Semantic Action Notes

Dan Beatty

March 19, 2006

Semantic Actions are embedded with the syntax analysis functions capturing the semantics of the program.. As syntax is being analyzed, a semantic representation of the program is produced. There is such a thing as a quad table, and this table captures the meaning of the program in a concise way.

Semantic actions are associated with the syntax production rules they accompany. Semantics are an intricate part of a languages constructs. Furthermore, some semantics can define the very paradigm that a language represents.

The main semantic types shared in common with most languages are:

1. Expression statements
2. Conditional statements
3. Flow Control Statements
4. Declaration types
   (a) Identifier
   (b) Identifier Lists
   (c) Array Sub lists (indices).

Two benefits of using intermediate - machine independent form:

1. Retargeting is facilitated; a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
2. A machine-independent code optimizer can be applied to the intermediate representation.

[**?**] page 463

Sections 5.2 and Chapter 8 of Ullman's book deal with semantic representation. There is a concept of three-address code from common programming language constructs. Question, is this the same as what Cooke is calling the quads?

> Three address code is a linearized representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph. The syntax tree and dag in Figure **??** are represented by the three-address code sequences in Figure [**?**]. Variable names can appear directly in three-address statements, so Figure **??** (a) has no statements corresponding to the leaves in figure **??**.

A quad table is a two dimensional table containing integers. The references are to the symbol table. References to quads which will be the row subscript for the quad in the quad table which may be any branch (loops or conditional branches).

Quad operations may be represented by 4 columns (operation, operand 1 and 2, and the result). The operands, and results may be references to the symbol table or the quad table. The operations must include Turing complete operations. The operations are referenced by integers mapped to specific operations.

Table 1: Turing Operations included in most Quad Tables

+	1
-	2
	3
/	4
<	5
>	6
>=	7
<=	8
read	9
write	10
readln	11
writeln	12
eq	13
neq	14
jmp	15
jmp-true	16
jmp-false	17
assign (:=)	18

The term "three address code" is defined by three addresses (operand 1, operand 2, and the result field). The other element is the operation code. Typically an user defined name is replaced by a symbol table reference. The operator is a type of Three-Address Statement.

1. Assignment (copy)
2. Unary operator
3. Binary operator
4. Unconditional jump
5. procedural calls (branches) and their parameters
6. indexed assignments
7. Address or pointer assignments

Page 408 contains syntax directed translation into Quads.

Two basic attributes associated with either expression or statement statements to be encoded into quads are:

1. Place
2. Code

The place attribute places a name that will hold the value of the statement/ expression. In Dr. Cooke's lingo, this is a gentemp command. The code attribute generates a quad, or sequence of quads. Statements have the additional attributes:

- **begin:** Start flow control statements
- **after:** End a previously generated flow control statement

Flow control generates labels which are reminscient of the labels in assembly.

1 Symbol Table

The symbol table was introduced in the lexical analyzer to store identifiers and constants. It is also used to differentiate reserved words and symbols from identifiers. In the syntax analyzer (parser), the token type is used to ascertain proper symbols where they should be in the productions. Now in the intermediate code generation, the majority of properties in the symbol table come to bear on the issue of translating a given grammar into a simple set of regular expressions.

1.1 Names in the Symbol Table

1. "Leximes for all tokens are acquired into ST by the lexical analyzer. " [?] The ST has a operator called lookup necessary for finding these leximes.

2. Semantic actions has an operation called emit (gen_quad). This operation implants basic operations into the quad table.

3. When the statement forming a procedure body is examined, a pointer to the symbol table for the procedure appears on top of the table stack.

4. Given a production for a procedure:

   $$D \rightarrow \textbf{proc id ;} N D_1; S$$

   "An names in an assignment generated by S must have been declared in either the procedure that S appears in, or in some enclosing procedure. " [?] In other words, the scope of statement determines the relevant symbol table paths.

5. There are critical semantic actions to take in the case of statements involving assignments, and Turing Essential Operations. Ordering is determined by syntax, and the calls then determine the semantics. (Reference figure 8.15 [?].

