Macros for Quintus Prolog

Peter Schachte

There are two things that are typically needed from a macro system: compile-time computation, and substitution of code at compile time. I propose separating these two concepts. Therefore this spec is actually two specs: one for compile-time computation, and one for substitution macros.

The aim of this design is for macros to be powerful and efficient, but most of all, for them to be as consistent as possible with the current Prolog system. A macro definition is meant to look just like a clause, and more importantly, to have the same denotation as a clause. It should always be possible to turn a macro definition into a procedure without changing the behavior of the system. Similarly for compile-time computation. Modulo differences between what procedures are available at compile-time vs. runtime, compile-time code should have the same denotation as if it were normal runtime code. This should make it easy to learn and use the macro system.

1. Substitution Macros

A substitution macro clause looks just like any other clause, except that instead of ":-" separating the head and body, it has ":->." The sole restriction on substitution macros is that they must not contain cuts. [1]For example, to code **nl**/0 as a macro, one might write:

```
nl :-> put(10).
```

Then, if one writes

```
hello :- print('hello'), nl.
```

this will generate code exactly as if one had written

```
hello :- print('hello'), put(10).
```

This "disappearing" seems OK, as replacing with 'fail' or giving an error gen. two much code in first case, is a pain in second. But doesn't this lead to easy bugs ?? ??

1.1. Multiple Macro Clauses

It is also permitted to have more than one clause defining a macro. For example, we might define

```
(1) (target_os(unix)) :-> !.

    nl :-> target_os(xerox), put(13).
    nl :-> target_os(unix), put(10).
```

!-> stuff.

This will produce exactly the same code as in the **hello**/0 example above. This is because the macro call **target_os**(xerox) fails, while **target_os**(unix) succeeds, so only the second of the macro clauses applies. 2 How does this work?

(2) Notice that the unit macro clause for **target_os** ends with ':-> .' This is essential to distinguish it from a normal unit clause.

OK, I think.

In cases where it is not possible to choose a single macro clause at compile time, all applicable clauses must be represented. For example

```
character_code(a,97) :-> .
character_code(b,98) :-> .
...

putchar(Char) :- character_code(Char, Code), put(Code).
```

[1]Allowing cuts in macros would greatly complicate the macro expander. Perhaps a later version of the macro system should allow cuts by turning several macro clauses with cuts into a single macro clause with an if-then-else. For now, users can get the same effect with if-then-else, so they shouldn't complain too much.

would generate the following code:

```
putchar(a) :- put(97).
putchar(b) :- put(98).
...
```

This code will be more efficient than a single clause with a disjunction, because this code can take advantage of clause indexing.

1.2. When Macros Fail

One important use of macros is to provide conditional compilation. For example, a routine to skip characters up to and including the next end of line might be written:

```
target_os(unix) :-> .

to_eol :- target_os(xerox), repeat, get(C), C=13.
to_eol :- target_os(unix), repeat, get(C), C=10.
```

This would generate the following code:

```
to_eol :- repeat, get(C), C=10.
```

Since it can be determined at compile-time that **target_os**(xerox) will fail, so the first clause of **to_eol**/0 will fail, so it can simply be dropped from the compiled code.

1.3. Restrictions

It was mentioned earlier that the only restriction on the content of a macro is that it must not contain cuts. There is, however, one restriction on the use of macros: it is not permitted to have a macro and a normal predicate by the same name. That is, one cannot write macro clauses for the same predicate one has normal clauses for. This is because the semantics of such a hybrid would be rather baroque. fine

If this were to be allowed, the semantics of macros could not be maintained without loosing much of the efficiency they provide, since then every macro would be implicitly disjoined by any clauses that might later be defined for that predicate. This would mean that all macro expansions would have to add another disjunct or another clause to call the macro predicate at runtime.

