CIT 3136 – Week 10 Lecture

Bottom-up Parsing:

- *Table-driven* **using an explicit stack (no recursion!).**
- **Stack can be viewed as containing both terminals and nonterminals.**
- **Basic operation is to** ***shift*** **terminals from the input to the stack until the right-hand side of an appropriate grammar rule is seen, and then to** ***reduce*** **the stuff on the stack that matches the rhs to the single nonterminal of the rule. Hence, bottom-up parsers are often called** ***shift-reduce parsers.***

Example

Grammar:

$E \rightarrow E + n \mid n$

Input: **2 + 3, or *n* + *n***

Parse: **($ is EOF in input, also bottom of stack)**

	Parsing stack	Input	Action
1	$	***n* + *n*** $	shift
2	$ ***n***	**+ *n*** $	reduce $E \rightarrow$ ***n***
3	$ *E*	**+ *n*** $	shift
4	$ *E* **+**	***n*** $	shift
5	$ *E* **+ *n***	$	reduce $E \rightarrow E$ **+ *n***
6	$ *E*	$	accept

Notes:

Left recursion is not a problem in bottom-up parsing. Indeed, as we shall see, lookahead is not as serious an issue.

Keeping track of what is on the stack, however, *is* an issue (note the difference in the grammar rule reductions at lines 2 and 5 of the previous example). *See later discussion on stack state.*

Right recursion is actually a bit of a problem, because it makes the stack grow large (see next example).

Example

Grammar:

$E \rightarrow n + E \mid n$

Input: 2 + 3, or *n* + *n*

Parse:

	Parsing stack	Input	Action
1	$\$$	$n + n\ \$$	shift
2	$\$\ n$	$+ n\ \$$	shift
3	$\$\ n +$	$n\ \$$	shift
4	$\$\ n + n$	$\$$	reduce $E \rightarrow n$
5	$\$\ n + E$	$\$$	reduce $E \rightarrow n + E$
6	$\$\ E$	$\$$	accept

Decision Problems in Bottom-up Parsing (parsing conflicts):

- **Shift-reduce conflicts: almost always come from ambiguities, and almost always the right disambiguating rule is to shift (dangling-else).**
- **Reduce-reduce conflicts are more difficult; bottom-up parsers try to resolve them using Follow contexts.**
- **There are no shift-shift conflicts.**

Dangling-else Example:

Grammar: $S \to I \mid \mathtt{o}$

$I \to \mathtt{i}\ S \mid \mathtt{i}\ S\ \mathtt{e}\ S$

Input: `i i o e o`

Parse:

	Parsing stack	Input	Action
1	$	i i o e o $	shift
2	$ i	i o e o $	shift
3	$ i i	o e o $	shift
4	$ i i o	e o $	reduce $S \to \mathtt{o}$
5	$ i i *S*	e o $	shift/reduce (shift)
6	$ i i *S* e	o $	shift
7	$ i i *S* e o	$	reduce $S \to \mathtt{o}$
8	$ i i *S* e *S*	$	reduce $I \to \mathtt{i}\ S\ \mathtt{e}\ S$
9	$ i *I*	$	reduce $S \to I$
10	$ i *S*	$	reduce $I \to \mathtt{i}\ S$
11	$ *I*	$	reduce $S \to I$
12	$ *S*	$	accept

Reduce-reduce Example

Grammar: $S \rightarrow A\ B$

$A \rightarrow \mathbf{x}$

$B \rightarrow \mathbf{x}$

Input: **x x**

Parse: **(Follow(*A*) = {x}, Follow(*B*) = {$})**

	Parsing stack	Input	Action
1	$	**x x** $	shift
2	$ **x**	**x** $	reduce $A \rightarrow$ **x** (reduce $B \rightarrow$ **x**)
3	$ *A*	**x** $	shift
4	$ *A* **x**	$	reduce $B \rightarrow$ **x**
5	$ *A B*	$	reduce $S \rightarrow A\ B$
6	$ *S*	$	accept

Shift-reduce parsers differ in their use of Follow information:

- **LR(0) parsers never consult the lookahead at all.**
- **SLR(1) parsers use the Follow sets as previously constructed.**
- **LR(1) parsers use context to split the Follow sets into subsets for different parsing paths (huge, inefficient parsers).**
- **LALR(1) parsers: like LR(1) but coarser subsets are used (achieves most of the benefit, but much smaller and faster).**

Technical Addendum

Shift-reduce parsers have trouble figuring out when to accept, so acceptance is turned into a reduction by a new rule $S' \rightarrow S$ with a new start symbol S'. Adding this rule is called *augmenting the grammar*:

