Compilers Assignment

Task 1

The first task was to add goto statements to the ppc4 compiler. The first part of this report outlines the changes that were made to the compiler and the new test cases that were written.

Abstract Syntax: A representation for the new label declarations, label statements and goto statements was added to the abstract syntax tree. In tree.mli:

```
label = LabelDecl of int list
```

```
stmt_guts = …
  | GotoStmt of int
  | LabelStmt of int * stmt
```

The definition of block was also updated to include label declarations before a list of other declarations.

```
block = Block of label * decl list * stmt * int ref * int ref
```

The same changes were made to tree.ml. In addition, the make_block function was updated to include label declarations.

```
make_block (labels, decls, stmts) = Block (labels,decls,stmts,ref 0,ref 0)
```

Concrete Syntax: The lexer and parser were extended to implement the new construct. In lexer.mll, tokens for the keywords goto and label were defined.

```
let symtable =
  Util.make_hash 100
    [ …; ("goto", GOTO); ("label", LABEL) ]
```

In parser.mly they were added as new token types.

```
%token                  GOTO LABEL
```

Furthermore, the context-free grammar for the language was updated to accommodate label declarations, goto statements and label statements.
For label declarations (here we also check the value of label is between 1 and 9999):

```
block :
    labels decl_list BEGIN stmts END    { make_block ($1, $2, $4) } ;
labels:
    /* empty */                         { LabelDecl([])}
  | LABEL COLON label_list SEMI         { LabelDecl($3) };

label_list:
    NUMBER                              { if ($1 >= 1 && $1 < 10000) then
[$1] else failwith "label out of range" }
  | NUMBER COMMA label_list             { if ($1 >= 1 && $1 < 10000) then
$1 :: $3 else failwith "label out of range" };
```

For goto and label statements:

```
stmt1 :
    …
  | NUMBER COLON stmt                    { LabelStmt ($1, $3) }
  | GOTO NUMBER                          { GotoStmt ($2) };
```

<u>Semantic Checks and Code Generation:</u> Both semantic checks and code generation are performed in kgen.ml. Semantically, we check that [1] :

1. A label is declared and used at most once in a scope and that all used labels have a valid declaration in their scope.
2. When a goto statement G is used with label statement S one of the following is satisfied:
    1. S contains G.
    2. S is a statement stmt in a sequence of statements stmt_list containing G.
    3. Let B be a block with the syntax "decl_list begin stmts end". B contains G (either in decl_list or stmts) and S must be a statement stmt contained in the sequence of statements stmt_list in the stmts part of B.

To perform these checks and generate Keiko code we keep a global hash table where labels are associated with pairs storing the level of the label's statement and its id which is used for jump instructions in Keiko code.

```
let table : (int, (int*codelab)) Hashtbl.t = Hashtbl.create 16
```

Whenever we are generating the code for a procedure, all the newly declared labels are added to the table assigning each label an id and setting its level to -1. Subsequently, code for the body of the procedure is generated. During this code generation, the level of each label that is used in a statement is updated. Afterwards, code for child procedures is also generated. Finally, the newly declared labels within this procedure are removed from the table.

Code generation for a procedure:

```
let rec gen_proc =
  function
      ProcDecl (Heading (x, _, _), Block (new_labels, locals, body, fsize,
nregv)) ->
        let d = get_def x in
        let p = get_proc d.d_type in
        begin
          match d.d_addr with
              Global lab ->
                add_labels table new_labels;
                do_proc lab d.d_level p.p_result !fsize body;
                gen_procs locals;
                remove_labels table new_labels
            | _ -> failwith "gen_proc"
        end
    | _ -> ()
```

While adding labels to the table, if they are already declared we throw an error as the label has been declared twice in this scope. Otherwise, the table is updated by adding this new label:

```
let add_label (table : (int, (int*codelab)) Hashtbl.t) (key : int) =
  let l = label () in if Hashtbl.mem table key then failwith "label
declared twice"
                      else Hashtbl.add table key (-1, l)

let add_labels table labels =
  match labels with
      LabelDecl ls -> List.iter (add_label table) ls
```

The level of each label is updated whenever gen_stmt is called on a sequence of statement (constructed with Seq) containing the label statement. If a label is found to have a level not equal to -1, it must be used twice - so we throw an error. Otherwise, the level of the label is updated to be the current level.

```
let update_table_label table s =
  match s.s_guts with
      LabelStmt (n,ss) -> if Hashtbl.mem table n then
                            let (lev,lab) = Hashtbl.find table n in
                              if lev<>(-1) then failwith "label used twice"
                              else Hashtbl.replace table n (!level, lab)
                          else failwith ("label undeclared: " ^
string_of_int(n))
    | _ -> ()

let rec gen_stmt s =
  let code =
    match s.s_guts with
        Skip -> NOP
      | Seq ss -> update_table table ss; SEQ (List.map gen_stmt ss )
```

Whenever we come across a label statement in gen_stmt, we look it up on the hash table to check it has been declared.

```
 …    | LabelStmt (n,ss) ->
          if Hashtbl.mem table n then
          let (level,lab) = Hashtbl.find table n in
          SEQ [LABEL lab; CONST 0; GLOBAL "__label"; STORE 4; gen_stmt ss]
          else failwith "label undeclared"
```

Similarly, for goto statements we look the label up on the table to make sure it has been declared.
Furthermore, we check the label's level to ensure that the nesting rules described in part 2 of the checks described on page 2 are met. In all three cases, code would have been generated for a sequence of statements containing the corresponding label statement which the goto is jumping to. Hence, the level would be set to its correct value and not -1.

By rule 2.3 on page 2, goto statements are allowed to jump to labels in their parent procedure. However, in Keiko, JUMP instructions are only permitted to jump to labels in the current procedure. Therefore, we facilitate such jumps through storing the label we are jumping to in a global variable named __label. This prevents a name clash as variable names cannot start with the underscore symbol.

```
      | GotoStmt (n) ->
          if Hashtbl.mem table n then
            let (lev, lab) = Hashtbl.find table n in
            if lev<>(-1) then
              if lev<>(!level) then SEQ [ CONST n; GLOBAL "__label"; STORE
4; RETURN]
              else JUMP lab
            else failwith ("label nesting invalid: " ^ string_of_int(n))
          else failwith "label undeclared"
```

Whenever we need to jump to a label in a parent procedure, we continually return from the inner procedures checking if the label is in this procedure and returning to its caller if not. This is analogous to following the static chain. And whenever we reach the correct level, we jump to the label. If we are not in the process of performing such a jump __label is set to zero and we just ignore the conditional logic to perform the checks described above. The variable __label is also reset to have the value zero whenever we reach a label statement as shown in the code for LabelStmt on page 3.

```
and jump_after_proc key value acc =
  let (lev,lab) = value and
      lab_ = label () in
    if lev<>(!level) then
      (SEQ [GLOBAL "__label"; LOAD 4; CONST key; JUMPC (Neq, lab_); RETURN;
LABEL lab_]) :: acc
    else
      (SEQ [GLOBAL "__label"; LOAD 4; CONST key; JUMPC (Eq,lab)] ) :: acc

and gen_call x args =
  …
        in let lab1 = label () in
        let casejump = SEQ (Hashtbl.fold jump_after_proc table []) in
        SEQ [code; GLOBAL "__label"; LOAD 4; CONST 0; JUMPC (Eq, lab1);
casejump; LABEL lab1]
```

When generating code for the whole program, a definition for the global variable __label is added at the bottom. Additionally, now the code for the main function is generated before the globally defined procedures to ensure we update the level of any label statements in the main function, allowing jumps from child procedures. Also, the lev parameter passed into do_proc for MAIN has been changed from 0 to -1 so that any labels in MAIN are labelled to have level 0, and any labels in global procedures have level 1. Previously for variables they would both have level 1, and this was unimportant as the levels of global variables were never used.

```
let translate (Prog (Block (labels, globals, main, _, _), glodefs)) =
  add_labels table labels;
  do_proc "MAIN" (-1) voidtype 0 main;
  gen_procs globals;
  List.iter gen_global !glodefs;
  printf "GLOVAR __label 4\n" [];
…
```

<u>Tests:</u> All existing test cases pass. In addition, test cases for goto statements are provided:

<u>These tests produce errors:</u>

- Label value is outside of the range 1 to 9999 (labelrange.p)
- Label value is not an unsigned integer (labelnotint.p)
- Label is undeclared (labelundeclared.p)
- Label is declared twice in a scope (labeldeclaredtwice.p)
- Label is used twice in a scope (labelusedtwice.p, labelusedtwice2.p)
- Corresponding label is not within scope of the label declaration (labelscope.p)

<u>These tests produce valid Keiko code:</u>

- Label whose statement is empty (labelempty.p)
- Label with same name declared twice but in different scopes (labelsamename.p)
- A program where the label statement contains the goto statement (condition 2.1 on page 2) (case1.p)
- A program where the label statement is in the same list of statements as the goto (condition 2.2 on page 2) (case2v1.p, case2v2.p)
- A program where the goto is to a label in a parent procedure (condition 2.3 on page 2) (case3.p)
- All examples provided in the assignment document (example1.p, example2.p and example3.p). Example 2 has been slightly changed to print a result.
- Program with infinite loop (infinite.p)
- Program with deep nesting of procedures and goto instructions between these procedures to produce a ladder pattern. (ladder.p)

In many of these cases the programs have been constructed to produce the wrong output if a slightly different behaviour than expected occurs. These test cases and the corresponding code generated can be found at the end of this report (pages 12 to 31).

<u>Task 2</u>

The second task was to add loop-invariant code motion to the ppc4 compiler. Loop-invariant code motion for while loops has been added to the compiler (with changes made to kgen.ml), and an outline of the changes that would be necessary for repeat statements and for loops are provided here. The second part of this report outlines the changes that were made to the compiler and the new test cases that were written.

<u>Constant Assignment Hoisting:</u> Firstly, the while loop is converted to a repeat statement hosted inside an if statement to make it easier for loop-invariant code to be moved outside of the loop. This way the loop-invariant code is only executed if we would have entered the loop, as required.

```
while (test) do

      …

end;
```

```
if (test) then

(*loop-invariant code moved here*)

      repeat

            …

      until (test);

end;
```

A statement is loop-invariant if all reaching definitions used by the statement occur outside of the loop [2]. A definition reaches a statement if there is a path from the definition to the statement, along which the defined variable is never redefined [3]. This requirement is equivalent to requiring there being no definitions of the variable inside the loop.

The conditions for being able to hoist a loop-invariant definition are [2]:

1. Loop invariant node must dominate all exists – guaranteed since the body of a repeat loop is always executed at least once.
2. There must be just one definition of this variable in the loop – we count the definitions of each variable in the loop using the defs hash table.
3. The variable being defined must not be liveOut from the loop's preheader – this is equivalent to the variable not being used in any statements before its definition inside the loop. This is the case assuming goto statements are not part of the language. Although the changes made in task 1 and task 2 are made to the same file, the intention is that they are independent tasks to be tested independently so we do not worry about goto statements here.

