Syntax-directed translation

Think back to the first or second week of the course:

We considered the following simple translation scheme.

\[
expr \rightarrow expr + term \{ printf(" + "); \}
| expr - term \{ printf(" - "); \}
| term
\]

\[
term \rightarrow id \{ printf(" %s ",id.lexeme); \}
| num \{ printf(" %s ",num.lexeme); \}
\]

We thought about this in the context of top-down parsing.

Draw the parse tree — with the semantic actions as additional “tokens” — and assume that the actions are performed in the order given by depth-first traversal.

Alternately, traverse the fringe, left-to-right (since semantic actions are leaves).

So what is the output on \texttt{a+b-2}?

For predictive parsing, we needed a modified translation scheme such as

\[
expr \rightarrow term rest
\]
\[
rest \rightarrow + term \{ printf(" + "); \} rest
| - term \{ printf(" - "); \} rest
| \epsilon
\]

\[
term \rightarrow id \{ printf(" %s ",id.lexeme); \}
| num \{ printf(" %s ",num.lexeme); \}
\]

(The underlying grammar is LL(1).)

Output on \texttt{a+b-2}?

In practice, we performed the actions without building the parse tree. No problem, since the top-down parser “traverses” the (implicit) parse tree in depth-first order.

But what happens when we write such a translation scheme for use in an LR parser? When can we expect the semantic actions to be performed?
Let's return to the first translation scheme:

\[
\begin{align*}
expr & \rightarrow \ expr \ + \ term \ \{ \ printf("+ "); \} \\
& \quad | \ expr \ - \ term \ \{ \ printf("- "); \} \\
& \quad | \ term \\
\end{align*}
\]

\[
\begin{align*}
term & \rightarrow \ id \ \{ \ printf("%s ", id.lexeme); \} \\
& \quad | \ num \ \{ \ printf("%s ", num.lexeme); \}
\end{align*}
\]

Recall from the description of a Yacc program — the semantic action is performed at the time the production is used in a reduction.

It seems to work on input \(a+b-2\)...
How about the second translation scheme we looked at?

\[
\begin{align*}
expr & \rightarrow \text{term rest} \\
\text{rest} & \rightarrow + \text{term} \ \{ \text{printf("+ ");} \} \ \text{rest} \\
| & \ - \ \text{term} \ \{ \text{printf("- ");} \} \ \text{rest} \\
| & \ \epsilon \\
\text{term} & \rightarrow \text{id} \ \{ \text{printf("%s ",id.lexeme);} \} \\
| & \ \text{num} \ \{ \text{printf("%s ",num.lexeme);} \}
\end{align*}
\]

Well, in describing the syntax of translation rules in Yacc, I showed semantic actions only at the “end” of productions.

So the question may seem a little strange.

But if you’ve investigated a little further, you may have discovered that Yacc too allows you to write such “embedded” actions.

What if they too were performed at the time of reduction?

Let’s look at the example again...

\[
\begin{align*}
expr & \rightarrow \text{term rest} \\
\text{rest} & \rightarrow + \text{term} \ \{ \text{printf("+ ");} \} \ \text{rest} \\
| & \ - \ \text{term} \ \{ \text{printf("- ");} \} \ \text{rest} \\
| & \ \epsilon \\
\text{term} & \rightarrow \text{id} \ \{ \text{printf("%s ",id.lexeme);} \} \\
| & \ \text{num} \ \{ \text{printf("%s ",num.lexeme);} \}
\end{align*}
\]

For better or worse (probably better!), that’s not what Yacc does...
Yacc turns

\[
\begin{align*}
expr & \rightarrow \ term \ rest \\
rest & \rightarrow \ + \ term \ \{ \ printf("+ "); \} \ rest \\
& \quad | \ - \ term \ \{ \ printf("- "); \} \ rest \\
& \quad | \ \epsilon \\
term & \rightarrow \ id \ \{ \ printf("\%s ",id.lexeme); \} \\
& \quad | \ num \ \{ \ printf("\%s ",num.lexeme); \}
\end{align*}
\]

into

\[
\begin{align*}
expr & \rightarrow \ term \ rest \\
rest & \rightarrow \ + \ term \ X_1 \ rest \\
& \quad | \ - \ term \ X_2 \ rest \\
& \quad | \ \epsilon \\
X_1 & \rightarrow \ \epsilon \ \{ \ printf("+ "); \} \\
X_2 & \rightarrow \ \epsilon \ \{ \ printf("- "); \} \\
term & \rightarrow \ id \ \{ \ printf("\%s ",id.lexeme); \} \\
& \quad | \ num \ \{ \ printf("\%s ",num.lexeme); \}
\end{align*}
\]

(Notice: in this case, two productions are added.)