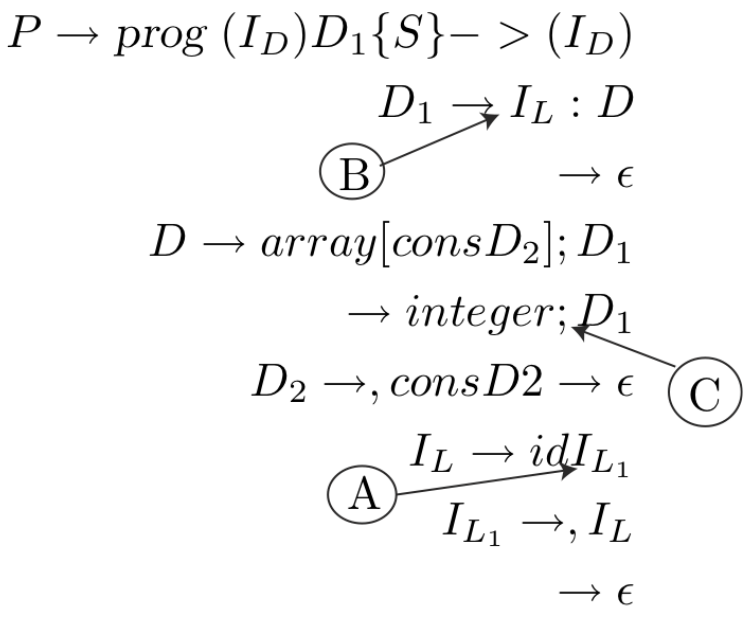

Figure 1: Declaration Semantics Example

Declaration statements primarily store information in the symbol table. The entries are initiated when a symbol is first detected in the tokenizer. The parser supplies the context of such a declaration. Declarations may include:

1. Identifiers
2. Identifier Lists
3. Array Sub lists (indices).

These three types have semantic actions that are required to obtain symbol entries from the parser. Once the parser indicates a enough information is acquired, then that information is used to augment the symbol table. Note that the quad table is not directly effected by a declaration statement.

Example:

Production
$P \rightarrow prog(I_L)D_1\{S\}- > (I_L)$
$D_1 \rightarrow I_L : D$
$\rightarrow \epsilon$
$D \rightarrow$ *array* [*cons* D_2]; D_1
$\rightarrow$ *integer* ; D_1
$D_2 \rightarrow$, *cons* D_2
$\rightarrow \epsilon$
$I_L \rightarrow I_D I_{L_1}$
$I_{L_1} \rightarrow, I_L \vert \epsilon$
$I_D \rightarrow id I'_D$
$I'_D \rightarrow$ [*Sublist*]
$\rightarrow$ *epsilon*

Algorithm 1 Semantic action α, a.k.a. Acquire Identifier, also ι Dimension Acquisition