We store the number of times a variable defined inside the loop in a hash table, defs. Additionally, a variable opt is kept which indicates whether we will perform loop-invariant code motion which is initially set to the value of the optflag. To be later used are also hash tables preheader, storing all variable definitions moved to the preheader, and if_tests storing all the test conditions of if statements moved to the preheader. Finally, the list used stores all the variables that have been in statements so far and updated is later used when we iterate code motion until no changes have been made.

```
and constant_while test body =
  let opt = ref (!optflag) in
  let defs = Hashtbl.create 16 in
  let preheader = Hashtbl.create 16 in
  let if_tests = Hashtbl.create 16 in
  let used = ref [] and updated = ref true in
  defs_update defs body;
```

The function defs_update recursively updates the hash table.

```
and defs_update defs s =
  match s.s_guts with
      Assign (v,e) -> (match v.e_guts with
                         Variable x -> let d = get_def x in
                           (if not (Hashtbl.mem defs d.d_tag) then
Hashtbl.add defs d.d_tag 1 else Hashtbl.replace defs d.d_tag (Hashtbl.find
defs d.d_tag + 1))
                       | _ -> ())
    | Seq ss -> List.iter (defs_update defs) ss
    | WhileStmt (test,body) -> defs_update defs body
    | IfStmt (test, thenpt, elsept) -> defs_update defs thenpt; defs_update
defs elsept
    | RepeatStmt (body, test) -> defs_update defs body
    | ForStmt (var, lo, hi, body) -> defs_update defs body
    | CaseStmt (sel, arms, deflt) -> List.iter (fun (ex,st) -> defs_update
defs st) arms; defs_update defs deflt
    | LabelStmt (n,ss) -> defs_update defs ss
    | _ -> ()
```

Next, we perform a liveness analysis – going sequentially through the body of the loop, updating which variables have been used so far to identify candidate assignments that might be hoisted out of the loop. For an assignment to be hoisted, it must not have been used in any statements before its definition in the loop, there must be one definition of it inside the loop, and there must be no definitions of variables that it uses inside the loop (or the only one has been moved to the preheader). Code is continually hoisted outside of the loop until no more updates are made.

```
and constant_while test body =
  …
  while !updated do
     updated := false;
     used := [];
     live_analysis used updated defs preheader if_tests true opt body;
  done;
  …
```

For liveness analysis we use the function below to get all the variables in an expression.

```
and vars_in_expr e opt =
  match e.e_guts with
      Variable x -> let d = get_def x in [d.d_tag]
    | Sub (a,i) -> union (vars_in_expr a opt) (vars_in_expr i opt)
    | Select (e,x) -> let d = get_def x in union [d.d_tag] (vars_in_expr e
opt)
    | Deref e -> vars_in_expr e opt
    | FuncCall (_,_) -> opt := false; []
    | Monop (op,e) -> vars_in_expr e opt
    | Binop (op,e1,e2) -> union (vars_in_expr e1 opt) (vars_in_expr e2 opt)
    | _ -> []
```

The function union is used to get the union of two lists in the above function.

```
and union lst1 lst2 =
  let rec add_unique acc lst =
    match lst with
    | [] -> acc
    | x :: xs ->
        if List.mem x acc then
          add_unique acc xs
        else
          add_unique (x :: acc) xs
  in
  add_unique lst2 lst1
```

The liveness analysis function adds variables whose definitions have been hoisted to the hash table preheader. We do not perform any code motion if a procedure call is found in the loop body since the procedure might have side effects which might affect the code motion (implemented in the penultimate line of the function below).

```
and live_analysis used updated defs preheader if_tests level opt s =
  match s.s_guts with
      Assign (v,e) -> let used_in_assign = vars_in_expr e opt in
                      used := union !used used_in_assign;
                      (match v.e_guts with
                           Variable x -> let d = get_def x in
                                         let invariant_func tag accum = (((not (Hashtbl.mem defs
tag))||((Hashtbl.find_opt defs tag = Some 1)&&(Hashtbl.mem preheader tag))) && accum) in
                                         let inv = List.fold_right invariant_func used_in_assign
true in
                                         if (inv && ((Hashtbl.find defs d.d_tag) = 1) && (not
(List.mem d.d_tag !used)) && (not (Hashtbl.mem preheader d.d_tag)) && level) then
                                           (updated := true; Hashtbl.add preheader d.d_tag 0)
                         | _ -> ())
    | Seq ss -> List.iter (live_analysis used updated defs preheader if_tests level opt) ss
    | WhileStmt (test,body) -> used := union !used (vars_in_expr test opt); live_analysis used
updated defs preheader if_tests false opt body
    | IfStmt (test, thenpt, elsept) -> let used_in_test = vars_in_expr test opt in
                                       let invariant_func tag accum = (((not (Hashtbl.mem defs
tag))||((Hashtbl.find_opt defs tag = Some 1)&&(Hashtbl.mem preheader tag))) && accum) in
                                       let inv = List.fold_right invariant_func used_in_test true in
                                       if (inv && level && (not (Hashtbl.mem if_tests test))) then
                                         (temps:=!temps+1; Hashtbl.add temps_size !temps (size_of
test.e_type); updated:= true; Hashtbl.add if_tests test (!temps));
                                       used := union !used used_in_test; live_analysis used updated
defs preheader if_tests false opt thenpt; live_analysis used updated defs preheader if_tests false
opt elsept
    | RepeatStmt (body, test) -> used := union !used (vars_in_expr test opt); live_analysis used
updated defs preheader if_tests false opt body
    | ForStmt (var, lo, hi, body) -> let used_in_for = union (union (vars_in_expr var opt)
(vars_in_expr lo opt)) (vars_in_expr hi opt) in
                                     used := union !used used_in_for;
                                     live_analysis used updated defs preheader if_tests false opt
body
    | CaseStmt (sel, arms, deflt) -> let used_in_case = List.fold_right (fun (ex,st) accum -> union
accum (vars_in_expr ex opt)) arms [] in
                                     used := union !used used_in_case;
                                     List.iter (fun (ex,st) -> live_analysis used updated defs
preheader if_tests false opt st) arms; live_analysis used updated defs preheader if_tests false opt
deflt
    | LabelStmt (n,ss) -> live_analysis used updated defs preheader if_tests level opt ss
    | ProcCall (p,args) -> opt := false
    | _ -> ()
```

Finally, code generation is performed after code motion has finished. The Keiko code implements the conversion of the while loop into a repeat statement hosted inside an if statement. The code below also includes details from the more general loop-invariant code motion involving if statements which will be described later.

```
and constant_while test body =
  …
  if !opt then
  (match body.s_guts with
      Seq ss ->
        let new_before, new_body0 = List.partition (partition1 preheader)
ss in
        let temp_defs = List.fold_right (temp_func if_tests) ss [] in
        let new_body = List.map (update_ifs if_tests) new_body0 in
        let const_stmt = make_stmt(Seq new_before, 0) and new_body_stmt =
make_stmt(Seq new_body,0) in
        let lab1 = label () and lab2 = label() and lab3 = label () in
        SEQ [gen_cond lab2 lab3 test; LABEL lab2; gen_stmt const_stmt; SEQ
temp_defs; LABEL lab1; gen_stmt new_body_stmt;
             gen_cond lab1 lab3 test; LABEL lab3]
    | _ -> let lab1 = label () and lab2 = label () and lab3 = label () in
      SEQ [JUMP lab2; LABEL lab1; gen_stmt body;
           LABEL lab2; gen_cond lab1 lab3 test; LABEL lab3])
  else
    let lab1 = label () and lab2 = label () and lab3 = label () in
      SEQ [JUMP lab2; LABEL lab1; gen_stmt body;
           LABEL lab2; gen_cond lab1 lab3 test; LABEL lab3]
```

We partition statements to be moved to the preheader using the function below.

```
and partition1 preheader s =
   match s.s_guts with
       Assign (v,e) -> (match v.e_guts with
                            Variable x -> let d = get_def x in Hashtbl.mem
preheader d.d_tag
                          | _ -> false)
     | _ -> false
```

A very similar method could be adopted to implement loop-invariant code motion for repeat statements and for loops. The process for repeat statements has already been implemented as this is essentially what we do for while loops, but additionally we host this code in an if statement. For loops could be converted to while loops as below.

```
for var := a to b do

      …

end;
```

```
var:= a

while (var <= b) do

      …

      var := var + 1

end;
```

Generalised Loop-Invariant Code Motion: The above code motion is limited to constant assignments. However, it can be generalized, for instance, by evaluating constant conditions in if statements outside the loop, storing the results in variables, and then using these variables directly within the loop.

In the liveness_analysis function on page 8, whenever we encounter an if statement we check if its test is an expression which could be hoisted – i.e. if all reaching definitions used in the expression are from outside the loop. If this is the case, we create a new temporary variable to store this value and assign it to the expression before the loop. We keep count of the number of temporary variables used and each variable's storage size in a global variable and hash table.

```
let temps = ref 0
let temps_size = Hashtbl.create 16
```

We keep track of which if statement uses which temporary variable through a hash table, if_tests, which associates test expressions with the count of their temporary variable. The temporary variables are named __t(i) (where i is the count of the variable) in Keiko to avoid name clashes with any existing variables in the program. The function temp_func is used to generate Keiko code for all the assignments before the loop, and update_ifs updates the tests in if statements to access the correct temporary variable.

```
and constant_while test body =
  …
  let if_tests = Hashtbl.create 16 in
  …
        let temp_defs = List.fold_right (temp_func if_tests) ss [] in
        let new_body = List.map (update_ifs if_tests) new_body0 in
  …
and temp_func if_tests s accum =
  match s.s_guts with
      IfStmt (test, thenpt, elsept) ->
        if Hashtbl.mem if_tests test then
          let t = Hashtbl.find if_tests test in
          SEQ [gen_expr test; GLOBAL ("__t"^string_of_int(t));STORE
(size_of test.e_type)] :: accum
        else
         accum
    | _ -> accum
and update_ifs if_tests s =
  match s.s_guts with
      IfStmt (test, thenpt, elsept) ->
        if Hashtbl.mem if_tests test then
          let t = Hashtbl.find if_tests test in
          let id = intern ("__t"^string_of_int(t)) in
          let name = make_name(id,0) in
          name.x_def <- Some
{d_tag=id;d_kind=VarDef;d_type=test.e_type;d_level=(!level);d_addr=Global
("__t"^string_of_int(t))};
```

```
          let expr = make_expr(Variable name) in
          expr.e_type <- test.e_type;
          make_stmt(IfStmt(expr,thenpt,elsept),s.s_line)
        else s
    | _ -> s
```

Finally, the definitions of all the temporary variables are added to the end of the Keiko program.

```
let translate (Prog (Block (labels, globals, main, _, _), glodefs)) =
  …
  for i = 1 to !temps do
    printf "GLOVAR __t$ $\n" [fNum i; fNum (Hashtbl.find temps_size i) ];
  done;
  …
```

Tests: All existing test cases pass. In addition, test cases for loop-invariant code motion (constant assignment hoisting and if statements) are provided:

- Assignment in loop to integer constant (example4.p)
- Nested loops (nested.p) – variable that could be hoisted without inner loop is used by inner loop. The inner loop contains another variable whose definition can be moved outside of the loop.
- Assignment in loop to expression containing constant variables (imp1.p)
- Variable defined multiple times in loop (imp2.p)
- Variable used before assignment in loop (liveOut from the loop's preheader) (imp3.p)
- If statement where the test expression contains variable hoisted to the loop's preheader and constants (if1.p)
- If statement contains variable, whose definition only contains variables which are hoisted, and constants. This variable is also used inside the if statement (if2.p)
- If-else statement's test uses a variable that is updated in the loop, but the body of the if statements use constants which can be hoisted (if3.p)
- If statement where the value of the variable used in the test changes in the body of the if statement (if4.p)

The comments with picoPascal code in the generated Keiko code can be ignored as they refer to the source code rather than the equivalent picoPascal after code motion is performed. These test cases and the corresponding code generated can be found below (pages 36 to 48).

