Outline

- An Overview of GIMPLE
- Using GIMPLE API in GCC-4.6.0
- Adding a GIMPLE Pass to GCC
- An Internal View of RTL
- Manipulating RTL IR
Part 1

An Overview of GIMPLE

GIMPLE: A Recap

- Language independent three address code representation
  - Computation represented as a sequence of basic operations
  - Temporaries introduced to hold intermediate values
- Control construct explicated into conditional and unconditional jumps
Motivation Behind GIMPLE

- Previously, the only common IR was RTL (Register Transfer Language)

- Drawbacks of RTL for performing high-level optimizations
  - Low-level IR, more suitable for machine dependent optimizations (e.g., peephole optimization)
  - High level information is difficult to extract from RTL (e.g. array references, data types etc.)
  - Introduces stack too soon, even if later optimizations do not require it

Why Not Abstract Syntax Trees for Optimization?

- ASTs contain detailed function information but are not suitable for optimization because
  - Lack of a common representation across languages
    - No single AST shared by all front-ends
    - So each language would have to have a different implementation of the same optimizations
    - Difficult to maintain and upgrade so many optimization frameworks
  - Structural Complexity
    - Lots of complexity due to the syntactic constructs of each language
    - Hierarchical structure and not linear structure
    - Control flow explication is required
Need for a New IR

- Earlier versions of GCC would build up trees for a single statement, and then lower them to RTL before moving on to the next statement.
- For higher level optimizations, entire function needs to be represented in trees in a language-independent way.
- Result of this effort - GENERIC and GIMPLE

What is GENERIC?

What?
- Language independent IR for a complete function in the form of trees
- Obtained by removing language specific constructs from ASTs
- All tree codes defined in $(SOURCE)/gcc/tree.def

Why?
- Each language frontend can have its own AST
- Once parsing is complete they must emit GENERIC
What is GIMPLE?

- GIMPLE is influenced by SIMPLE IR of McCat compiler
- But GIMPLE is not same as SIMPLE (GIMPLE supports GOTO)
- It is a simplified subset of GENERIC
  - 3 address representation
  - Control flow lowering
  - Cleanups and simplification, restricted grammar
- Benefit: Optimizations become easier

GIMPLE Goals

The Goals of GIMPLE are

- Lower control flow
  - Sequenced statements + conditional and unconditional jumps
- Simplify expressions
  - Typically one operator and at most two operands
- Simplify scope
  - Move local scope to block begin, including temporaries
Tuple Based GIMPLE Representation

- Earlier implementation of GIMPLE used trees as internal data structure.
- Tree data structure was much more general than was required for three address statements.
- Now a three address statement is implemented as a tuple.
- These tuples contain the following information:
  - Type of the statement
  - Result
  - Operator
  - Operands
- The result and operands are still represented using trees.

Observing Internal Form of GIMPLE

test.c.004t.gimple with compilation option
-fdump-tree-all

\[
\begin{align*}
x &= 10; \\
y &= 5; \\
D.1954 &= x * y; \\
a.0 &= a; \\
x &= D.1954 + a.0; \\
a.1 &= a; \\
D.1957 &= a.1 * x; \\
y &= y - D.1957; \\
\end{align*}
\]

gimple_assign <integer_cst, x, 10, NULL>
gimple_assign <integer_cst, y, 5, NULL>
gimple_assign <mult_expr, D.1954, x, ... <var_decl, a.1, a, NULL>
gimple_assign <mult_expr, D.1957, a.1, x>gimple_assign <minus_expr, y, y, D.1957>
Observing Internal Form of GIMPLE

if (a < c)
    goto <D.1953>;
else
    goto <D.1954>;
<D.1953>:
    a = b + c;
    goto <D.1955>;
<D.1954>:
    a = b - c;
<D.1955>:

gimple_cond <lt_expr, a, c, <D.1953>, <D.1954>>
gimple_label <<D.1953>>
gimple_assign <plus_expr, a, b, c>
gimple_goto <<D.1955>>
gimple_label <<D.1954>>
gimple_assign <minus_expr, a, b, c>
gimple_label <<D.1955>>

Using GIMPLE API in GCC-4.6.0
Iterating Over GIMPLE Statements

• A basic block contains a doubly linked-list of GIMPLE statements
• The statements are represented as GIMPLE tuples, and the operands are represented by tree data structure
• Processing of statements can be done through iterators

```c
basic_block bb;
gimple_stmt_iterator gsi;

FOR_EACH_BB (bb) {
    for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi);
         gsi_next (&gsi))
        analyze_statement (gsi_stmt (gsi));
}
```
Iterating Over GIMPLE Statements

