The Select Statement

Jack Hamilton
First Health
West Sacramento, California

Abstract
The SELECT statement is an alternative to nested IF statements. It is often easier to code and understand (especially six months later) than the equivalent IF-based code. This paper presents the syntax and several typical uses of the SELECT statement.

Keywords
IF Statement, IN= Dataset option, MERGE Statement, SELECT Statement

Syntax of the SELECT Statement
The SELECT statement is described in Chapter 8, SAS Language Statements, of the SAS Language Reference. It has two variations, with and without a select-expression.

SELECT with a select-expression
Use this form of the SELECT statement when you want to base your action on the value of a single variable or expression.

```
SELECT (select-expression);
  WHEN (expression1, etc.)
    statement-1;
  WHEN (expression-n, etc.)
    statement-n;
  [lots more WHEN clauses if you want]
  OTHERWISE
    statement;
END;
```

SAS first evaluates `select-expression` (if `select-expression` is a simple variable or constant, this doesn’t amount to much). It then compares the result with each of the values in the `WHEN` clauses. The first time the two are found to be equal, the associated `statement` is executed, and processing of the SELECT group stops. If `select-expression` isn’t equal to any of the values in any of the `WHEN` clauses, the statement associated with OTHERWISE is executed. If there is no match, and you haven’t coded an OTHERWISE clause, it’s an error.

A few things to notice here:
° You can have any number of when clauses, and any number of expressions within each when clause (well, maybe not any number, but a large number).

° `statement` can be a single statement or a DO-group. IF-THEN-ELSE counts as a single statement, as does another SELECT statement.

° Only the statement associated with the first true when clause will be executed. The rest will not be executed, even if they are true.

° As with IF statements, you should have the most likely event tested first to reduce further testing.

° You’d better have an otherwise statement unless (a) you’re sure one of the when clauses will be true, and (b) you’re too lazy to write better error-handling.
**SELECT without a SELECT Expression**

Use this form of the SELECT statement when you want to select a statement to execute based on true/false tests.

```
SELECT;
  WHEN (expression-1, etc.)
    statement-1;
  WHEN (expression-n)
    statement-2;
[lots more WHEN clauses if you want]
OTHERWISE
  statement;
END;
```

Without a select-expression, the SELECT statement evaluates expressions until it finds one that’s true, then executes the associated statement. If no expressions evaluate to True, and there’s no otherwise clause, it’s an error.

**The LEAVE Statement**

The LEAVE statement, when used inside a SELECT group, causes execution to continue processing with the next statement following the SELECT group. It’s like a GOTO statement, but you don’t have to name the place you’re going to. You might use this inside an IF-THEN-ELSE statement nested in the SELECT group.

**Example 1: Checking An Expression Against A List Of Values**

It’s often desirable to see whether the value of a variable matches one of a set of values. The IF statement coding might be:

```
if (title = 'Sir') or (title = 'Lord') or (title = 'HRH') then
  paper = 'Rag';
else
  paper = 'Recycled';
```

The SELECT equivalent is shorter and less repetitious, and it’s clearer that only one value is on the left side of the comparison:

```
select (title);
  when ('Sir', 'Lord', 'HRH')
    paper = 'Rag';
  otherwise
    paper = 'Recycled';
end;
```

**Example 2: As a Replacement for the IN Operator**

This example is very similar to the first example. The IN operator is very handy in an IF statement. It lets you compare an expression to several different expressions:

```
if title in
  ('Sir', 'Lord', 'HRH') then
  paper = 'Rag';
else
  paper = 'Recycled';
```

But the IN operator has an unfortunate restriction: the items in the IN list must be constants. They can’t be expressions, so you can’t read them from data or calculate them at runtime.

The SELECT statement provides a way around this:
data called;

infile cards missover;

length name keyword1-keyword4 $8.;

input name keyword1-keyword4;
cards;
AB1234 AUTO POLICY CADDY COUPE
BC8324 MOTO BMW POLICY
DEXXXX NOTHING SPECIAL HERE OK?
*****; run;

data chosen;
set called (keep=name
    keyword1 keyword2
    keyword3 keyword4);
select ('POLICY');
    when (keyword1, keyword2,
        Keyword3, keyword4)
        output chosen;
    otherwise;
end;

Run;

**Example 3: After a Merge**

A common use of the MERGE statement is to update a master file with transactions (or to perform a table lookup, which is just a variation).

