LECTURE 11

Semantic Analysis and Yacc

REVIEW OF LAST LECTURE

In the last lecture, we introduced the basic idea behind semantic analysis.

Instead of merely specifying valid structures with a context-free grammar, we can attach meaning to the structures with an attribute grammar.

Attribute grammars contain productions, as in a context-free grammar, as well as semantic actions to perform when the production is taken.

production	**semantic rule**
<*A*> → <*B*> <*C*>	*A*.a := ...; *B*.a := ...; *C*.a := ...

REVIEW OF LAST LECTURE

We *decorate* a parse tree by evaluating the attributes of the nodes. The attributes may have a complicated dependency hierarchy.

An attribute flow algorithm propagates attribute values through the parse tree by traversing the tree according to the set (write) and use (read) dependencies (an attribute must be set before it is used).

Some attribute grammars, such as S-attributed and L-attributed grammars, give rise to straightforward attribute flow algorithms (such as one depth-first traversal of the tree).

ATTRIBUTE GRAMMAR EXAMPLES

$P \rightarrow A\ B\ C$
$A_1 \rightarrow A_2\ a$
$A \rightarrow a$
$B_1 \rightarrow B_2\ b$
$B \rightarrow b$
$C_1 \rightarrow C_2\ c$
$C \rightarrow c$

Write the semantic actions such that P has an attribute *st* whose value is true when the number of a's, b's and c's are the same in the sentence, and false otherwise.

ATTRIBUTE GRAMMAR EXAMPLES

$P \rightarrow A\ B\ C$

$A_1 \rightarrow A_2\ a$

$A \rightarrow a$ $\quad \{A.num = 1;\}$

$B_1 \rightarrow B_2\ b$

$B \rightarrow b$ $\quad \{B.num = 1;\}$

$C_1 \rightarrow C_2\ c$

$C \rightarrow c$ $\quad \{C.num = 1;\}$

Write the semantic actions such that P has an attribute *st* whose value is true when the number of a's, b's and c's are the same in the sentence, and false otherwise.

ATTRIBUTE GRAMMAR EXAMPLES

$P \rightarrow A\ B\ C$	
$A_1 \rightarrow A_2\ a$	$\{A_1.num = A_2.num + 1;\}$
$A \rightarrow a$	$\{A.num = 1;\}$
$B_1 \rightarrow B_2\ b$	$\{B_1.num = B_2.num + 1;\}$
$B \rightarrow b$	$\{B.num = 1;\}$
$C_1 \rightarrow C_2\ c$	$\{C_1.num = C_2.num + 1;\}$
$C \rightarrow c$	$\{C.num = 1;\}$

Write the semantic actions such that P has an attribute *st* whose value is true when the number of a's, b's and c's are the same in the sentence, and false otherwise.

ATTRIBUTE GRAMMAR EXAMPLES

$P \rightarrow A\ B\ C$	$\{\ if\ ((A.num == B.num)\ \&\&\ (A.num == C.num))$ $P.st\ =\ true;$ $else\ P.st\ =\ false;\}$
$A_1 \rightarrow\ A_2\ a$	$\{A_1.num =\ A_2.num + 1;\}$
$A \rightarrow a$	$\{A.num = 1;\}$
$B_1 \rightarrow B_2\ b$	$\{B_1.num =\ B_2.num + 1;\}$
$B \rightarrow b$	$\{B.num = 1;\}$
$C_1 \rightarrow C_2\ c$	$\{C_1.num =\ C_2.num + 1;\}$
$C \rightarrow c$	$\{C.num = 1;\}$

Write the semantic actions such that P has an attribute *st* whose value is true when the number of a's, b's and c's are the same in the sentence, and false otherwise.

ATTRIBUTE GRAMMAR EXAMPLES

Here's another example grammar.

$$P \rightarrow W$$
$$W \rightarrow B$$
$$W_1 \rightarrow B\ W_2$$
$$B \rightarrow 1$$
$$B \rightarrow 0$$

Let's write an attribute grammar that converts the binary number to a decimal number and prints it out.

ATTRIBUTE GRAMMAR EXAMPLES

$P \rightarrow W$

$W \rightarrow B$

$W_1 \rightarrow B\ W_2$

$B \rightarrow 1$ $\quad \{B.num = 1;\}$

$B \rightarrow 0$ $\quad \{B.num = 0;\}$

Let's write an attribute grammar that converts the binary number to a decimal number and prints it out.