Another way to try to make sense out of mixing macros and normal procedures with the same name and arity would be to change the semantics of macro failure. We could say that if a macro call fails at compile time, then the call is simply left as is in the source clause, and is not expanded at all. There are two problems with this approach, however. Firstly, we could no longer use macros to effect conditional compilation, since a clause that begins with a failing macro call could not be dropped from the source code. Secondly, and more importantly, the semantics of a macro clause would change radically. The new semantics would be that macro clauses would implicitly end with a baroque sort of cut that cuts all non-macro clauses for the procedure, but not any macro clauses.

Seems sensible to make the restriction...

1.4. When To Use Macros

After writing a bit of code using this sort of macro it suddenly becomes unclear what code should be written as macros and what should be written normally. Any procedure that can be written without cuts can be coded as a macro. That is to say, *any* procedure can be written as a macro, since any procedure with cuts can be written without cuts by using \+/1 or ->/2. So which things do you write as macros?

The trade-offs to be considered are speed *vs.* code size. Macros will never make code run slower, and can sometimes make code run much faster. Macros can sometimes even make for smaller code. However, macros can make code much larger without giving much of a speed-up. Following are a few instances when it would be a good idea to use a macro:

- When a predicate is defined by a single clause with at most one goal in the body.

- When a predicate is defined by a single clause, and is not called from very many places in the program.
- When a predicate is called in some time-critical section of code.
- When conditional compilation is needed.
- When it will frequently be possible to select a single clause of many for a predicate at compile-time.

This list is by no means exhaustive. After gaining more experience using macros to write real code, I'm sure we could draft a better set of heuristics.

2. Compile-time Computation

Any code appearing between "{" and "}"[2] in the input file will be evaluated at compile-time. Any variable bindings produced by this evaluation will, of course, be propagated throughout the rest of the enclosing clause. The only restriction on a compile-time computation is that, like a substitution macro, it must not contain any cuts. ✱ The semantics of a clause containing compile-time code is exactly the same as if the code were not enclosed in these brackets, assuming that all the predicates called within the brackets are defined at compile-time.

?? =/2 ?

(there is this funny time-dependency...

So, for example,

```
    first5(X, L) :- {length(L, 5), append(L, _, X)}.
```

would compile into the more efficient clause

```
    first5([X1,X2,X3,X4,X5|_], [X1,X2,X3,X4,X5]).
```

2.1. Nondeterministic compile-time code

Compile-time computations may be nondeterministic, just like any Prolog code. The behavior of the code will be just as if the code were evaluated at runtime. For example,

```
    vowel(L) :- {member(L, [a,e,i,o,u])}.
```

would compile into

```
    vowel(a).
    vowel(e).
    vowel(i).
    vowel(o).
    vowel(u).
```

3. Optimization Restrictions

There are three classes of restrictions on optimizations for both substitution macros and compile-time computations: variable binding restrictions, multiple clause restrictions, and dropped clause restrictions. These restrictions become important when extra-logical features of Prolog clash with logical optimizations. Each of these classes of restrictions is described below.

[2] For concreteness, I'll use "{" and "}" to delimit compile-time code, but if anybody has a better pair in mind, I'm open to substitutions. {} are already used in grammar rules, and the semantics here are different, so we should probably pick another pair.

3.1. Variable Binding Restrictions

There are times when it is not permissible to propagate a variable binding leftward in a clause. This occurs when a macro (or compile-time code) appears to the right of some non-logical goal in the clause, and the macro tries to bind a variable that can be "seen" by the non-logical goal. A variable can be "seen" by a goal if it appears in that goal, or if it appears in any other goal that mentions any variable that can be "seen" by the goal. For these purposes, the head of the clause is included in the set of goals to the left of the macro.

