Disjunction for Prolog-X

1. The languages features covered by this design.

The point of this design is to extend the existing Prolog-X design so that a number of Prolog features supported by the Dec-10 interpreter (and in one case by a library routine) may be handled by the Prolog-X compiler and byte-code interpreter. These features are to retain their Dec-10 forms and as far as possible are to have exactly the same effects.

1.1. Disjunction.

If G1 and G2 are goals, G1;G2 is a goal.

To solve this goal, the interpreter attempts to solve G1. After all attempts to solve G1 have failed, the interpreter attempts to solve G2. The goal G1;G2;...;Gn has the same effect as calling a predicate with n clauses, one for each disjunct, *except* that a cut textually inside any Gi has the effect of cutting the the parent predicate as well as the disjunction. This effect of cuts means that defining ';'/2 as a normal predicate calling its arguments does not give ';'/2 the desired semantics.

For compatibility with Dec-10 Prolog, the parser should accept '|' as a lexical alternative to ';'. The rule is that inside list brackets '|' signifies the cdr of a list, inside application brackets p(...) '|' is forbidden, and elsewhere it is a synonym for ';'. Thus [H|T] means .(H,T), p(X|Y) is forbidden, (p|q) means (p;q), and {p|q} means {p;q}. This notation is particularly suited to grammar rules. Making the ZIP parser accept this notation requires a trivial change which I have already made to the C code.

1.2. If-Then-Else.

If P, T, and F are goals, (P -> T | F) and (P -> T) are goals. This includes the cases where P, T, and F are conjunctions. The effect of (P -> T) is exactly the same as that of (P -> T | fail), and the effect of (P -> T | F) is almost the effect defined by

```
        (P -> T | F) :- call(P), !, call(T).
        (P -> T | F) :- call(F).
```

except that cuts inside P, T, or F cut the parent clause just like disjunction.

It should be noted that (P -> T | F) is equivalent to ((P->T) | F). In a disjunction G1;G2;...;Gn some of the disjuncts might have arrows and some might not. An arrow is effectively a local cut, which just cuts the disjunction. There is no reason why a disjunct may not contain more than one arrow. If we introduce a local cut symbol ->, then A->B is equivalent to (A,->,B), and A->(B->C) is equivalent to (A,->,B,->C). However, A->(B->C) and (A->B)->C are *not* equivalent.

Good style in interpreted Prolog is to use if-then-else in preference to cuts. However, while the Dec-10 compiler does support disjunction, it does not support arrows, so most Edinburgh Prolog programs avoid arrows and use cuts so that they can be compiled. A typical example would be

```
        lookup(t(K,V,L,R), Key, Val) :-
                compare(C, Key, K),
                (   C = (<) -> lookup(L, Key, Val)
                |   C = (>) -> lookup(R, Key, Val)
                |   C = (=) -> Val = V
                ).
```

which is rendered into compilable Prolog as

```
        lookup(t(K,V,L,R), Key, Val) :-
                compare(C, Key, K),
                (   C = (<), !, lookup(L, Key, Val)
                |   C = (>), !, lookup(R, Key, Val)
                |   Val = V
                ).
```

which only has the same effect when compare/3 is determinate (which it is) or when we don't want multiple solutions. This mental translation is undesirable, and has been known to introduce bugs. The main point of this design is to let people write better programs; section 2 shows how a very simple design can achieve what the Dec-10 compiler does.

1.3. Negation by finite failure.

Negation is defined in all Edinburgh Prologs as

```
        \+ Goal :- call(Goal), !, fail.
        \+ Goal.
```

where PDP-11 and C Prolog use not/1, and Dec-10 and C Prolog use '\+'/1. This is almost the same as (Goal->fail;true), except that cuts in the Goal are *not* supposed to cut the parent procedure. This is not much of a problem for two reasons. One is that the only time I have ever seen a negated goal with a cut in it is when I wrote one to see what it would do. Negated goals are almost always simple procedure calls, though occasionally they are conjunctions or disjunctions. The other reason is that we can turn cuts in the negated goal into local cuts cutting this disjunction. Section 3 shows how this is done.