Bibliography

[1] “Compilers Assignment Oxford,” 2024.

[2] P. Kelly, “Loop-Invariant Code Motion,” [Online]. Available: https://www.doc.ic.ac.uk/~phjk/Compilers/Lectures/pdfs/Ch7-part3-LoopInvariantCodeMotionV02.pdf.

[3] “Reaching Definitions and SSA,” 2023. [Online]. Available: https://www.cs.cornell.edu/courses/cs4120/2023sp/notes.html?id=reachdef.

<u>The diff file:</u>

```
diff --git a/ppc4/check.ml b/ppc4/check.ml
index 1989477..0c0be56 100644
--- a/ppc4/check.ml
+++ b/ppc4/check.ml
@@ -346,6 +346,8 @@ let rec check_stmt s env =
         let vs = List.map check_arm arms in
         check_dupcases vs;
         check_stmt deflt env
+    | LabelStmt (l, s') -> check_stmt s' env
+    | GotoStmt (l) -> ()


 (* TYPES AND DECLARATIONS *)
@@ -500,7 +502,7 @@ and check_decls ds alloc env =
   Util.accum (fun d -> check_decl d alloc) ds env

 (* |check_block| -- check a local block *)
-let rec check_block rt env (Block (ds, ss, fsize, nregv)) =
+let rec check_block rt env (Block (ls, ds, ss, fsize, nregv)) =
   let env' =
     check_decls ds (local_alloc fsize) (new_block env) in
   check_bodies env' ds;
@@ -564,7 +566,7 @@ let init_env =
       ("bitnot", operator BitNot [integer], integer)] empty

 (* |annotate| -- annotate the whole program *)
-let annotate (Prog (Block (globals, ss, _, _), glodefs)) =
+let annotate (Prog (Block (labels, globals, ss, _, _), glodefs)) =
   level := 0;
   let env = check_decls globals global_alloc (new_block init_env) in
   check_bodies env globals;
```

