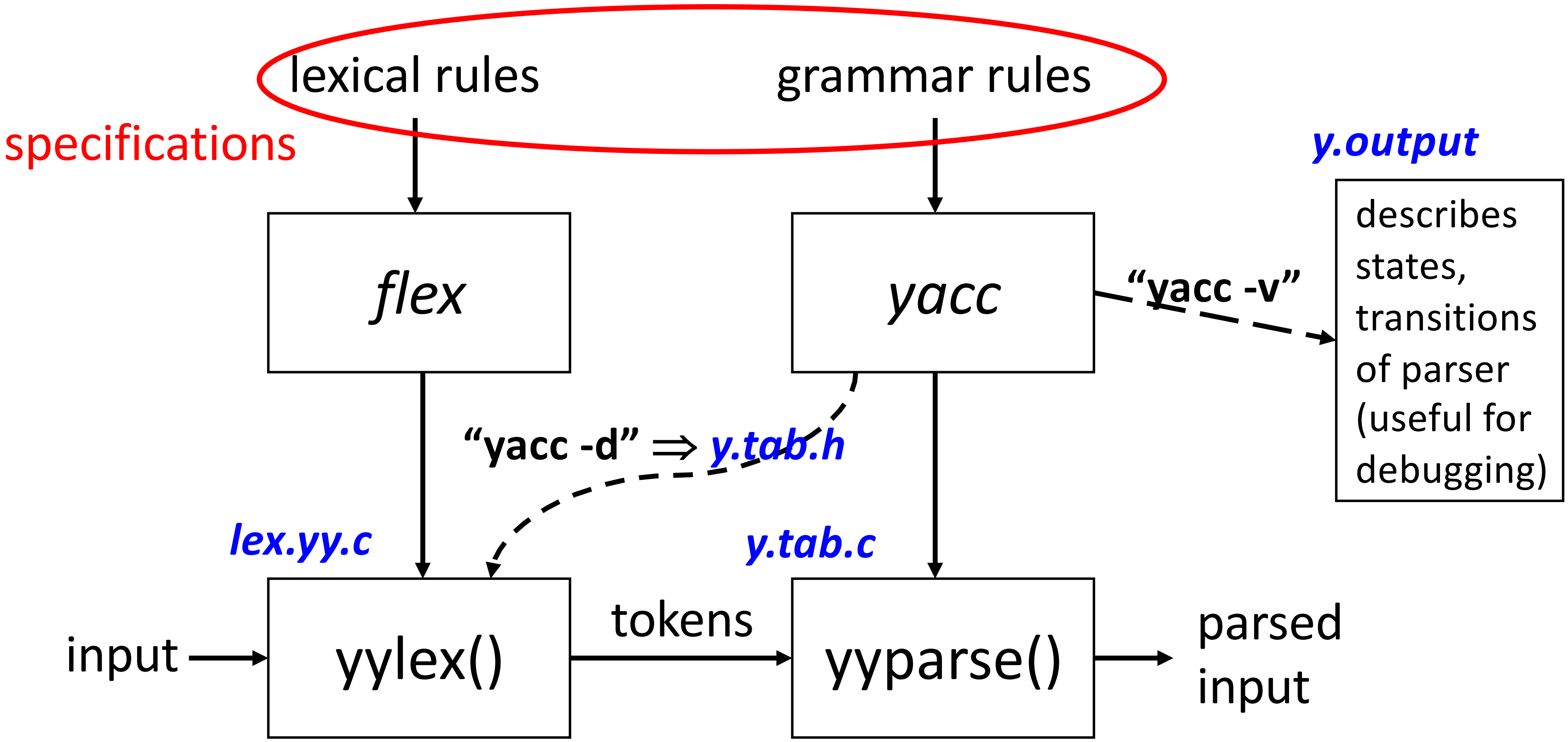
- Takes a specification for a context-free grammar
- Produces code for a parser