ATTRIBUTE GRAMMAR EXAMPLES

$P \rightarrow W$	
$W \rightarrow B$	$\{W.val = B.num; W.order = 1;\}$
$W_1 \rightarrow B\ W_2$	
$B \rightarrow 1$	$\{B.num = 1;\}$
$B \rightarrow 0$	$\{B.num = 0;\}$

Let's write an attribute grammar that converts the binary number to a decimal number and prints it out.

ATTRIBUTE GRAMMAR EXAMPLES

$P \rightarrow W$	
$W \rightarrow B$	$\{W.val = B.num; W.order = 1;\}$
$W_1 \rightarrow B\ W_2$	$\{W_1.order = W_2.order * 2;$ $W_1.val = W_2.val + (B.num * W_1.order);\ \}$
$B \rightarrow 1$	$\{B.num = 1;\}$
$B \rightarrow 0$	$\{B.num = 0;\}$

Let's write an attribute grammar that converts the binary number to a decimal number and prints it out.

ATTRIBUTE GRAMMAR EXAMPLES

$P \rightarrow W$	$\{printf("\%d", W.val);\}$
$W \rightarrow B$	$\{W.val = B.num; W.order = 1;\}$
$W_1 \rightarrow B\ W_2$	$\{W_1.order = W_2.order * 2;$ $W_1.val = W_2.val + (B.num * W_1.order);\ \}$
$B \rightarrow 1$	$\{B.num = 1;\}$
$B \rightarrow 0$	$\{B.num = 0;\}$

Let's write an attribute grammar that converts the binary number to a decimal number and prints it out.

INTRODUCTION TO YACC

Yacc (Yet Another Compiler Compiler)

Automatically generate an LALR (Look-ahead LR) parser for a context free grammar.

- Parses a subset of LR grammars.
- Can parse LL(1) grammars as long as symbols with empty derivations also have non-empty derivations.

Allows syntax-directed translation by writing grammar productions and semantic actions.

Works with Lex by calling yylex() to get the next token.

YACC

Yacc file format:

```
declarations                    /* specify tokens, non-terminals, and other C constructs */
%%
translation rules               /* specify grammar here */
%%
supporting C-routines
```

The command “yacc yaccfile” produces y.tab.c, which contains a routine yyparse().

- yyparse() calls yylex() to get tokens.
- yyparse() returns 0 if the program is grammatically correct, non-zero otherwise.

YACC DECLARATIONS

```
%{
        #include <stdio.h>
        #include <string.h>
%}

%token NUMBER NAME

%%

%%
```

Include some necessary libraries.

Specify the tokens we will be using.

YACC TRANSLATION RULES

```
statement:      NAME '=' expression
                | expression
                ;

expression:     expression '+' NUMBER
                | expression '-' NUMBER
                | NUMBER
                ;
```

S → *name* = E
S → E
E → E + *num*
E → E – *num*
E → *num*

YACC ATTRIBUTE GRAMMARS

Each symbol can be associated with some attributes and semantic actions can be associated with productions.

```
item :  LPAREN exp RPAREN    {$$ = $2;}
      | NUMBER               {$$ = $1;}
      ;
```

$$ is the attribute associated with the left-hand side of the production.
$1 is the attribute associated with the first symbol in the right-hand side,
$2 for the second symbol, …

YACC ENVIRONMENT

Yacc processes a yacc specification file and produces a y.tab.c file. An integer function yyparse() is produced by Yacc.

- Calls yylex() to get tokens.
- Return non-zero when an error is found, or 0 if the program is accepted.
- Need main() and and yyerror() functions.

```
yyerror(const char *str){
   printf("yyerror: %s at line %d\n", str, yyline);
}

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

LEX AND YACC TOGETHER

ex.l

```
%{
#include “y.tab.h"
extern int yylval;
%}

%%
[0-9]+      { yylval = atoi (yytext);
              printf ("scanned the number %d\n", yylval);
              return NUMBER; }
[ \t]       { printf ("skipped whitespace\n"); }
\n          { printf ("reached end of line\n");
              return 0;}
.           { printf ("found other data \"%s\"\n", yytext);
              return yytext[0];}
%%
```

Include the header file for the parser.

yylval accessible by Yacc as image of token.

What kinds of tokens does this lex file identify?

LEX AND YACC TOGETHER

ex.l