- A basic block contains a doubly linked-list of GIMPLE statements
- The statements are represented as GIMPLE tuples, and the operands are represented by tree data structure
- Processing of statements can be done through *iterators*

```c
basic_block bb;
gimple_stmt_iterator gsi;

FOR_EACH_BB (bb) {
    for (gsi = gsi_start_bb (bb); !gsi_end_p (gsi);
        gsi_next (&gsi))
        analyze_statement (gsi_stmt (gsi));
}
```

**Notes**

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Iterating Over GIMPLE Statements

- A basic block contains a doubly linked-list of GIMPLE statements
- The statements are represented as GIMPLE tuples, and the operands are represented by tree data structure
- Processing of statements can be done through **iterators**

```c
basic_block bb;
gimple_stmt_iterator gsi;

FOR_EACH_BB (bb)
{
    gsi = gsi_start_bb (bb);  // Get the first statement of bb
    gsi_end_p (gsi);
    gsi_next (&gsi);
    analyze_statement (gsi_stmt (gsi));
}
```

Notes
Iterating Over GIMPLE Statements

- A basic block contains a doubly linked-list of GIMPLE statements
- The statements are represented as GIMPLE tuples, and the operands are represented by tree data structure
- Processing of statements can be done through iterators

```c
basic_block bb;
gimple_stmt_iterator gsi;

FOR_EACH_BB (bb)
{
    for (gsi = gsi_start_bb (bb); !gsi_end_p (gsi);
         gsi_next (&gsi))
        analyze_statement (gsi_stmt (gsi));
}
```

Return the current statement
Other Useful APIs for Manipulating GIMPLE

Extracting parts of GIMPLE statements:

- `gimple_assign_lhs`: left hand side
- `gimple_assign_rhs1`: left operand of the right hand side
- `gimple_assign_rhs2`: right operand of the right hand side
- `gimple_assign_rhs_code`: operator on the right hand side

A complete list can be found in the file `gimple.h`

Part 3

Adding a GIMPLE Pass to GCC
Adding a GIMPLE Intraprocedural Pass in GCC-4.6.0 (1)

Add the following `gimple_opt_pass` struct instance to the file

```c
struct gimple_opt_pass pass_intra_gimple_manipulation =
{
    GIMPLE_PASS,     /* optimization pass type */
    "gm",            /* name of the pass*/
    gate_gimple_manipulation, /* gate. */
    intra_gimple_manipulation, /* execute (driver function) */
    NULL,            /* sub passes to be run */
    NULL,            /* next pass to run */
    0,               /* static pass number */
    0,               /* timevar_id */
    0,               /* properties required */
    0,               /* properties provided */
    0,               /* properties destroyed */
    0,               /* todo_flags start */
    0,               /* todo_flags end */
};
```

1. Write the driver function in file `new-pass.c`

2. Declare your pass in file `tree-pass.h`:
   ```c
   extern struct gimple_opt_pass
   pass_intra_gimple_manipulation;
   ```

3. Add your pass to the intraprocedural pass list in
   ```c
   init_optimization_passes()
   ```
   ```c
   ... NEXT_PASS (pass_build_cfg);
   NEXT_PASS (pass_intra_gimple_manipulation);
   ... ```
4. In \$SOURCE/gcc/Makefile.in, add new-pass.o to the list of language independent object files. Also, make specific changes to compile new-pass.o from new-pass.c

5. Configure and build gcc (For simplicity, we will make cc1 only)

6. Debug cc1 using ddd/gdb if need arises (For debugging cc1 from within gcc, see: http://gcc.gnu.org/ml/gcc/2004-03/msg01195.html)

```c
struct register_pass_info dynamic_pass_info = {
    &(pass_intra_gimple_manipulation.pass),
    /* Address of new pass, here, the struct opt_pass field of simple_ipa_opt_pass defined above */
    "cfg",
    /* Name of the reference pass (string in the pass structure specification) for hooking up the new pass. */
    0,
    /* Insert the pass at the specified instance number of the reference pass. Do it for every instance if it is 0. */
    PASS_POS_INSERT_AFTER
};
```
Registering Callback for Our Pass for a Dynamic Plugins

