```
simplify(X+Y, XX)    :- simplify(Y, 0), simplify(X,XX).
simplify(X-Y, XX)    :- simplify(Y, 0), simplify(X,XX).
simplify(X-Y, YY)    :- simplify(X, 0), simplify(-Y,YY).

simplify(-(-X), XX) :- simplify(X,XX).
simplify(-0, 0).
simplify(+X,XX)      :- simplify(X,XX).
```

last part:

No because '!' is not reversible. In any case it is could only integrate expressions that were identical to ones that could be the result of df. There are an infinite number of terms that could differentiate to 0. It may loop trying to find on.

The situation is even worse the the simplifying version of df.

\newpage

\leftline{\bf 2 Prolog for AI} 2002

p647
MR

A simple D-type flip-flop is represented by the Prolog preficate {\tt dff} whose definition is as follows:

\centerline{\tt dff(D, 0, Q, Q).}
\centerline{\tt dff(D, 1, Q, D).}

The first argument is the input to the flip-flop, the second is the clock with {\tt0} representing a falling edge and {\tt1} representing a rising edge. The third and fourth arguments are the previous and next states of the flip-flop. As can be seen the state of the flip-flop changes on a rising edge of the clock.

A clocked circuit consists of three d-type flip-flops with inputs and states (D_1,Q_1), (D_2,Q_2) and (D_3,Q_3). They are wired in such a way that

\begin{center}
\begin{tabular}{l}
$D1=(Q_1\wedge Q_2)\vee(\bar{Q_1}\wedge\bar{Q_2})$ \\
$D2=(\bar{Q_1}\wedge Q_3)\vee (Q_2\wedge\bar{Q_3})$ \\
$D3=(Q_1\wedge Q_3)\vee(\bar{Q_2}\wedge\bar{Q_3})$
\end{tabular}
\end{center}
Using {\tt s(Q1, Q2, Q3)} to represent the state of the circuit, define a predicate that will compute the state after the next rising edge of the clock. You may find it helpful to define predicates to represent {\em and}, {\em or} and {\em not} gates.\hfill[14~marks]

Define a predicate {\tt testcc(N, s(Q1,Q2,Q3), List)} that will compute the list of states ({\tt List}) through which the circuit passes from the given initial state {\tt s(Q1,Q2,Q3)} as a result of a sequence of {\tt N} rising edges of the clock.\hfill[6~marks]

```
\begin{verbatim}
ANSWER:

and(0,0,0).
and(0,1,0).
and(1,0,0).
and(1,1,1)
or(0,0,0).
or(0,1,1).
or(1,0,1).
or(1,1,1).
inv(0,1).
inv(1,0).

step(s(Q1,Q2,Q3), s(Z1,Z2,Z3)) :-
  inv(Q1, NQ1), inv(Q2, NQ2), inv(Q3, NQ3),
  and(Q1,   Q3, A),
  and(Q2,  NQ3, B),
  and(NQ1,  Q3, C),
  and(Q1,   Q2, D),
  and(NQ1, NQ2, E),
  or(A, B, D1), or(B, C, D2), or(D, E, D3),
  dff(D1, 1, Q1, Z1).
  dff(D2, 1, Q2, Z2),
  dff(D3, 1, Q3, Z3).

Last part:

testcc(0, S, [S]).
testcc(N, S, [S|SS]) :-
  step(S,NS),
  N1 is N-1,
  testcc(N1, NS, SS).
\newpage

\leftline{\bf 1 Comparative Programming languages}

This question concerns the representation of parse tree nodes for
expressions composed on integer constants, identifiers, and integer
operators for addition, subtraction, multiplication and division.  In
a typeless language, such as BCPL, each node can be implemented as a
vector whose first element holds an integer identifying the node
operator. The size of the vector and the kinds of value held in the
other elements then depends on this node operator.

Complete the description of how you would represent such integer
expressions in a typeless language.\hfill[5~marks]

Suggest how you would represent such integer expressions in C and
either ML or Java.  \hfill[10~marks]

Briefly discuss the relative merits of the your C data structure
compared with that used in the typeless approach.\hfill[5~marks]
```

P5 Q5
MR