PUSH NT Token to be adjusted in the symbol table

Algorithm 2 Semantic action β, a.k.a. start declarations

PUSH # Push a token stopper to the stack

Algorithm 3 Semantic action γ, a.k.a. start array dimensions

PUSH # Push a token stopper to the stack
PUSH NT Push a dimension on the stack

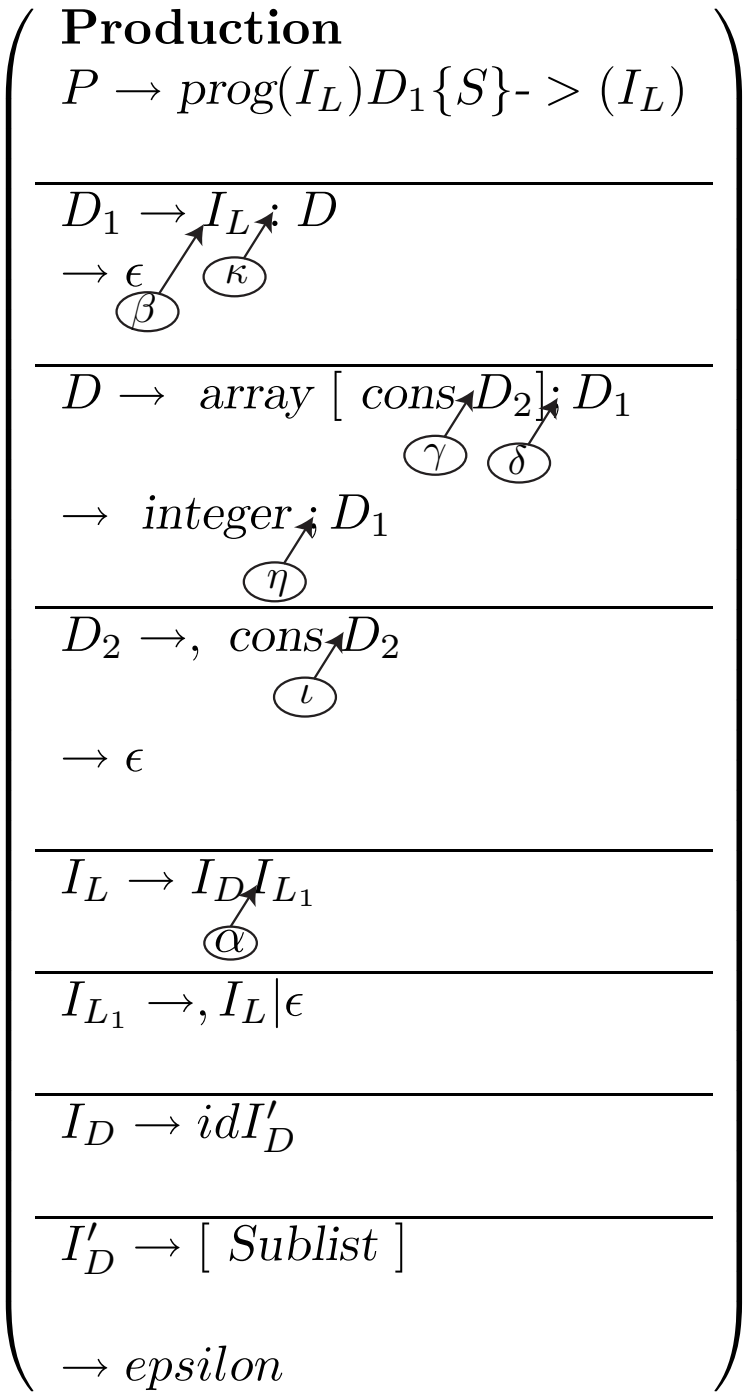

Figure 2: Declaration Semantics Example

Algorithm 4 Semantic action η

```
a := pop (SAS)
while a ≠ # do
    ST[a].type := integer
    a := pop (SAS)
```

Algorithm 5 Semantic action δ

```
b := new vector
a := pop (SAS)
while a ≠ # do
    b.add(a)
    a := pop (SAS)
a := pop (SAS)
while a ≠ # do
    ST[a].type := array
    ST[a].dimensions := rev(b)
    a := pop (SAS)
```

1.2 Evaluations

Evaluation semantics: $<$, $>$, $\leq$, $\geq$, eq, neq, $+$, $-$, $*$, $\div$, and mod all have similar semantics:

$$E \rightarrow \nu E_1 E_2$$

$$C \rightarrow \nu E_1 E_2$$

Algorithm 6 Semantic action α

```
a := POP (SAS)   Results of E1
b := POP (SAS)   Results of E2
c := gentemp
genquad ( ν, b, a, c)
```

1.3 Reusing Temporary Names

What actions should occur with semantic action operation newtemp?

1. The operation requests a new entry in the scope's symbol table.
2. Also, it is with these temporary variables that optimizations can be made.
3. Where are temporaries generated:

- Majority in syntax directed translation of expression.
- "The lifetimes of these temporaries are nested like matching pairs of balanced parentheses. " An optimization may involve modifying gentemp (newtemp) such that "it uses a small array in a procedure's data area to hold the temporaries." This array is organized as a stack, and tends to be used in this fashion.

4. "A reasonable strategy is to create a new name whenever we create an additional definition or use for a temporary or move its computation."

1.4 Addressing Array Elements

1. "Elements of an array can be accessed quickly if the elements are stored in a block of consecutive locations." The equation is

   $$A[i] = A_{base} + (i - \textit{lower-bound}) \times \textit{width}$$

   Various means of computing this equation in the semantic actions can be performed depending on how the declaration of the array is handled. For example, the algebraic rewrite of $A[i]$ is

   $$A[i] = A_{base} - \textit{lower-bound} \times \textit{width} + i \times \textit{width}$$

   The first sum

   $$C = A_{base} - \textit{lower-bound} \times \textit{width}$$

2. "Compile-time precalculation can also be applied to address calculations of elements of multi-dimensional arrays." This is a classic representation issue that must be addressed in most stages of language and compiler design. It can obviously be ignored in lexical analysis.

