More on Yacc’s value stack

As we noted last time, Yacc uses a second stack to store the attribute values of the tokens and terminals in the parse stack.

• For a token, the attributes are computed by the lexical analyzer, and pushed on the value stack when the token is shifted.

• For a variable, the attributes are computed when (just before) a reduction to that variable is carried out, and pushed onto the value stack when the variable is pushed onto the parse stack. (After popping the values corresponding to the rhs of the production used in the reduction.)

All of this works nicely for S-attributed definitions: syntax-directed definitions that use only synthesized attributes.

As we have seen, token attribute values are supplied via yylval, as in

```c
%token NUMBER

expr : expr '+' NUMBER { $$ = $1 + $3; }
    | expr '-' NUMBER { $$ = $1 - $3; }
    | NUMBER { $$ = $1; }
    | '(' expr ')' { $$ = $2; }
    
#include "lex.yy.c"

yyerror(char *s) {
    printf("%s\n", s);
}
main() {
    yyparse();
}

number 0[1-9][0-9]*

[ ] {}
{number} { sscanf(yytext, "%d", &yylval);
    return NUMBER; }
\n. { return yytext[0]; }

int yywrap() {
    return (1);
}
```

When the token is shifted, yylval is pushed onto the attribute stack.
The type used in the value stack can be changed. For instance, we can use doubles, as in your hwk...%include <stdio.h>
#define YYSTYPE double
%token NUMBER
%left '+', '-'
%left '*', '/'
%right UMINUS
%
lines : lines expr '
' { printf("%g\n", $2); }
| lines 'n'
| ;
expr : expr '+' expr { $$ = $1 + $3; }
| expr '-' expr { $$ = $1 - $3; }
| expr '*' expr { $$ = $1 * $3; }
| expr '/' expr { $$ = $1 / $3; }
| '(' expr ')' { $$ = $2; }
| '-' expr %prec UMINUS { $$ = -$2; }
| NUMBER
%
#include "lex.yy.c"
...
number [0-9]+\.[0-9]*\.[0-9]+
[ ]{}
{number} { sscanf(yytext, "%lf", &yylval);
    return NUMBER; }
\n]. { return yytext[0]; }
%
.
There are other possibilities...
%
#include <ctype.h>
%
union { char s[64]; char c; }
%type <s> expr
%token <c> DIGIT
%left '<', '>
%
line : line expr 'n' { printf("%a\n", $2); }
| ;
expr : expr '+' expr { strcpy($$,strcat(strcat($1,$3),"+"); }
| expr '-' expr { strcpy($$,strcat(strcat($1,$3),"-")); }
| '(' expr ')' { strcpy($$, $2); }
| DIGIT { $$[0] = $1; $$[1] = '\0'; }
|
%
yylex() {
    int c;
    while (((c=getchar()) == ' '));
    if (isdigit(c)) {
        yylval.c = c;
        return DIGIT;
    }
    return c;
}
yyerror(char *s) {
    printf("%s\n", s);
}
main() {
    yyparse();
}
Although we assume that the attributes of tokens are synthesized, those attributes may still depend on “other aspects of the parse”…

For example, in a desk calculator (as in Appendix A of the Yacc document) we may allow assignments of values to identifiers. In this case it may be convenient to have synthesized attributes _lexeme_ and _value_ for token _id_. Interestingly, the _value_ attribute at a node would presumably be computed by looking up _id_.lexeme in a “symbol table” —

- If the lexeme is not already in the table, make an entry for it (and perhaps give it a default value).
- If the lexeme is already in the table, then get the corresponding value from the table. Notice that this value is in some sense inherited: it is there because of a previous occurrence of an identifier with the same lexeme, perhaps in an assignment statement.

Here _entry_ should return the _id_ pointer for the lexeme’s table entry — if the lexeme is new, _entry_ enters it in the table.
Inherited attributes

So far we have focused on synthesized attributes, with good reason — they are convenient for use with LR parsers.

Once we allow inherited attributes, it becomes more difficult to annotate the parse tree. In fact, if there are “cycles” in the semantic rules, it may be impossible. Certainly there is no guarantee that the attributes can be computed in a single pass through the tree — in any order — much less bottom-up order.

Unfortunately, inherited attributes are in some cases more “natural.” Here is an example involving variable declarations:

<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>( D \rightarrow T \ L ) ;</td>
<td>$2.type := $1.type</td>
</tr>
<tr>
<td>( T \rightarrow \text{int} )</td>
<td>$$.type := \text{integer}</td>
</tr>
<tr>
<td>( T \rightarrow \text{real} )</td>
<td>$$.type := \text{real}</td>
</tr>
</tbody>
</table>
| \( L \rightarrow \text{id} \) \( \) | \$1.type := \$$.type \) \( \text{addentry}($3.\text{lexeme},\$$.\text{type}) \)
| \( L \rightarrow \text{id} \) | \( \text{addentry}($1.\text{lexeme},\$$.\text{type}) \)

Not difficult to build and annotate a parse tree for \textbf{int \ x, y;} for example, but let’s parse it bottom up and try to think about computing attributes as we go:

<table>
<thead>
<tr>
<th>STACK</th>
<th>INPUT</th>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{int id, id}$</td>
<td>\shift</td>
<td></td>
</tr>
<tr>
<td>\text{id, id}$</td>
<td>( \text{r2 { T.type = integer } )</td>
<td></td>
</tr>
<tr>
<td>\text{id, id}$</td>
<td>\shift</td>
<td></td>
</tr>
<tr>
<td>\text{T id, . id}$</td>
<td>( \text{r5 { addentry(id.\text{lexeme},L.type) } ) !!</td>
<td></td>
</tr>
<tr>
<td>\text{T L, . id}$</td>
<td>\shift</td>
<td></td>
</tr>
<tr>
<td>\text{T L, id}$</td>
<td>\shift</td>
<td></td>
</tr>
<tr>
<td>\text{T L, id}$</td>
<td>$</td>
<td>( \text{r4 { L.type = L1.type . addentry(id.\text{lexeme},L1.type) } ) !!</td>
</tr>
<tr>
<td>\text{T L1}$</td>
<td>$</td>
<td>( \text{r1 { L1.type = T.type } )</td>
</tr>
<tr>
<td>\text{D}$</td>
<td>$</td>
<td>( \text{accept} )</td>
</tr>
</tbody>
</table>

As the book explains, this is a case when we can finesse the problem, but you have to “manipulate” the attribute value stack...
In this case, an LR parser can manage to make the crucial symbol table entries.

The key is that you can get by without computing $L$.type attributes — instead notice that $T$.type sits at the bottom of the attribute stack throughout the parse.

Just before you reduce by the fourth production, you will have $id$.lexeme on top of the attribute stack, and $T$.type will be three down from the top. So you are still able to make the type entry in the symbol table (by “reaching down into the stack”).

Similarly, just before you reduce by the fifth production, you will have $id$.lexeme on top of the attribute stack, and $T$.type will be one down from the top.

So what does this approach look like?

Instead of the above, we have:

Instead of the above, we have:
The book discusses how this might be approached systematically (when it can be done at all).

Of course another approach is available here. We can alter the syntax-directed definition, making it possible to perform all actions at the time of reduction during a bottom up parse, without having to know about values of non-rhs elements that may be buried in the value stack.

<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D \rightarrow T \ L \ ;$</td>
<td>$2.\text{type} := 1.\text{type}$</td>
</tr>
<tr>
<td>$T \rightarrow \text{int}$</td>
<td>$$.\text{type} := \text{integer}$</td>
</tr>
<tr>
<td>$T \rightarrow \text{real}$</td>
<td>$$.\text{type} := \text{real}$</td>
</tr>
<tr>
<td>$L \rightarrow L\ , \ id$</td>
<td>$1.\text{type} := $.\text{type}$ addentry($3.\text{lexeme},$.\text{type}$)</td>
</tr>
<tr>
<td>$L \rightarrow id$</td>
<td>addentry($1.\text{lexeme},$.\text{type}$)</td>
</tr>
</tbody>
</table>

In fact, in this case we eliminate all inherited attributes.

This syntax-directed definition is arguably less “natural” though.

Let’s look at the sneak-a-peak mechanism in action in Yacc. . .

```c
#include <stdio.h>

%token NUMBER

trick : NUMBER list \n ;
list : list \x \ { printf("%d down 2\n",$0); }
    \ | \x \ { printf("%d down 1\n",$0); }
    %%
#include "lex.yy.c"

yyerror(char *s) {
    printf("%s\n",s);
}

main() {
    yyparse();
}

number 0[1-9][0-9]*
%
[ ] {}
{number} { sscanf(yytext, "%d", &yylval);
    return NUMBER; }
\n. { return yytext[0]; }
%
int yywrap() {
    return (1);
}
```
> yacc l26.y
> lex l26.l
> cc -o l26y y.tab.c
> ./l26y
777 x x x
777 down 1
777 down 2
777 down 2

<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T \rightarrow \text{NUMBER}\ L \ \n$</td>
<td></td>
</tr>
<tr>
<td>(L \rightarrow L\ x)</td>
<td>printf(&quot;%d down 2\n&quot;,$0);</td>
</tr>
<tr>
<td>(L \rightarrow x)</td>
<td>printf(&quot;%d down 1\n&quot;,$0);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>STACK</th>
<th>INPUT</th>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBERxxx\n$</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>NUMBERxxx\n$</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>NUMBER x xx\n$</td>
<td>r3 { printf(&quot;%d down 1\n&quot;,$0); }</td>
<td></td>
</tr>
<tr>
<td>NUMBER L xx\n$</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>NUMBER L x x\n$</td>
<td>r2 { printf(&quot;%d down 2\n&quot;,$0); }</td>
<td></td>
</tr>
<tr>
<td>NUMBER L x x\n$</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>NUMBER L x \n$</td>
<td>r2 { printf(&quot;%d down 2\n&quot;,$0); }</td>
<td></td>
</tr>
<tr>
<td>NUMBER L \n$</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>NUMBER L \n$</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

---

For next time and after

Last homework due.

Final on Saturday 12/16 at noon. Open book, open notes.

Cumulative.

Assigned reading, lectures, lecture notes, homeworks, labs.