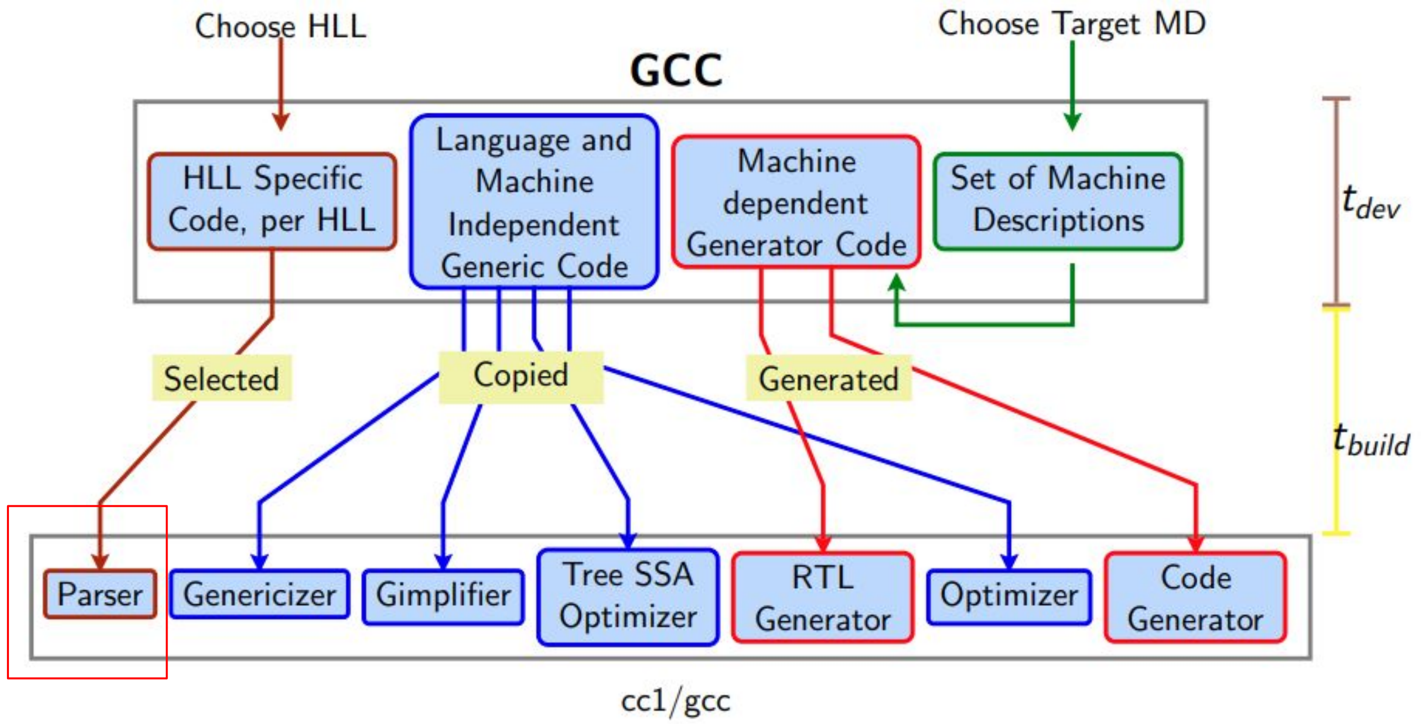
编译技术入门与实战——RISCV-GNU-GCC (从语法树到GIMPLE)

PLCT实验室

sinan@iscas.ac.cn

2021/07/28

回顾



高级语言到语法树AST

Fig1. GCC Framework. ref from IITB

回顾

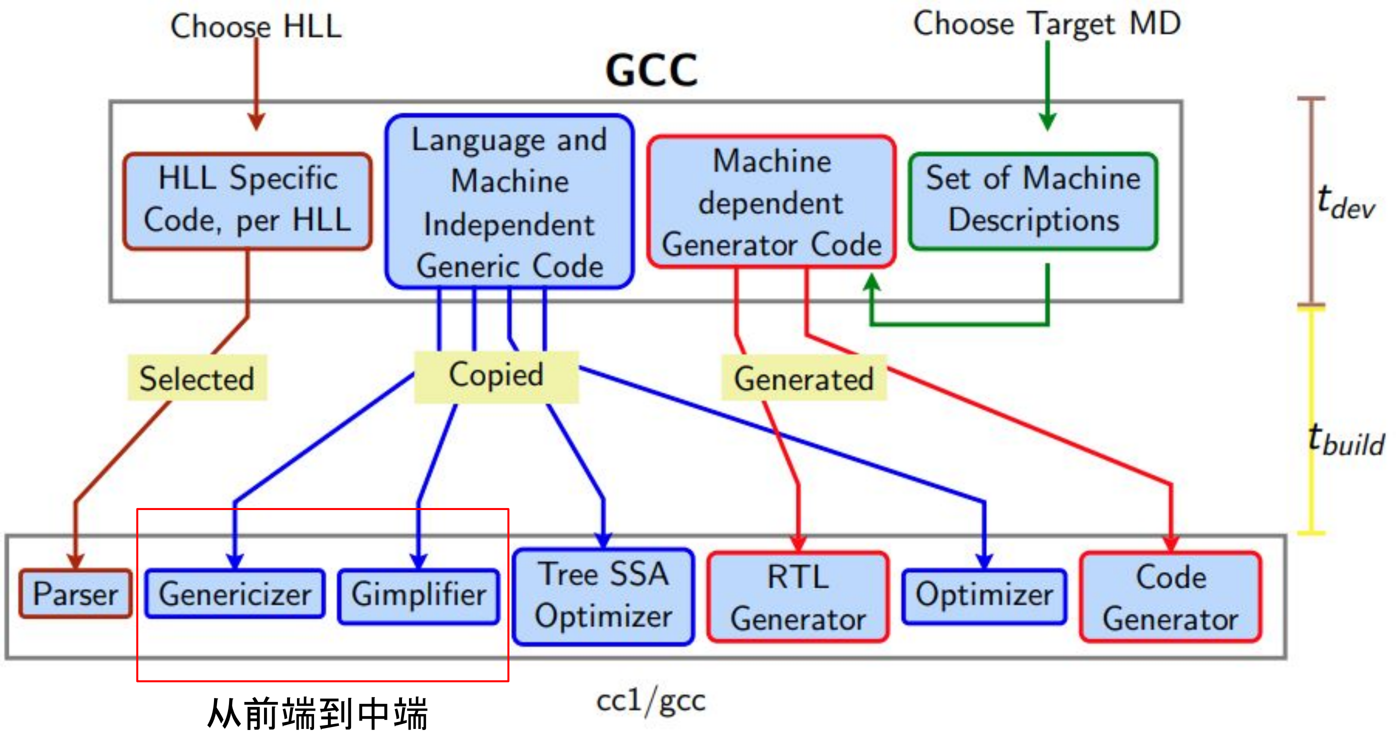
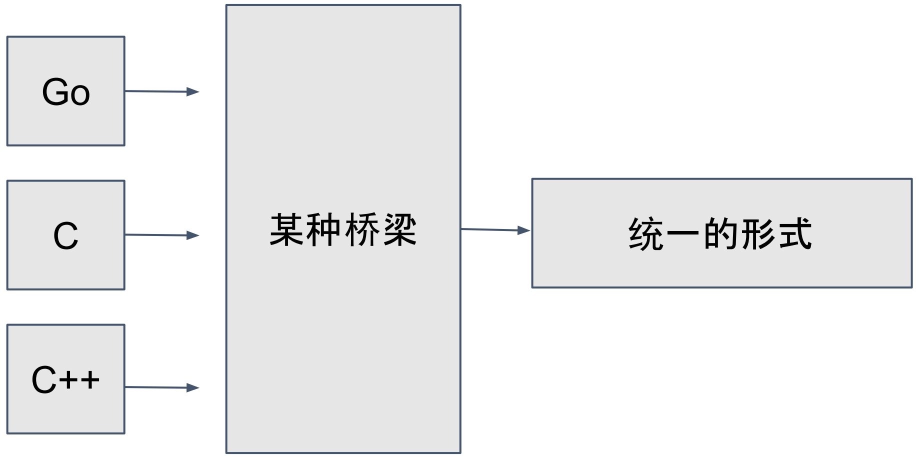