So a goal \+ (G1|G2|...|Gn) becomes (G1',->,fail|...|Gn',->,fail|true) where the Gi' are the Gi with cuts changed to suitable local cuts.

1.4. Iteration.

A form of iteration which is very useful for data base operations, and which neatly expresses bounded universal quantification, is

```
        forall(Generator, Test) :-
                \+ (Generator, \+ Test).
```

The logical reading of forall(G,P) is "there is no instance of G for which the corresponding instance of P is false", e.g. forall(member(X,L), integer(X)) expresses the condition that every element of the list L should be an integer. The procedural reading is "for each instance of G, do P", e.g. forall(p(X), (write(X),nl)). If we can compile negation in line, it follows that we can compile forall/2 in line.

2. A Simple Solution for Disjunction Only.

Ordinary choice points are handled by recording suitable information in the local frames. The novel thing about Prolog compared with other languages is that there may be local frames above the current one (the one which variable references access), and that control may re-enter them. The simple solution is to push a dummy local frame for each disjunction, but never to enter it.

In the following, I assume registers

L	points to the top of the local stack
CL	points to the current local frame
BL	points to the frame of the most recent choice point
BP	is the alternative address in the BL frame

The code that is generated for (G1|G2|...|Gn) is

```
        BP := label 1
        construct a new local frame on the stack
        BL := L
        L +:= framesize
           G1
           go to exit
label 1:# the frame has been popped but is intact #
        BP := label 2
        set the BP field of the frame to BP
        L +:= framesize
           G2
           go to exit
label 2:# as label 1 #
label n-1:
        BL := the BL field of the frame
           Gn
exit:
```

I assume that failure pops the stacks back to the latest choice point, and resumes executing code at the BP address, but does not change the BL register, so that it is up to the resumed code to say whether it is still a choice point or not.

This solution automatically works correctly with respect to cuts. A cut discards this dummy local frame like any other, and it is only the alternatives which care whether the local frame is still there or not.

We can already see one useful optimisation. If the disjunction is the last goal of a clause, we can compile each disjunct as if it were a clause body, so that the disjuncts end with Depart instructions or whatever. I believe that the Dec-10 compiler does not do this, but don't know for certain. If this is done, the second form of the lookup/3 example would be tail recursive.

However, this solution only handles disjunction and cuts. It does not handle local cuts, so it cannot handle if then else or negation, as the Dec-10 compiler cannot.

3. A Complete Solution.

The basic idea of pushing dummy frames is clearly right. There are two basic problems which we have to solve. The first concerns local cuts. The second concerns clause cuts.

3.1. Local cuts.

Suppose we have a goal (p, q -> r). By the time we get to the local cut, there may be any number of local frames above the dummy frame we are looking for. The CL register is pointing to the correct frame for this clause, but there may be any number of local frames above that and below the dummy.

If we have some way of distinguishing between ordinary frames and dummy frames, we may easily scan down the local stack until we come to a dummy frame whose CL field matches the CL register. This is just a matter of chaining down the BL fields of the local frames, starting from the BL register. This means that we must leave disjunction frames in the BL chain even when they are executing their last disjunct, and so are not really choice points at all. This is a small price to pay.

On ZIP we do have a way of distinguishing between ordinary frames and dummy frames, and that is the form of the BP field. The BP field of an ordinary frame is either a clause pointer or Termin. What we do is to introduce a new tag (there are plenty of spare values), say ALT, and make the BP field of a dummy frame be [ALT:d] or [ALT:0] signifying that the resumption point is at byte offset d in the clause the CL frame is running, or that there is no resumption point.