   In most cases, it is simply a matter of define the following parameters

   - Lower bound
   - Fastest Variance in Subscripts

3. A more significant issue in intermediate code generation is generating references for array(s) and their indices.

   - Left most expression semantic action association:
     This idea binds the semantic actions of the array to the first index of the list as opposed to the array identifier itself.
   - Number of dimensions and array of dimension values must be two properties of the symbol table record(s).
   - Included in the symbol table must be location and offset of the array.

1.5 Translation Scheme for Addressing Array Elements

Diagram for the example.

1.6 Type Conversion within Assignment

> In practice, there would be many different types of variables and constants, so the compiler must either reject certain mixed-type operations or generate appropriate coercion (type conversion) instructions.

This issue is not in the grammar that is being presented in Cooke's class, but is common in procedural languages.

Figure 8.19 shows the algorithm of Type Conversion for simple addition expressions. The concept deals with converting arguments into compatible types. Obviously, there is not always a common type between two arguments. This issue is critical as data types are allowed to become more complex.

The example presented by Ullman is relatively simple and present in most commonly used procedural and object oriented languages.

1.7 Accessing Fields in Records

> The compiler must keep track of both the types and relative addresses of the fields of a record. An advantage of keeping this information in symbol-table entries for the field names is that the routine for looking up names in the symbol tables can also be used for field names.

Note that in all of this, words may have been stated about operations based on type. However, indirect memory access was not one of them.

It should be noted that procedural and logic programming have two separate concepts of the boolean or logic expressions in programming. In procedural, the concept of logic is for controlling the flow of the program. Although a procedural program can evaluate a set of boolean expressions to either true or false, and hence be used to evaluate a complex expression, it is not the same as a answer set or other resolution type concepts.

This section examines the procedural component of logic evaluations. Another section in the context of logic oriented semantics handles answer set semantics and reviews existing papers on the subject.

2 Statements

Statement semantics include some boolean and numeric evaluations as well as boolean expressions.

$$\left(\begin{array}{l} S \rightarrow :=idE; S \\ \rightarrow \{S\}S^3 \\ \rightarrow \epsilon \\ \hline S' \rightarrow \ else\ \{S\}; S \\ \rightarrow ; S \\ \hline S'' \rightarrow C; S \\ \rightarrow E; S \\ \hline S^3 \rightarrow \ when\ CS' \\ \hat{}\ when\ S'' \end{array}\right)$$

Algorithm 7 Semantic action λ and γ

```
PUSH NQ
genquad (jmp, , , )
```

Algorithm 8 Semantic action μ

```
a := POP (SAS)  conditional result
b := POP (SAS)  end of the statement
c := POP (SAS)  beginning of the statement (at the jump)
genquad ( jtrue, a, , c + 1)
QUADS [c,4] := b+ 1
QUADS [b,4] := NQ
```

Algorithm 9 Semantic action η

```
a := POP (SAS)  conditional result
b := POP (SAS)  end of the statement
c := POP (SAS)  beginning of the statement (at the jump)
genquad ( jtrue, a, , c + 1)
QUADS [c,4] := b+ 1
PUSH b
```

Algorithm 10 Semantic action χ

```
b := POP
QUADS [b,4] := NQ
```

Algorithm 11 Semantic action ι

```
c := POP  Results of Conditional
b := POP  Jump coordinates after statement
a := POP  Jump coordinates before statement
QUADS [a,4] := b + 1
QUADS [b,4] := b + 1
genquad (jtrue, c, , a + 1 )
```

Algorithm 12 Semantic action κ

```
c := POP  Results of Expression
b := POP  Jump coordinates after statement
a := POP  Jump coordinates before statement
genquad ( jmp, , , a + 1 )  Jump to the statement
d := NQ  Location of the test
genquad ( - , c, 1, c )  decrement the count
e := gentemp
genquad ( ≤, c , 0, e )
genquad (jtrue, e, , a + 1 )
QUADS [a,4] := b + 1
QUADS [b,4] := d
```

