順方向変換とは，プログラムpにxを入力したときに出力yが得られ，メモリの状態がSからMに変化するのであれば，プログラムpの順方向変換後のプログラムp'はxを入力すると出力yが得られ，メモリの常態をMからSに変化させるための情報を補助記憶装置に書き出すようにする変換である.

逆方向変換とは，プログラムpにxを入力したときに出力yが得られ，メモリの状態がSからMに変化するのであれば，プログラムpの順方向変換後のプログラムp'が実行済みであるときにプログラムpの逆方向変換後のプログラムrev_p'を実行すると，補助記憶装置の情報を使用してメモリの状態をMからSに変化させるようにする変換である.

ここでの逆関数は，メモリが初期化された状態をSとし，ある関数f()が実行後のメモリの状態をMとすると，関数f()に対する逆関数f'()が存在し，メモリの状態がMの場合にf'()を実行すると，実行後のメモリの状態がSになる関数である.

	元のコード	T[]	R[]
オリジナル	v = f();	SAVE(v); v = f();	RESTORE(v);
提案	v = f();	SAVE(v); v = f();	rev_f(); RESTORE(v);

・関数呼出しの元々の変換は，順方向変換では何もせず，逆方向変換は関数の名前を逆関数の名前に変更するとされている.
・しかし，関数呼出しはすべてf();かv = f();の形であるため，単純に名前を置き換えた場合では，v = f();の形の逆方向変換では，RESTORE(v);となってしまい，逆関数が呼び出されなくなってしまう
・逆関数の呼び出しが行われなければ，関数f()の本体で副作用がある演算を行っていた場合，変数の値を復元できなくなり可逆性が保たれなくなる
・したがって，逆方向変換では逆関数を呼び出し，その後にRESTORE();を行う必要がある

	元のコード	T[]	R[]

オリジナル	{
	　s1
	　goto label;
	　s2
	　goto label;
	　s3
	label:
	　s4
	}
		{
		　char c;
		　T[s1]
		　{c = 1; SAVE(c)}
		　goto label;
		　T[s2]
		　{c = 2; SAVE(c)}
		　goto label;
		　T[s3]
		　{c = 0; SAVE(c)}
		label:
		　T[s4]
		}
			{
			　char c;
			　R[s4]
			　RESTORE(c);
			　swicth(c){
			　　case 0: break;
			　　case 1: goto rlabel 1;
			　　case 2: goto rlabel 2;
			　}
			　R[s3]
			rlabel2:
			　R[s2]
			rlabel1:
			　R[s1]
			}

提案	{
	　s1
	　goto label;
	　s2
	　goto label;
	　s3
	label:
	　s4
	}
		{
		　unsigned char c;
		　T[s1]
		　{c = 1; SAVE(c)}
		　goto label;
		　T[s2]
		　{c = 2; SAVE(c)}
		　goto label;
		　T[s3]
		　{c = 0; SAVE(c)}
		label:
		　T[s4]
		}
		
		（cはfresh.）
		（同じ関数内の同じラベルに対するgoto文は255個以内）
			{
			　unsigned char c;
			　R[s4]
			　RESTORE(c);
			　swicth(c){
			　　case 0: break;
			　　case 1: goto rlabel 1;
			　　case 2: goto rlabel 2;
			　}
			　R[s3]
			rlabel2:
			　R[s2]
			rlabel1:
			　R[s1]
			}
			
			（cはfresh.）

・goto文の元々の変換は，1つ以上のgoto文が対応するlabelについて変換を行っている
・元々の変換は，順方向変換では追加の情報を使用して，どのgoto文が実行されたのかを保存している
・元々の変換は，逆方向変換では保存された情報を使用して，実行されたgoto文の位置に戻れるように変換が行われている
・追加の情報はchar型かshort int型で定義されるが，それぞれunsigned char型とunsigned int型にすることで空間計算量を増やさずに，より多くのgoto文の位置を記憶することができる．
・また，同じ関数内の同じラベルに対するgoto文が128個以上255個以内の場合でもunsigned short int型の変数を用意しなくてよいため，この範囲であれば元々の変換よりも空間計算量が少なくなる

	元のコード	T[]	R[]

オリジナル	int f(int arg){
	　int x, y;
	　s
	　return(x);
	}
		int f(int arg){
		　int x, y;
		　T[s]
		　return(x);
		　SAVE(arg);
		　SAVE(x);
		　SAVE(y);
		}
			int rev_f(int arg){
			　int x, y;
			　RESTORE(y);
			　RESTORE(x);
			　RESTORE(arg);
			　R[s]
			　return(x);
			}

提案	int f(int arg){
	　int x, y;
	　s
	}
		int f(int arg){
		　int x, y;
		　T[s]
		　SAVE(arg);
		　SAVE(x);
		　SAVE(y);
		　return(x);
		}
			int rev_f(int arg){
			　int x, y;
			　RESTORE(y);
			　RESTORE(x);
			　RESTORE(arg);
			　R[s]
			　return(x);
			}

・元々のブロック構文の変換は，ブロック構文内で定義されるローカル変数をブロック構文の最後で補助記憶装置に保存するように変換される
・元々の逆方向変換では，ブロック構文でローカル変数が定義された直後に補助記憶装置から変数の値の読み出しを行う
・しかし，ブロック構文が関数の本体だった場合
・元々の変換では引数をローカル変数として扱い，ブロック構文と同じ変換を行うとされており，厳密に定義はされていない
・この場合，returnが存在しなければ問題はないが，returnが存在する場合，ブロック構文の末尾はreturnの後となる解釈も存在してしまう
・そのため，順方向変換ではreturnの直前か，ブロック構文の末尾でSAVE();を呼び出すように変換を行う
・逆方向変換では変数宣言の直後かブロック構文の頭でRESTORE();を呼び出すように変換を行う
・また，複数のローカル変数を保存する場合，逆方向変換では保存した変数を逆の順序で読み出すように変換を行う