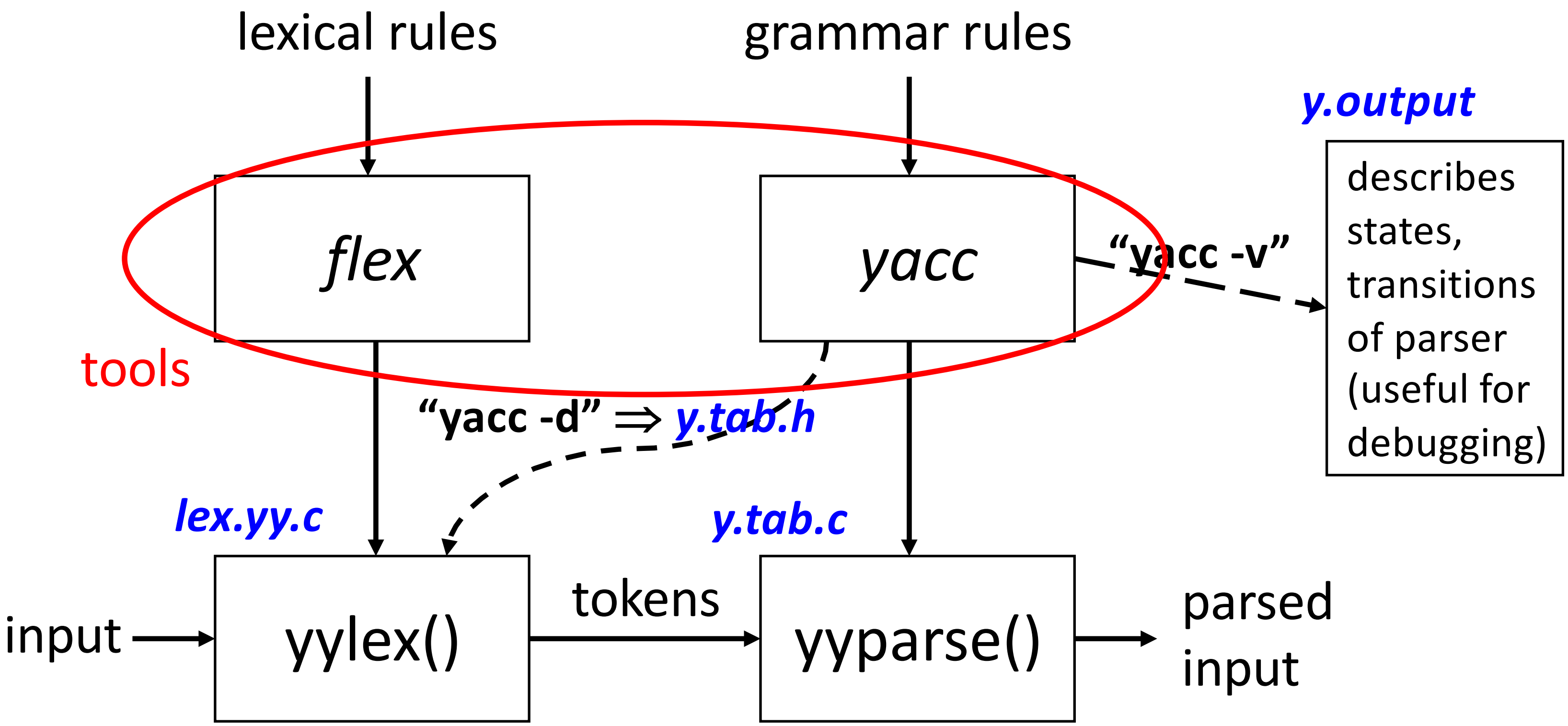
Using parser generators



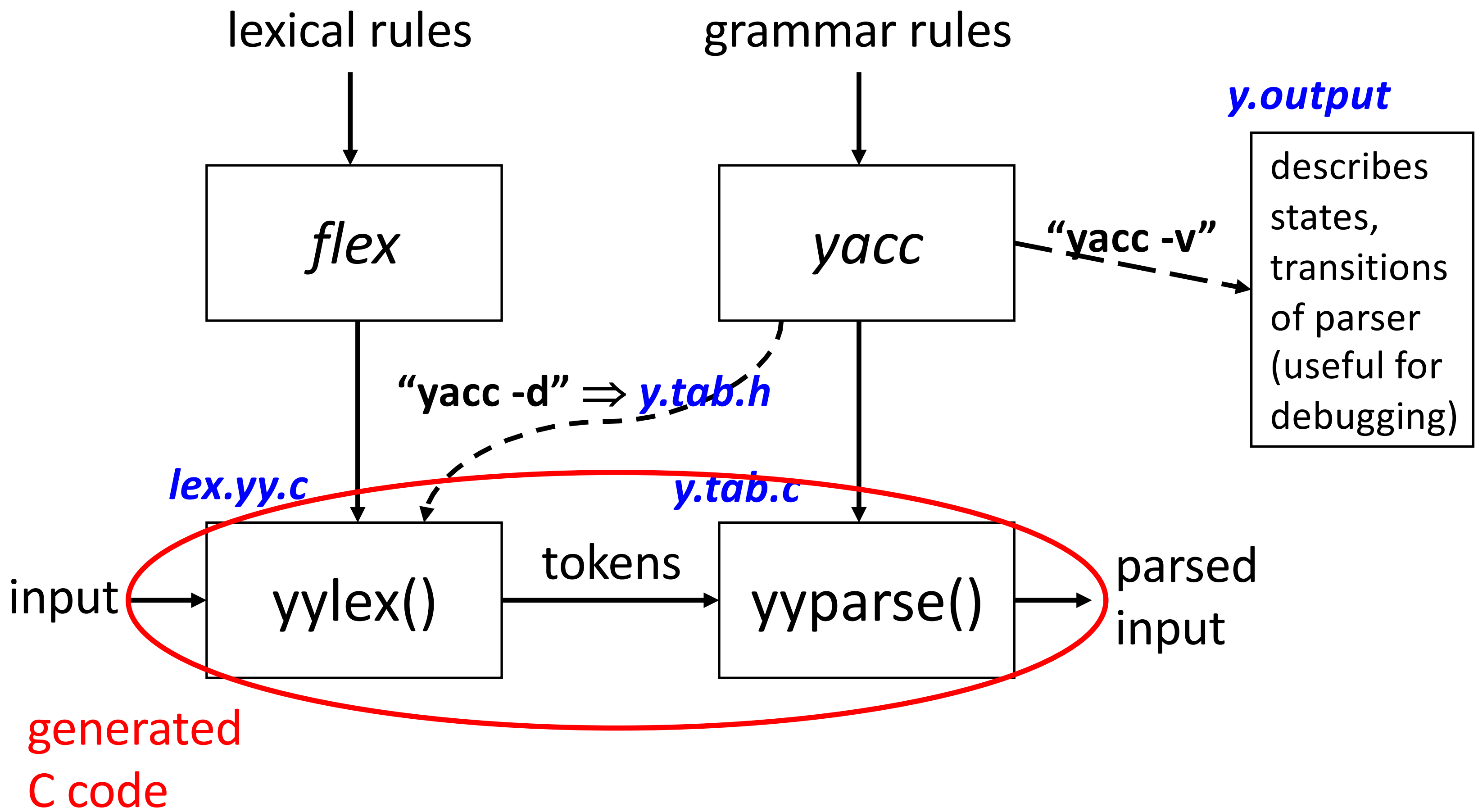
Using parser generators



Using parser generators



Using parser generators



Using yacc/bison

Structure of input file:

definitions (optional)

%%

rules (optional)

%%

user code (optional)

Format of rules:

Grammar rules:

$A \rightarrow B_1 \; B_2$

$A \rightarrow C_1 \; C_2 \; C_3$

Yacc rules:

$A : B_1 \; B_2$

$\quad | \;\; C_1 \; C_2 \; C_3$

You can embed semantic actions (C code) anywhere on the RHS of a rule

Example: parsing expressions

Calculator:

```
extern int intval;
int power(int x, int y);

void yyerror(char *s);
extern int yylex();
%}

%token NUM

%left '+' '-'
%left '*' '/'
%right '^'

%%

toplev : expr        { printf("ANS: %
       ;

expr : expr '+' expr     { $$ = $1 +
     | expr '-' expr     { $$ = $1 - $3; }
     | expr '*' expr     { $$ = $1 * $3; }
     | expr '/' expr     { $$ = $1 / $3; }
     | expr '^' expr     { $$ = power($1, $3); }
     | '(' expr ')'      { $$ = $2; }
     | '-' expr  %prec '^' { $$ = - $2; }