An example would better illustrate the problem. If we wrote, for example,

```
    one(1)  :-> .
    foo(X)  :- var(X), one(X).
```

then we would expect **foo**(A) to succeed only when A is not bound, and would then bind A to 1. If we performed the usual expansion of this compile-time code, however, we would get

```
    foo(1)  :- var(1).
```

This code, of course, makes no sense; **foo** can never succeed. Therefore, we must forgo the usual optimization of propagating the binding of X, and place a call to =/2 in the resulting code:

```
    foo(X)  :- var(X), X=1.
```

Unfortunately, the Prolog system cannot determine which goals are non-logical, so it must assume that all user-defined predicates are non-logical.[3] The macro system includes a table of all logical built-in predicates, and assumes that any predicate not mentioned there is non-logical. This table will be described in greater detail below. This means that in order to be most effective, macros that try to bind previously-appearing variables should usually appear at the front of a clause, right after the :-.

How much efficiency is lost here ??

3.2. Multiple Clause Restrictions

In the **vowel**/1 example above, we saw how more than one clause could be produced when only one clause appeared in the input file. This optimization must not be done when the macro (or compile-time code) appears after any code that can have side-effects. This is because if backtracking should occur, the side-effects could happen more than once. By the same token, if the macro (or compile-time code) appears after any declarative goals that may take some time to compute, it would not make sense to generate multiple clauses, since the work might be done more than once.

what ???

???

In these cases, the macro system must generate a disjunction. For example, if the **vowel**/1 example had been written:

```
    vowel(L)  :- write(L), {member(L, [a,e,i,o,u])}.
```

then it would have to be compiled as:

```
    vowel(L)  :- write(L), (L=a; L=e; L=i; L=o; L=u).
```

but this could be a general-purpose call, though!

So the assumption must be made

3.3. Dropped Clause Restrictions

In the **to_eol**/0 example above, the macro system was able to drop one clause from the input program. Whenever a macro call that fails follows any goal that will have a side-effect, the clause cannot be dropped, since if it were, the side-effect would not happen. Instead, the macro system must generate an explicit **fail**. Note that clauses *may* be dropped if the failing macro call is after a metalogical goal or a time-consuming declarative goal; only a true side-effect goal invalidates this optimization. Of course, in the absence of explicit declarations to the contrary, we must assume that any user-defined predicates will have side-effects.

If, for example, we defined a runtime predicate

messy "listing"
macro expansion does not immediately give you "what you'd expect"

[3]We will discuss the option of allowing users to declare their code to be declarative below.

```
    target_os(unix) :-> .

    non_unix :- !, target_os(xerox).
    non_unix.
```

then we could not simply drop the first clause for non_unix. We would have to generate the code

```
    non_unix :- !, fail.
    non_unix.
```

4. Declarations

There are two metapredicates needed by the macro system to contain declarations about predicates: one for higher-order predicates, and one to indicate degree of declarativeness of predicates.

4.1. Higher-Order Predicates

The macro system *must* know of the existence of any higher-order predicates, in order that subgoals be macro expanded. This declaration is necessary to the correct operation of the macro system. Since the module system will need similar information for similar reasons, the two requirements should be folded together.

For each higher-order goal, we must be able to get the arguments that are to be called, process them, and rebuild the goal with the processed arguments. One way to do this is with a 4-ary predicate that has as arguments

1. The form of the higher-order goal;
2. The name/arity of the predicate;
3. A list of the arguments to be called;
4. A list of the arguments not to be called.

This might look like:

```
    higher_order_pred(bagof(X,Y,Z), bagof/3, [Y], [X,Z]).
    higher_order_pred(call(X), call/1, [X], []).
    ...
```

4.2. Declarativeness of Predicates

Each of the three types of optimizations described in the last section rely upon information about the goals occurring before a macro or compile-time code. The macro system distinguishes four classes of predicates: *trivial*, *declarative*, *metalogical*, and *procedural*.

Procedural predicates may do anything. Cut and all Input/Output predicates are procedural. User predicates are assumed to be procedural (see the discussion below about user declarations).

Metalogical predicates do not have side-effects, but calls to them may be effected by whether their args are bound or not. Metalogical predicates include **var**/1, **number**/1, *etc.*

Declarative predicates do not have side-effects, and calls to them cannot be made incorrect by binding previously unbound variables. The may, however, take some time to evaluate. **=..**/2, **length**/2, **arg**/3, and many more are declarative.