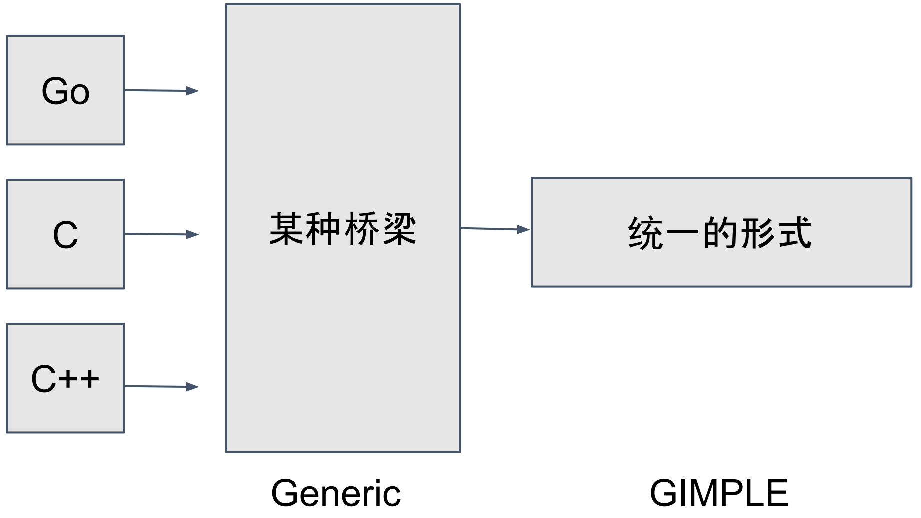
Fig1. GCC Framework. ref from IITB

中间表示（IR, Immediate representation)



高级语言通过GCC前端转换到统一的language independent形式

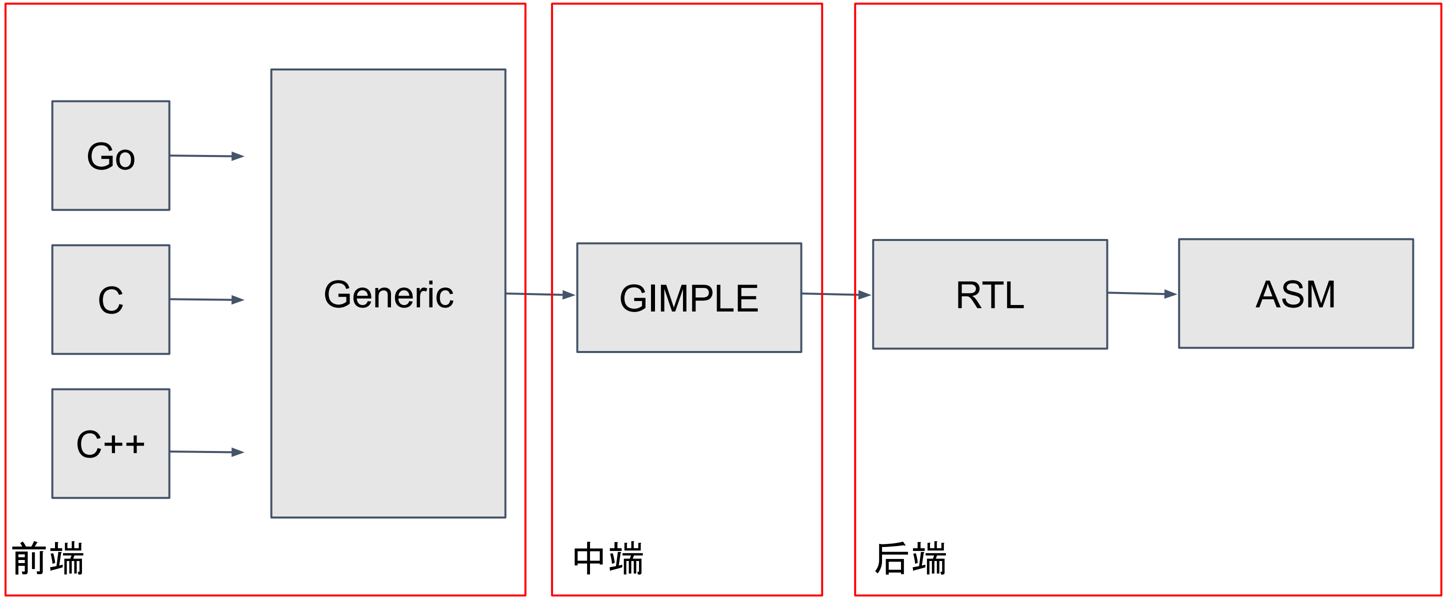
中间表示（IR, Immediate representation)



高级语言通过GCC前端转换到统一的language independent形式

GCC 中主要的IR

- GIMPLE
- Register Transfer Language (RTL)