     | NUM               { $$ = intval; }
     ;

%%
```

associativity + precedence

```
/*
 * power(x, y) -- return x raised to the power y.
 */
int power(int x, int y) {
  int neg = 1, i, pow;

  fprintf(stderr, "%s\n", s);
  exit(1);
}

int main() {
  yyparse();
  return 0;
}
```

```
% make clean
/bin/rm -f *.BAK *.o lex.yy.c y.tab.c y.tab.h y.output calc
% make
flex scanner.l
yacc -dv calc.y
gcc -Wall -c lex.yy.c
gcc -Wall -c y.tab.c
gcc -Wall lex.yy.o y.tab.o -o calc -lfl
% ./calc
5 - 3 - 1 + 2^2^2 + 2
ANS: 19
%
```

Example: parsing expressions

AST Builder:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "tree.h"

extern int intval;
void yyerror(char *s);
extern int yylex();
%}

%union {
  int nval;
  TreeNode *tnptr;
}

%token <nval> NUM
%type <tnptr> expr;

%left '+' '-'
%left '*' '/'
%right '^'
%%
toplev : expr              { pri
;

expr : expr '+' expr      { $$
     | expr '-' expr      { $$
     | expr '*' expr      { $$
     | expr '/' expr      { $$
     | expr '^' expr      { $$
     | '(' expr ')'       { $$
     | '-' expr     %prec '^'
     | NUM                { $$
;