Trivial predicates are declarative and take effectively no time to compute. This class of predicate includes **true**/0, **fail**/0, and **otherwise**/0.

by this

I'm suspicious of this 'time to compute' stuff

4.3. User Declarations

Should users be permitted to make these kinds of declarations about their code? If they are to be permitted to write their own higher-order predicates, an important capability, they must be allowed (required) to declare them, both for the macro system, and for modules. But should they be permitted to make declarations about the degree of declarativeness of predicates?

The argument against allowing such declarations is that this will complicate user's code, and, if users make incorrect declarations, create the possibility of erroneous code. The argument in favor of allowing users to make declarations is that they are an optional way for users to increase the quality of their compiled code, and that the information users supply for macro optimization may be useful for general source-level optimizations.

On balance, I think the advantages of allowing user declarations outweigh the disadvantages. Perhaps more thought should be given to the nature of the declarations allowed, so the release of this capability to users should be held back until a later Prolog release, but I think the capability should be provided eventually. We should probably make some policy decision about this.

5. User Interface Issues

There are several questions about the user interface for macros. What should macros, and compile-time code, look like to the debugger? How can we make sure that macros are defined before they are used? How can we handle multifile macros? What happens when all the clauses for a predicate get dropped due to macro failure? What do we do when the user changes a macro without recompiling all the procedures that use it? How do we insure the consistency of the code available at compile-time versus what's around at runtime?

3.0: Could not expand dynamic code, and expand all else, if that distinction will still be made...

5.1. Macros and the Debugger

It's not really clear what is the *right* way to debug code that uses macros.

One appealing option would be to make use of the fact that macros are "just like" normal predicates, and not expand them at all for interpreted code. That is, we could treat macro definition clauses as any other clause when consulting files.

This would be a very natural way to debug code that uses macros. But its advantage of distancing the user from the macro-expansion mechanism has the disadvantage for sophisticated users of not allowing them to see what kind of code the system generates. This will hinder them from using macros to do what they are meant for: optimize code without sacrificing clarity. We could give a "test expansion" option...

I propose modifying **expand_term**/2 to also do macro expansion, giving the sophisticated user the ability to experiment with macros in order to produce the best possible code. Additionally, we should make macro definitions be valid clauses, so macro expansion would not be done at consult-time, only at compile-time. A definition such as

```
{A} :- call(A).
```

yes!

should handle compile-time evaluation reasonably. Maintaining the consistency of code available at compile-time and runtime is discussed below.

5.2. Ensuring Macros are Defined Before Use

A problem occurs with macros that doesn't occur with normal procedures: macros *must* be defined before they are used (where "used" means compiled, or invoked by interpreted code). And since a macro call looks just like a normal goal, a call to an undefined macro will compile into a call to a predicate of the same name and arity. (We made use of this fact in the previous subsection.) So how do we make sure

IS a problem if we go from static/dynamic to normal/fastcode that all the definitions happen in the right order?[4]

There are two levels at which we must ensure that macros are defined before their use: intra-file and inter-file. Intra-file consistency is relatively easy to maintain. It is permissible for a macro call to appear in a *macro* before the called macro is defined, however all macros called, directly or indirectly, by a clause must be defined before that clause is compiled. Therefore it is sufficient to insure that all macros used in a given file appear before any clauses in that file

Unfortunately, things are often not so simple. How can we maintain consistency across multiple files, or multiple modules? The only safe and simple policy is to insure that all macros are defined before any clauses are read. This can be guaranteed by fixing the order of the various parts of a module in the file as follows:

1. Module heading stuff,
2. Macro definitions,
3. Imports,
4. Clauses.

This is not really satisfactory, though, because it puts unnatural constraints on programmers, and may force them to separate related parts of a program. It also does not cover cases where macros may be generated by other code. Many such problems arise when the model of the programming environment is file-based. If the mode of use of the system were such that all files were loaded before any compiled code could be generated, these problems would all disappear. (This would also allow many classes of global optimization that are not possible using our one-clause-at-a-time model of compilation.) In the long run, I think we should move toward this type of system; in the mean time, putting macros first will have to suffice.