以中间语言为角度的GCC工作流

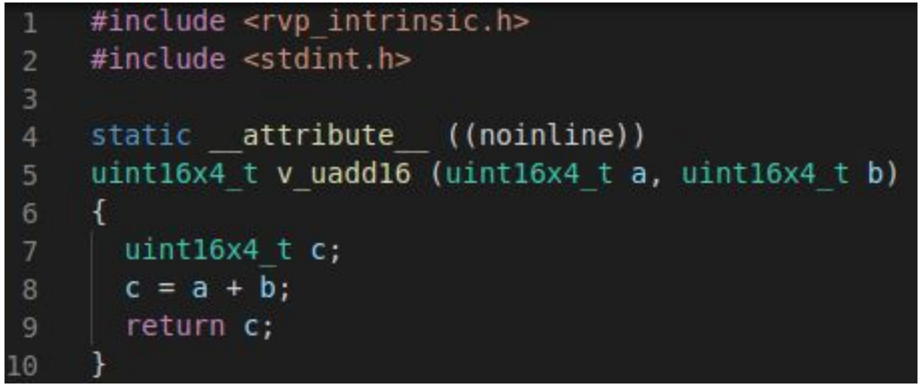
GIMPLE

High GIMPLE, Low GIMPLE,

SSA GIMPLE, CFG GIMPLE

从源码到GIMPLE

➡ source code



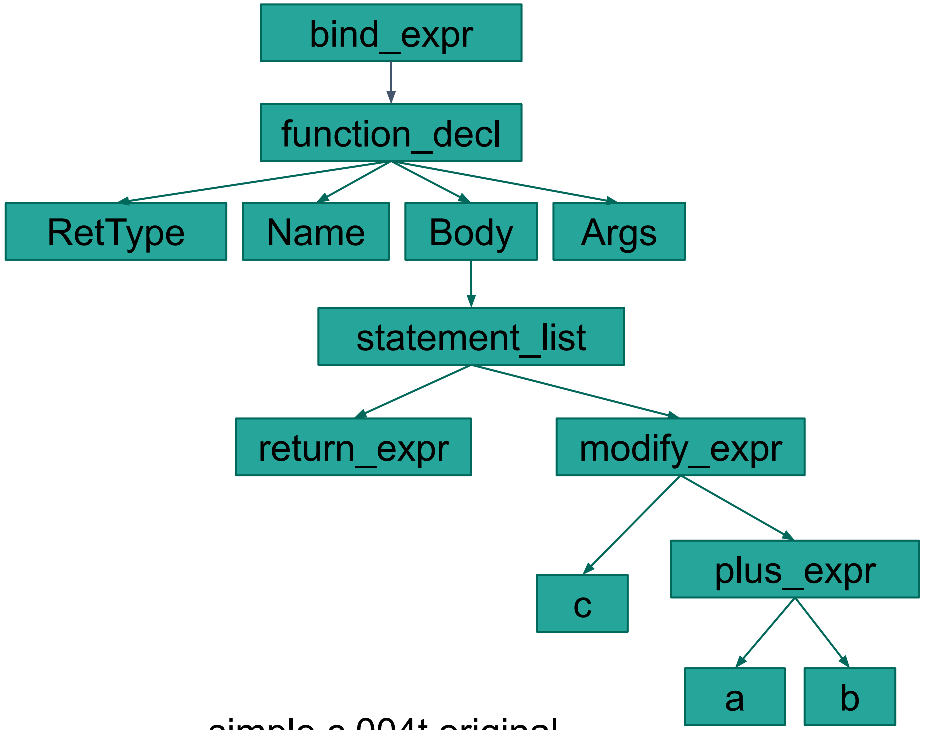
```c
#include <rvp_intrinsic.h>
#include <stdint.h>

static __attribute__ ((noinline))
uint16x4_t v_uadd16 (uint16x4_t a, uint16x4_t b)
{
    uint16x4_t c;
    c = a + b;
    return c;
}
```

sample.c

从源码到GIMPLE

source code

➡ AST



simple.c.004t.original

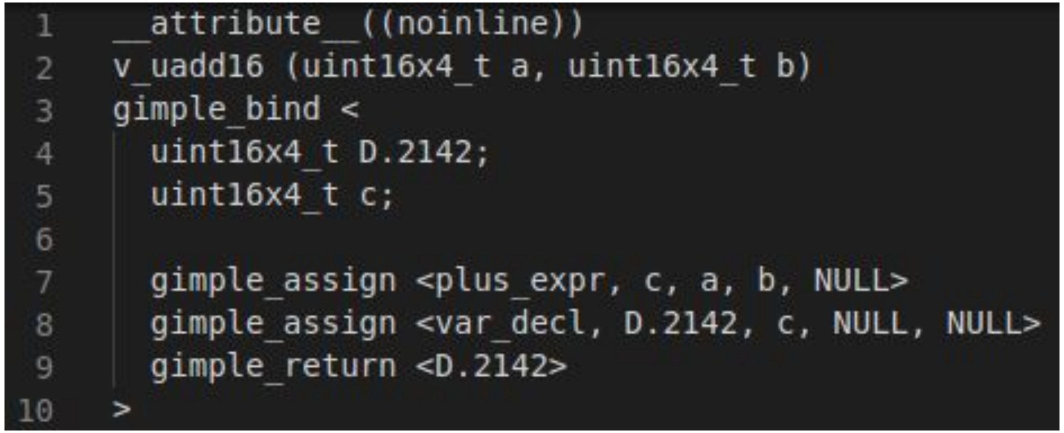
```
gcc -fdump-tree-original-raw -g -S sample.c
```

从源码到GIMPLE

source code

AST

GIMPLE



```
__attribute__((noinline))
v_uadd16 (uint16x4_t a, uint16x4_t b)
gimple_bind <
  uint16x4_t D.2142;
  uint16x4_t c;

  gimple_assign <plus_expr, c, a, b, NULL>
  gimple_assign <var_decl, D.2142, c, NULL, NULL>
  gimple_return <D.2142>
>
```

simple.c.005t.gimple

gcc -fdump-tree-original-raw -g -S sample.c

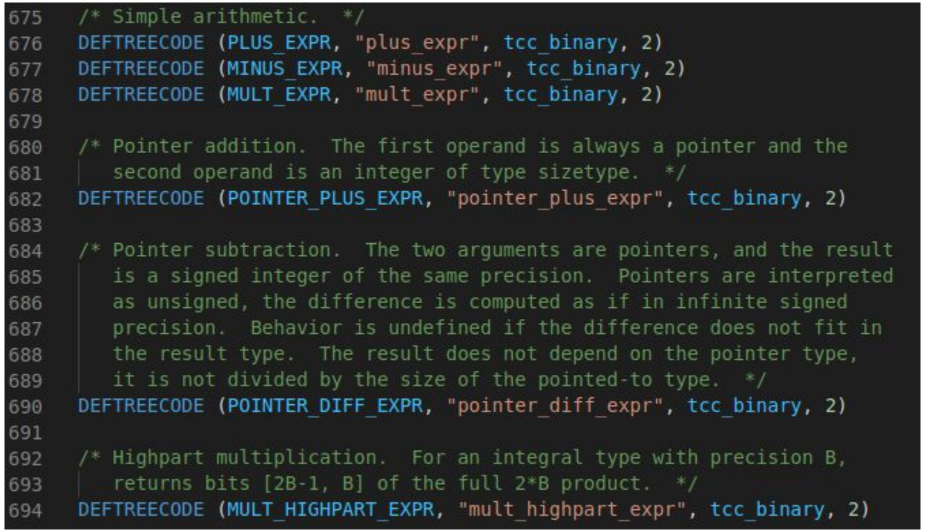
Generic

一种独立语言的表达方式来表达函数 树，通过实现GCC的提供的接口来达到 语言无关的中间表达的生 产(gimplify)。

GCC所有提供的Generic表达节点可以在gcc/tree.def中找到。

Generic

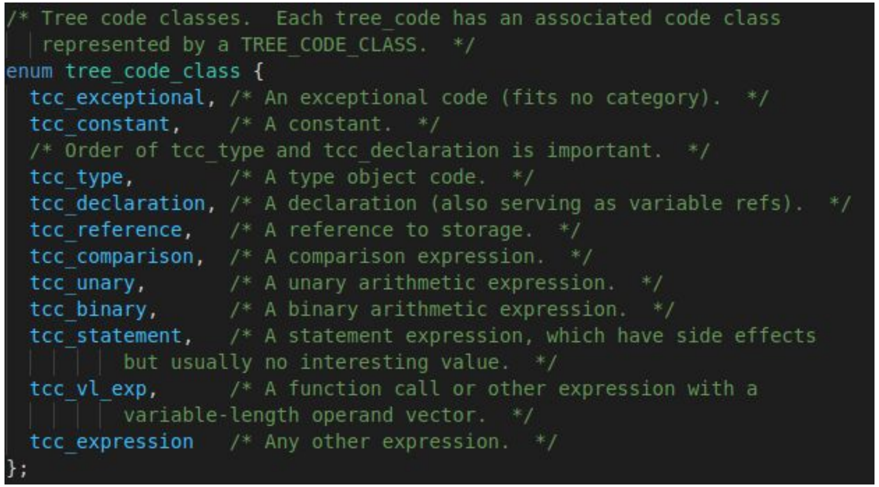
所有节点



```
/* Simple arithmetic.  */
DEFTREECODE (PLUS_EXPR, "plus_expr", tcc_binary, 2)
DEFTREECODE (MINUS_EXPR, "minus_expr", tcc_binary, 2)
DEFTREECODE (MULT_EXPR, "mult_expr", tcc_binary, 2)

/* Pointer addition.  The first operand is always a pointer and the
   second operand is an integer of type sizetype.  */
DEFTREECODE (POINTER_PLUS_EXPR, "pointer_plus_expr", tcc_binary, 2)

/* Pointer subtraction.  The two arguments are pointers, and the result
   is a signed integer of the same precision.  Pointers are interpreted
   as unsigned, the difference is computed as if in infinite signed
   precision.  Behavior is undefined if the difference does not fit in
   the result type.  The result does not depend on the pointer type,
   it is not divided by the size of the pointed-to type.  */
DEFTREECODE (POINTER_DIFF_EXPR, "pointer_diff_expr", tcc_binary, 2)

/* Highpart multiplication.  For an integral type with precision B,
   returns bits [2B-1, B] of the full 2*B product.  */
DEFTREECODE (MULT_HIGHPART_EXPR, "mult_highpart_expr", tcc_binary, 2)
```

