Complete macro expansion algorithm

March 7, 2008

Here is a complete implementation of macro expansion that meets the requirements of the Standard. It defines a behavior for the two currently unspecified parts of the Standard's macro expansion process. Be assured that these two parts only come into play when the expansion process is abused (when varying *hide sets* are intermixed) and as such do not have any effect on ''real'' programs.

Naming conventions: all uppercase names are variables, binding of values to variables occurs at calls and conditionals. *TS* is a ''token sequence''; *T* is a ''token''; *HS* is a ''hide set''; apostrophes are used to distinguish separate instances of these. A token and its hide set are specified as "T^{HS}". Initially, each token has an empty hide set. The operator "•" separates elements of a list (or an element from the rest of a sequence).

```
expand(TS)  /* recur, substitute, pushback, rescan */
{
    if TS is {} then
        return {};

    else if TS is T^HS • TS' and T is in HS then
        return T^HS • expand(TS');

    else if TS is T^HS • TS' and T is a "()-less macro" then
        return expand(subst(ts(T),{},{},HS∪{T},{}) • TS');

    else if TS is T^HS • ( • TS' and T is a "()'d macro" then
      check TS' is actuals • )^HS' • TS" and actuals are "correct for T"
        return expand(subst(ts(T),fp(T),actuals,(HS∩HS')∪{T},{}) • TS");

    note TS must be T^HS • TS'
    return T^HS • expand(TS');
}
```

The entire program's text as a token sequence is to be handed to *expand*, which is invoked with a single token sequence argument, *TS*. First, if *TS* is the empty set, the result is the empty set.

Otherwise, if the token sequence begins with a token whose hide set contains that token, then the result is the token sequence beginning with that token (including its hide set) followed by the result of expand on the rest of the token sequence.

Otherwise, if the token sequence begins with an object-like macro, the result is the expansion of the rest of the token sequence beginning with the sequence returned by subst invoked with the replacement token sequence for the macro, two empty sets, the union of the macro's hide set and the macro itself, and an empty set.

Otherwise, if the token sequence begins with a function-like macro and a left parenthesis, then first verify that there is, after the possibly empty token sequence comprising the actuals, a right parenthesis. If it is so, the result is the expansion of the rest of the token sequence beginning with the sequence returned by subst invoked with the replacement token sequence for the macro, the formal parameters for the macro, the actuals token sequence, the union of the intersection of the macro's hide set and the hide set of the right parenthesis and the macro itself, and an empty set.

Otherwise, the token sequence must be some other token. The result is the expansion of the rest of the

token sequence, preceded by the token (including its hide set).

The intersection operation above and the one below in the glue function are both one possible implementation choice that the draft leaves unspecified. This algorithm chose the intersection operation as it gives the most amount of macro replacement, still without causing infinite loops, etc.

```
subst(IS,FP,AP,HS,OS )  /* substitute args, handle stringize and paste */
{
    if IS is {} then
        return hsadd(HS,OS );

    else if IS is #•T•IS' and T is FP[i] then
        return subst(IS',FP,AP,HS,OS • stringize(select(i,AP )));

    else if IS is ##•T•IS' and T is FP[i] then
    {
        if select(i,AP ) is {} then         /* only if actuals can be empty */
            return subst(IS',FP,AP,HS,OS );
        else
            return subst(IS',FP,AP,HS,glue(OS,select(i,AP )));
    }

    else if IS is ##•T^HS'•IS' then
        return subst(IS',FP,AP,HS,glue(OS,T^HS' ));

    else if IS is T•##^HS'•IS' and T is FP[i] then
    {
        if select(i,AP ) is {} then         /* only if actuals can be empty */
        {
            if IS' is T'•IS" and T' is FP[j] then
                return subst(IS",FP,AP,HS,OS • select(j,AP ));
            else
                return subst(IS',FP,AP,HS,OS );
        }
        else
            return subst(##^HS'•IS',FP,AP,HS,OS • select(i,AP ));
    }
    else if IS is T•IS' and T is FP[i] then
        return subst(IS',FP,AP,HS,OS • expand(select(i,AP )));

    note IS must be T^HS'•IS'
    return subst(IS',FP,AP,HS,OS • T^HS' );
}
```

A quick overview of *subst* is that it walks through the input sequence, *IS*, building up an output sequence, *OS*, by handling each token from left to right. (The order that this operation takes is left to the implementation also, walking from left to right is more natural since the rest of the algorithm is constrained to this ordering.) Stringizing is easy, pasting requires trickier handling because the operation has a bunch of combinations. After the entire input sequence is finished, the updated hide set is applied to the output sequence, and that is the result of *subst.*

```
glue(LS,RS)  /* paste last of left side with first of right side */
{
    if LS is L^HS and RS is R^HS' • RS' then
        return L&R^(HS∩HS') • RS';          /* undefined if L&R is invalid */

    note LS must be L^HS • LS'
    return L^HS • glue(LS',RS);
}

hsadd(HS,TS)  /* add to token sequence's hide sets */
{
    if TS is {} then
        return {};

    note TS must be T^HS' • TS'
    return T^(HS∪HS') • hsadd(HS,TS');
}
```

The remaining support functions are:

ts(*T*)	Given a macro-name token, *ts* returns the replacement token sequence from the macro's definition.
fp(*T*)	Given a macro-name token, *fp* returns the (ordered) list of formal parameters from the macro's definition.
select(*i*,*TS*)	Given a token sequence and an index *i*, *select* returns the *i*-th token sequence using the comma tokens (not between nested parenthesis pairs) in the original token sequence as delimiters.
stringize(*TS*)	Given a token sequence, *stringize* returns a single string literal token containing the concatenated spellings of the tokens.

-- D. F. Prosser

$Id: cpp.algo.mm 1.9 2008/03/07 12:19:47 dds Exp $