NO WAY!

NO. Procedure compilation could give this w/o doing all files at a time!

5.3. Multifile Macros

Using the order of parts of a module recommended above, multifile macros present no problems at all. In fact, since macros are not permitted to contain cuts, the order of clauses defining a macro is immaterial, so the loading order of the files containing multifile macros is immaterial, too. The only provision we must make is to extend the **multifile** declaration to work for macros, too.

5.4. Completely Dropped Procedures

Since under Quintus Prolog calling an undefined predicate does not simply fail, we must be careful about dropping *all* the clauses for a procedure. The simple solution to this problem is to make a small extension to the undefined procedure trapping code so that it checks to make sure that it is not a procedure all of whose clauses were dropped before it starts tracing. This should be easy to implement.

but is time-consuming. Why not generate one foo :- fail. clause?

5.5. When a Macro Is Changed

But what happens when the user changes a macro? What must then be done in order to make the environment reflect the change? The first thing to notice is that only compiled code will be affected, since macros are treated as regular procedures by interpreted code. But what can be done for compiled code?

One thing is clear: the macro system must maintain a list of all predicates that call each macro, at least for compiled predicates. This is easily done when the macros are being expanded. But what do you do when the user changes one of these macros?

hmm

[4]It is worth noting that Lisp systems face the same problem, and none of them are able to handle it properly.

One complete but time-consuming solution would be to use **source_file**/2 to find the files that contain the predicates needing to be recompiled, and then go through the files recompiling all ~~clauses~~ that need to be recompiled due to the macro redefinitions. This is probably the best solution. procedures

This is another case where loading files, and then compiling the incore version, would make a problem easier to handle. In this case, we could just recompile the predicates that had changed from the incore source code.

No way!

I am not convinced that this is the right view of it

5.6. Consistency of Compile-time vs. Runtime Code

Here, once more, we see the advantage of separating compilation from loading. If all predicates were loaded and interpretable before any attempt was made to compile them, then compile-time would be the same as runtime.

Since our compiler doesn't work that way, all we can really do is provide users with a few rules that will keep them out of trouble. The simplest rule is that any built-in predicate, or any predicate defined in a loaded module can be used freely in compile-time code. The second rule is that if the user does get the order wrong and calls an undefined predicate at compile time, the Prolog system will tell him he is calling an undefined predicate; he won't wind up compiling erroneous code without knowing it. Beyond that, things get tricky.

When compile-time code appears in a normal clause, it is pretty easy to verify that the called predicate is defined earlier in the file. The interesting case is when compile-time code appears in a macro. If the macro is not used in the module it is defined in (it is only exported), and all the predicates used as compile-time code in the macro are defined in the module, or one it imports, everything is alright. If the module *is* used in the module, however, the user will just have to be careful.

6. A Few Larger Examples

6.1. Repeat a goal

repeat(Goal, N) will repeat Goal N times. Note that N must be bound to a positive integer at compile-time.

```
repeat(_, 0) :-> .
repeat(Goal, N) :-> {N>0, N1 is N-1}, Goal, repeat(Goal, N1).
```

So, the code produced by

```
test :- repeat(nl, 5).
```

would be

```
test :- nl, nl, nl, nl, nl.
```

?

6.2. Conditional Compile-time Code Expansion

expand_if(Test, Goal) is equivalent to **call**(Goal), except that if **call**(Test) succeeds at compile-time, then Goal will be evaluated at compile-time, else it is evaluated at runtime.

expand_if_bound(Term, Goal) is equivalent to **call**(Goal), except that if Term is bound at compile-time, then Goal will be evaluated at compile-time, else it is evaluated at runtime.

expand_if_ground(Term, Goal) is equivalent to **call**(Goal), except that if Term is bound to a ground term at compile-time, then Goal will be evaluated at compile-time, else it is evaluated at runtime.