gcc/tree.def

节点类型



```
/* Tree code classes.  Each tree_code has an associated code class
   represented by a TREE_CODE_CLASS.  */
enum tree_code_class {
  tcc_exceptional, /* An exceptional code (fits no category).  */
  tcc_constant,    /* A constant.  */
  /* Order of tcc_type and tcc_declaration is important.  */
  tcc_type,        /* A type object code.  */
  tcc_declaration, /* A declaration (also serving as variable refs).  */
  tcc_reference,   /* A reference to storage.  */
  tcc_comparison,  /* A comparison expression.  */
  tcc_unary,       /* A unary arithmetic expression.  */
  tcc_binary,      /* A binary arithmetic expression.  */
  tcc_statement,   /* A statement expression, which have side effects
                      but usually no interesting value.  */
  tcc_vl_exp,      /* A function call or other expression with a
                      variable-length operand vector.  */
  tcc_expression   /* Any other expression.  */
};
```

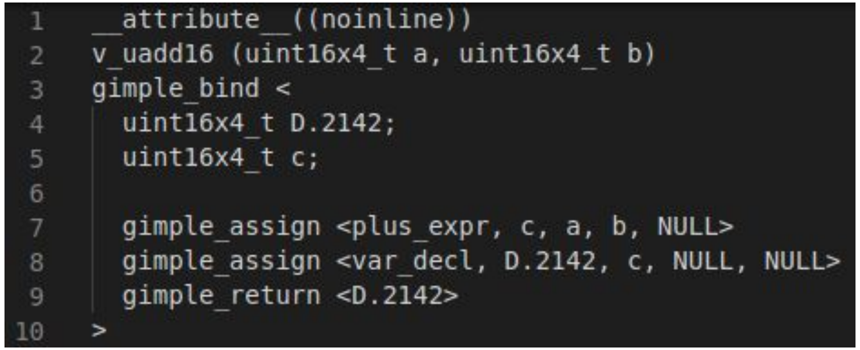
riscv-gcc/gcc/tree-core.h

通过TREE_* 宏对操作节点
TREE_CODE，TREE_CODE_CLASS，TREE_TYPE

GIMPLE

GIMPLE是一种包含最多三个操作数的中 间语言，其独立于编程语言的编译器内部表示。GIMPLE 主要用于大规模目标和语言无关的优化，例如内联、常数折叠。

GIMPLE是Generic的更简化且更多限制的子集，唯一的控制结构是条件跳转和词法范围被删除。这意味着GCC通过GIMPLE，程序会变得更加扁平化（flattened）。



```
__attribute__((noinline))
v_uadd16 (uint16x4_t a, uint16x4_t b)
gimple_bind <
  uint16x4_t D.2142;
  uint16x4_t c;

  gimple_assign <plus_expr, c, a, b, NULL>
  gimple_assign <var_decl, D.2142, c, NULL, NULL>
  gimple_return <D.2142>
>
```

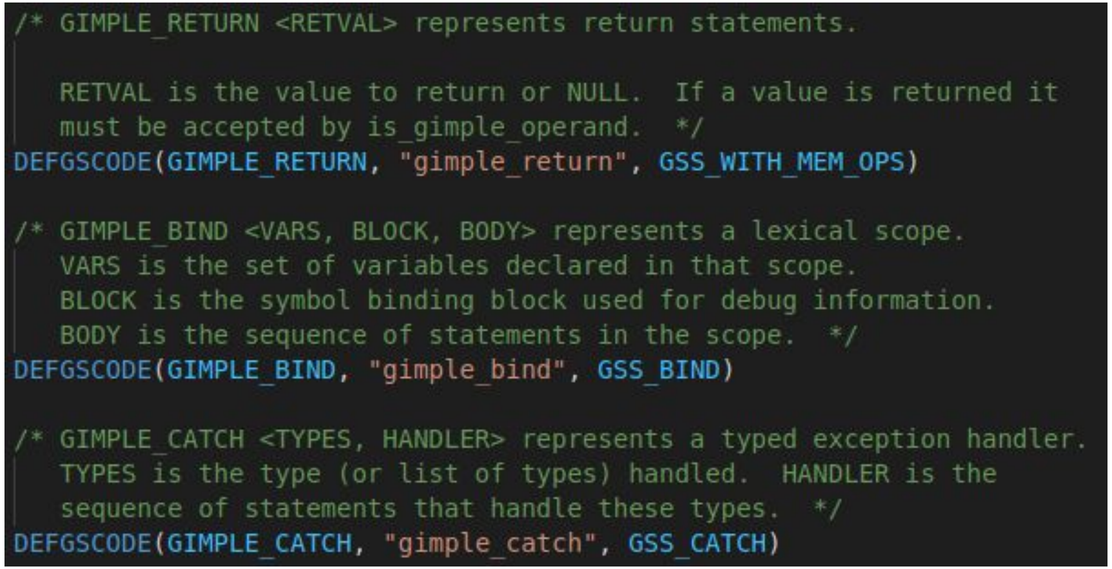
simple.c.005t.gimple

GIMPLE

GCC中端中的多种GIMPLE

- High GIMPLE: 复杂表达式被切分成三地址码，显示表示中间变量
- Low GIMPLE: 控制流结构线性化得,包括嵌套函数、异常处理和循环
- SSA GIMPLE: 以静态单赋值形式重写的GIMPLE

GIMPLE


```
/* GIMPLE_RETURN <RETVAL> represents return statements.

   RETVAL is the value to return or NULL.  If a value is returned it
   must be accepted by is_gimple_operand.  */
DEFGSCODE(GIMPLE_RETURN, "gimple_return", GSS_WITH_MEM_OPS)

/* GIMPLE_BIND <VARS, BLOCK, BODY> represents a lexical scope.
   VARS is the set of variables declared in that scope.
   BLOCK is the symbol binding block used for debug information.
   BODY is the sequence of statements in the scope.  */
DEFGSCODE(GIMPLE_BIND, "gimple_bind", GSS_BIND)

/* GIMPLE_CATCH <TYPES, HANDLER> represents a typed exception handler.
   TYPES is the type (or list of types) handled.  HANDLER is the
   sequence of statements that handle these types.  */
DEFGSCODE(GIMPLE_CATCH, "gimple_catch", GSS_CATCH)
```

gimple指令的介绍 gcc/gimple.def

GIMPLIFY

GIMPLE化，即从语法树或Generic形式（前端）转为GIMPLE中间语言的这个过程。在GIMPLE化的过程中一个复杂的表达式会展开为一系列的GIMPLE指令，从而使程序对于编译器而言更好分析。