<u>GOTO TESTS</u>

```
diff --git a/ppc4/gototests/case1.p b/ppc4/gototests/case1.p
new file mode 100644
index 0000000..c7fe525
--- /dev/null
+++ b/ppc4/gototests/case1.p
@@ -0,0 +1,79 @@
+label: 1,2;
+var x, y: integer;
+begin
```

```
+  x := 0;
+  y := 2;
+  1: if x = y then
+    goto 2
+  else
+    x:= x+1;
+    goto 1
+  end;
+  2: print_num(x); newline()
+end.
+
+(*<<
+ 2
+>>*)
+
+(*[[
+MODULE Main 0 0
+IMPORT Lib 0
+ENDHDR
+
+FUNC MAIN 0
+! x := 0;
+CONST 0
+GLOBAL _x
+STOREW
+! y := 2;
+CONST 2
+GLOBAL _y
+STOREW
+! 1: if x = y then
+LABEL L1
+CONST 0
+GLOBAL __label
+STOREW
+GLOBAL _x
+LOADW
+GLOBAL _y
+LOADW
+JEQ L3
+JUMP L4
+LABEL L3
+! goto 2
+JUMP L2
+JUMP L5
+LABEL L4
+! x:= x+1;
+GLOBAL _x
+LOADW
+CONST 1
+PLUS
+GLOBAL _x
+STOREW
+! goto 1
+JUMP L1
+LABEL L5
+! 2: print_num(x); newline()
+LABEL L2
+CONST 0
+GLOBAL __label
+STOREW
+GLOBAL _x
+LOADW
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
```

```
+CONST 0
+GLOBAL lib.newline
+PCALL 0
+RETURN
+END
+
+GLOVAR _x 4
+GLOVAR _y 4
+GLOVAR __label 4
+! End
+]]*)
diff --git a/ppc4/gototests/case2v1.p b/ppc4/gototests/case2v1.p
new file mode 100644
index 0000000..69c4cc8
--- /dev/null
+++ b/ppc4/gototests/case2v1.p
@@ -0,0 +1,68 @@
+label: 1,2;
+var x: integer;
+begin
+  x := 0;
+  if x = 0 then
+    goto 2
+  end;
+  1: x := x + 1;
+  2: print_num(x); newline()
+end.
+
+(*<<
+ 0
+>>*)
+
+(*[[
+MODULE Main 0 0
+IMPORT Lib 0
+ENDHDR
+
+FUNC MAIN 0
+! x := 0;
+CONST 0
+GLOBAL _x
+STOREW
+! if x = 0 then
+GLOBAL _x
+LOADW
+CONST 0
+JEQ L3
+JUMP L4
+LABEL L3
+! goto 2
+JUMP L2
+JUMP L5
+LABEL L4
+LABEL L5
+! 1: x := x + 1;
+LABEL L1
+CONST 0
+GLOBAL __label
+STOREW
+GLOBAL _x
+LOADW
+CONST 1
+PLUS
+GLOBAL _x
+STOREW
```

```
+! 2: print_num(x); newline()
+LABEL L2
+CONST 0
+GLOBAL __label
+STOREW
+GLOBAL _x
+LOADW
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
+CONST 0
+GLOBAL lib.newline
+PCALL 0
+RETURN
+END
+
+GLOVAR _x 4
+GLOVAR __label 4
+! End
+]]*)
diff --git a/ppc4/gototests/case2v2.p b/ppc4/gototests/case2v2.p
new file mode 100644
index 0000000..c261f97
--- /dev/null
+++ b/ppc4/gototests/case2v2.p
@@ -0,0 +1,68 @@
+label: 1,2;
+var x: integer;
+begin
+  x := 1;
+  if x = 0 then
+    goto 2
+  end;
+  1: x := x + 1;
+  2: print_num(x); newline()
+end.
+
+(*<<
+ 2
+>>*)
+
+(*[[
+MODULE Main 0 0
+IMPORT Lib 0
+ENDHDR
+
+FUNC MAIN 0
+! x := 1;
+CONST 1
+GLOBAL _x
+STOREW
+! if x = 0 then
+GLOBAL _x
+LOADW
+CONST 0
+JEQ L3
+JUMP L4
+LABEL L3
+! goto 2
+JUMP L2
+JUMP L5
+LABEL L4
+LABEL L5
+! 1: x := x + 1;
+LABEL L1
```

```
+CONST 0
+GLOBAL __label
+STOREW
+GLOBAL _x
+LOADW
+CONST 1
+PLUS
+GLOBAL _x
+STOREW
+! 2: print_num(x); newline()
+LABEL L2
+CONST 0
+GLOBAL __label
+STOREW
+GLOBAL _x
+LOADW
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
+CONST 0
+GLOBAL lib.newline
+PCALL 0
+RETURN
+END
+
+GLOVAR _x 4
+GLOVAR __label 4
+! End
+]]*)
diff --git a/ppc4/gototests/case3.p b/ppc4/gototests/case3.p
new file mode 100644
index 0000000..5c93d32
--- /dev/null
+++ b/ppc4/gototests/case3.p
@@ -0,0 +1,148 @@
+label: 1, 2;
+var i: integer;
+proc foo(x : integer): integer;
+  proc bar(y : integer): integer;
+  begin
+    goto 1;
+    print_num(2); newline()
+  end;
+begin
+  print_num(3); newline();
+  return bar(x);
+  print_num(4); newline()
+end;
+begin
+  i := 1;
+  i := foo(i);
+  goto 2;
+  1: i:= 0;
+  2: print_num(i); newline()
+end.
+
+(*<<
+ 3
+ 0
+>>*)
+
+(*[[
+MODULE Main 0 0
+IMPORT Lib 0
+ENDHDR
```

```
+
+FUNC MAIN 0
+! i := 1;
+CONST 1
+GLOBAL _i
+STOREW
+! i := foo(i);
+GLOBAL _i
+LOADW
+CONST 0
+GLOBAL _foo
+PCALLW 1
+GLOBAL __label
+LOADW
+CONST 0
+JEQ L3
+GLOBAL __label
+LOADW
+CONST 1
+JEQ L1
+GLOBAL __label
+LOADW
+CONST 2
+JEQ L2
+LABEL L3
+GLOBAL _i
+STOREW
+! goto 2;
+JUMP L2
+! 1: i:= 0;
+LABEL L1
+CONST 0
+GLOBAL __label
+STOREW
+CONST 0
+GLOBAL _i
+STOREW
+! 2: print_num(i); newline()
+LABEL L2
+CONST 0
+GLOBAL __label
+STOREW
+GLOBAL _i
+LOADW
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
+CONST 0
+GLOBAL lib.newline
+PCALL 0
+RETURN
+END
+
+FUNC _foo 0
+! print_num(3); newline();
+CONST 3
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
+CONST 0
+GLOBAL lib.newline
+PCALL 0
+! return bar(x);
+LOCAL 16
+LOADW
```

```
+LOCAL 0
+GLOBAL _foo.bar
+PCALLW 1
+GLOBAL __label
+LOADW
+CONST 0
+JEQ L6
+GLOBAL __label
+LOADW
+CONST 1
+JNEQ L8
+RETURN
+LABEL L8
+GLOBAL __label
+LOADW
+CONST 2
+JNEQ L7
+RETURN
+LABEL L7
+LABEL L6
+RETURN
+! print_num(4); newline()
+CONST 4
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
+CONST 0
+GLOBAL lib.newline
+PCALL 0
+ERROR E_RETURN 0
+END
+
+FUNC _foo.bar 0
+! goto 1;
+CONST 1
+GLOBAL __label
+STOREW
+RETURN
+! print_num(2); newline()
+CONST 2
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
+CONST 0
+GLOBAL lib.newline
+PCALL 0
+ERROR E_RETURN 0
+END
+
+GLOVAR _i 4
+GLOVAR __label 4
+! End
+]]*)
diff --git a/ppc4/gototests/example1.p b/ppc4/gototests/example1.p
new file mode 100644
index 0000000..8efeec9
--- /dev/null
+++ b/ppc4/gototests/example1.p
@@ -0,0 +1,79 @@
+label: 1, 2;
+var x, y: integer;
+begin
+  x:=0;
+  y:=2;
+  1: x:=x+1;
+  if x = y then
+    goto 2
+  else
+    goto 1
+  end;
+  2: print_num(x); newline()
+end.
+
+(*<<
+ 2
+>>*)
+
+(*[[
+MODULE Main 0 0
+IMPORT Lib 0
+ENDHDR
+
+FUNC MAIN 0
+! x:=0;
+CONST 0
+GLOBAL _x
+STOREW
+! y:=2;
+CONST 2
+GLOBAL _y
+STOREW
+! 1: x:=x+1;
+LABEL L1
+CONST 0
+GLOBAL __label
+STOREW
+GLOBAL _x
+LOADW
+CONST 1
+PLUS
+GLOBAL _x
+STOREW
+! if x = y then
+GLOBAL _x
+LOADW
+GLOBAL _y
+LOADW
+JEQ L3
+JUMP L4
+LABEL L3
+! goto 2
+JUMP L2
+JUMP L5
+LABEL L4
+! goto 1
+JUMP L1
+LABEL L5
+! 2: print_num(x); newline()
+LABEL L2
+CONST 0
+GLOBAL __label
+STOREW
+GLOBAL _x
+LOADW
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
+CONST 0
+GLOBAL lib.newline
+PCALL 0
```

```
+RETURN
+END
+
+GLOVAR _x 4
+GLOVAR _y 4
+GLOVAR __label 4
+! End
+]]*)
diff --git a/ppc4/gototests/example2.p b/ppc4/gototests/example2.p
new file mode 100644
index 0000000..6bea6d6
--- /dev/null
+++ b/ppc4/gototests/example2.p
@@ -0,0 +1,120 @@
+label: 1, 2;
+var i: integer;
+proc foo(x : integer): integer;
+  proc bar(y : integer): integer;
+  begin
+    goto 1
+  end;
+begin
+  return bar(x)
+end;
+begin
+  i := 1;
+  i := foo(i);
+  goto 2;
+  1: i:= 0;
+  2: print_num(i); newline()
+end.
+
+(*<<
+ 0
+>>*)
+
+(*[[
+MODULE Main 0 0
+IMPORT Lib 0
+ENDHDR
+
+FUNC MAIN 0
+! i := 1;
+CONST 1
+GLOBAL _i
+STOREW
+! i := foo(i);
+GLOBAL _i
+LOADW
+CONST 0
+GLOBAL _foo
+PCALLW 1
+GLOBAL __label
+LOADW
+CONST 0
+JEQ L3
+GLOBAL __label
+LOADW
+CONST 1
+JEQ L1
+GLOBAL __label
+LOADW
+CONST 2
+JEQ L2
+LABEL L3
```

```
+GLOBAL _i
+STOREW
+! goto 2;
+JUMP L2
+! 1: i:= 0;
+LABEL L1
+CONST 0
+GLOBAL __label
+STOREW
+CONST 0
+GLOBAL _i
+STOREW
+! 2: print_num(i); newline()
+LABEL L2
+CONST 0
+GLOBAL __label
+STOREW
+GLOBAL _i
+LOADW
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
+CONST 0
+GLOBAL lib.newline
+PCALL 0
+RETURN
+END
+
+FUNC _foo 0
+! return bar(x)
+LOCAL 16
+LOADW
+LOCAL 0
+GLOBAL _foo.bar
+PCALLW 1
+GLOBAL __label
+LOADW
+CONST 0
+JEQ L6
+GLOBAL __label
+LOADW
+CONST 1
+JNEQ L8
+RETURN
+LABEL L8
+GLOBAL __label
+LOADW
+CONST 2
+JNEQ L7
+RETURN
+LABEL L7
+LABEL L6
+RETURN
+ERROR E_RETURN 0
+END
+
+FUNC _foo.bar 0
+! goto 1
+CONST 1
+GLOBAL __label
+STOREW
+RETURN
+ERROR E_RETURN 0
+END
+
+GLOVAR _i 4
+GLOVAR __label 4
+! End
+]]*)
diff --git a/ppc4/gototests/example3.p b/ppc4/gototests/example3.p
new file mode 100644
index 0000000..1908913
--- /dev/null
+++ b/ppc4/gototests/example3.p
@@ -0,0 +1,11 @@
+label: 1;
+var x, y: integer;
+begin
+  x:=0;
+  y:=2;
+  goto 1;
+  if x = y then
+    1: x:=x+1
+  end;
+  print_num(x); newline()
+end.
diff --git a/ppc4/gototests/infinite.p b/ppc4/gototests/infinite.p
new file mode 100644
index 0000000..77b4270
--- /dev/null
+++ b/ppc4/gototests/infinite.p
@@ -0,0 +1,17 @@
+label: 1;
+
+proc loop();
+label: 2;
+begin
+  goto 1;
+  2: goto 2;
+end;
+
+begin
+  loop();
+  1: print_num(0); newline()