Suppose you two files called CLAIM and HOSPNAME. CLAIM has hospital claims, including a hospital code, and HOSPITAL has hospital codes and full hospital names:

<table>
<thead>
<tr>
<th>CLAIM</th>
<th>CLAIMNO</th>
<th>HOSPCODE</th>
<th>CHGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLAIM</td>
<td>9837</td>
<td>ABCD</td>
<td>250.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HOSPITAL</th>
<th>HOSPCODE</th>
<th>HOSPNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOSPNAME</td>
<td>LPMP</td>
<td>Lucille Packard Medical Pavilion</td>
</tr>
</tbody>
</table>

Suppose you wanted to match the two tables on HOSPCODE and create a new table containing the CLAIM variables plus HOSPNAME from the HOSPITAL table. A HOSPCODE in CLAIM without a match record in HOSPITAL is considered an error, and so is a HOSPCODE in CLAIM without a match in CLAIM (in other words, you want to match two datasets by key, and find out which output observations came from both input datasets, and which came from only one input dataset).
One obvious way to code this is with a MERGE statement in a data step:

```sas
data newclaim
    noname (drop=hospname)
    noclaim (drop=claimno patid chgs);
merge claim (in-inclaim)
    hospname (in=inhosp);
by hospcode;
```

After this, you can use IF statements to see where an observation’s data came from:

```sas
if inclaim then
    if inhosp then
        output newclaim;
    else
        output noname;
else
    output noclaim;
```

You could also use a SELECT statement:

```sas
select;
    when (inclaim and inhosp)
        output newclaim;
    when (inclaim)
        output noname;
    otherwise
        output noclaim;
end;
```

I claim that the SELECT construction is easier to read and understand (some might disagree, of course). So let’s try a more complicated example:

### Example 4: After a Three-Way Merge

Suppose we also have a HOSPTYPE table identifying whether the hospital is a county hospital, a university hospital, or a private hospital:

<table>
<thead>
<tr>
<th>HOSPCODE</th>
<th>HOSPTYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPMP</td>
<td>P</td>
</tr>
</tbody>
</table>

If we want to check all the combinations with IF statements, it’s suddenly much more complicated:

```sas
data cyhyty
    cyhytn (drop=hosptype)
    cyhnty (drop=hospname)
    cyhntn (drop=hospname hosptype)
    chhyty (drop=claimno chgs)
    chhytn (drop=claimno chgs hosptype)
    chhnty (drop=claimno chgs hospname)
;
merge claim (in-inclaim)
    hospname (in=inname)
    hospotype (in=intype)
by hospocode;
```
The IF statement would be complicated, too:

```plaintext
if inclaim then
    if inname then
        if intype
            output cyhyty;
        else
            output cyhyyn;
    else
        if intype
            output cyhnty;
        else
            output cyhnyn;
else
    if inname then
        if intype
            output cnhyty;
        else
            output cnhyyn;
    else
        if intype
            output cnhnty;
        else
            error "Can't happen";
```

Getting rough, huh? Imagine what it would be like with a four-way merge.

The SELECT statement version, while probably less efficient, is easier to write, and I think it’s easier to understand.

First, draw a truth table with each of the possibilities, and associate a dataset name with each. (This is almost completely brain-free; after the first time you probably won’t need the table. Having a good dataset naming scheme helps a lot.):

<table>
<thead>
<tr>
<th>INCLAIM</th>
<th>INNAME</th>
<th>INTYPE</th>
<th>Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>cyhyty</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>cyhyyn</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>cyhnty</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>cyhntn</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>cnhntn</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>cnhytn</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>cnhntn</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>
Next, write the SELECT statement, substituting the IN= variables for the T and F values in the table:

```sql
select;
    when (inclaim & inname & intype) output cyhyty;
    when (inclaim & inname & ^intype) output cyhyn;
    when (inclaim & ^inname & intype) output cyhnty;
    when (inclaim & ^inname & ^intype) output cyhnny;
    when (^inclaim & inname & intype) output cnhyty;
    when (^inclaim & inname & ^intype) output cnhyn;
    when (^inclaim & ^inname & intype) output cnhnty;
    when (^inclaim & ^inname & ^intype) output cnhntn;
    otherwise error "Can't happen";
end;
```

It’s easier to tell what’s happening with the SELECT statement, and it’s much easier to move the test for the most frequently occurring case to the front. It’s also a few lines shorter than the IF-THEN-ELSE. It’s probably less efficient, but I consider my time more valuable than the computer’s.

References


SAS Version

The examples in this paper were creating using SAS 6.10 under Microsoft® Windows® 3.1 running on a 60 MHz Pentium Processor® with 16MB of RAM.

Trademarks

SAS is a registered trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Microsoft and Windows are “either trademarks or registered trademarks of Microsoft Corporation in the USA and other countries” (you’d think Microsoft would know which it is, wouldn’t you?).