GIMPLIFY流程（C语言）

c_parser_declaration_or_fndef()	gcc/c/c-parser.c
finish_function()	gcc/c/c-decl.c
c_genericize()	gcc/c-family/c-gimplify.c
gimplify_function_tree()	gcc/gimplify.c
gimplify_body()	gcc/gimplify.c
gimplify_stmt()	gcc/gimplify.c
gimplify_expr()	gcc/gimplify.c

cc1创建GIMPLE表达式的流程

GIMPLIFY流程（C语言）

```
             c_parser_declaration_or_fndef()              gcc/c/c-parser.c
genericize      finish_function()                         gcc/c/c-decl.c
                   c_genericize()                         gcc/c-family/c-gimplify.c
                      gimplify_function_tree()            gcc/gimplify.c
                        gimplify_body()                   gcc/gimplify.c
                          gimplify_stmt()                 gcc/gimplify.c
                            gimplify_expr()               gcc/gimplify.c
```

cc1创建GIMPLE表达式的流程

GIMPLIFY流程（C语言）

genericize	c_parser_declaration_or_fndef()	gcc/c/c-parser.c
	finish_function()	gcc/c/c-decl.c
	c_genericize()	gcc/c-family/c-gimplify.c
gimplifier	gimplify_function_tree()	gcc/gimplify.c
	gimplify_body()	gcc/gimplify.c
	gimplify_stmt()	gcc/gimplify.c
	gimplify_expr()	gcc/gimplify.c

cc1创建GIMPLE表达式的流程

GIMPLIFY流程（C语言）

```
              c_parser_declaration_or_fndef()           gcc/c/c-parser.c
genericize      finish_function()                       gcc/c/c-decl.c
                  c_genericize()                        gcc/c-family/c-gimplify.c
                    gimplify_function_tree()            gcc/gimplify.c
gimplifier            gimplify_body()                   gcc/gimplify.c
                        gimplify_stmt()                 gcc/gimplify.c
                          gimplify_expr()               gcc/gimplify.c
```

cc1创建GIMPLE表达式的流程

函数体的转换，gimplify_paramaters()进行函数参数的转换

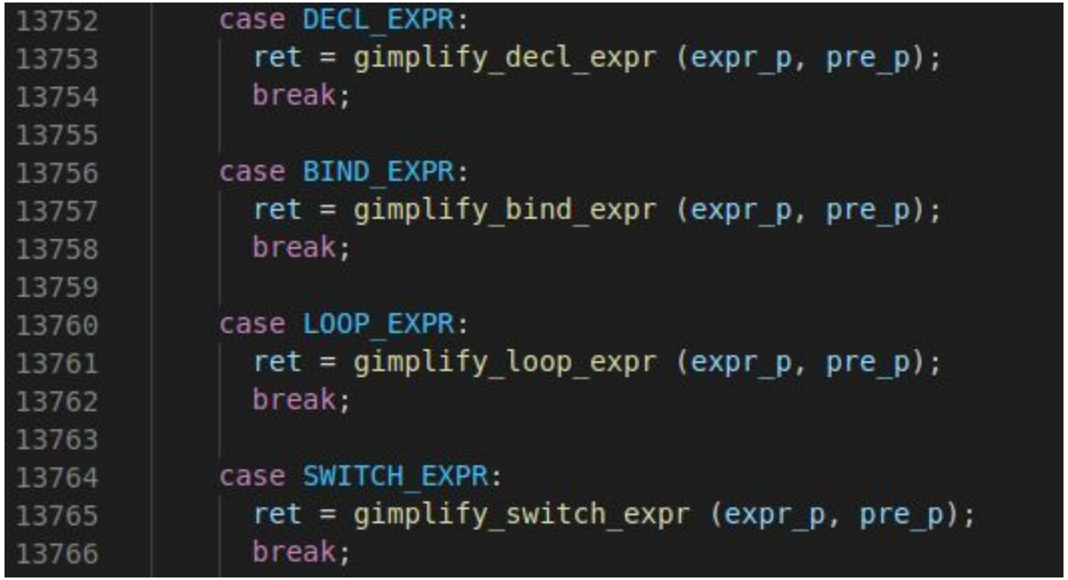
GIMPLIFY流程（C语言）

genericize	c_parser_declaration_or_fndef()	gcc/c/c-parser.c
	finish_function()	gcc/c/c-decl.c
	c_genericize()	gcc/c-family/c-gimplify.c
gimplifier	gimplify_function_tree()	gcc/gimplify.c
	gimplify_body()	gcc/gimplify.c
	gimplify_stmt()	gcc/gimplify.c
	gimplify_expr()	gcc/gimplify.c

cc1创建GIMPLE表达式的流程

1. Generic节点转GIMPLE节点
2. 语言特定的转换（由GCC language hook）

GIMPLIFY流程（C语言）



```
    case DECL_EXPR:
      ret = gimplify_decl_expr (expr_p, pre_p);
      break;

    case BIND_EXPR:
      ret = gimplify_bind_expr (expr_p, pre_p);
      break;

    case LOOP_EXPR:
      ret = gimplify_loop_expr (expr_p, pre_p);
      break;

    case SWITCH_EXPR:
      ret = gimplify_switch_expr (expr_p, pre_p);
      break;
```

gimplify_expr in gcc/gimplify.c

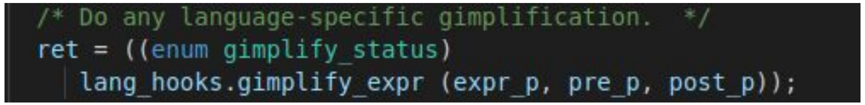
将对应的Generic节点转为GIMPLE节点

GIMPLIFY流程（C语言）

	c_parser_declaration_or_fndef()	gcc/c/c-parser.c
genericize	finish_function()	gcc/c/c-decl.c
	c_genericize()	gcc/c-family/c-gimplify.c
gimplifier	gimplify_function_tree()	gcc/gimplify.c
	gimplify_body()	gcc/gimplify.c
	gimplify_stmt()	gcc/gimplify.c
	gimplify_expr()	gcc/gimplify.c

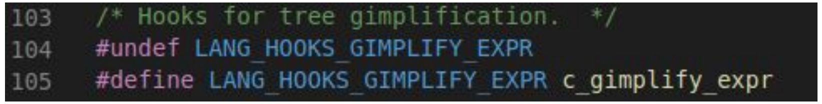
cc1创建GIMPLE表达式的流程

通过函数指针lang_hooks::gimplify_expr调用语言特定的gimpify回调函数



```
/* Do any language-specific gimplification.  */
ret = ((enum gimplify_status)
  lang_hooks.gimplify_expr (expr_p, pre_p, post_p));
```

gcc/gimplify.c

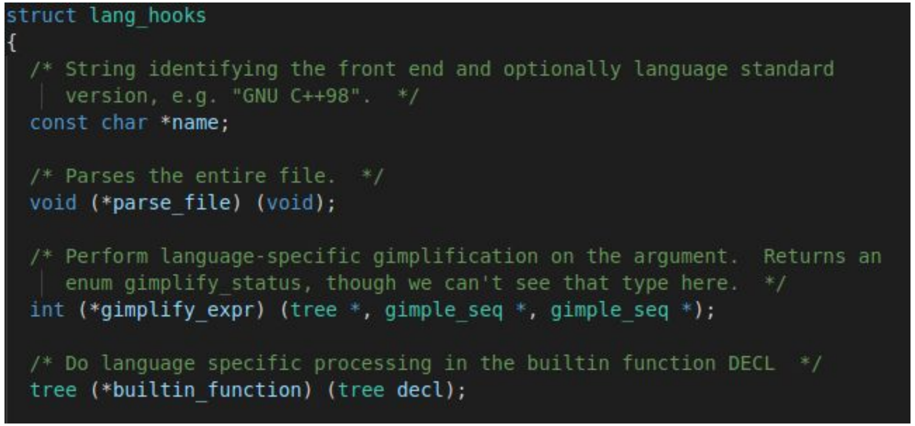


```
/* Hooks for tree gimplification.  */
#undef LANG_HOOKS_GIMPLIFY_EXPR
#define LANG_HOOKS_GIMPLIFY_EXPR c_gimplify_expr
```

gcc/c/c-objc-common.h

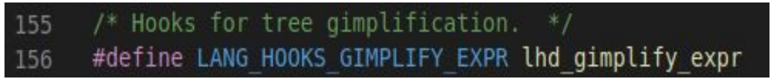
GIMPLIFY接口(language hooks)