```c
int plugin_init(struct plugin_name_args *plugin_info,
               struct plugin_gcc_version *version)
{
    /* Plugins are activated using this callback */

    register_callback ( plugin_info->base_name, /* char *name: Plugin name, could be any name. */
                       plugin_info->base_name gives this filename */
                       PLUGIN_PASS_MANAGER_SETUP, /* int event: The event code. Here, setting up a new pass */
                       NULL, /* The function that handles the event */
                       &dynamic_pass_info); /* plugin specific data */

    return 0;
}
```

Makefile for Creating and Using a Dynamic Plugin

```bash
CC = $(INSTALL_D)/bin/gcc
PLUGIN_SOURCES = new-pass.c
PLUGIN_OBJECTS = $(patsubst %.c,%.o,$(PLUGIN_SOURCES ))
GCCPLUGINS_DIR ... -fPIC -O2
INCLUDE = -I/plugin/include

%.o : %.c
$<
$CC $(CFLAGS) $(INCLUDE) -c $<
new-pass.so: $(PLUGIN_OBJECTS)
    $<
    $CC $(CFLAGS) $(INCLUDE) -shared $^ -o $@

test_plugin: test.c
    $<
    $CC -fplugin=./new-pass.so $^ -o $@ -fdump-tree-all
```
An Intrprocedural Analysis for Discovering Pointer Usage

Calculate the number of pointer statements in GIMPLE (i.e. result or an operand is a pointer variable)

```
int *p, *q;
void callme (int);
int main (){ int a, b;
p = &b;callme (a);
return 0;
}
void callme (int a){ int * p,0;
int a.1;
p.0 = p;
a.1 = MEM[(int *)p.0 + 12B];
a = a.1;
q = &a;
}
```
static unsigned int
intra_gimple_manipulation (void)
{
    basic_block bb;
gimple_stmt_iterator gsi;

    initialize_var_count ();
    FOR_EACH_BB_FN (bb, cfun)
    {
        for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi);
            gsi_next (&gsi))
            analyze_gimple_stmt (gsi_stmt (gsi));
    }
p
    print_var_count ();
    return 0;
}
static unsigned int
intra_gimple_manipulation (void)
{
    basic_block bb;
gimple_stmt_iterator gsi;

    initialize_var_count ();
FOR_EACH_BB_FN (bb, cfun)
{
    for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi);
        gsi_next (&gsi))
        analyze_gimple_stmt (gsi_stmt (gsi));
    }
print_var_count ();
return 0;
}
static void analyze_gimple_stmt (gimple stmt) {
    if (is_gimple_assign (stmt)) {
        tree lhsop = gimple_assign_lhs (stmt);
        tree rhsop1 = gimple_assign_rhs1 (stmt);
        tree rhsop2 = gimple_assign_rhs2 (stmt);
        /* Check if either LHS, RHS1 or RHS2 operands can be pointers. */
        if ((lhsop && is_pointer_var (lhsop)) || (rhsop1 && is_pointer_var (rhsop1)) || (rhsop2 && is_pointer_var (rhsop2))) {
            if (dump_file)
                fprintf (dump_file, "Pointer Statement : ");
            print_gimple_stmt (dump_file, stmt, 0, 0);
            num_ptr_stmts++;
        }
    }
}
static void analyze_gimple_stmt (gimple stmt)
{
    if (is_gimple_assign (stmt))
    {
        tree lhsop = gimple_assign_lhs (stmt);
        tree rhsop1 = gimple_assign_rhs1 (stmt);
        tree rhsop2 = gimple_assign_rhs2 (stmt);
        /* Check if either LHS, RHS1 or RHS2 operands
         * can be pointers. */
        if ((lhsop && is_pointer_var (lhsop)) ||
            (rhsop1 && is_pointer_var (rhsop1)) ||
            (rhsop2 && is_pointer_var (rhsop2)))
        {
            if (dump_file)
                fprintf (dump_file, "Pointer Statement :");
            print_gimple_stmt (dump_file, stmt, 0, 0);
            num_ptr_stmts++;
        }
    }
}

Returns first operand of RHS

Returns second operand of RHS
### Analysing GIMPLE Statement