You can check by hand that this alternative grammar works for the example (that is yields the translation we want).

Or you can run Yacc...
> cat y.output

state 0
$accept : _lines _end
lines : _ (3)
. reduce 3

lines goto 1
lines : lines NL_ (2)
. reduce 2

state 1
$accept : lines_$end
lines : lines_expr NL
lines : lines_NL
expr : term_rest
rest : _ (9)
$end accept
NL shift 3
+ shift 9
ID shift 5
- shift 10
NUM shift 6
. reduce 9
. error
rest goto 8
expr goto 2
term goto 4

state 5
term : ID_ (10)
. reduce 10

state 6
term : NUM_ (11)
ID shift 5
 NUM shift 6
. reduce 11
. error
term goto 12

lines : lines expr NL_ (1)
state 7
lines : lines_NL

state 8
. reduce 5
expr : term_rest_ (4)
 $$5 goto 13
 . reduce 4

state 9
rest : + term_$$_5 rest
rest : + term $$5 $$7 rest
$$5 goto 13

state 10
rest : _-term $$7 rest

state 11
rest : + term $$5 rest
$$5 : _ (5)

state 12
rest : - term $$7 rest
rest : _+term $$5 $$7 rest
$$7 : _ (7)

ID shift 5
. reduce 7
NUM shift 6
. error
$$7 goto 14
term goto 11
Annotated parse trees

In a parse tree, each node is labeled with a grammar symbol.

We can associate with each label a set of “attributes.”

For example, in a parse tree constructed with the grammar

\[
E \rightarrow E + n \\
| E - n \\
| n \\
| (E)
\]

each \(E\) node and \(n\) node may have a \textit{val} attribute — of type integer say.

Consider the attribute values in a parse tree for \(3-2+1\), for instance. (We’ll assume we know how to compute them; later we’ll start to give a general account of how it can be done.)

A parse tree showing the values of attributes at each node is called an \textit{annotated parse tree}.
We'll define syntax-directed definitions in a moment. First, here’s an example based on the previous grammar:

\[
\begin{align*}
E & \rightarrow E + n \quad \text{\$$.val := \$1.val + \$3.val} \\
E & \rightarrow E - n \quad \text{\$$.val := \$1.val - \$3.val} \\
E & \rightarrow n \quad \text{\$$.val := \$1.val} \\
E & \rightarrow (E) \quad \text{\$$.val := \$2.val}
\end{align*}
\]

A syntax-directed definition is a generalization of a context-free grammar in which each grammar symbol has an associated set of attributes of two kinds — synthesized attributes and inherited attributes — and each grammar production

\[A \rightarrow \alpha\]

has an associated set of semantic rules of the form

\[b := f(c_1, c_2, \ldots, c_k)\]

where \(f\) is a function and either

- \(b\) is a synthesized attribute of \(A\) and \(c_1, c_2, \ldots, c_k\) are attributes of the symbols of \(\alpha\), or
- \(b\) is an inherited attribute of one of the symbols of \(\alpha\) and \(c_1, c_2, \ldots, c_k\) are attributes of the symbols of the production (possibly including attributes of \(A\)).

Attributes of tokens are assumed to be synthesized attributes.

In practice, they are typically supplied by the lexical analyzer.

So in implementing our example syntax-directed definition, the lexical analyzer should supply a \textit{val} attribute for each \textit{n} token.

\[
\begin{align*}
E & \rightarrow E + n \quad \text{\$$.val := \$1.val + \$3.val} \\
E & \rightarrow E - n \quad \text{\$$.val := \$1.val - \$3.val} \\
E & \rightarrow n \quad \text{\$$.val := \$1.val} \\
E & \rightarrow (E) \quad \text{\$$.val := \$2.val}
\end{align*}
\]

We have seen Yacc programs in which this is done...
In some cases, semantic rules in syntax-directed definitions are used to produce side effects.