GCC接口(language hook)通过重定义宏的方式，添加回调函数以实现特定于语言的行为。



```
struct lang_hooks
{
  /* String identifying the front end and optionally language standard
     version, e.g. "GNU C++98".  */
  const char *name;

  /* Parses the entire file.  */
  void (*parse_file) (void);

  /* Perform language-specific gimplification on the argument.  Returns an
     enum gimplify_status, though we can't see that type here.  */
  int (*gimplify_expr) (tree *, gimple_seq *, gimple_seq *);

  /* Do language specific processing in the builtin function DECL  */
  tree (*builtin_function) (tree decl);
```

gcc/langhooks.h

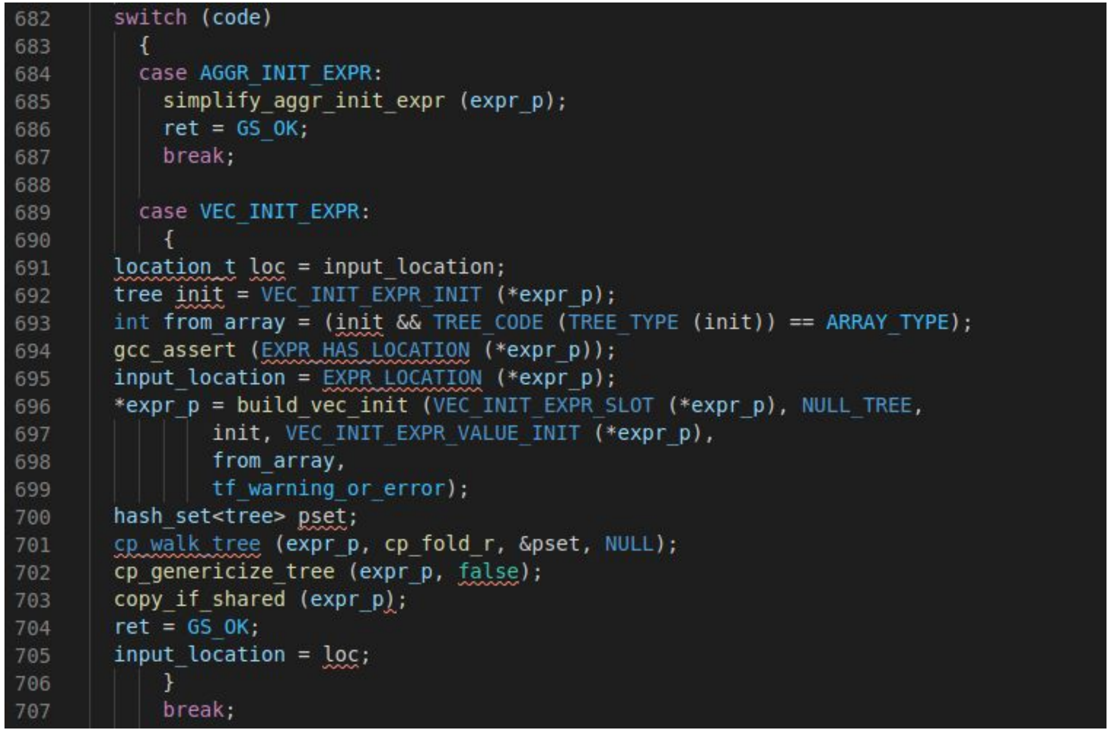


```
/* Hooks for tree gimplification.  */
#define LANG_HOOKS_GIMPLIFY_EXPR lhd_gimplify_expr
```

默认回调函数

gcc/langhooks-def.h

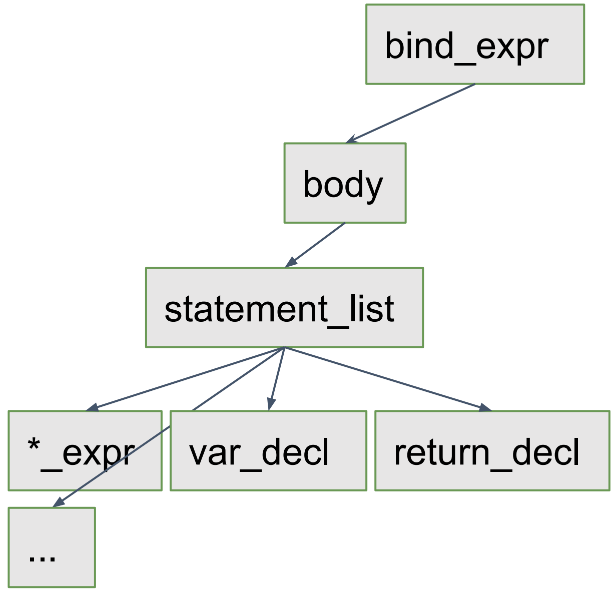
GIMPLIFY流程（C语言）



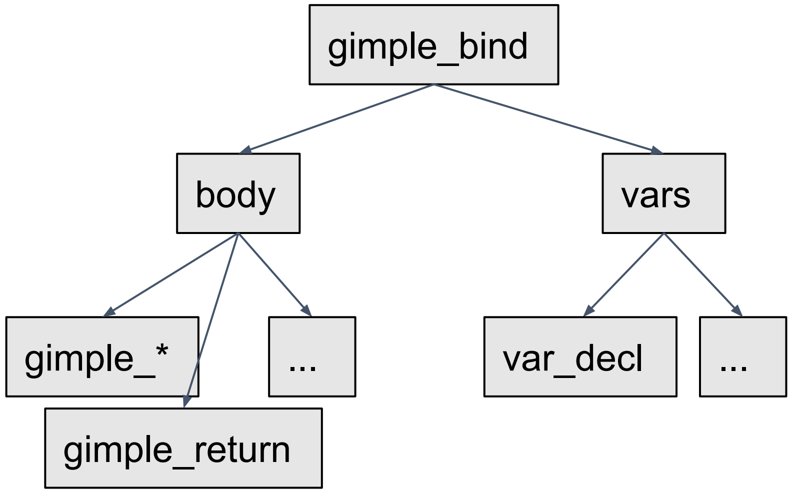
```
  switch (code)
    {
    case AGGR_INIT_EXPR:
      simplify_aggr_init_expr (expr_p);
      ret = GS_OK;
      break;

    case VEC_INIT_EXPR:
      {
  location_t loc = input_location;
  tree init = VEC_INIT_EXPR_INIT (*expr_p);
  int from_array = (init && TREE_CODE (TREE_TYPE (init)) == ARRAY_TYPE);
  gcc_assert (EXPR_HAS_LOCATION (*expr_p));
  input_location = EXPR_LOCATION (*expr_p);
  *expr_p = build_vec_init (VEC_INIT_EXPR_SLOT (*expr_p), NULL_TREE,
          init, VEC_INIT_EXPR_VALUE_INIT (*expr_p),
          from_array,
          tf_warning_or_error);
  hash_set<tree> pset;
  cp_walk_tree (expr_p, cp_fold_r, &pset, NULL);
  cp_genericize_tree (expr_p, false);
  copy_if_shared (expr_p);
  ret = GS_OK;
  input_location = loc;
      }
      break;
```