```c
static void analyze_gimple_stmt (gimple stmt)
{
    if (is_gimple_assign (stmt))
    {
        tree lhsop = gimple_assign_lhs (stmt);
        tree rhsop1 = gimple_assign_rhs1 (stmt);
        tree rhsop2 = gimple_assign_rhs2 (stmt);
        /* Check if either LHS, RHS1 or RHS2 operands
         * can be pointers. */
        if ((lhsop && is_pointer_var (lhsop)) ||
            (rhsop1 && is_pointer_var (rhsop1)) ||
            (rhsop2 && is_pointer_var (rhsop2)))
        {
            if (dump_file)
                fprintf (dump_file, "Pointer Statement :");
            print_gimple_stmt (dump_file, stmt, 0, 0);
            num_ptr_stmts++;
        }
    }
}
```

### Discovering Pointers

```c
static bool is_pointer_var (tree var)
{
    return is_pointer_type (TREE_TYPE (var));
}

static bool is_pointer_type (tree type)
{
    if (POINTER_TYPE_P (type))
        return true;
    if (TREE_CODE (type) == ARRAY_TYPE)
        return is_pointer_var (TREE_TYPE (type));
    /* Return true if it is an aggregate type. */
    return AGGREGATE_TYPE_P (type);
}
```
static bool is_pointer_var (tree var)
{
    return is_pointer_type (TREE_TYPE (var));
}

static bool is_pointer_type (tree type)
{
    if (POINTER_TYPE_P (type))
        return true;
    if (TREE_CODE (type) == ARRAY_TYPE)
        return is_pointer_var (TREE_TYPE (type));
    /* Return true if it is an aggregate type. */
    return AGGREGATE_TYPE_P (type);
}
Intraprocedural Analysis Results

main ()
{ ...
    p = &b;
callme (a);
D.1965 = 0;
return D.1965;
}
callme (int a)
{ ...
    p.0 = p;
a.1 = MEM[(int *)p.0 + 12B];
a = a.1;
q = &a;
}

Information collected by intraprocedural Analysis pass
• For main: 1
• For callme: 2

Why is the pointer in the red statement being missed?

Discovering Local Variables

static void gather_local_variables ()
{
    tree list = cfun->local_decls;
    if (!dump_file)
        return;
    fprintf(dump_file, "Local variables : ");
    while (list)
    {
        if (!DECL_ARTIFICIAL (list) && dump_file)
        {
            fprintf(dump_file, get_name(list));
            fprintf(dump_file, "\n");
        }
        list = TREE_CHAIN (list);
    }
}
Discovering Global Variables

static void gather_global_variables ()
{
    struct varpool_node *node;

    if (!dump_file)
        return;

    fprintf(dump_file, "Global variables: ");
    for (node = varpool_nodes; node; node = node->next)
    {
        tree var = node->decl;
        if (!DECL_ARTIFICIAL(var))
        {
            fprintf(dump_file, get_name(var));
            fprintf(dump_file, " ");
        }
    }
}

Adding Interprocedural Pass as a Static Plugin

1. Add the following gimple_opt_pass struct instance to the file

struct simple_ipa_opt_pass pass_inter_gimple_manipulation =
{
    SIMPLE_IPA_PASS, /* optimization pass type */
    "gp", /* name of the pass*/
    gate_gimple_manipulation, /* gate. */
    inter_gimple_manipulation, /* execute (driver function) */
    NULL, /* sub passes to be run */
    NULL, /* next pass to run */
    0, /* static pass number */
    0, /* timevar_id */
    0, /* properties required */
    0, /* properties provided */
    0, /* properties destroyed */
    0, /* todo_flags start */
    0 /* todo_flags end */
};
2. Write the driver function in file `new-pass.c`
3. Declare your pass in file `tree-pass.h`:
   ```c
   extern struct simple_ipa_opt_pass
       pass_inter_gimple_manipulation;
   ```
4. Add your pass to the interprocedural pass list in `init_optimization_passes()`
   ```c
   ...  
   p = &all_regular_ipa_passes;  
   NEXT_PASS (pass_ipa_whole_program_visibility);  
   NEXT_PASS (pass_inter_gimple_manipulation);  
   NEXT_PASS (pass_ipa_cp);  
   ...  
   ```
5. In `$SOURCE/gcc/Makefile.in`, add `new-pass.o` to the list of language independent object files. Also, make specific changes to compile `new-pass.o` from `new-pass.c`
6. Configure and build gcc for `cc1`
7. Debug using `ddd/gdb` if a need arises
   (For debugging `cc1` from within `gcc`, see: http://gcc.gnu.org/ml/gcc/2004-03/msg01195.html)
static unsigned int
ter_gimple_manipulation (void) {
    struct cgraph_node *node;
    basic_block bb;
    gimple_stmt_iterator gsi;
    initialize_var_count ();
    for (node = cgraph_nodes; node; node=node->next) {
        /*Nodes without a body, and clone nodes are not interesting. */
        if (!gimple_has_body_p (node->decl) || node->clone_of)
            continue;
        push_cfun (DECL_STRUCT_FUNCTION (node->decl));
        FOR_EACH_BB (bb) {
            for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi); gsi_next (&gsi))
                analyze_gimple_stmt (gsi_stmt (gsi));
        }
        pop_cfun ();
    }
    print_var_count ();
    return 0;
}
Discovering Pointer Usage Interprocedurally