	Parsing stack	Input	Action
	<previous example>	…	…
5	$\$ A\ B$	$	reduce $S \rightarrow A\ B$
6	$\$ S$	$	reduce $S' \rightarrow S$
7	$\$ S'$	$	accept

Yacc

- **“Yet another compiler compiler” (historical term for parser generator)**
- **Written by Steve Johnson at Bell Labs 1975.**
- **Bison: Gnu version of Yacc written by Robert Corbett and Richard Stallman circa 1985.**
- **Follows same basic conventions as Lex/Flex.**
- **Complete Bison documentation at** http://www.gnu.org/manual/bison-1.25/

Format of a Yacc/Bison definition file

```
{definitions}
%%
{rules}
%%
{auxiliary C functions}
```

Typical file extensions: .y .yacc .bison

Yacc Example

```
%token NUMBER
%%
command : exp        { printf("%d\n",$1);}
   ;  /* allows printing of the result */
exp   : exp '+' term  {$$ = $1 + $3;}
   | exp '-' term  {$$ = $1 - $3;}
   | term   {$$ = $1;}
   ;
term  : term '*' factor  {$$ = $1 * $3;}
   | factor  {$$ = $1;}
   ;
factor      : NUMBER       {$$ = $1;}
      | '(' exp ')'  {$$ = $2;}
      ;
```

Yacc insists on defining tokens itself (except single chars can be matched directly).

Actions can use a "value" stack to compute results (yylval); number is position.

The value of a token must be assigned to yylval by the scanner

Yacc Example, continued

```
%%
main()
{ return yyparse(); }
int yylex(void)
{ int c;  while((c = getchar()) == ' ');
  if ( isdigit(c) ) {
    ungetc(c,stdin); scanf("%d",&yylval);
    return(NUMBER);
  }
  if (c == '\n') return 0;/* makes the parse stop */
  return(c);
}
void yyerror(char * s) /* prints an error message */
{ fprintf(stderr,"%s\n",s);}
```

Interfacing Yacc/Bison

- **Yacc generates a C file named y.tab.c (Bison: <filename>.tab.c)**
- **Yacc/Bison will generate a header file with token information for a scanner with the -d option: bison -d tiny.y *produces* tiny.tab.c and tiny.tab.h**
- **The .tab.h file for the above grammar looks as follows:**

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define  NUMBER      258
extern YYSTYPE yylval;
```

Yacc/Bison Parsing Tables

With the -v option ("verbose") Yacc generates a file y.output (Bison: <filename>.output) describing its parsing actions. For example, for the grammar

$$S \rightarrow A\ B$$

$$A \rightarrow x$$

$$B \rightarrow x$$

the output file looks as on the next slide.

y.output file:

`state 0`
` 'x'   shift, and go to state 1`
` S go to state 5`
` A go to state 2`
`state 1`
` A  ->  'x' .(rule 2)`
` $default reduce using rule 2 (A)`
`state 2`
` S  ->  A . B (rule 1)`
` 'x'   shift, and go to state 3`
` B go to state 4`
	`state 3`
	` B  ->  'x'. (rule 3)`
	` $default    reduce using rule 3 (B)`
	`state 4`
	` S  ->  A B . (rule 1)`
	` $default reduce using rule 1 (S)`
	`state 5`
	`    $ go to state 6`
	`state 6`
	`    $ go to state 7`
	`state 7`
	`    $default  accept`

Stack states and the description of shift-reduce parsing tables

- **Represent the state of a parse by a *position* in a grammar rule (indicated by some symbol - a period in the text).**
- **Track positions using a DFA, with transitions labeled by symbols (terminals and nonterminals).**
- **Transitions on terminals represent shifts**
- **Transitions on nonterminals represent reductions ("gotos")**
- ***To be continued in next slide set….***

Bison Parsing Conflicts in C-Minus?

Only the dangling else:

```
state 95
  sel_stmt  ->  IF '(' expr ')' stmt .   (rule 29)
  sel_stmt  ->  IF '(' expr ')' stmt . ELSE stmt
(rule 30)


  ELSE    shift, and go to state 98


  ELSE    [reduce using rule 29 (sel_stmt)]
  $default      reduce using rule 29 (sel_stmt)
```

Bison and TINY

- **No parsing conflicts at all (no dangling else)**
- **Tokens are communicated to scanner by including tiny.tab.h in globals.h**
- **tokenString is communicated to parser by including scan.h in tiny.y**
- **yylval and YYSTYPE not used by scanner**

scan.h

```
#ifndef _SCAN_H_
#define _SCAN_H_

/* MAXTOKENLEN is the maximum size of a token */
#define MAXTOKENLEN 40

/* tokenString array stores the lexeme of each token */
extern char tokenString[MAXTOKENLEN+1];