Algorithm 13 Semantic action β

```
b := pop
a := id
genquad (:= , b, , a )
```

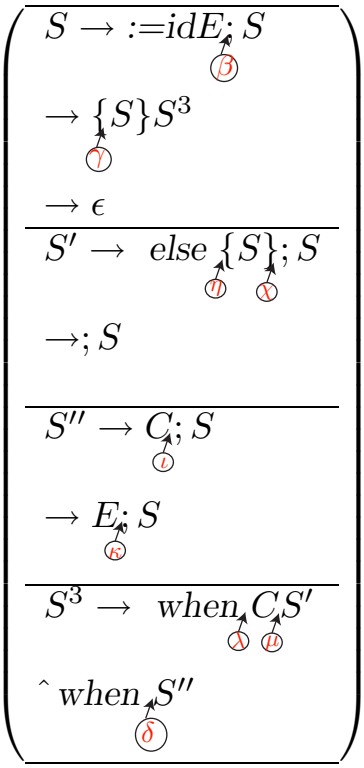

Figure 3: Statements Semantic Actions

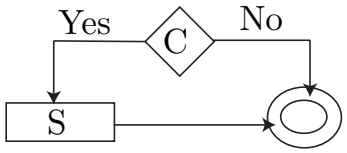

Figure 4: When Condition Semantic Actions

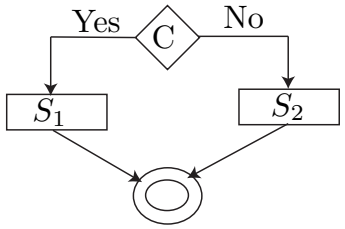

Figure 5: When Condition Else Semantic Actions

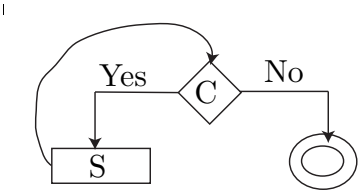

Figure 6: Carrot Condition Semantic Actions, a.k.a. a while loop

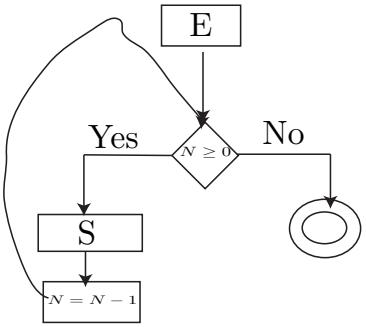

Figure 7: Carrot Expression Semantic Actions, a.k.a. a for loop

2.1 Methods of Translating Boolean Expressions

1. Principle means of encoding boolean expressions
   (a) Represent the states of boolean expressions numerically
      i. Operations become a form of boolean mathematics which inherent to the intermediate machine.
      ii. The logical operators $\wedge$, $\vee$ and $\neg$ (and, or, and not) can be represented by branching statements.
      iii. The logical operators can also be represented as actual quad operations.
   (b) Flow control representation: The method identify the boolean state by position in the program.
2. Optimizations in the flow control and numerical evaluation can be made in cases where by the outcome is determined with out evaluating the full condition.

2.1.1 Example: Conditionals

$$\begin{pmatrix} \rightarrow \ and\ C\ C \\ \rightarrow \ or\ C\ C \\ \rightarrow \ not\ E \end{pmatrix}$$

There is a brute force method for handling conditionals such as these. Three semantic actions are necessary for this prefix example. The general flow is the same for other notations, but this notation provides a very simple set of actions.

Algorithm 14 Semantic action β, a.k.a. the "And" action

```
a := pop C2 result
b := pop C1 result
c := NQ  First quad operation
d := gentemp Result of this conditional
genquad (jfalse, b, , , )
genquad (jfalse, a, , , )
genquad (:=, d, , , true )
genquad (jmp, , , , )
genquad (:=, d, , , false )
QUADS[c].results := c + 4
QUADS[c+1].results := c + 4
QUADS[c + 3].results := c + 5
```

Algorithm 15 Semantic action γ, a.k.a. the "or" action

```
a := pop C2 result
b := pop C1 result
c := NQ  First quad operation
d := gentemp Result of this conditional
genquad (jtrue, b, , , )
genquad (jtrue, a, , , )
genquad (:=, d, , , false )
genquad (jmp, , , , )
genquad (:=, d, , , true )
QUADS[c].results := c + 4
QUADS[c+1].results := c + 4
QUADS[c + 3].results := c + 5
```

Algorithm 16 Semantic action η, a.k.a. the 'not" action

```
a := pop C1 result
c := NQ  First quad operation
d := gentemp Result of this conditional
genquad (jtrue, a, , , )
genquad (:=, d, , , true )
genquad (jmp, , , , )
genquad (:=, d, , , false )
QUADS[c].results := c + 3
QUADS[c+1].results := c + 3
QUADS[c + 2].results := c + 4
```