X **est** Y is equivalent to X **is** Y, except that if Y is bound to a ground term at compile-time, the calculation will take place at compile-time.

```
    expand_if(Test, Goal) :-> {Test, Goal}.
    expand_if(Test, Goal) :-> {\+ Test}, Goal.

    expand_if_bound(Term, Goal) :-> expand_if(nonvar(Term), Goal).

    expand_if_ground(Term, Goal) :-> expand_if(ground(Term), Goal).

    :- op(700, xfx, est).

    X est Y :-> expand_if_ground(Y, (X is Y)).

    ground(X) :- numbervars(X, 0, 0).
```

So if you wrote

```
    foo(X, Y) :- Z est 256*256, Y est X*Z.
```

the macro expander would produce

```
    foo(X, Y) :- Y is X*65536.
```

correctly determining that the first **est** could be evaluated at compile-time, while the second could not.

6.3. Abstract Datatypes

This kind of macro will support the efficient compilation of a simple abstract datatype paradigm. The following geometric example gives the idea, albeit without the syntactic sugar that would make the paradigm usable. Such a syntactic sugar could presumably be found; in the mean time, this gives the rough idea.

```
/* Type predicates */
polygon(X) :-> quadrilateral(X).
polygon(X) :-> triangle(X).

quadrilateral(quad(_,_,_,_,_,_,_,_)) :-> .
quadrilateral(X) :-> rectangle(X).

rectangle(rect(_,_)) :-> .
rectangle(X) :-> square(X).

square(sq(_)) :-> .

triangle(tri(_,_,_,_,_,_)) :-> .

/* Selector predicates */
side1(quad(S,_,_,_,_,_,_,_), S) :-> .
side2(quad(_,S,_,_,_,_,_,_), S) :-> .
side3(quad(_,_,S,_,_,_,_,_), S) :-> .
side4(quad(_,_,_,S,_,_,_,_), S) :-> .

side1(rect(S,_), S) :-> .
side2(rect(_,S), S) :-> .
side3(rect(S,_), S) :-> .
side4(rect(_,S), S) :-> .

side1(square(S), S) :-> .
side2(square(S), S) :-> .
side3(square(S), S) :-> .
side4(square(S), S) :-> .

side1(tri(S,_,_,_,_,_), S) :-> .
side2(tri(_,S,_,_,_,_), S) :-> .
side3(tri(_,_,S,_,_,_), S) :-> .

/* compute perimeter */
perimeter(X, P) :->
        quadrilateral(X),
        side1(X, S1),
        side2(X, S2),
        side3(X, S3),
        side4(X, S4),
        P est S1+S2+S3+S4.
perimeter(X, P) :->
        triangle(X),
        side1(X, S1),
        side2(X, S2),
        side3(X, S3),
        P est S1+S2+S3.
```

Given this code,

```
test1(P) :- square(X), side1(X, 7), perimeter(X, P).
```

would produce

```
test1(28).
```

and

```
test2(S, P) :- square(X), side1(X, S), perimeter(X, P).
```

would produce

```
    test2(S, P) :- P is S+S+S+S.
```

More realistically,

```
    quad_perim(Q, P) :- quadrilateral(Q), perimeter(Q, P).
```

would produce

```
    quad_perim(quad(S1,S2,S3,S4,_,_,_,_), P) :-
            P is S1+S2+S3+S4.
    quad_perim(rect(W,H), P) :-
            P is W+H+W+H.
    quad_perim(sq(S), P) :- P is S+S+S+S.
```

And the final test case:

```
    poly_perim(X, P) :- polygon(X), perimeter(X, P).
```

would produce

```
    poly_perim(quad(S1,S2,S3,S4,_,_,_,_), P) :-
            P is S1+S2+S3+S4.
    poly_perim(rect(W,H), P) :-
            P is W+H+W+H.
    poly_perim(sq(S), P) :- P is S+S+S+S.
    poly_perim(tri(S1,S2,S3,_,_,_), P) :-
            P is S1+S2+S3.
```