C++中的向量初始化 gcc/cp/cp-gimplify.c

GIMPLE

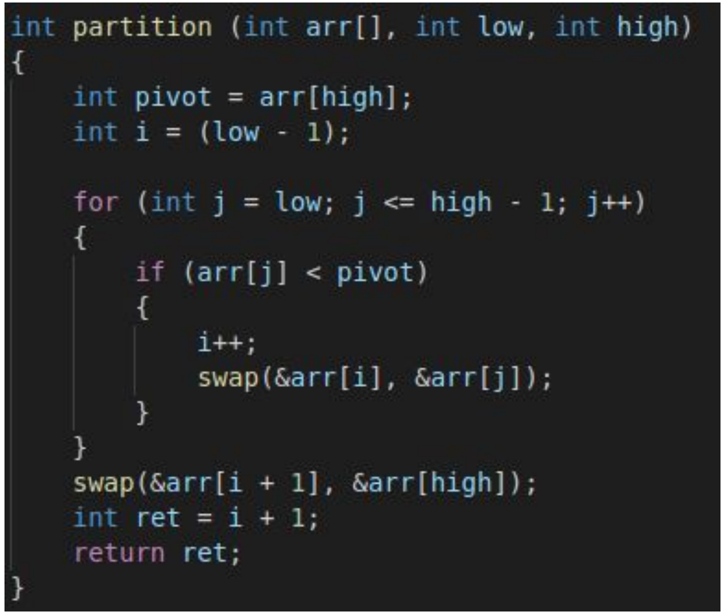


Generic



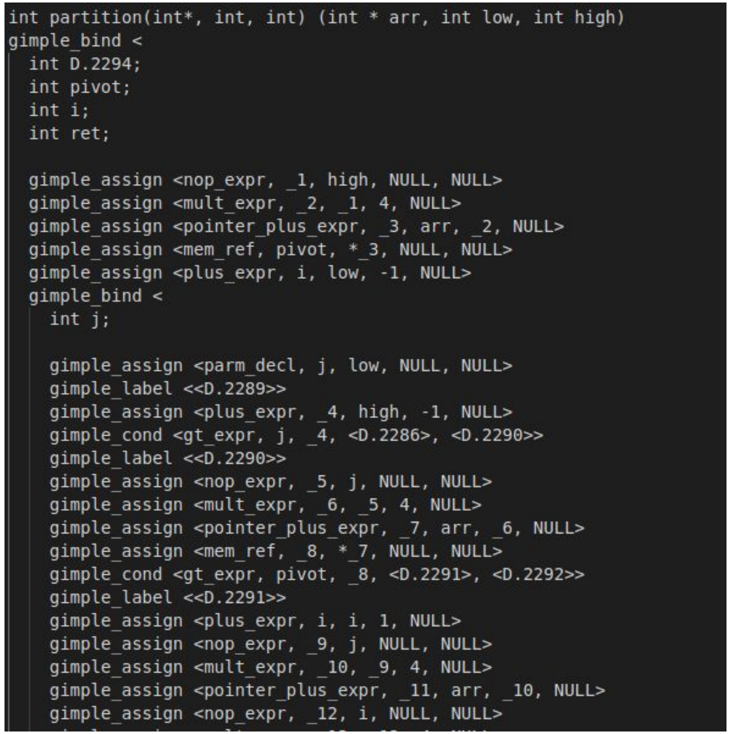
GIMPLE

GIMPLE



```c
int partition (int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    int ret = i + 1;
    return ret;
}
```

C语言源代码

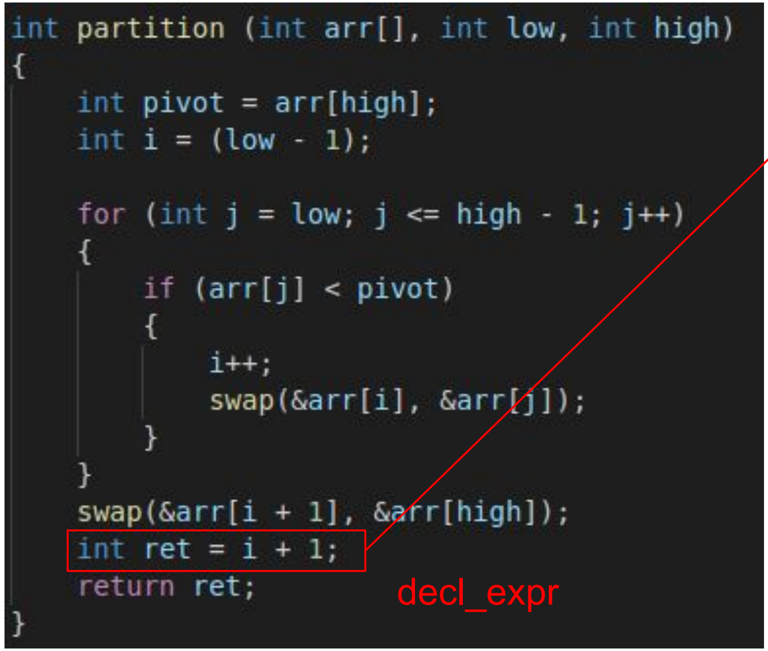


```
int partition(int*, int, int) (int * arr, int low, int high)
gimple_bind <
  int D.2294;
  int pivot;
  int i;
  int ret;

  gimple_assign <nop_expr, _1, high, NULL, NULL>
  gimple_assign <mult_expr, _2, _1, 4, NULL>
  gimple_assign <pointer_plus_expr, _3, arr, _2, NULL>
  gimple_assign <mem_ref, pivot, *_3, NULL, NULL>
  gimple_assign <plus_expr, i, low, -1, NULL>
  gimple_bind <
    int j;

    gimple_assign <parm_decl, j, low, NULL, NULL>
    gimple_label <<D.2289>>
    gimple_assign <plus_expr, _4, high, -1, NULL>
    gimple_cond <gt_expr, j, _4, <D.2286>, <D.2290>>
    gimple_label <<D.2290>>
    gimple_assign <nop_expr, _5, j, NULL, NULL>
    gimple_assign <mult_expr, _6, _5, 4, NULL>
    gimple_assign <pointer_plus_expr, _7, arr, _6, NULL>
    gimple_assign <mem_ref, _8, *_7, NULL, NULL>
    gimple_cond <gt_expr, pivot, _8, <D.2291>, <D.2292>>
    gimple_label <<D.2291>>
    gimple_assign <plus_expr, i, i, 1, NULL>
    gimple_assign <nop_expr, _9, j, NULL, NULL>
    gimple_assign <mult_expr, _10, _9, 4, NULL>
    gimple_assign <pointer_plus_expr, _11, arr, _10, NULL>
    gimple_assign <nop_expr, _12, i, NULL, NULL>
```