+end.
+
+(*
+ 0
+*)
diff --git a/ppc4/gototests/labeldeclaredtwice.p b/ppc4/gototests/labeldeclaredtwice.p
new file mode 100644
index 0000000..8e34f37
--- /dev/null
+++ b/ppc4/gototests/labeldeclaredtwice.p
@@ -0,0 +1,10 @@
+label: 1, 2, 2;
+var x: integer;
+begin
+  x := 0;
+  if x = 0 then
+    goto 2
+  end;
+  1: x := x + 1;
+  2: print_num(x); newline()
+end.
diff --git a/ppc4/gototests/labelempty.p b/ppc4/gototests/labelempty.p
new file mode 100644
index 0000000..f128a42
--- /dev/null
+++ b/ppc4/gototests/labelempty.p
```

```
@@ -0,0 +1,84 @@
+label: 1,2;
+var x, y: integer;
+begin
+  x := -5;
+  y := 3;
+  while x > 0 do
+    while y > 0 do
+      goto 1
+    end;
+  end;
+  goto 2;
+  1: print_num(x); newline();
+  2:
+end.
+
+(*<<
+>>*)
+
+(*[[
+MODULE Main 0 0
+IMPORT Lib 0
+ENDHDR
+
+FUNC MAIN 0
+! x := -5;
+CONST -5
+GLOBAL _x
+STOREW
+! y := 3;
+CONST 3
+GLOBAL _y
+STOREW
+! while x > 0 do
+JUMP L4
+LABEL L3
+! while y > 0 do
+JUMP L7
+LABEL L6
+! goto 1
+JUMP L1
+LABEL L7
+GLOBAL _y
+LOADW
+CONST 0
+JGT L6
+JUMP L8
+LABEL L8
+! end;
+LABEL L4
+GLOBAL _x
+LOADW
+CONST 0
+JGT L3
+JUMP L5
+LABEL L5
+! goto 2;
+JUMP L2
+! 1: print_num(x); newline();
+LABEL L1
+CONST 0
+GLOBAL __label
+STOREW
+GLOBAL _x
+LOADW
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
+CONST 0
+GLOBAL lib.newline
+PCALL 0
+! 2:
+LABEL L2
+CONST 0
+GLOBAL __label
+STOREW
+! end.
+RETURN
+END
+
+GLOVAR _x 4
+GLOVAR _y 4
+GLOVAR __label 4
+! End
+]]*)
diff --git a/ppc4/gototests/labelnotint.p b/ppc4/gototests/labelnotint.p
new file mode 100644
index 0000000..33485f4
--- /dev/null
+++ b/ppc4/gototests/labelnotint.p
@@ -0,0 +1,14 @@
+label: 1.5, 2;
+var x, y: integer;
+begin
+  x:=0;
+  y:=2;
+  1.5: x:=x+1;
+  if x = y then
+    goto 2
+  else
+    goto 10000
+  end;
+  2: print_num(x); newline()
+end.
+
diff --git a/ppc4/gototests/labelrange.p b/ppc4/gototests/labelrange.p
new file mode 100644
index 0000000..1e3aac9
--- /dev/null
+++ b/ppc4/gototests/labelrange.p
@@ -0,0 +1,14 @@
+label: 10000, 2;
+var x, y: integer;
+begin
+  x:=0;
+  y:=2;
+  10000: x:=x+1;
+  if x = y then
+    goto 2
+  else
+    goto 10000
+  end;
+  2: print_num(x); newline()
+end.
+
diff --git a/ppc4/gototests/labelsamename.p b/ppc4/gototests/labelsamename.p
new file mode 100644
index 0000000..5308f8e
--- /dev/null
+++ b/ppc4/gototests/labelsamename.p
```

```
@@ -0,0 +1,113 @@
+var i,j : integer;
+
+proc foo(x : integer);
+label: 2;
+begin
+  goto 2;
+  2: print_num(x); newline()
+end;
+
+proc goo(x: integer);
+label: 2;
+begin
+  goto 2;
+  print_num(x); newline();
+  2:
+end;
+
+begin
+  i := 5;
+  j := 1;
+  foo(i);
+  goo(i);
+end.
+
+(*<<
+ 5
+>>*)
+
+(*[[
+MODULE Main 0 0
+IMPORT Lib 0
+ENDHDR
+
+FUNC MAIN 0
+! i := 5;
+CONST 5
+GLOBAL _i
+STOREW
+! j := 1;
+CONST 1
+GLOBAL _j
+STOREW
+! foo(i);
+GLOBAL _i
+LOADW
+CONST 0
+GLOBAL _foo
+PCALL 1
+GLOBAL __label
+LOADW
+CONST 0
+JEQ L1
+LABEL L1
+! goo(i);
+GLOBAL _i
+LOADW
+CONST 0
+GLOBAL _goo
+PCALL 1
+GLOBAL __label
+LOADW
+CONST 0
+JEQ L2
+LABEL L2
+! end.
+RETURN
+END
+
+FUNC _foo 0
+! goto 2;
+JUMP L3
+! 2: print_num(x); newline()
+LABEL L3
+CONST 0
+GLOBAL __label
+STOREW
+LOCAL 16
+LOADW
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
+CONST 0
+GLOBAL lib.newline
+PCALL 0
+RETURN
+END
+
+FUNC _goo 0
+! goto 2;
+JUMP L4
+! print_num(x); newline();
+LOCAL 16
+LOADW
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
+CONST 0
+GLOBAL lib.newline
+PCALL 0
+! 2:
+LABEL L4
+CONST 0
+GLOBAL __label
+STOREW
+! end;
+RETURN
+END
+
+GLOVAR _i 4
+GLOVAR _j 4
+GLOVAR __label 4
+! End
+]]*)
diff --git a/ppc4/gototests/labelscope.p b/ppc4/gototests/labelscope.p
new file mode 100644
index 0000000..41234a0
--- /dev/null
+++ b/ppc4/gototests/labelscope.p
@@ -0,0 +1,18 @@
+label: 2;
+var i: integer;
+proc foo(x : integer): integer;
+  label: 1;
+  proc bar(y : integer): integer;
+  begin
+    goto 1
+  end;
+begin
+  return bar(x)
```

```
+end;
+begin
+  i := 1;
+  i := foo(i);
+  goto 2;
+  1: i:= 0;
+  2: print_num(i); newline()
+end.
diff --git a/ppc4/gototests/labelundeclared.p b/ppc4/gototests/labelundeclared.p
new file mode 100644
index 0000000..b06f733
--- /dev/null
+++ b/ppc4/gototests/labelundeclared.p
@@ -0,0 +1,10 @@
+label: 1;
+var x: integer;
+begin
+  x := 0;
+  if x = 0 then
+    goto 2
+  end;
+  1: x := x + 1;
+  2: print_num(x); newline()
+end.
diff --git a/ppc4/gototests/labelusedtwice.p b/ppc4/gototests/labelusedtwice.p
new file mode 100644
index 0000000..3f6ae7b
--- /dev/null
+++ b/ppc4/gototests/labelusedtwice.p
@@ -0,0 +1,14 @@
+label: 1,2;
+var x, y: integer;
+begin
+  x := 0;
+  y := 2;
+  1: if x = y then
+    goto 2
+  else
+    x:= x+1;
+    goto 1
+  end;
+  2: print_num(x); newline();
+  1: print_num(x); newline()
+end.
diff --git a/ppc4/gototests/labelusedtwice2.p b/ppc4/gototests/labelusedtwice2.p
new file mode 100644
index 0000000..1984072
--- /dev/null
+++ b/ppc4/gototests/labelusedtwice2.p
@@ -0,0 +1,14 @@
+label: 1,2;
+var x, y: integer;
+begin
+  x := 2;
+  y := 2;
+  1: if x = y then
+    goto 2
+  else
+    x:= x+1;
+    goto 1;
+    1: print_num(x); newline()
+  end;
+  2: print_num(x); newline();
+end.
diff --git a/ppc4/gototests/ladder.p b/ppc4/gototests/ladder.p
```

```
new file mode 100644
index 0000000..017d34b
--- /dev/null
+++ b/ppc4/gototests/ladder.p
@@ -0,0 +1,214 @@
+var step : integer;
+
+proc ladder1();
+label: 1;
+  proc ladder2();
+    proc ladder3();
+      proc ladder4();
+      begin
+        print_num(4); newline()
+      end;
+    begin
+      print_num(3);
+      if step = 3 then
+        goto 1
+      end;
+      ladder4()
+    end;
+  begin
+     print_num(2);
+     if step=2 then
+        goto 1
+     end;
+      ladder3()
+  end;
+begin
+  1: step:=step+1; if step<>1 then newline() end; print_num(1);
+  if step=1 then
+    goto 1;
+  end;
+  ladder2();
+end;
+
+begin
+ ladder1();
+end.
+
+(*<<
+ 1
+ 12
+ 123
+ 1234
+>>*)
+
+(*[[
+MODULE Main 0 0
+IMPORT Lib 0
+ENDHDR
+
+FUNC MAIN 0
+! ladder1();
+CONST 0
+GLOBAL _ladder1
+PCALL 0
+GLOBAL __label
+LOADW
+CONST 0
+JEQ L1
+LABEL L1
+! end.
+RETURN
```

```
+END
+
+FUNC _ladder1 0
+! 1: step:=step+1; if step<>1 then newline() end; print_num(1);
+LABEL L2
+CONST 0
+GLOBAL __label
+STOREW
+GLOBAL _step
+LOADW
+CONST 1
+PLUS
+GLOBAL _step
+STOREW
+GLOBAL _step
+LOADW
+CONST 1
+JNEQ L3
+JUMP L4
+LABEL L3
+CONST 0
+GLOBAL lib.newline
+PCALL 0
+JUMP L5
+LABEL L4
+LABEL L5
+CONST 1
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
+! if step=1 then
+GLOBAL _step
+LOADW
+CONST 1
+JEQ L6
+JUMP L7
+LABEL L6
+! goto 1;
+JUMP L2
+! end;
+JUMP L8
+LABEL L7
+LABEL L8
+! ladder2();
+LOCAL 0
+GLOBAL _ladder1.ladder2
+PCALL 0
+GLOBAL __label
+LOADW
+CONST 0
+JEQ L9
+GLOBAL __label
+LOADW
+CONST 1
+JEQ L2
+LABEL L9
+! end;
+RETURN
+END
+
+FUNC _ladder1.ladder2 0
+! print_num(2);
+CONST 2
+CONST 0
+GLOBAL lib.print_num
```

```
+PCALL 1
+! if step=2 then
+GLOBAL _step
+LOADW
+CONST 2
+JEQ L11
+JUMP L12
+LABEL L11
+! goto 1
+CONST 1
+GLOBAL __label
+STOREW
+RETURN
+JUMP L13
+LABEL L12
+LABEL L13
+! ladder3()
+LOCAL 0
+GLOBAL _ladder1.ladder2.ladder3
+PCALL 0
+GLOBAL __label
+LOADW
+CONST 0
+JEQ L14
+GLOBAL __label
+LOADW
+CONST 1
+JNEQ L15
+RETURN
+LABEL L15
+LABEL L14
+RETURN
+END
+
+FUNC _ladder1.ladder2.ladder3 0
+! print_num(3);
+CONST 3
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
+! if step = 3 then
+GLOBAL _step
+LOADW
+CONST 3
+JEQ L16
+JUMP L17
+LABEL L16
+! goto 1
+CONST 1
+GLOBAL __label
+STOREW
+RETURN
+JUMP L18
+LABEL L17
+LABEL L18
+! ladder4()
+LOCAL 0
+GLOBAL _ladder1.ladder2.ladder3.ladder4
+PCALL 0
+GLOBAL __label
+LOADW
+CONST 0
+JEQ L19
+GLOBAL __label
+LOADW
```

```
+CONST 1
+JNEQ L20
+RETURN
+LABEL L20
+LABEL L19
+RETURN
+END
+
+FUNC _ladder1.ladder2.ladder3.ladder4 0
+! print_num(4); newline()
+CONST 4
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
+CONST 0
+GLOBAL lib.newline
+PCALL 0
+RETURN
+END
+
+GLOVAR _step 4
+GLOVAR __label 4
+! End
+]]*)
```