%%
```

```
void yyerror(char *s) {
  fprintf(stderr, "%s\n", s);
  exit(1);
```

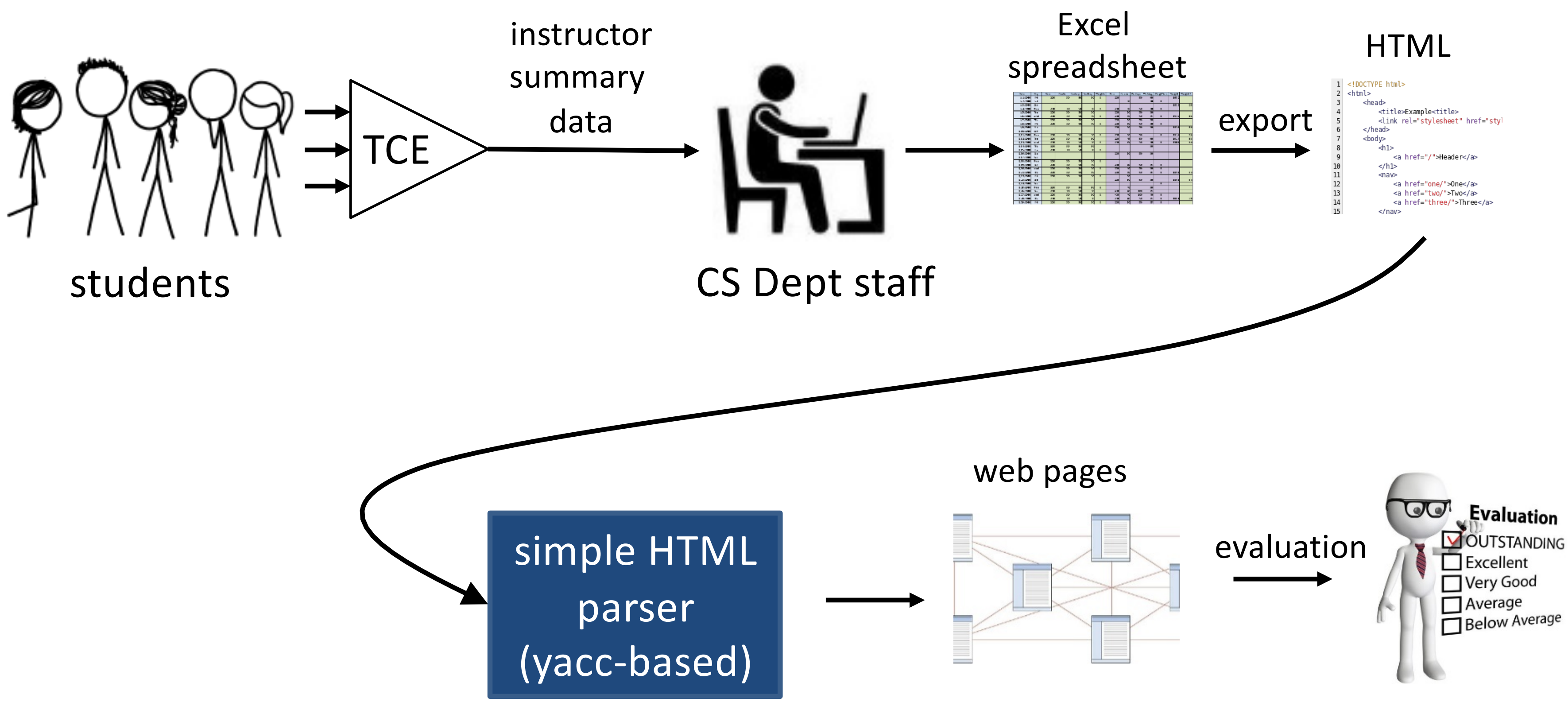
```
, ADD, SUB, MUL, DIV, EXP};

d;

child, rchild) -- return a tree
d fields ival, lchild, rchild.

ype nodetype, int ival,
lchild, TreeNode *rchild) {
izeof(*tptr));
("Out of memory!");
```

```
% make clean
/bin/rm -f *.BAK *.o lex.yy.c y.tab.c y.tab.h y.output calc
% make
flex scanner.l
yacc -dv calc.y
gcc -Wall -c lex.yy.c
gcc -Wall -c y.tab.c
gcc -Wall   -c -o tree.o tree.c
gcc -Wall lex.yy.o y.tab.o tree.o -o calc -lfl
% ./calc
5 - 3 - 1 + 2^2^2 + 2
(ADD:
    (ADD:
        (SUB:
            (SUB:
                intval: 5
                intval: 3
            )
            intval: 1
        )
        (EXP:
            intval: 2
            (EXP:
                intval: 2
                intval: 2
            )
        )
    )
    intval: 2
)
%
```

SOME NON-STANDARD PARSERS

1. Faculty teaching evaluations

(circa 2012)



1. Faculty teaching evaluations

scanner (input to flex)