The code generated for a disjunction (G1|...|Gn) is then

```
L[0]:   Or d[1]
            G[1]
            Goto Exit
L[1]:   Alt d[2]
            G[2]
            Goto Exit
L[2]:   ...                         d[i] = L[i]-L[i-1]
L[n]:   Alt 0
            G[n]
Exit:   EndOr
```

where Or d pushes a dummy frame with BP field [ALT:d+PC], Alt d restores the dummy frame and sets its BP field to [ALT:d+PC], in both cases d=0 is taken as 0, not as PC, Goto does the obvious thing, and EndOr checks whether the top frame on the local stack is a dummy for this clause, and if it is and has BP = [ALT:0] pops it. One more instruction is needed to complete this picture, and that is Arrow, which does a local cut. The EndOr instruction need not appear after all the code for the disjunction. When the disjunction is the last thing in the clause, we can turn the final goal in each disjunct into a Depart preceded by an EndOr. Thus the translation of the lookup/3 example would end with

```
L0:     Or d1
            Var C
            Atom <
            Call =/2
            Arrow
            EndOr
            Var L
            Var Key
            Var Val
            Depart lookup/3
L1:     Alt d2
            Var C
            Atom >
            Call =/2
            Arrow
            EndOr
```

```
                Var R
                Var Key
                Var Val
                Depart lookup/3
L2:     Alt 0
                Var C
                Atom =
                Call =/2
                Arrow
                EndOr
                Var Val
                Var V
                Depart =/2
```

This solution handles disjunction, if then else, negation, and forall. Provided, that is, that there are no cuts inside them. It also handles disjunction with cuts provided there are no local cuts. We can detect at compile time that a particular disjunction contains negations and foralls (none of which contain cuts), cuts, and nested disjunctions of a similar sort. In that case we generate exactly the same code, but not the EndOr instructions. We rely on cuts to remove the dummy frames so that the depart instructions work. Thus we could code the other version of lookup/3 as

```
L0:     Or d1
                Var C
                Atom <
                Call =/2
                Cut
                Var L
                Var Key
                Var Val
                Depart lookup/3
L1:     Alt d2
                Var C
                Atom >
                Call =/2
                Cut
                Var R
                Var Key
                Var Val
                Depart lookup/3
L2:     Alt 0
                EndOr                   # get rid of frame #
                Var Val
                Var V
                Depart =/2
```

This is a very attractive scheme. It takes care of almost all the programs I have ever seen that use disjunction, because either they use if-then-else or they use cuts, but not both.

3.2. Cuts and Local cuts both.

There is no problem with cuts alone. The Dec-10 Prolog system is proof of that. There is no problem with if-then-else alone, as I hope the outline above shows. The problem is when you have something like

```
        (p, !, q -> r)
```

which is a rather silly thing to have. The problem is that the Arrow expects to find the dummy frame still there, and the Cut thinks it has a right to remove it. The compiler could just reject such mixtures as nonsense. I do not think anyone would be inconvenienced by that. But we can do better.

What we want when cuts mingle with disjunctions is for the cut to prune away all the real local frames, tidy up the trail, and remove the choice point for this clause just like it always does. We also want it to reset the BP fields of the dummy frames to [ALT:0]. Then the dummy frames have to be moved down over the reclaimed parts of the local stack. But there is no difficulty in that. We define a new instruction Slice which does the following steps.

```
t := NIL
p := BL
while p > CL do
    if Tag of BP of p is ALT and CL of p = CL then
        CL of p := t
        t := p
    fi
    p := BL of p
od
cut as usual
while p =/= NIL do
    copy p down to L
    t := CL of p
    CL of L := CL
    p := t
od
```

That is all it takes to make disjunction and if-then-else work. Cuts inside disjunctions are replaced by Slices, and all else is as in section 3.1.

3.3. Negation and Iteration with Cuts.

There is one more place that cuts cause problems. That is inside negations and iterations. We can handle

```
\+ (G1, !, G2)
\+ (G1 -> G2)
```

(where G1 and G2 are arbitrary conjunctions and may be empty) by turning the ! or -> into an Arrow. We can handle cuts and arrows at the top level of the arguments to forall/2 in the same way. What we cannot handle is cuts inside disjunctions inside negations. The cut would have to be turned into an Arrow to stop it escaping from the negation, but it can't be turned into an arrow because then it would only cut the disjunction and not the whole of the negated goal.

When presented with such a messy negation, one thing we could do is forget about compiling it, and just call '\+'/1 with the whole thing as argument. The attractive thing about this solution is that it is guaranteed to be correct. Indeed, the only argument in favour of compiling negations and iterations at all is efficiency; not and forall are defined as ordinary Prolog predicates in the Dec-10 system.