--END OF GOTO TESTS--

```
diff --git a/ppc4/keiko.ml b/ppc4/keiko.ml
index 0b52228..479722e 100644
--- a/ppc4/keiko.ml
+++ b/ppc4/keiko.ml
@@ -58,7 +58,7 @@ type code =

   | SEQ of code list            (* Sequence of other instructions *)
   | NOP                         (* Null operation *)
-
+
 let mark_line n ys =
   if n = 0 then ys else
     match ys with
diff --git a/ppc4/kgen.ml b/ppc4/kgen.ml
index 2de04a9..def24d1 100644
--- a/ppc4/kgen.ml
+++ b/ppc4/kgen.ml
@@ -78,6 +78,8 @@ let gen_closure d =
           address d; load_addr]
     | _ -> failwith "missing closure"

+let table : (int, (int*codelab)) Hashtbl.t = Hashtbl.create 16
+
 (* |gen_addr| -- code for the address of a variable *)
 let rec gen_addr v =
   match v.e_guts with
@@ -137,6 +139,14 @@ and gen_expr e =
             | _ -> failwith "gen_expr"
         end

+and jump_after_proc key value acc =
+  let (lev,lab) = value and
+      lab_ = label () in
+    if lev<>(!level) then
+      (SEQ [GLOBAL "__label"; LOAD 4; CONST key; JUMPC (Neq, lab_); RETURN; LABEL lab_]) ::
acc
+    else
+      (SEQ [GLOBAL "__label"; LOAD 4; CONST key; JUMPC (Eq,lab)] ) :: acc
+
 (* |gen_call| -- generate code to call a procedure *)
 and gen_call x args =
   let d = get_def x in
@@ -145,10 +155,14 @@ and gen_call x args =
         gen_libcall q args
     | _ ->
         let p = get_proc d.d_type in
+        let code =
        SEQ [
          SEQ (List.map gen_arg (List.rev (List.combine p.p_fparams args)));
          gen_closure d;
          PCALL (p.p_pcount, count_of p.p_result)]
+        in let lab1 = label () in
+        let casejump = SEQ (Hashtbl.fold jump_after_proc table []) in
+        SEQ [code; GLOBAL "__label"; LOAD 4; CONST 0; JUMPC (Eq, lab1); casejump; LABEL lab1]

 (* |gen_arg| -- generate code to push a procedure argument *)
 and gen_arg (f, a) =
@@ -241,12 +255,39 @@ let gen_jtable tab0 deflab =
     SEQ [CONST lob; BINOP Minus; JCASE (tab lob table); JUMP deflab]
   end

+let add_label (table : (int, (int*codelab)) Hashtbl.t) (key : int) =
+  let l = label () in if Hashtbl.mem table key then failwith "label declared twice"
+                      else Hashtbl.add table key (-1, l)
+
+let add_labels table labels =
+  match labels with
+      LabelDecl ls -> List.iter (add_label table) ls
+
+let remove_labels table labels =
+  match labels with
+      LabelDecl ls -> List.iter (Hashtbl.remove table) ls
+
+let update_table_label table s =
+  match s.s_guts with
+      LabelStmt (n,ss) -> if Hashtbl.mem table n then
+                            let (lev,lab) = Hashtbl.find table n in
+                              if lev<>(-1) then failwith "label used twice"
+                              else Hashtbl.replace table n (!level, lab)
+                          else failwith ("label undeclared: " ^ string_of_int(n))
+    | _ -> ()
+
+let update_table table ss =
+  List.iter (update_table_label table) ss
+
+let temps = ref 0
+let temps_size = Hashtbl.create 16
+
 (* |gen_stmt| -- generate code for a statement *)
 let rec gen_stmt s =
   let code =
     match s.s_guts with
         Skip -> NOP
-      | Seq ss -> SEQ (List.map gen_stmt ss)
+      | Seq ss -> update_table table ss; SEQ (List.map gen_stmt ss )
       | Assign (v, e) ->
           if scalar v.e_type || is_pointer v.e_type then
            SEQ [gen_expr e; gen_addr v; STORE (size_of v.e_type)]
@@ -267,9 +308,7 @@ let rec gen_stmt s =
             LABEL lab1; gen_stmt thenpt; JUMP lab3;
            LABEL lab2; gen_stmt elsept; LABEL lab3]
       | WhileStmt (test, body) ->
```

```
-         let lab1 = label () and lab2 = label () and lab3 = label () in
-           SEQ [JUMP lab2; LABEL lab1; gen_stmt body;
-             LABEL lab2; gen_cond lab1 lab3 test; LABEL lab3]
+           constant_while test body
       | RepeatStmt (body, test) ->
           let lab1 = label () and lab2 = label () in
           SEQ [LABEL lab1; gen_stmt body;
@@ -292,8 +331,150 @@ let rec gen_stmt s =
           SEQ [gen_expr sel; gen_jtable table deflab;
             SEQ (List.map2 gen_case labs arms);
             LABEL deflab; gen_stmt deflt;
-             LABEL donelab] in
-  SEQ [if s.s_line <> 0 then LINE s.s_line else NOP; code]
+             LABEL donelab]
+      | LabelStmt (n,ss) ->
+           if Hashtbl.mem table n then
+           let (level,lab) = Hashtbl.find table n in
+           SEQ [LABEL lab; CONST 0; GLOBAL "__label"; STORE 4; gen_stmt ss]
+           else failwith "label undeclared"
+      | GotoStmt (n) ->
+           if Hashtbl.mem table n then
+             let (lev, lab) = Hashtbl.find table n in
+             if lev<>(-1) then
+               if lev<>(!level) then SEQ [ CONST n; GLOBAL "__label"; STORE 4; RETURN]
+               else JUMP lab
+             else failwith ("label nesting invalid: " ^ string_of_int(n))
+           else failwith "label undeclared"
+  in SEQ [if s.s_line <> 0 then LINE s.s_line else NOP; code]
+
+and defs_update defs s =
+  match s.s_guts with
+      Assign (v,e) -> (match v.e_guts with
+                         Variable x -> let d = get_def x in
+                           (if not (Hashtbl.mem defs d.d_tag) then Hashtbl.add defs d.d_tag 1
else Hashtbl.replace defs d.d_tag (Hashtbl.find defs d.d_tag + 1))
+                       | _ -> ())
+    | Seq ss -> List.iter (defs_update defs) ss
+    | WhileStmt (test,body) -> defs_update defs body
+    | IfStmt (test, thenpt, elsept) -> defs_update defs thenpt; defs_update defs elsept
+    | RepeatStmt (body, test) -> defs_update defs body
+    | ForStmt (var, lo, hi, body) -> defs_update defs body
+    | CaseStmt (sel, arms, deflt) -> List.iter (fun (ex,st) -> defs_update defs st) arms;
defs_update defs deflt
+    | LabelStmt (n,ss) -> defs_update defs ss
+    | _ -> ()
+
+and vars_in_expr e opt =
+  match e.e_guts with
+      Variable x -> let d = get_def x in [d.d_tag]
+    | Sub (a,i) -> union (vars_in_expr a opt) (vars_in_expr i opt)
+    | Select (e,x) -> let d = get_def x in union [d.d_tag] (vars_in_expr e opt)
+    | Deref e -> vars_in_expr e opt
+    | FuncCall (_,_) -> opt := false; []
+    | Monop (op,e) -> vars_in_expr e opt
+    | Binop (op,e1,e2) -> union (vars_in_expr e1 opt) (vars_in_expr e2 opt)
+    | _ -> []
+
+and union lst1 lst2 =
+  let rec add_unique acc lst =
+    match lst with
+    | [] -> acc
+    | x :: xs ->
+        if List.mem x acc then
+          add_unique acc xs
+        else
+          add_unique (x :: acc) xs
+  in
+  add_unique lst2 lst1
+
+and partition1 preheader s =
+   match s.s_guts with
+       Assign (v,e) -> (match v.e_guts with
+                           Variable x -> let d = get_def x in Hashtbl.mem preheader d.d_tag
+                         | _ -> false)
+     | _ -> false
+
+and live_analysis used updated defs preheader if_tests level opt s =
+  match s.s_guts with
+      Assign (v,e) -> let used_in_assign = vars_in_expr e opt in
+                      used := union !used used_in_assign;
+                      (match v.e_guts with
+                           Variable x -> let d = get_def x in
+                                         let invariant_func tag accum = (((not (Hashtbl.mem
defs tag))||((Hashtbl.find_opt defs tag = Some 1)&&(Hashtbl.mem preheader tag))) && accum) in
+                                         let inv = List.fold_right invariant_func
used_in_assign true in
+                                         if (inv && ((Hashtbl.find defs d.d_tag) = 1) && (not
(List.mem d.d_tag !used)) && (not (Hashtbl.mem preheader d.d_tag)) && level) then
+                                           (updated := true; Hashtbl.add preheader d.d_tag 0)
+                         | _ -> ())
+    | Seq ss -> List.iter (live_analysis used updated defs preheader if_tests level opt) ss
+    | WhileStmt (test,body) -> used := union !used (vars_in_expr test opt); live_analysis
used updated defs preheader if_tests false opt body
+    | IfStmt (test, thenpt, elsept) -> let used_in_test = vars_in_expr test opt in
+                                       let invariant_func tag accum = (((not (Hashtbl.mem
defs tag))||((Hashtbl.find_opt defs tag = Some 1)&&(Hashtbl.mem preheader tag))) && accum) in
+                                       let inv = List.fold_right invariant_func used_in_test
true in
+                                       if (inv && level && (not (Hashtbl.mem if_tests test)))
then
+                                         (temps:=!temps+1; Hashtbl.add temps_size !temps
(size_of test.e_type); updated:= true; Hashtbl.add if_tests test (!temps));
+                                       used := union !used used_in_test; live_analysis used
updated defs preheader if_tests false opt thenpt; live_analysis used updated defs preheader
if_tests false opt elsept
+    | RepeatStmt (body, test) -> used := union !used (vars_in_expr test opt); live_analysis
used updated defs preheader if_tests false opt body
+    | ForStmt (var, lo, hi, body) -> let used_in_for = union (union (vars_in_expr var opt)
(vars_in_expr lo opt)) (vars_in_expr hi opt) in
+                                     used := union !used used_in_for;
+                                     live_analysis used updated defs preheader if_tests false
opt body
+    | CaseStmt (sel, arms, deflt) -> let used_in_case = List.fold_right (fun (ex,st) accum ->
union accum (vars_in_expr ex opt)) arms [] in
+                                     used := union !used used_in_case;
+                                     List.iter (fun (ex,st) -> live_analysis used updated
defs preheader if_tests false opt st) arms; live_analysis used updated defs preheader if_tests
false opt deflt
+    | LabelStmt (n,ss) -> live_analysis used updated defs preheader if_tests level opt ss
+    | ProcCall (p,args) -> opt := false
+    | _ -> ()
+
+and temp_func if_tests s accum =
+  match s.s_guts with
+      IfStmt (test, thenpt, elsept) ->
+        if Hashtbl.mem if_tests test then
+          let t = Hashtbl.find if_tests test in
+          SEQ [gen_expr test; GLOBAL ("__t"^string_of_int(t));STORE (size_of test.e_type)] ::
accum
+        else
+         accum
+    | _ -> accum
+
+and update_ifs if_tests s =
+  match s.s_guts with
+      IfStmt (test, thenpt, elsept) ->
+        if Hashtbl.mem if_tests test then
+          let t = Hashtbl.find if_tests test in
+          let id = intern ("__t"^string_of_int(t)) in
+          let name = make_name(id,0) in
+          name.x_def <- Some
{d_tag=id;d_kind=VarDef;d_type=test.e_type;d_level=(!level);d_addr=Global
("__t"^string_of_int(t))};
+          let expr = make_expr(Variable name) in
+          expr.e_type <- test.e_type;
+          make_stmt(IfStmt(expr,thenpt,elsept),s.s_line)
+        else s
+    | _ -> s
+
+and constant_while test body =
+  let opt = ref (!optflag) in
+  let defs = Hashtbl.create 16 in
+  let preheader = Hashtbl.create 16 in
+  let if_tests = Hashtbl.create 16 in
+  let used = ref [] and updated = ref true in
+  defs_update defs body;
+  while !updated do
+     updated := false;
+     used := [];
+     live_analysis used updated defs preheader if_tests true opt body;
+  done;
+  if !opt then
+  (match body.s_guts with
+      Seq ss ->
+        let new_before, new_body0 = List.partition (partition1 preheader) ss in
+        let temp_defs = List.fold_right (temp_func if_tests) ss [] in
+        let new_body = List.map (update_ifs if_tests) new_body0 in
+        let const_stmt = make_stmt(Seq new_before, 0) and new_body_stmt = make_stmt(Seq
new_body,0) in
+        let lab1 = label () and lab2 = label() and lab3 = label () in
+        SEQ [gen_cond lab2 lab3 test; LABEL lab2; gen_stmt const_stmt; SEQ temp_defs; LABEL
lab1; gen_stmt new_body_stmt;
+             gen_cond lab1 lab3 test; LABEL lab3]
+    | _ -> let lab1 = label () and lab2 = label () and lab3 = label () in
+      SEQ [JUMP lab2; LABEL lab1; gen_stmt body;
+           LABEL lab2; gen_cond lab1 lab3 test; LABEL lab3])
+  else
+    let lab1 = label () and lab2 = label () and lab3 = label () in
+      SEQ [JUMP lab2; LABEL lab1; gen_stmt body;
+           LABEL lab2; gen_cond lab1 lab3 test; LABEL lab3]

 (* |do_proc| -- generate code for a procedure *)
 let do_proc lab lev rtype fsize body =
@@ -305,17 +486,20 @@ let do_proc lab lev rtype fsize body =
   Keiko.output (if !optflag then Peepopt.optimise code else code);
   printf "END\n\n" []

+
 (* |gen_proc| -- translate a procedure, ignore other declarations *)
 let rec gen_proc =
   function
-      ProcDecl (Heading (x, _, _), Block (locals, body, fsize, nregv)) ->
+      ProcDecl (Heading (x, _, _), Block (new_labels, locals, body, fsize, nregv)) ->
        let d = get_def x in
        let p = get_proc d.d_type in
        begin
          match d.d_addr with
              Global lab ->
+               add_labels table new_labels;
                do_proc lab d.d_level p.p_result !fsize body;
-               gen_procs locals
+               gen_procs locals;
+               remove_labels table new_labels
            | _ -> failwith "gen_proc"
        end
    | _ -> ()
@@ -346,14 +530,19 @@ let gen_global d =
   match d.d_kind with
       VarDef ->
        (match d.d_addr with
-           Global lab ->
+           Global lab ->
              printf "GLOVAR $ $\n" [fStr lab; fNum (size_of d.d_type)]
          | _ -> failwith "gen_global")
    | _ -> ()
-
+
 (* |translate| -- generate code for the whole program *)
-let translate (Prog (Block (globals, main, _, _), glodefs)) =
+let translate (Prog (Block (labels, globals, main, _, _), glodefs)) =
+  add_labels table labels;
+  do_proc "MAIN" (-1) voidtype 0 main;
   gen_procs globals;
-  do_proc "MAIN" 0 voidtype 0 main;
   List.iter gen_global !glodefs;
+  printf "GLOVAR __label 4\n" [];
+  for i = 1 to !temps do
+    printf "GLOVAR __t$ $\n" [fNum i; fNum (Hashtbl.find temps_size i) ];
+  done;
   List.iter gen_string (string_table ())
diff --git a/ppc4/lexer.mll b/ppc4/lexer.mll
index 8a2e59c..b2af70e 100644
--- a/ppc4/lexer.mll
+++ b/ppc4/lexer.mll
@@ -20,7 +20,7 @@ let symtable =
       ("repeat", REPEAT); ("until", UNTIL); ("for", FOR);
       ("elsif", ELSIF); ("case", CASE);
      ("and", MULOP And); ("div", MULOP Div); ("or", ADDOP Or);
-     ("not", NOT); ("mod", MULOP Mod) ]
+     ("not", NOT); ("mod", MULOP Mod); ("goto", GOTO); ("label", LABEL) ]

 let lookup s =
   try Hashtbl.find symtable s with
```