```
ignore     ([[:space:]]|[[:cntrl:]])
%%

<*>"\n"                          { LineNo++; }
<INITIAL>"<body "[^>]*">"        { BEGIN(InBody); }
<INITIAL>.                       ;

<InBody>"<!--"                   { BEGIN(InComment); }
<InBody>"<table "[^>]*">"        { BEGIN(InTbl); }
<InBody>"</body>"                { BEGIN(INITIAL); }

<InTbl>"<col "[^>]*">"           ;
<InTbl>"</table"[^>]*">"         { BEGIN(InBody); }
<InTbl>"<!--"                    { BEGIN(InTblComment); }
<InTbl>"<td "[^>]*">"            { BEGIN(InData);
                                   bufidx = 0;
                                   ch = strstr(yytext, "colspan");
                                   if (ch) {
                                     while (!isdigit(*ch)) {
                                       ch++;
                                     }
                                     colspan = atoi(ch);
                                   }
                                   else {
                                     colspan = 1;
                                   }
                                 }
<InTbl>"<tr "[^>]*">"            { fieldnum = 0; return TR_begin; }
<InTbl>"</tr>"                   { return TR_end; }
<InTbl>.                         ;

<InData>{ignore}*"</td>"         { buf[bufidx] = '\0';
                                   if (fieldnum >= MAXFIELDS) {
                                     fprintf(stderr,
                                             "*** Field no. exceeds MAXFIELDS
",
                                             LineNo);
                                     PrintRecord(dataPtr);
                                     exit(-1);
                                   }
                                   Data(dataPtr,fieldnum) = strdup(buf);
                                   fieldnum += colspan;
                                   BEGIN(InTbl);
                                   return FIELD;
                                 }
```

parser (input to yacc)

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "global.h"

extern char *yytext;
extern int LineNo;

extern eptr dataList;
eptr dataPtr;
%}

%token TR_begin TR_end FIELD

%start Doc

%%
Doc  :  Doc Record
     |
     ;

Record  :  TR_begin
           { NewNode(dataPtr); Attach(dataPtr, dataList); }
           FieldSeq
           TR_end
        ;

FieldSeq  :  FieldSeq FIELD
          |
          ;

%%
void yyerror(char *s)
{
  printf("line %d: %s near: %s\n", LineNo, s, yytext);
}
```

1. Faculty teaching evaluations

generated output

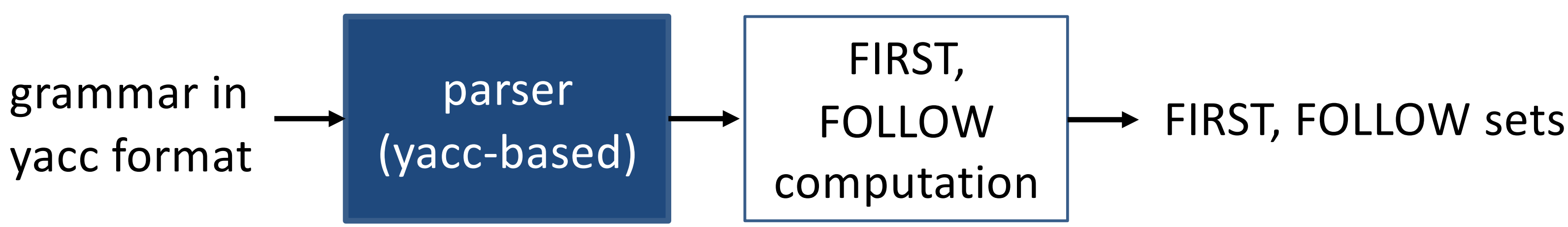
CS Course Evaluations: [illegible]