High GIMPLE

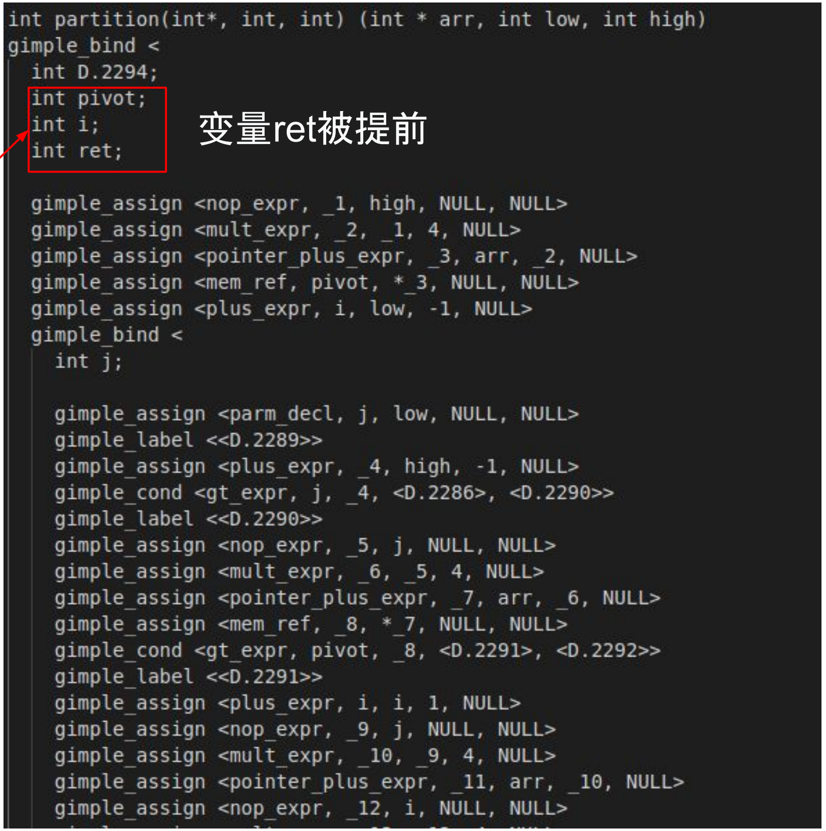
每个gimple语句的定义gcc/gimple.def

GIMPLE



```
int partition (int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    int ret = i + 1;
    return ret;
}
```

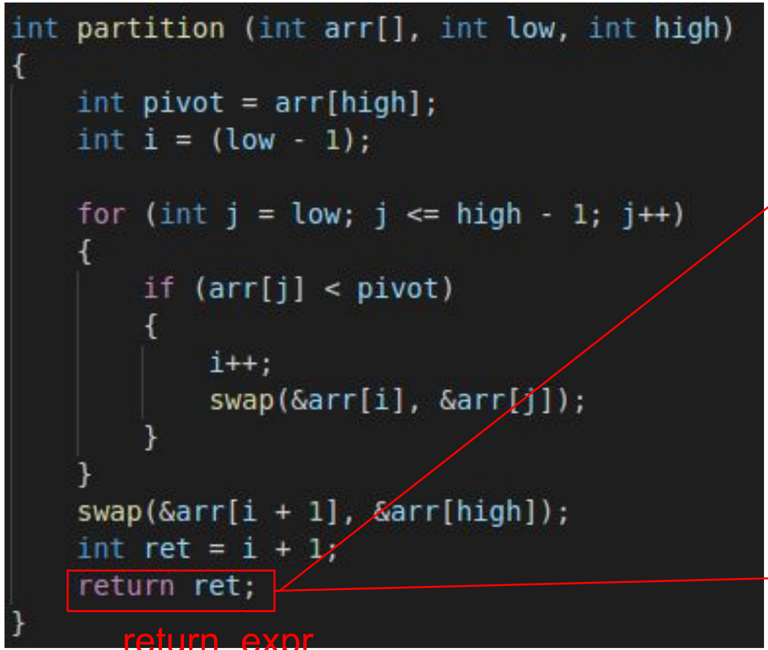
C语言源代码



```
int partition(int*, int, int) (int * arr, int low, int high)
gimple_bind <
  int D.2294;
  int pivot;
  int i;
  int ret;

  gimple_assign <nop_expr, _1, high, NULL, NULL>
  gimple_assign <mult_expr, _2, _1, 4, NULL>
  gimple_assign <pointer_plus_expr, _3, arr, _2, NULL>
  gimple_assign <mem_ref, pivot, *_3, NULL, NULL>
  gimple_assign <plus_expr, i, low, -1, NULL>
  gimple_bind <
    int j;

    gimple_assign <parm_decl, j, low, NULL, NULL>
    gimple_label <<D.2289>>
    gimple_assign <plus_expr, _4, high, -1, NULL>
    gimple_cond <gt_expr, j, _4, <D.2286>, <D.2290>>
    gimple_label <<D.2290>>
    gimple_assign <nop_expr, _5, j, NULL, NULL>
    gimple_assign <mult_expr, _6, _5, 4, NULL>
    gimple_assign <pointer_plus_expr, _7, arr, _6, NULL>
    gimple_assign <mem_ref, _8, *_7, NULL, NULL>
    gimple_cond <gt_expr, pivot, _8, <D.2291>, <D.2292>>
    gimple_label <<D.2291>>
    gimple_assign <plus_expr, i, i, 1, NULL>
    gimple_assign <nop_expr, _9, j, NULL, NULL>
    gimple_assign <mult_expr, _10, _9, 4, NULL>
    gimple_assign <pointer_plus_expr, _11, arr, _10, NULL>
    gimple_assign <nop_expr, _12, i, NULL, NULL>
```

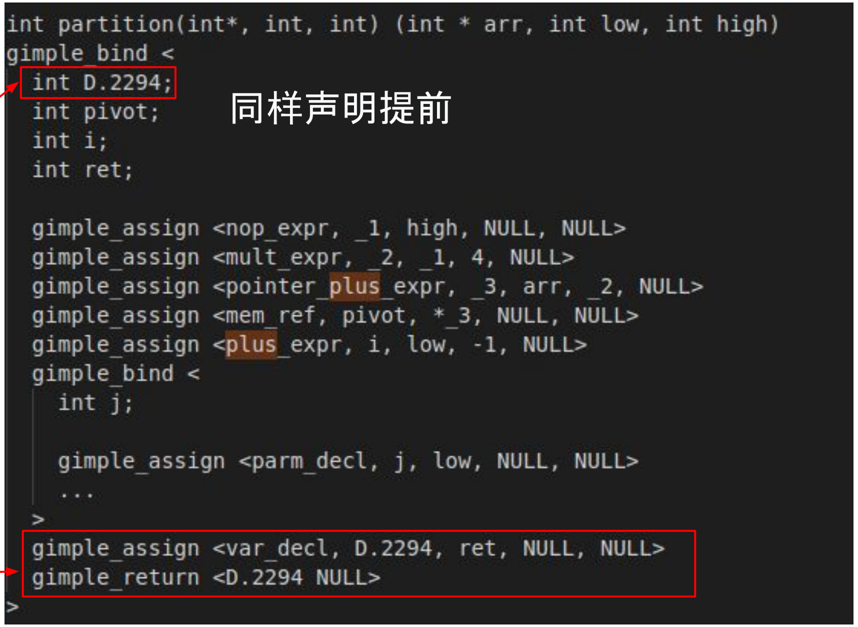
High GIMPLE

GIMPLE



```
int partition (int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    int ret = i + 1;
    return ret;
}
```

C语言源代码



```
int partition(int*, int, int) (int * arr, int low, int high)
gimple_bind <
  int D.2294;
  int pivot;
  int i;
  int ret;

  gimple_assign <nop_expr, _1, high, NULL, NULL>
  gimple_assign <mult_expr, _2, _1, 4, NULL>
  gimple_assign <pointer_plus_expr, _3, arr, _2, NULL>
  gimple_assign <mem_ref, pivot, *_3, NULL, NULL>
  gimple_assign <plus_expr, i, low, -1, NULL>
  gimple_bind <
    int j;

    gimple_assign <parm_decl, j, low, NULL, NULL>
    ...
  >
  gimple_assign <var_decl, D.2294, ret, NULL, NULL>
  gimple_return <D.2294 NULL>
>
```

High GIMPLE

GIMPLIFY

1) 表达式简化
    控制流 -> gimple_goto/gimple_cond
2) 变量声明语句提前
3) 显式加入临时变量
    函数返回语句
    函数调用语句

参考

深入分析GCC
编译系统透视：图解编译原理

GCC internal
IITB GCC lecture

谢 谢

欢迎交流合作