Or we could introduce yet another version of Cut which is told how many dummy frames to cut. There is nothing hard about that. All we have to do is take the Slice instruction presented in the previous section, and terminate the first loop at the kth dummy frame, and terminate the cut at that frame instead of CL. There are two reasons for doing this. The first is that it means we can compile ALL the control structures of a Prolog program with the exception of setof/3 and bagof/3. The next subsection shows how we can compile them too.

The other reason is that it permits unfolding where the Dec-10 does not.

What do I mean by this? Well, one of the tools we want in the Prolog toolkit if we are to use abstract data types efficiently is a way of expanding some procedure calls in line. Suppose we have a goal G which calls a sacred procedure that has already been defined. [Tapu would be a better word than sacred, by the way. Read-only might be even better.] If there is no clause of that procedure which matches G, we can replace G by 'fail'. If there is exactly one clause which matches G, we can replace G by the body of that clause. But this only works if there are no cuts in the body of that clause, or if the clause has the form

```
        Head :- Tests, !, Body.
```

where we evaluated all the tests at compile time. (In that case, e.g. matching a goal p(a) against a clause p(X) :- atom(X), !, Body; we use the part after the cut to replace the goal.) However, we can put any body at all inside a disjunction with one disjunct, and turn cuts into TimidCuts.

Suppose we are expanding a clause Head :- Tests, !, G1, !, ..., !, Gn. where the Gi do not have cuts except possibly inside disjunctions. We can expand this in-line, after evaluating the Tests, as

```
        (G1,->,G2,->, ... ->), Gn
```

If any Gi contains cuts inside disjunctions, it is those cuts which have to be turned into TimidCuts. If Gn contains such cuts, it has to be moved inside the disjunction as well.

This is very much a topic for the future, but we might as well adopt a solution which permits it now.

3.4. setof/3 and bagof/3

This section is partly baked, and is outside the proper scope of this design.

The basic action of setof(Template, Generator, Answer) is to generate a list of instantiations of the Template by backtracking through all ways of satisfying the Generator, to sort this list, and then to enumerate concordant subsets of the list. Concordant subsets are ones which bind the free variables of the generator to the same value.

Once we have a list of all the solutions, sorting it and backtracking through concordant subsets can be done in ordinary Prolog, as in fact it is done in Dec-10 and C Prolog. The only difference between bagof and setof is in this part, so a solution to the problem of making a list of all the answers is a solution to setof, bagof, and the inferior findall.

We can easily see how to backtrack through all the solutions. All we do is write (Generator,fail;true). Thus a solution to disjunction is an important part of compiling setof. The new problem is, how do we keep track of the answers? In Dec-10 and C Prolog, the answers are asserted into the data base, and when the last one has been found they are retracted out again. This involves two lots of copying. If the recorded data base in ZIP just copies and doesn't compile, we will be able to use that solution unchanged. We can do better though. What we want is to write a loop that looks like

```
        V := NIL
        (Generator, V := cons(straighten(Template),V), fail; true)
```

Clearly we can't write such a loop in Prolog. However, we can easily do so in our fictitious machine code. What we do is allocate a new local variable V, and bind V to NIL, and trail it. There are problems here.

4. The New Instructions.

4.1. Or a d

Disjunction is used for four different things. The decompiler cannot possibly tell which is which without assistance. The assistance is the a parameter of this instruction, which is an index into the XR table. That entry will be one of three atoms: ';' for disjunction and if-then-else, '\+' for negation, and 'forall' for iteration. The d parameter is either 0 or the offset of the second alternative from the beginning of the clause.

4.2. Alt d

The d parameter is either 0 or the offset of the second alternative from the beginning of the clause.

4.3. Goto d

The d parameter is the offset of the destination from the beginning of the clause. Whenever d is not 0 it is a forward branch. Although this instruction is meant for disjunction alone, it could be used for cross-jumping. E.g. in the lookup example, the first two cases share most of their code.

4.4. EndOr

4.5. Arrow

4.6. Slice

4.7. TimidCut n

The n parameter is the number of dummy frames to cut past.

5. Compiling into these Instructions.