More information: Color Codes | Enrollment data | Historical data

Semester	Course	#Resp	Enrollment	%Resp	Overall Teaching Effectiveness (TCE Q1)	Overall Rating of Course (TCE Q2)	Amount Learned (TCE Q3)	Overall Instructor Comparison (TCE Q4)	Difficulty Level (TCE Q9)	Grades
spr08	127A	80	130 (142)	62% (56%)	3.70 A:23; B:28; C:19; D:5; E:5; ***hist***: *mean: 4.095, SD: 0.362*	3.6 ***hist***: *mean: 3.855, SD: 0.288*	4 ***hist***: *mean: 4.027, SD: 0.253*	3.3 ***hist***: *mean: 3.765, SD: 0.433*	3.5 ***hist***: *mean: 3.614, SD: 0.321*	A: 33 (25%); B: 35 (27%); C: 20 (15%); D: 20 (15%); E: 18 (14%); I: 4 (3%);
fall07	127A	90	160 (174)	56% (52%)	4.20 A:38; B:35; C:17; D:0; E:0; ***hist***: *mean: 4.095, SD: 0.362*	3.9 ***hist***: *mean: 3.855, SD: 0.288*	4 ***hist***: *mean: 4.027, SD: 0.253*	3.6 ***hist***: *mean: 3.765, SD: 0.433*	3.4 ***hist***: *mean: 3.614, SD: 0.321*	A: 46 (29%); B: 36 (23%); C: 30 (19%); D: 28 (18%); E: 16 (10%); I: 1 (1%);
spr07	127A	67	124 (145)	54% (46%)	4.10 A:26; B:25; C:14; D:2; E:0; ***hist***: *mean: 4.095, SD: 0.362*	4.1 ***hist***: *mean: 3.855, SD: 0.288*	4.4 ***hist***: *mean: 4.027, SD: 0.253*	3.8 ***hist***: *mean: 3.765, SD: 0.433*	3.3 ***hist***: *mean: 3.614, SD: 0.321*	A: 47 (38%); B: 32 (26%); C: 14 (11%); D: 19 (15%); E: 12 (10%); I: 0 (0%);
fall06	127A	95	148 (165)	64% (58%)	4.00 A:33; B:37; C:16; D:4; E:2; ***hist***: *mean: 4.095, SD: 0.362*	3.9 ***hist***: *mean: 3.855, SD: 0.288*	4.1 ***hist***: *mean: 4.027, SD: 0.253*	3.6 ***hist***: *mean: 3.765, SD: 0.433*	3.3 ***hist***: *mean: 3.614, SD: 0.321*	A: 69 (47%); B: 31 (21%); C: 14 (9%); D: 14 (9%); E: 16 (11%); I: 2 (1%);
spr06	127B	45	86 (97)	52% (46%)	4.30 A:16; B:25; C:4; D:0; E:0; ***hist***: *mean: 4.095, SD: 0.362*	4.2 ***hist***: *mean: 3.855, SD: 0.288*	4.4 ***hist***: *mean: 4.027, SD: 0.253*	4 ***hist***: *mean: 3.765, SD: 0.433*	3.6 ***hist***: *mean: 3.614, SD: 0.321*	A: 27 (31%); B: 29 (34%); C: 9 (10%); D: 6 (7%); E: 14 (16%); I: 0 (0%);
fall05	127A	78	180 (200)	43% (39%)	4.00 A:23; B:38; C:13; D:3; E:1; ***hist***: *mean: 4.095, SD: 0.362*	3.8 ***hist***: *mean: 3.855, SD: 0.288*	4.1 ***hist***: *mean: 4.027, SD: 0.253*	3.6 ***hist***: *mean: 3.765, SD: 0.433*	3.3 ***hist***: *mean: 3.614, SD: 0.321*	A: 64 (36%); B: 37 (21%); C: 43 (24%); D: 18 (10%); E: 16 (9%); I: 0 (0%);
fall04	127a	82	158 (184)	52% (45%)	4.37 A:39; B:32; C:7; D:0; E:1; ***hist***: *mean: 4.095, SD: 0.362*	4 ***hist***: *mean: 3.855, SD: 0.288*	4 ***hist***: *mean: 4.027, SD: 0.253*	4 ***hist***: *mean: 3.765, SD: 0.433*	3.6 ***hist***: *mean: 3.614, SD: 0.321*	A: 46 (29%); B: 42 (27%); C: 22 (14%); D: 20 (13%); E: 25 (16%); I: 0 (0%);
spr04	345	89	120 (132)	74% (67%)	3.94 A:24; B:41; C:18; D:4; E:1; ***hist***: *mean: 4.055, SD: 0.466*	3.3 ***hist***: *mean: 3.762, SD: 0.467*	3.4 ***hist***: *mean: 3.944, SD: 0.409*	3.5 ***hist***: *mean: 3.729, SD: 0.555*	3.4 ***hist***: *mean: 3.904, SD: 0.455*	A: 38 (32%); B: 55 (46%); C: 18 (15%); D: 6 (5%); E: 2 (2%); I: 0 (0%);
spr04	346			0% ()	***hist***: *mean: 4.055, SD: 0.466*	***hist***: *mean: 3.762, SD: 0.467*				A: 0 (0%); B: 1 (50%); C: 1 (50%); D: 0 (0%); E: 0 (0%); I: 0 (0%);
fall03	127a	97	174 (180)	56% (54%)	3.88 A:30; B:35; C:24; D:6; E:2; ***hist***: *mean: 4.095, SD: 0.362*	3.5 ***hist***: *mean: 3.855, SD: 0.288*	3.9 ***hist***: *mean: 4.027, SD: 0.253*	3.5 ***hist***: *mean: 3.765, SD: 0.433*	3.4 ***hist***: *mean: 3.614, SD: 0.321*	A: 55 (32%); B: 40 (23%); C: 34 (20%); D: 26 (15%); E: 15 (9%); I: 0 (0%);