LOOP-INVARIANT CODE MOTION TESTS

```
diff --git a/ppc4/motiontests/example4.p b/ppc4/motiontests/example4.p
new file mode 100644
index 0000000..e2f3513
--- /dev/null
+++ b/ppc4/motiontests/example4.p
@@ -0,0 +1,56 @@
+var x, y: integer;
+begin
+  x:=0;
+  while x < 10 do
+    y := 1;
+    x:=x+y
+  end;
+  print_num(x); newline()
+end.
```

```
+
+(*<<
+ 10
+>>*)
+
+(*[[
+MODULE Main 0 0
+IMPORT Lib 0
+ENDHDR
+
+FUNC MAIN 0
+! x:=0;
+CONST 0
+STGW _x
+! while x < 10 do
+LDGW _x
+CONST 10
+JGEQ L3
+! y := 1;
+CONST 1
+STGW _y
+LABEL L1
+! x:=x+y
+LDGW _x
+LDGW _y
+PLUS
+STGW _x
+LDGW _x
+CONST 10
+JLT L1
+LABEL L3
+! print_num(x); newline()
+LDGW _x
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
+CONST 0
+GLOBAL lib.newline
+PCALL 0
+RETURN
+END
+
+GLOVAR _x 4
+GLOVAR _y 4
+GLOVAR __label 4
+! End
+]]*)
diff --git a/ppc4/motiontests/if1.p b/ppc4/motiontests/if1.p
new file mode 100644
index 0000000..4e47a57
--- /dev/null
+++ b/ppc4/motiontests/if1.p
@@ -0,0 +1,72 @@
+var x, y: integer;
+begin
+  x:=0;
+  while x < 10 do
+    y:= 1;
+    if y+4*8=33 then
+      x:=x+y
+    end;
+  end;
+  print_num(x); newline()
+end.
+
+(*<<
+ 10
+>>*)
+
+(*[[
+MODULE Main 0 0
+IMPORT Lib 0
+ENDHDR
+
+FUNC MAIN 0
+! x:=0;
+CONST 0
+STGW _x
+! while x < 10 do
+LDGW _x
+CONST 10
+JGEQ L3
+! y:= 1;
+CONST 1
+STGW _y
+LDGW _y
+CONST 32
+PLUS
+CONST 33
+EQ
+STGC __t1
+LABEL L1
+! if y+4*8=33 then
+LDGC __t1
+JNEQZ L4
+JUMP L6
+LABEL L4
+! x:=x+y
+LDGW _x
+LDGW _y
+PLUS
+STGW _x
+LABEL L6
+! end;
+LDGW _x
+CONST 10
+JLT L1
+LABEL L3
+! print_num(x); newline()
+LDGW _x
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
+CONST 0
+GLOBAL lib.newline
+PCALL 0
+RETURN
+END
+
+GLOVAR _x 4
+GLOVAR _y 4
+GLOVAR __label 4
+GLOVAR __t1 1
+! End
+]]*)
diff --git a/ppc4/motiontests/if2.p b/ppc4/motiontests/if2.p
new file mode 100644
index 0000000..ecc026c
--- /dev/null
+++ b/ppc4/motiontests/if2.p
```

```
@@ -0,0 +1,77 @@
+var x, y, z: integer;
+begin
+  x:=0;
+  while x < 10 do
+    y:= 1;
+    z := y;
+    if z+4*8=33 then
+      x:=x+z
+    end;
+  end;
+  print_num(x); newline()
+end.
+
+(*<<
+ 10
+>>*)
+
+(*[[
+MODULE Main 0 0
+IMPORT Lib 0
+ENDHDR
+
+FUNC MAIN 0
+! x:=0;
+CONST 0
+STGW _x
+! while x < 10 do
+LDGW _x
+CONST 10
+JGEQ L3
+! y:= 1;
+CONST 1
+STGW _y
+! z := y;
+LDGW _y
+STGW _z
+LDGW _z
+CONST 32
+PLUS
+CONST 33
+EQ
+STGC __t1
+LABEL L1
+! if z+4*8=33 then
+LDGC __t1
+JNEQZ L4
+JUMP L6
+LABEL L4
+! x:=x+z
+LDGW _x
+LDGW _z
+PLUS
+STGW _x
+LABEL L6
+! end;
+LDGW _x
+CONST 10
+JLT L1
+LABEL L3
+! print_num(x); newline()
+LDGW _x
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
```

```
+CONST 0
+GLOBAL lib.newline
+PCALL 0
+RETURN
+END
+
+GLOVAR _x 4
+GLOVAR _y 4
+GLOVAR _z 4
+GLOVAR __label 4
+GLOVAR __t1 1
+! End
+]]*)
diff --git a/ppc4/motiontests/if3.p b/ppc4/motiontests/if3.p
new file mode 100644
index 0000000..926751a
--- /dev/null
+++ b/ppc4/motiontests/if3.p
@@ -0,0 +1,79 @@
+var x, y, z: integer;
+begin
+  x:=0;
+  while x < 10 do
+    y:= 1;
+    z:=2;
+    if x<>9 then
+      x:=x+y
+    else
+      x:=x+z
+    end;
+  end;
+  print_num(x); newline()
+end.
+
+(*<<
+ 11
+>>*)
+
+(*[[
+MODULE Main 0 0
+IMPORT Lib 0
+ENDHDR
+
+FUNC MAIN 0
+! x:=0;
+CONST 0
+STGW _x
+! while x < 10 do
+LDGW _x
+CONST 10
+JGEQ L3
+! y:= 1;
+CONST 1
+STGW _y
+! z:=2;
+CONST 2
+STGW _z
+LABEL L1
+! if x<>9 then
+LDGW _x
+CONST 9
+JEQ L5
+! x:=x+y
+LDGW _x
+LDGW _y
```

```
+PLUS
+STGW _x
+JUMP L6
+LABEL L5
+! x:=x+z
+LDGW _x
+LDGW _z
+PLUS
+STGW _x
+LABEL L6
+! end;
+LDGW _x
+CONST 10
+JLT L1
+LABEL L3
+! print_num(x); newline()
+LDGW _x
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
+CONST 0
+GLOBAL lib.newline
+PCALL 0
+RETURN
+END
+
+GLOVAR _x 4
+GLOVAR _y 4
+GLOVAR _z 4
+GLOVAR __label 4
+! End
+]]*)
+
diff --git a/ppc4/motiontests/if4.p b/ppc4/motiontests/if4.p
new file mode 100644
index 0000000..f11e2b8
--- /dev/null
+++ b/ppc4/motiontests/if4.p
@@ -0,0 +1,69 @@
+var x, y, z: integer;
+begin
+  x := 2;
+  y := 1;
+  while x < 10 do
+    x := x + y;
+    if y = 1 then
+       y := 2
+     end;
+  end;
+  print_num(x); newline()
+end.
+
+(*<<
+ 11
+>>*)
+
+(*[[
+MODULE Main 0 0
+IMPORT Lib 0
+ENDHDR
+
+FUNC MAIN 0
+! x := 2;
+CONST 2
+STGW _x
```

```
+! y := 1;
+CONST 1
+STGW _y
+! while x < 10 do
+LDGW _x
+CONST 10
+JGEQ L3
+LABEL L1
+! x := x + y;
+LDGW _x
+LDGW _y
+PLUS
+STGW _x
+! if y = 1 then
+LDGW _y
+CONST 1
+JNEQ L6
+! y := 2
+CONST 2
+STGW _y
+LABEL L6
+! end;
+LDGW _x
+CONST 10
+JLT L1
+LABEL L3
+! print_num(x); newline()
+LDGW _x
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
+CONST 0
+GLOBAL lib.newline
+PCALL 0
+RETURN
+END
+
+GLOVAR _x 4
+GLOVAR _y 4
+GLOVAR _z 4
+GLOVAR __label 4
+! End
+]]*)
diff --git a/ppc4/motiontests/imp1.p b/ppc4/motiontests/imp1.p
new file mode 100644
index 0000000..32cd017
--- /dev/null
+++ b/ppc4/motiontests/imp1.p
@@ -0,0 +1,80 @@
+var t, a, b, i: integer;
+var m : array 10 of integer;
+begin
+  a := 5;
+  b := 3;
+  t:=0;
+  i := 0;
+  while i < 10 do
+    t:=a+b;
+    m[i] := t;
+    i := i + 1
+  end;
+  print_num(t); newline()
+end.
+
+(*<<
+ 8
+>>*)
+
+(*[[
+MODULE Main 0 0
+IMPORT Lib 0
+ENDHDR
+
+FUNC MAIN 0
+! a := 5;
+CONST 5
+STGW _a
+! b := 3;
+CONST 3
+STGW _b
+! t:=0;
+CONST 0
+STGW _t
+! i := 0;
+CONST 0
+STGW _i
+! while i < 10 do
+LDGW _i
+CONST 10
+JGEQ L3
+! t:=a+b;
+LDGW _a
+LDGW _b
+PLUS
+STGW _t
+LABEL L1
+! m[i] := t;
+LDGW _t
+GLOBAL _m
+LDGW _i
+STIW
+! i := i + 1
+LDGW _i
+CONST 1
+PLUS
+STGW _i
+LDGW _i
+CONST 10
+JLT L1
+LABEL L3
+! print_num(t); newline()
+LDGW _t
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
+CONST 0
+GLOBAL lib.newline
+PCALL 0
+RETURN
+END
+
+GLOVAR _t 4
+GLOVAR _a 4
+GLOVAR _b 4
+GLOVAR _i 4
+GLOVAR _m 40
+GLOVAR __label 4
+! End
+]]*)
diff --git a/ppc4/motiontests/imp2.p b/ppc4/motiontests/imp2.p
```

```
new file mode 100644
index 0000000..b7be8a0
--- /dev/null
+++ b/ppc4/motiontests/imp2.p
@@ -0,0 +1,97 @@
+var t, a, b, i, j: integer;
+var m : array 10 of integer;
+begin
+  a := 5;
+  b := 3;
+  t:=0;
+  i := 0;
+  j := 5;
+  while i < 10 do
+    t:=a+b;
+    m[i] := t;
+    t := 0;
+    m[j] := t;
+    i := i + 1
+  end;
+  print_num(m[j]); newline()
+end.
+
+(*<<
+ 0
+>>*)
+
+(*[[
+MODULE Main 0 0
+IMPORT Lib 0
+ENDHDR
+
+FUNC MAIN 0
+! a := 5;
+CONST 5
+STGW _a
+! b := 3;
+CONST 3
+STGW _b
+! t:=0;
+CONST 0
+STGW _t
+! i := 0;
+CONST 0
+STGW _i
+! j := 5;
+CONST 5
+STGW _j
+! while i < 10 do
+LDGW _i
+CONST 10
+JGEQ L3
+LABEL L1
+! t:=a+b;
+LDGW _a
+LDGW _b
+PLUS
+STGW _t
+! m[i] := t;
+LDGW _t
+GLOBAL _m
+LDGW _i
+STIW
+! t := 0;
+CONST 0
```

```
+STGW _t
+! m[j] := t;
+LDGW _t
+GLOBAL _m
+LDGW _j
+STIW
+! i := i + 1
+LDGW _i
+CONST 1
+PLUS
+STGW _i
+LDGW _i
+CONST 10
+JLT L1
+LABEL L3
+! print_num(m[j]); newline()
+GLOBAL _m
+LDGW _j
+LDIW
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
+CONST 0
+GLOBAL lib.newline
+PCALL 0
+RETURN
+END
+
+GLOVAR _t 4
+GLOVAR _a 4
+GLOVAR _b 4
+GLOVAR _i 4
+GLOVAR _j 4
+GLOVAR _m 40
+GLOVAR __label 4
+! End
+]]*)
diff --git a/ppc4/motiontests/imp3.p b/ppc4/motiontests/imp3.p
new file mode 100644
index 0000000..d886636
--- /dev/null
+++ b/ppc4/motiontests/imp3.p
@@ -0,0 +1,89 @@
+var t, a, b, i: integer;
+var m : array 10 of integer;
+begin
+  a := 5;
+  b := 3;
+  t:=0;
+  i := 0;
+  while i < 5 do
+    m[9-i] := t;
+    t:=a+b;
+    m[i] := t;
+    i := i + 1
+  end;
+  print_num(m[9]); newline()
+end.
+
+(*<<
+ 0
+>>*)
+
+(*[[
+MODULE Main 0 0
```

```
+IMPORT Lib 0
+ENDHDR
+
+FUNC MAIN 0
+! a := 5;
+CONST 5
+STGW _a
+! b := 3;
+CONST 3
+STGW _b
+! t:=0;
+CONST 0
+STGW _t
+! i := 0;
+CONST 0
+STGW _i
+! while i < 5 do
+LDGW _i
+CONST 5
+JGEQ L3
+LABEL L1
+! m[9-i] := t;
+LDGW _t
+GLOBAL _m
+CONST 9
+LDGW _i
+MINUS
+STIW
+! t:=a+b;
+LDGW _a
+LDGW _b
+PLUS
+STGW _t
+! m[i] := t;
+LDGW _t
+GLOBAL _m
+LDGW _i
+STIW
+! i := i + 1
+LDGW _i
+CONST 1
+PLUS
+STGW _i
+LDGW _i
+CONST 5
+JLT L1
+LABEL L3
+! print_num(m[9]); newline()
+GLOBAL _m
+LDNW 36
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
+CONST 0
+GLOBAL lib.newline
+PCALL 0
+RETURN
+END
+
+GLOVAR _t 4
+GLOVAR _a 4
+GLOVAR _b 4
+GLOVAR _i 4
+GLOVAR _m 40
+GLOVAR __label 4
```

```
+! End
+]]*)
diff --git a/ppc4/motiontests/nested.p b/ppc4/motiontests/nested.p
new file mode 100644
index 0000000..6011456
--- /dev/null
+++ b/ppc4/motiontests/nested.p
@@ -0,0 +1,78 @@
+var x, y, z: integer;
+begin
+  x:=0;
+  while x < 10 do
+    y := 1;
+    while y<2 do
+      z:= 2;
+      y := y + 1
+    end;
+    x:=x+y
+  end;
+  print_num(x); newline()
+end.
+
+(*<<
+ 10
+>>*)
+
+(*[[
+MODULE Main 0 0
+IMPORT Lib 0
+ENDHDR
+
+FUNC MAIN 0
+! x:=0;
+CONST 0
+STGW _x
+! while x < 10 do
+LDGW _x
+CONST 10
+JGEQ L3
+LABEL L1
+! y := 1;
+CONST 1
+STGW _y
+! while y<2 do
+LDGW _y
+CONST 2
+JGEQ L6
+! z:= 2;
+CONST 2
+STGW _z
+LABEL L4
+! y := y + 1
+LDGW _y
+CONST 1
+PLUS
+STGW _y
+LDGW _y
+CONST 2
+JLT L4
+LABEL L6
+! x:=x+y
+LDGW _x
+LDGW _y
+PLUS
+STGW _x
```

```
+LDGW _x
+CONST 10
+JLT L1
+LABEL L3
+! print_num(x); newline()
+LDGW _x
+CONST 0
+GLOBAL lib.print_num
+PCALL 1
+CONST 0
+GLOBAL lib.newline
+PCALL 0
+RETURN
+END
+
+GLOVAR _x 4
+GLOVAR _y 4
+GLOVAR _z 4
+GLOVAR __label 4
+! End
+]]*)
```