```c
static unsigned int inter_gimple_manipulation (void)
{
    struct cg_node *node;
    basic_block bb;
    gimple_stmt_iterator gsi;
    initialize_var_count ();
    for (node = cg_nodes; node; node=node->next) {
        /* Nodes without a body, and clone nodes are not interesting. */
        if (!gimple_has_body_p (node->decl) || node->clone_of)
            continue;
        push_cfun (DECL_STRUCT_FUNCTION (node->decl));
        FOR_EACH_BB (bb) {
            gsi = gsi_start_bb (bb);
            gsi_next (&gsi);
            for (; gsi < gsi_end_bb (bb); !gsi_end_p (gsi))
                analyze_gimple_stmt (gsi_stmt (gsi));
        }
        pop_cfun ();
        print_var_count ();
    }
    return 0;
}
```

Essential Abstractions in GCC
static unsigned int inter_gimple_manipulation (void) {
    struct cgraph_node *node;
    basic_block bb;
    gimple_stmt_iterator gsi;
    initialize_var_count ();
    for (node = cgraph_nodes; node; node = node->next) {
        /* Nodes without a body, and clone nodes are not interesting. */
        if (!gimple_has_body_p (node->decl) || node->clone_of)
            continue;
        push_cfun (DECL_STRUCT_FUNCTION (node->decl));
        FOR_EACH_BB (bb) {
            for (gsi = gsi_start_bb (bb); !gsi_end_p (gsi); gsi_next (&gsi))
                analyze_gimple_stmt (gsi_stmt (gsi));
        }
        pop_cfun ();
    }
    print_var_count ();
    return 0;
}
Interprocedural Results

Number of Pointer Statements = 3

Observation:

- Information can be collected for all the functions in a single pass
- Better scope for optimizations

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay

Part 4
An Overview of RTL
What is RTL?

**RTL = Register Transfer Language**

*Assembly language for an abstract machine with infinite registers*

Why RTL?

A lot of work in the back-end depends on RTL. Like,

- Low level optimizations like loop optimization, loop dependence, common subexpression elimination, etc
- Instruction scheduling
- Register Allocation
- Register Movement
Why RTL?

For tasks such as those, RTL supports many low level features, like,

- Register classes
- Memory addressing modes
- Word sizes and types
- Compare and branch instructions
- Calling Conventions
- Bitfield operations

The Dual Role of RTL

- For specifying machine descriptions
  Machine description constructs:
  - define_insn, define_expand, match_operand
- For representing program during compilation
  IR constructs
  - insn, jump_insn, code_label, note, barrier

This lecture focusses on RTL as an IR
Part 5

An Internal View of RTL

Notes

RTL Objects

- Types of RTL Objects
  - Expressions
  - Integers
  - Wide Integers
  - Strings
  - Vectors

- Internal representation of RTL Expressions
  - Expressions in RTX are represented as trees
  - A pointer to the C data structure for RTL is called rtx
RTX Codes

RTL Expressions are classified into RTX codes:

- Expression codes are names defined in rtl.def
- RTX codes are C enumeration constants
- Expression codes and their meanings are machine-independent
- Extract the code of a RTX with the macro GET_CODE(x)

RTL Classes

RTL expressions are divided into few classes, like:

- RTX_UNARY : NEG, NOT, ABS
- RTX_BINT : INT, INT
- RTX_BINTARITH : MINUS, DIV
- RTX_COMMARITH : PLUS, MULT
- RTX_OBJ : REG, MEM, SYMBOL_REF
- RTX_COMPARE : GE, LT
- RTX_TERNARY : IF_THEN_ELSE
- RTX_INSN : INSN, JUMP_INSN, CALL_INSN
- RTX_EXTRA : SET, USE
**RTX Codes**

The RTX codes are defined in `rtl.def` using cpp macro call `DEF_RTL_EXPR`, like:

- `DEF_RTL_EXPR(INSN, "insn", "iuuBieie", RTX_INSN)
- `DEF_RTL_EXPR_SET, "set", "ee", RTX_EXTRA)
- `DEF_RTL_EXPR_PLUS, "plus", "ee", RTX_COMM_ARITH)
- `DEF_RTL_EXPR_IF_THEN_ELSE, "if_then_else", "eee", RTX_TERNARY)

The operands of the macro are:

- Internal name of the `rtx` used in C source. It's a tag in enumeration `enum rtx_code`
- name of the `rtx` in the external ASCII format
- Format string of the `rtx`, defined in `rtx_format[]`
- Class of the `rtx`

**RTX Formats**

`DEF_RTL_EXPR(INSN, "insn", "iuuBieie", RTX_INSN)`

- `i`: Integer
- `u`: Integer representing a pointer
- `B`: Pointer to basic block
- `e`: Expression
### RTL statements

- RTL statements are instances of type `rtx`
- RTL insns contain embedded links
- Types of RTL insns:
  - `INSN`: Normal non-jumping instruction
  - `JUMP_INSNS`: Conditional and unconditional jumps
  - `CALL_INSNS`: Function calls
  - `CODE_LABEL`: Target label for `JUMP_INSNS`
  - `BARRIER`: End of control flow
  - `NOTE`: Debugging information

### Basic RTL APIs

- `XEXP, XINT, XWINT, XSTR`
  - Example: `XINT(x, 2)` accesses the 2nd operand of `rtx x` as an integer
  - Example: `XEXP(x, 2)` accesses the same operand as an expression
- Any operand can be accessed as any type of RTX object
  - So operand accessor to be chosen based on the format string of the containing expression
- Special macros are available for Vector operands
  - `XVEC(exp, idx)`: Access the vector-pointer which is operand number `idx` in `exp`
  - `XVECLEN(exp, idx)`: Access the length (number of elements) in the vector which is in operand number `idx` in `exp`. This value is an int
  - `XVECEXP(exp, idx, eltnum)`: Access element number “eltnum” in the vector which is in operand number `idx` in `exp`. This value is an RTX
• A function’s code is a doubly linked chain of INSN objects
• Insns are rtxs with special code
• Each insn contains at least 3 extra fields:
  ▶ Unique id of the insn, accessed by INSN_UID(i)
  ▶ PREV_INSN(i) accesses the chain pointer to the insn preceding i
  ▶ NEXT_INSN(i) accesses the chain pointer to the insn succeeding i
• The first insn is accessed by using get_insns()
• The last insn is accessed by using get_last_insn()
Adding an RTL Pass

Similar to adding GIMPLE intraprocedural pass except for the following

- Use the data structure `struct rtl_opt_pass`
- Replace the first field `GIMPLE_PASS` by `RTL_PASS`

Sample Demo Program

Problem statement: Counting the number of SET objects in a basic block by adding a new RTL pass

- Add your new pass after `pass_expand`
- `new_rtl_pass_main` is the main function of the pass
- Iterate through different instructions in the doubly linked list of instructions and for each expression, call `eval_rtx` for that expression which recurse in the expression tree to find the set statements
Sample Demo Program