2. Computing FIRST, FOLLOW sets

gff: A tool to compute FIRST and FOLLOW sets for an arbitrary grammar [CSc 453]



2. Computing FIRST, FOLLOW sets

yacc specification for the yacc-format input grammar:

```
%token KEYWD_TOK KEYWD_START TOKEN IDENT COLON SEMI BAR PP

%union {
  plist prod_list;
  pptr  prod_body;
}

%type <prod_body> body;
%type <prod_list> body_list

%start Gram
%%
Gram : tokens PP prods
;

tokens : tokens KEYWD_TOK toklist
       | tokens KEYWD_START IDENT { set_startsym(yytext); }
       |
;

toklist : toklist IDENT { sym_insert(yytext, TRUE); }
        | IDENT { sym_insert(yytext, TRUE); }
;
```

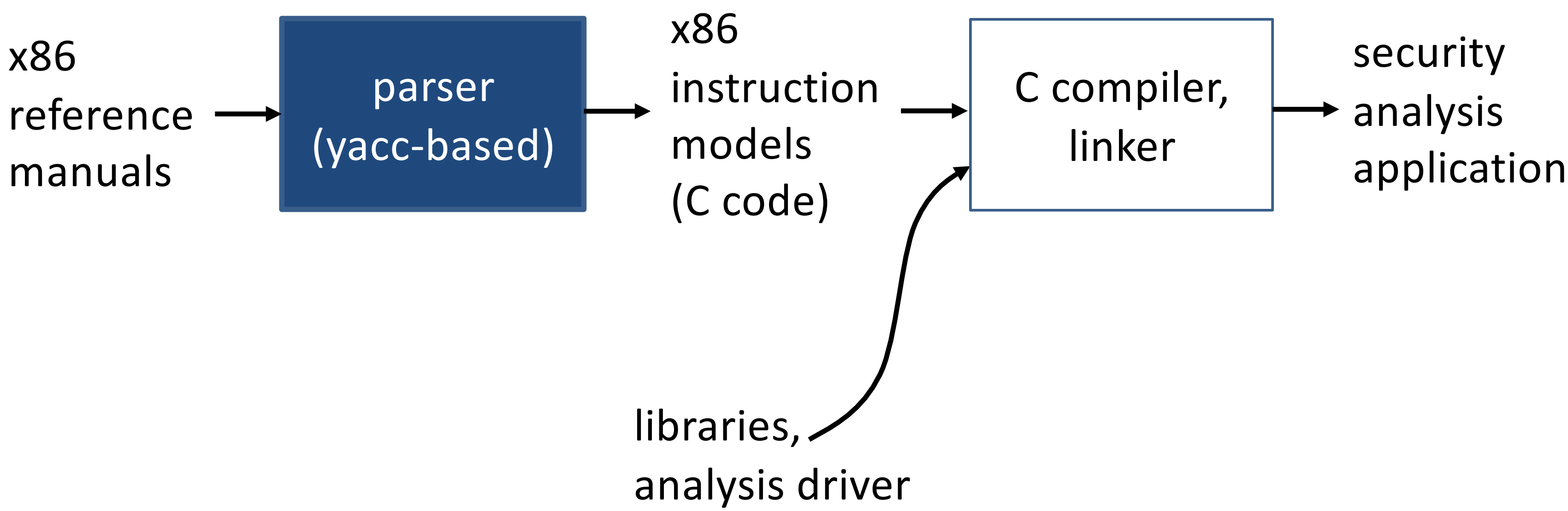
```
prods : prods prod
      |
;

prod : IDENT
       { stmp = sym_insert(yytext, FALSE); }
       COLON
       body_list
       SEMI
       { Prods(stmp) = $4; }
;

body_list : body_list BAR body { $$ = MkProdList($1, $3); }
     | body { $$ = MkProdList(NULL, $1); }
;

body : body IDENT { $$ = ProdAttach($1, sym_add(yytext, FALSE)); }
     | body TOKEN  { $$ = ProdAttach($1, sym_add(yytext, TRUE)); }
     | { $$ = NULL; }
;

%%
```