```
%{
#include “y.tab.h"
extern int yylval;
%}

%%
[0-9]+      { yylval = atoi (yytext);
              printf ("scanned the number %d\n", yylval);
              return NUMBER; }
[ \t]       { printf ("skipped whitespace\n"); }
\n          { printf ("reached end of line\n");
              return 0;}
.           { printf ("found other data \"%s\"\n", yytext);
              return yytext[0];}
%%
```

Include the header file for the parser.

yylval accessible by Yacc as image of token (note that it is of type int here).

What kinds of tokens does this lex file identify?

- Integers
- Whitespace
- Newlines
- Single characters that do not fall into any other category

LEX AND YACC TOGETHER

```
%{
#include <stdio.h>
%}

%token NAME NUMBER
%%
statement:   NAME '=' expression       { printf("pretending to assign %s the value %d\n",  $1, $3); }
           |  expression                { printf("= %d\n", $1); }
           ;
expression:  expression '+' NUMBER     { $$ = $1 + $3; printf ("Recognized '+' expression.\n");}
           |  expression '-' NUMBER     { $$ = $1 - $3; printf ("Recognized '-' expression.\n");}
           |  NUMBER                    { $$ = $1;  printf ("Recognized a number.\n");}
           ;
%%
int main (void) { return yyparse();}
int yyerror (char *msg) { return fprintf (stderr, "YACC: %s\n", msg); }
int yywrap(){ return 1; }
```

LEX AND YACC TOGETHER

```
carnahan@diablo:~>flex ex.l
carnahan@diablo:~>yacc -d ex.y
carnahan@diablo:~>gcc lex.yy.c y.tab.c
carnahan@diablo:~>./a.out < test.txt
scanned the number 3
Recognized a number.
skipped whitespace
found other data "+"
skipped whitespace
scanned the number 4
Recognized '+' expression.
reached end of line
= 7
```

test.txt

```
3 + 4
```

yacc –d: cause the header file y.tab.h to be written.

LEX AND YACC TOGETHER

ex.l

```
%{
#include “y.tab.h"
extern int yylval;
%}

%%
[0-9]+      { yylval = atoi (yytext);
              printf ("scanned the number %d\n", yylval);
              return NUMBER; }
[ \t]       { printf ("skipped whitespace\n"); }
\n          { printf ("reached end of line\n");
              return 0;}
.           { printf ("found other data \"%s\"\n", yytext);
              return yytext[0];}
%%
```

Notice that yylval is an int here. What about when we want the image of strings?

LEX AND YACC TOGETHER

Let's make yylval a union so that it can represent multiple types. In our yacc file:

```
%{
#include <stdio.h>
%}

%union {
  int number;
  char *string;
}

%token <string> NAME
%token <number> NUMBER
%type <number> expression
%%
```

Makes yylval a union of int and char*

Now, we specify the type we need to use from the union for not only the tokens but also the non-terminals which are used in semantic actions.

LEX AND YACC TOGETHER

In our lex file:

```
%{
#include "y.tab.h"
%}

%%
[a-zA-Z]+ {yylval.string=yytext; printf(“scanned the name %s\n”, yylval); return NAME;}
[0-9]+      { yylval.number = atoi (yytext); printf ("scanned the number %d\n", yylval); return NUMBER; }
[ \t]       { printf ("skipped whitespace\n"); }
….
```

Now, we must specify the type as an attribute of yylval.

Also, we added rules for picking up the NAME token.

LEX AND YACC TOGETHER

```
carnahan@diablo:~>lex ex.l
carnahan@diablo:~>yacc -d ex.y
carnahan@diablo:~>gcc lex.yy.c y.tab.c
carnahan@diablo:~>./a.out < test.txt
scanned the number a
skipped whitespace
found other data "="
skipped whitespace
scanned the number 3
Recognized a number.
skipped whitespace
found other data "+"
skipped whitespace
scanned the number 4
Recognized '+' expression.
reached end of line
pretending to assign a = 3 + 4
 the value 7
```

MULTIPLE ATTRIBUTES

It is possible to define multiple attributes for a symbol by adding structs to your union. For example, let's say you want a symbol to be associated with both a type and id.

```
%union{
  struct{
    int type;
    char *id;
  } type_id;
}

%type <type_id> var
```

CONFLICTS

Keep in mind that Yacc produces an LALR parser.

If the grammar is not LALR, there may be conflicts that prevent parsing. Yacc will try to resolve them automatically, but some conflicts will need your intervention.

Where it is necessary, specify precedence and associativity of operators: use keywords %left, %right, %nonassoc in the declarations section.

- All tokens on the same line are the same precedence level and associativity.

- The lines are listed in order of increasing precedence.

%left PLUSnumber, MINUSnumber
%left TIMESnumber, DIVIDEnumber