```c
int new_rtl_pass_main(void)
{
    basic_block bb;
    rtx last, insn, opd1, opd2;
    int bbno, code, type;
    count = 0;
    for (insn=get_insns(), last=get_last_insn(),
        last=NEXT_INSN(last); insn!=last; insn=NEXT_INSN(insn))
    {
        int is_insn;
        is_insn = INSN_P (insn);
        if(flag_dump_new_rtl_pass)
            print_rtl_single(dump_file,insn);
        code = GET_CODE(insn);
        if(code==NOTE){ ... }
        if(is_insn)
            { rtx subexp = XEXP(insn,5);
                eval_rtx(subexp);
            }

    }
}
```

Sample Demo Program

```c
void eval_rtx(rtx exp)
{
    rtx temp;
    int veclen, i,
    int rt_code = GET_CODE(exp);
    switch(rt_code)
    {
        case SET:
            if(flag_dump_new_rtl_pass){
                fprintf(dump_file, "%Set statement %d : %t", count+1);
                print_rtl_single(dump_file,exp);
                count++; break;
            case PARALLEL:
                veclen = XVECLEN(exp, 0);
                for(i = 0; i < veclen; i++)
                    { temp = XVECEXP(exp, 0, i);
                        eval_rtx(temp);
                    }
                    break;
                    default: break;
    }
}