/* function getToken returns the
 * next token in source file
 */
TokenType getToken(void);

#endif
```

globals.h

```
. . .
#ifndef YYPARSER
/* the name of the following file may change */
#include "tiny.tab.h"
/* ENDFILE is implicitly defined by Yacc/Bison,
 * and not included in the tab.h file
 */
#define ENDFILE 0
#endif
. . .
/* Yacc/Bison generates its own integer values
 * for tokens
 */
typedef int TokenType;
. . .
```

tiny.tab.h

```
#ifndef BISON_TINY_TAB_H
#define BISON_TINY_TAB_H
#ifndef YYSTYPE
#define YYSTYPE int
#define YYSTYPE_IS_TRIVIAL 1
#endif
#define     IF    257
#define     THEN  258
#define     ELSE  259
#define     END   260
. . .
# define    RPAREN      275
# define    SEMI  276
# define    ERROR 277
extern YYSTYPE yylval;
#endif /* not BISON_TINY_TAB_H */
```

tiny.y (part 1)

```
%{
#define YYPARSER /* distinguishes Yacc output
 from other code files */

#include "globals.h"
#include "util.h"
#include "scan.h"
#include "parse.h"

#define YYSTYPE TreeNode *
static char * savedName; /* for use in assignments */
static int savedLineNo;  /* ditto */
static TreeNode * savedTree; /* stores syntax tree
for later return */

%}
```

tiny.y (part 2)

```
%token IF THEN ELSE END REPEAT UNTIL READ WRITE
%token ID NUM
%token ASSIGN EQ LT PLUS MINUS TIMES OVER
%token LPAREN RPAREN SEMI ERROR
%% /* Grammar for TINY */
program     : stmt_seq { savedTree = $1;}
            ;
stmt_seq    : stmt_seq SEMI stmt
       { YYSTYPE t = $1;
         if (t != NULL)
         { while (t->sibling != NULL) t = t->sibling;
           t->sibling = $3;
           $$ = $1; }
         else $$ = $3;
       }
            | stmt  { $$ = $1; }
            ;
```

tiny.y (part 3)

```
stmt          : if_stmt { $$ = $1; }
              | repeat_stmt { $$ = $1; }
              | assign_stmt { $$ = $1; }
              | read_stmt { $$ = $1; }
              | write_stmt { $$ = $1; }
              | error  { $$ = NULL; }
              ;
if_stmt       : IF exp THEN stmt_seq END
                   { $$ = newStmtNode(IfK);
                     $$->child[0] = $2;
                     $$->child[1] = $4; }
         | IF exp THEN stmt_seq ELSE stmt_seq END
                   { $$ = newStmtNode(IfK);
                     $$->child[0] = $2;
                     $$->child[1] = $4;
                     $$->child[2] = $6; }
              ;
```

Error production

tiny.y (part 4)

Embedded action

```
assign_stmt : ID
              { savedName = copyString(tokenString);
                savedLineNo = lineno; }
                ASSIGN exp
                   { $$ = newStmtNode(AssignK);
                     $$->child[0] = $4;
                     $$->attr.name = savedName;
                     $$->lineno = savedLineNo;
                   }
              ;
. . .
factor        : . . .
        | NUM
        { $$ = newExpNode(ConstK);
          $$->attr.val = atoi(tokenString);
        } . . . /* also an error production */
```

tiny.y (part 5)

```
%%
int yyerror(char * message)
{ fprintf(listing,"Syntax error at line %d: %s\n",
                   lineno,message);
  fprintf(listing,"Current token: ");
  printToken(yychar,tokenString);
  Error = TRUE;
  return 0;
}

int yylex(void)
{ return getToken(); }

TreeNode * parse(void)
{ yyparse();
  return savedTree;
}
```

Yacc/Bison internal names

Yacc internal name	Meaning/Use
`y.tab.c`	Yacc output file name
`y.tab.h`	Yacc-generated header file containing token definitions
`yyparse`	Yacc parsing routine
`yylval`	value of current token in stack
`yyerror`	user-defined error message printer used by Yacc
`error`	Yacc error pseudotoken
`yyerrok`	procedure that resets parser after error
`yychar`	contains the lookahead token that caused an error
`YYSTYPE`	preprocessor symbol that defines the value type of the parsing stack
`yydebug`	variable which, if set by the user to 1, causes the generation of runtime information on parsing actions

Yacc/Bison definition mechanisms

Yacc definition mechanism	Meaning/Use
`%token`	defines token preprocessor symbols
`%start`	defines the start nonterminal symbol
`%union`	defines a union **`YYSTYPE`**, allowing values of different types on parser stack
`%type`	defines the variant union type returned by a symbol
`%left %right %nonassoc`	defines the associativity and precedence (by position) of operators