--END OF LOOP-INVARIANT CODE MOTION TESTS--

```
diff --git a/ppc4/parser.mly b/ppc4/parser.mly
index dd48264..237a539 100644
--- a/ppc4/parser.mly
+++ b/ppc4/parser.mly
@@ -22,7 +22,7 @@ open Tree
 %token                 ARRAY BEGIN CONST DO ELSE END IF OF
 %token                 PROC RECORD RETURN THEN TO TYPE
 %token                 VAR WHILE NOT POINTER NIL
-%token                 REPEAT UNTIL FOR ELSIF CASE
+%token                 REPEAT UNTIL FOR ELSIF CASE GOTO LABEL

 /* operator priorities */
 %left                  RELOP EQUAL
@@ -43,7 +43,15 @@ program :
     block DOT                          { Prog ($1, ref []) } ;

 block :
-    decl_list BEGIN stmts END          { make_block ($1, $3) } ;
+    labels decl_list BEGIN stmts END   { make_block ($1, $2, $4) } ;
+
+labels:
+    /* empty */                        { LabelDecl([])}
+  | LABEL COLON label_list SEMI        { LabelDecl($3) };
+
+label_list:
+    NUMBER                             { if ($1 >= 1 && $1 < 10000) then [$1] else failwith
"label out of range" }
+  | NUMBER COMMA label_list            { if ($1 >= 1 && $1 < 10000) then $1 :: $3 else
failwith "label out of range" };

 decl_list :
    /* empty */                        { [] }
@@ -125,7 +133,9 @@ stmt1 :
   | FOR name ASSIGN expr TO expr DO stmts END
                                        { let v = make_expr (Variable $2) in
                                          ForStmt (v, $4, $6, $8) }
-  | CASE expr OF arms else_part END    { CaseStmt ($2, $4, $5) } ;
+  | CASE expr OF arms else_part END    { CaseStmt ($2, $4, $5) }
+  | NUMBER COLON stmt                  { LabelStmt ($1, $3) }
+  | GOTO NUMBER                        { GotoStmt ($2) };
```

```
 elses :
     /* empty */                         { make_stmt (Skip, 0) }
diff --git a/ppc4/tree.ml b/ppc4/tree.ml
index ffeaa71..4919b65 100644
--- a/ppc4/tree.ml
+++ b/ppc4/tree.ml
@@ -14,7 +14,9 @@ type name =
 (* abstract syntax *)
 type program = Prog of block * def list ref

-and block = Block of decl list * stmt * int ref * int ref
+and block = Block of label * decl list * stmt * int ref * int ref
+
+and label = LabelDecl of int list

 and decl =
     ConstDecl of ident * expr
@@ -40,6 +42,8 @@ and stmt_guts =
   | RepeatStmt of stmt * expr
   | ForStmt of expr * expr * expr * stmt
   | CaseStmt of expr * (expr * stmt) list * stmt
+  | GotoStmt of int
+  | LabelStmt of int * stmt

 and expr =
   { e_guts: expr_guts;
@@ -69,7 +73,7 @@ let make_expr e =
   { e_guts = e; e_type = voidtype; e_value = None }

 (* |make_stmt| -- construct a stmt node *)
-let make_stmt (s, n) = { s_guts = s; s_line = n }
+let make_stmt (s, n) = { s_guts = s; s_line = n}

 (* |make_name| -- contruct a name node with dummy annotations *)
 let make_name (x, n) = { x_name = x; x_line = n; x_def = None }
@@ -86,7 +90,7 @@ let get_def x =
     | None -> failwith (sprintf "missing def of $" [fId x.x_name])

 (* |make_block| -- construct a block node with dummy annotations *)
-let make_block (decls, stmts) = Block (decls, stmts, ref 0, ref 0)
+let make_block (labels, decls, stmts) = Block (labels, decls, stmts, ref 0, ref 0)


 (* Grinder *)
@@ -101,7 +105,7 @@ let fList f =

 let fName x = fId x.x_name

-let rec fBlock (Block (decls, stmts, _, _)) =
+let rec fBlock (Block (labels, decls, stmts, _, _)) =
   match decls with
       [] -> fMeta "(BLOCK $)" [fStmt stmts]
     | _ -> fMeta "(BLOCK (DECLS$) $)" [fTail(fDecl) decls; fStmt stmts]
@@ -151,6 +155,7 @@ and fStmt s =
     | CaseStmt (sel, arms, deflt) ->
         let fArm (lab, body) = fMeta "($ $)" [fExpr lab; fStmt body] in
        fMeta "(CASE $ $ $)" [fExpr sel; fList(fArm) arms; fStmt deflt]
+    | _ -> fStr ""

 and fExpr e =
   match e.e_guts with
diff --git a/ppc4/tree.mli b/ppc4/tree.mli
index bf78de3..15cef02 100644
--- a/ppc4/tree.mli
+++ b/ppc4/tree.mli
```

```
@@ -29,7 +29,9 @@ val get_def : name -> def
 (* abstract syntax *)
 type program = Prog of block * def list ref

-and block = Block of decl list * stmt * int ref * int ref
+and block = Block of label * decl list * stmt * int ref * int ref
+
+and label = LabelDecl of int list

 and decl =
     ConstDecl of ident * expr
@@ -55,6 +57,8 @@ and stmt_guts =
   | RepeatStmt of stmt * expr
   | ForStmt of expr * expr * expr * stmt
   | CaseStmt of expr * (expr * stmt) list * stmt
+  | GotoStmt of int
+  | LabelStmt of int * stmt

 and expr =
   { e_guts: expr_guts;
@@ -92,7 +96,7 @@ val make_stmt : stmt_guts * int -> stmt
 val make_name : ident * int -> name

 (* |make_block| -- construct a block node with dummy annotations *)
-val make_block : decl list * stmt -> block
+val make_block : label * decl list * stmt -> block

 (* |print_tree| -- pretty-print a tree *)
 val print_tree : out_channel -> string -> program -> unit
```