Syntax Directed Translation-
Evaluation of Attributes

Evaluation of S-attributed Definitions

Recall that an S-attributed syntax-directed definition contains only synthesized attributes. Thus, in some production $A \rightarrow \alpha$, we can compute attributes for A with attributes for α. This makes S-attributed definitions very easily computed in conjunction with a bottom-up parser.

If we keep values for attributes on the parser stack (or another stack maintained concurrently with the parser stack), we can access these attributes when rules are reduced.

	State	Val
	...	...
	X	X.x
	Y	Y.x
top $\rightarrow$	Z	Z.x
	...	...

Table 1: Parser stack with attributes.

If we have a production $\texttt{A} \rightarrow \texttt{XYZ}$, we can compute

`A.x` (the attribute(s) of `A`) by accessing `val[top]`, `val[top-1]` and `val[top-2]`.

`bison/yacc` has a mechanism for accessing stack values, with the `$N` and `$$`, as we have already seen. Of course, types must be specified in order for it to work in `bison`.

L-attributed Definitions

L-attributed definitions encompass a larger class than that of S-attributed definitions. A syntax-directed definition is L-attributed if each inherited attribute of X_j on the right side of $A \to X_1 X_2 \ldots X_n$ with $1 \leq j \leq n$ only depends on

1. The symbols $X_1 \ldots X_{j-1}$.

2. The inherited attributes of A.

Note that an S-attributed grammar is also L-attributed since it doesn't contain any inherited attributes and the above restrictions apply only to definitions with inherited attributes (i.e., it's trivially true).

Evaluating L-attributes

The definition of L-attributes lends itself very nicely to a depth-first traversal for evaluation (that is, start at the root and recursively visit the children left-to-right). This is because symbols on the right side of a production can only rely on symbols to the left of that symbol. Thus, you can evaluate the attributes with a function:

```
function dfvisit (node)
    foreach child m of node from left to right do
        evaluate inherited attributes of m
        dfvisit (m)
    end
    evaluate synthesized attributes of node
```

Translation Schemes

A translation scheme is a context-free grammar where attributes are associated with each symbol and actions are specified within the productions, usually with braces ({}). Note that this is not the same as a syntax-directed definition where semantics have simply been associated with the grammar rules. We must actually specify *when* the actions are to take place.

Here again is an infix-to-postfix translator, as a translation scheme:

$E \rightarrow T R$

$R \rightarrow \mathbf{addop}\, T\ \{\texttt{print(addop.val)}\}\ R_1$

$R \rightarrow \epsilon$

$T \rightarrow \mathbf{num}\ \{\texttt{print(num.val)}\}$

With S-attributed definitions, translation schemes are essentially the same as the definition (attributes are computed at the end). However, with L-attributed definitions, you must follow some rules:

1. An inherited attribute for a symbol must be computed in an action before that symbol.

2. No action can refer to a synthesized attribute of a symbol to the right.

3. A synthesized attribute is computed at the end of the rule.

Example:

$S \rightarrow A_1 A_2$ {`A1.in = 1; A2.in = 2;`}

$A \rightarrow a$ {`print (A.in);`}

This translation scheme fails condition 1. To fix it, we must put the action before the `print` statement:

$S \rightarrow$ {`A1.in = 1; A2.in = 2;`} $A_1 A_2$

$A \rightarrow a$ {`print (A.in);`}

Evaluating L-attributed Definitions Bottom-up

While it may not seem intuitive, it is possible to evaluate L-attributed definitions using a bottom-up technique. If you haven't noticed, all L-attributed definitions have been evaluated top-down (like the previous example). Earlier, bottom-up evaluation relied on actions being executed just prior to the rule being reduced.

To perform actions inside of a production (like the previous translation scheme), you insert "dummy" rules whose sole purpose is to execute a given action.

$E \rightarrow TR$

$R \rightarrow \mathbf{addop}\, T\ M\ R_1$

$M \rightarrow \epsilon\ \{\texttt{print(addop.val)}\}$

$R \rightarrow \epsilon$

$T \rightarrow \mathbf{num}\ \{\texttt{print(num.val)}\}$

`bison` forces you to put actions at the end of the rule, but you can embed actions inside the rule. In this case, `bison` will automatically generate a dummy rule to perform the action given.

A Caveat:

Inserting these embedded actions can cause shift/reduce conflicts. Observe:

```
%%
thing: abcd | abcz
abcd: ‘A’ ‘B’ ‘C’ ‘D’
abcz: ‘A’ ‘B’ ‘C’ ‘D’
--------------------
%%
thing: abcd | abcz
abcd: ‘A’ ‘B’ {action} ‘C’ ‘D’
abcz: ‘A’ ‘B’ ‘C’ ‘D’
```

The first is unambiguous. The second, with the dummy rule, makes it such that the parser does not know whether to shift `‘C’` or reduce the dummy rule.

Part of the reason that these dummy rules work is that when the rule is entered, the context is known and it is allowed to reference items below itself on the stack (note that each dummy rule is uniquely named, else this wouldn't work). If context can be *assured*, it is quite legal to reference below the top of the stack to get attribute values. (see Tables 3 and 4)

When context cannot be assured, you can use dummy variables to make sure back-references work.

$S \to aAC$	`C.i = A.s`
$S \to bABC$	`C.i = A.s`
$C \to c$	`C.s = g(C.i)`
$S \to aAC$	`C.i = A.s`
$S \to bABMC$	`M.i = A.s; C.i = M.s`
$M \to \epsilon$	`M.s = M.i`
$C \to c$	`C.s = g(C.i)`

Table 2: Inserting M allows assures that `C.i` gets the needed value.

Translation scheme	Code Fragment
$D \rightarrow T$ {`L.in = T.type`} L	
$T \rightarrow$ **int** {`T.type = integer`}	val[ntop] = integer
$T \rightarrow$ **real** {`T.type = real`}	val[ntop] = real
$L \rightarrow$ {`L1.in = L.in`}	
L_1, **id** {`addtype`}	addtype(val[top], val[top - 3])
$L \rightarrow$ **id** {`addtype`}	addtype(val[top], val[top-1])

Table 3: 'top' (top of the stack before a reduction), 'ntop' (after).

Input	State	Production
`real p, q, r`	-	
`p, q, r`	**real**	
`p, q, r`	T	$T \rightarrow$ **real**
`, q, r`	T `p`	
`, q, r`	$T\ L$	$L \rightarrow$ **id**
`q, r`	$T\ L$ `,`	
`, r`	$T\ L$ `, q`	
`, r`	$T\ L$	$L \rightarrow L,$ **id**
`r`	$T\ L$ `,`	
	$T\ L$ `, r`	
	$T\ L$	$L \rightarrow L,$ **id**
	D	$D \rightarrow TL$

Table 4: Note how T is always just below L.

Avoiding Inherited Attributes

One way of dealing with inherited attributes is to avoid them whenever possible. This usually involves changing the grammar in some way. For identifiers, you could make the type come at the end of the declaration.

Another method, if possible based on the dependencies, is to do a combination of bottom-up and top-down. While parsing and building a parse/syntax tree, you calculate synthesized attributes. When you get to the point where you have the needed values, you descend the tree and calculate as needed.

For example, the identifier declarations in C. Since you can have a long list of identifiers with the same type, build the list of identifiers then, when you get to the node to reduce it to a declaration, run through the list and insert the type into the symbol table for each identifier.