3. Parsing Intel x86 manuals

- Want C code that models the behavior of individual x86 instructions
  - for analyzing executables for a security project
- Too many to do manually
  - xed reports 1503 different instructions

Parse the x86 Instruction Reference Manual

3. Parsing Intel x86 manuals



3. Parsing Intel x86 manuals

INSTRUCTION SET REFERENCE, A-M

IMUL—Signed Multiply

Opcode	Instruction
F6 /5	IMUL *r/m8**
F7 /5	IMUL *r/m16*
F7 /5	IMUL *r/m32*
REX.W + F7 /5	IMUL *r/m64*
0F AF /*r*	IMUL *r16, r/m16*
0F AF /*r*	IMUL *r32, r/m32*
REX.W + 0F AF /*r*	IMUL *r64, r/m64*
6B /*r ib*	IMUL *r16, r/m16, imm8*
6B /*r ib*	IMUL *r32, r/m32, imm8*
REX.W + 6B /*r ib*	IMUL *r64, r/m64, imm8*
69 /*r iw*	IMUL *r16, r/m16, imm16*
69 /*r id*	IMUL *r32, r/m32, imm32*
REX.W + 69 /*r id*	IMUL *r64, r/m64, imm32*

NOTES:

* In 64-bit mode, r/m8 can not be encoded to acce

Op/En	Operand 1
M	ModRM:r/m (r, w)
RM	ModRM:reg (r, w)

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. Use of REX.W modifies the three forms of the instruction as follows.

- **One-operand form** —The source operand (in a 64-bit general-purpose register or memory location) is multiplied by the value in the RAX register and the product is stored in the RDX:RAX registers.
- **Two-operand form** — The source operand is promoted to 64 bits if it is a register or a memory location. The destination operand is promoted to 64 bits.
- **Three-operand form** — The first source operand (either a register or a memory location) and destination operand are promoted to 64 bits. If the source operand is an immediate, it is sign extended to 64 bits.

Operation

```
IF (NumberOfOperands = 1)
   THEN IF (OperandSize = 8)
      THEN
         TMP_XP ← AL ∗ SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *);
         AX ← TMP_XP[15:0];
         SF ← TMP_XP[7];
         IF SignExtend(TMP_XP[7:0]) = TMP_XP
            THEN CF ← 0; OF ← 0;
            ELSE CF ← 1; OF ← 1; FI;
      ELSE IF OperandSize = 16
         THEN
            TMP_XP ← AX ∗ SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
            DX:AX ← TMP_XP[31:0];
            SF ← TMP_XP[15];
            IF SignExtend(TMP_XP[15:0]) = TMP_XP
               THEN CF ← 0; OF ← 0;
               ELSE CF ← 1; OF ← 1; FI;
         ELSE IF OperandSize = 32
            THEN
               TMP_XP ← EAX ∗ SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC*)
               EDX:EAX ← TMP_XP[63:0];
               SF ← TMP_XP[32];
               IF SignExtend(TMP_XP[31:0]) = TMP_XP
                  THEN CF ← 0; OF ← 0;
                  ELSE CF ← 1; OF ← 1; FI;
```

want to parse this

3. Parsing Intel x86 manuals

It's a *lot* harder than I had thought

3. Parsing Intel x86 manuals

The input to yacc looks like this:

```
StmtTerminator : ';'
      |  /* empty */
;

assg_stmt : expr Op_ASSG expr
;

if_stmt : Kw_IF expr Kw_THEN stmts opt_else Kw_FI
;

opt_else : Kw_ELSE stmts
      |  /* empty */
;

case_stmt : Kw_CASE expr Kw_OF case_body Kw_ESAC

case_body : case_body  case_body_1
      | case_body_1
;

case_body_1 : INTCON ':' stmts StmtTerminator
;

expr : ID
      | INTCON
      | Kw_OpSz
      | Kw_CondTRUE
      | Mode_64bit
      | Mode_IA32e
      | Kw_Bitpos INTCON
      | Kw_Bitpos '(' expr ',' expr ')'
      | '(' expr ')'
      | '!' expr
      | '-' expr  %prec '!'
      | expr '+' expr
      | expr '-' expr
      | expr '*' expr
```

3. Parsing Intel x86 manuals

Current status: working on it

Summary

- Bottom-up parsers can be an alternative to recursive-descent
  - painful to write by hand
  - much more convenient via parser generators
- They construct the parse tree bottom-up
  - start at the tokens
  - work by repeatedly undoing derivation steps
- Parsers also find non-standard applications