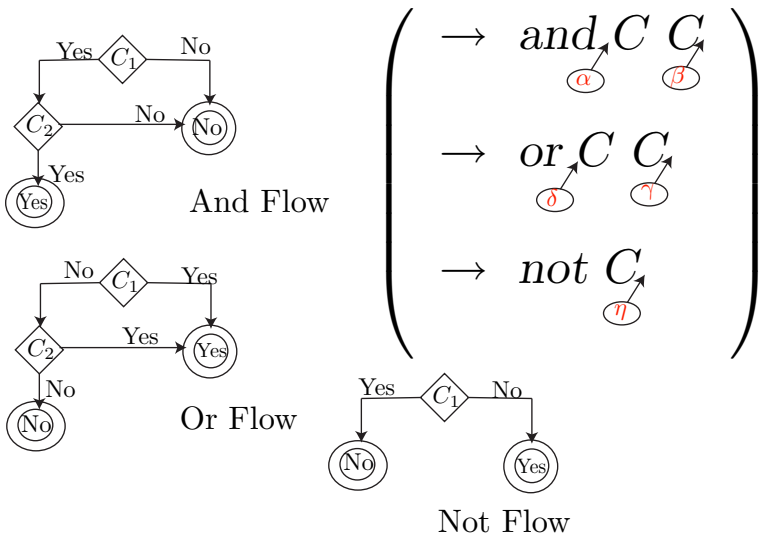

Figure 8: Conditional Statement

2.2 Flow Control Statements

The three basic procedural control constructs:

1. if - then
2. if - then - else
3. while - do

In each of these cases, there is a conditional production part of these flow control statements which is evaluated true or false, and any branching can be based those two conditions.

2.3 Mixed-Mode Boolean Expresion

A semantic to be considered in procedural programming is

$$(a < b) + (b < a)$$

In this case, it is an arithmetic expression. Not all procedural syntax allows this sort of mixing.

1. Conditionals are special types of expressions.
2. The case of mixed expressions forces conditional expressions include type information. Reference Figure 8.25.

3 Case Statements

There is a selector expression which is to evaluated, followed by n constant values that the expression might take, perhaps including a default "value," which always matches the expression if no other value does. The intended translation of a switch is code to:

1. Evaluate the expression.
2. Find which value in the list of cases is the same as the value of the expression. Recall that the default value matches the expression if non of the values explicitly mentioned in case does.
3. Execute the statement associated with the value found.

Two methods exist for implementing such a construct. One is to use a series of branch statements. The other is to use a lookup table consisting of pairs (value and code location pairs). This becomes a symbol table problem for efficiency.

Reference Figures 8.27 and 8.28 .

We process each statement **case** V_1: S_i by emitting the newly created created label L_i, followed by the code for S_i, followed by the jump *goto next*. Then when when the keyword **end** terminating the body of the switch is found, we are ready to generate the code for the n-way branch. Reading the pointer-value pairs on the case stack from the bottom to the top, we can generate a sequence of three-address statements of the form.

```
case V1, L1
case V2 L2
...
case Vn Ln-1
case t  Ln
label next
```

4 Backpatching

The main problem with generating code for boolean expressions and flow-of-control statements in a single pass is that during one single pass we may not know the labels that control must go to at the time jump statements are generated.

Labels will be indices into this array. To manipulate lists of labels, we use three functions:

1. Makelist (i)
2. Merge (p1, p2)
3. backpatch (p,i)

"Synthesized attributes truelist and falselist of nonterminal E are used to generate jumping code for boolean expressions. ... These incomplete jumps are placed on lists pointed to by E.truelist and E.falselist, as appropriate." A relational operation conditional tends to have at least two jumps: the conditional jump and the unconditional jump. Typically, the truelist belongs to the code pointed to by the conditional jump, and the false list is associated with the code following the unconditional jump.

Other productions end up getting fill-in values for their quad-labels by the backtrack operations. This is due to the fact some operations fall within the statements such that they are part of branched code. Each of these productions maintain "a list of all conditional and unconditional jumps to the quadruple following the statement S in execution order."

Flow control statements also include markers to indicate the start of productions nested within. The markers are updated by any backpatch. Another example semantic actions are prepared on Ullman's page 505.