<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L \rightarrow L E \text{ nl}$</td>
<td><code>printf(&quot;%d\n&quot;, $2)</code></td>
</tr>
<tr>
<td>$L \rightarrow \epsilon$</td>
<td><code>$.val := $1.val + $3.val</code></td>
</tr>
<tr>
<td>$E \rightarrow E + n$</td>
<td><code>$.val := $1.val + $3.val</code></td>
</tr>
<tr>
<td>$E \rightarrow E - n$</td>
<td><code>$.val := $1.val - $3.val</code></td>
</tr>
<tr>
<td>$E \rightarrow n$</td>
<td><code>$.val := $1.val</code></td>
</tr>
<tr>
<td>$E \rightarrow (E)$</td>
<td><code>$.val := $2.val</code></td>
</tr>
</tbody>
</table>

The new semantic rule does not have the prescribed form. That’s ok. If we wanted, we could introduce a “dummy” synthesized attribute for the lhs occurrence of variable $L$, and assign the result of `printf("%d\n", $2)` to it.

For that matter, we could similarly introduce a “dummy” inherited attribute for any of the other occurrences of variables in the first production.

But we’ll see that synthesized attributes are generally easier to handle...
A syntax-directed definition that uses only synthesized attributes is called an *S-attributed definition*. A parse tree for an S-attributed definition can always be annotated by evaluating the semantic rules for the attributes at each node from the bottom up.

Hence, **S-attributed definitions are perfectly suited to LR parsers.**

And we’ve seen the elegant manner in which S-attributed definitions are specified in Yacc.

```yacc
%%
lines : lines expr \n \n    \n| \nexpr : \n    \n    expr \+ NUMBER \{ $$ = $1 + $3; \} \n    \n| expr \* NUMBER \{ $$ = $1 * $3; \} \n    \n| NUMBER \{ $$ = $1; \} \n    \n| \(', expr ')\' \{ $$ = $2; \} \n
lines expr \n\n```

In fact, given an S-attributed definition, it is natural for an LR parser to use a second stack to store attributes.

Each item in the parse stack has a set of attributes stored in the corresponding position in the attribute stack:

- For a token, the attributes are computed by the lexical analyzer, and pushed on the attribute 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 attribute stack when the variable is pushed onto the parse stack. (After popping the attributes corresponding to the rhs of the production used in the reduction.)
Notice that the translation scheme we started with today is essentially an S-attributed definition:

\[
\begin{align*}
  \text{expr} & \rightarrow \text{expr} + \text{term} \{ \text{printf}("+ "); \} \\
  & \mid \text{expr} - \text{term} \{ \text{printf}("- "); \} \\
  & \mid \text{term} \\
  \text{term} & \rightarrow \text{id} \{ \text{printf}(\"\%s\", \text{id}.lexeme); \} \\
  & \mid \text{num} \{ \text{printf}(\"\%s\", \text{num}.lexeme); \}
\end{align*}
\]

In fact, it is a particularly simple one. No attributes are computed for variables (except “dummy” attributes) and no attributes need be stored.

Hence we can write a Yacc program like:

```
%token ID NUM

%token NUMBER

expr : expr '+' term { printf("+ "); }
  | expr '-' term { printf("- "); }
  | term
  
| num { printf("\%s ", yytext); }

term : ID { printf("\%s ", yytext); }
  | NUM { printf("\%s ", yytext); }

%include "lex.yy.c"
```

But if Yacc is to place a token attribute in the attribute stack, in order to make further use of it in computing attributes of nonterminals, the token attribute value should be supplied via `yylval`, as in

```
%token NUMBER

expr : expr '+' NUMBER { $$ = $1 + $3; }
  | expr '-' NUMBER { $$ = $1 - $3; }
  | NUMBER { $$ = $1; }
  | '(' expr ')' { $$ = $2; }

%include "lex.yy.c"
```

```c
yyerror(char *s) {
    printf("%s\n", s);
}
main() {
    yyparse();
}
number 0|\([\-9]\)[0-9]*

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

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