GCC 的语法树结构

编译器的第一遍是语法/词法分析器，此遍完成对整个输入文件的扫描分析，在分析过程中，每条语句被转换成语法树。一旦一个函数被扫描分析，则以语句或函数为单位生成对应的 RTL 中间代码。下面分以下三个方面介绍语法树的表示方式：

·语法树的结点种类
·语法树的结点编码
·语法树举例

1. 标准的树结点（即语言无关的树结点）介绍

1.1. 语法树的结点种类

语法树由树结点构成，树结点是定义如下的一种联合：

```
union tree_node
{
  struct tree_common common;
  struct tree_int_cst int_cst;
  struct tree_real_cst real_cst;
  struct tree_vector vector;   /***GCC3.1 新增****/
  struct tree_string string;
  struct tree_complex complex;
  struct tree_identifier identifier;
  struct tree_decl decl;
  struct tree_type type;
  struct tree_list list;
  struct tree_vec vec;
  struct tree_exp exp;
  struct tree_block block;
 };
```

语法树有 13 种结点，它们分别为：

·TREE_COMMON 结点，这是一类基本结点，其余 12 种结点都包含此类结点。
·INTEGER_CST 结点，此类结点代表一个双字整常数，若此常数为有符号常数，则将值扩展成双字。
·REAL_CST 结点，此类结点代表一个以‘double’或‘字符串’形式表示的实数值，若此值以字符串形式表示，则不允许对它做任何优化，但允许交叉编译。
·STRING_CST 结点，此类结点代表一个字符串常量。
·COMPLEX_CST 结点，此类结点代表一个复型常量。
·TREE_IDENTIFIER 结点，此类结点代表一个标识符。
·TREE_LIST 结点，此类结点代表一串相关结点。
·TREE_VEC 结点，此类结点代表一组结点向量。
·TREE_EXP 结点，此类结点代表一个表达式。
·TREE_BLOCK 结点，此类结点表示一个程序块的相关信息。
·TREE_TYPE 结点，此类结点代表数据类型。
·TREE_DECL 结点，此类结点代表一个被说明的符号名。
·TREE_VECTOR 结点，此类结点对应于机器模式中的 vector 模式，主要用于并行执行表达式，

如 ix86 中的 SSE 扩展，对应的 vector 模式有 V8QImode、V16QImode、V16Simode 等。这类树结点往往由 RTL vector 中的值形成。

Expmed.c 中的 make_tree 函数表述了这一形成过程：

```
Make_tree(tree type,rtx x)
{
    tree t;
    switch(GET_CODE(X))
    {
        case   CONST_VECTOR:
               int I,units;
               rtx elt;
               tree t=NULL_TREE;
               units=CONST_VECTOR_NUNITS(x);
        /*** build a tree with vector elements***/
               for(I=units-1;I>=0;--I)
               { elt = CONST_VECTOR_ELT(x,i);
                 t=tree-cons(NULL_TREE,make_tree(type,elt),t);
               }
               return build_vector(type,t);   /**build_vector 返回一个
                                                VECTOR_CST  树结点****/
  }。
```

VECTOR_CST 结点的形成主要是因为将 VECTOR 看成一种类型，因而在常数折叠、常数表达式判别（如是否为 0）等过程中应考虑到。

程序的很多地方调用了 make_tree 过程，如 expr.c 中的 store_exp(tree exp, rtx target, int want_value)。

而具有 CONST_VECTOR 代码的 RTX 表达式则在 emit-rtl.c 中的 gen_const_vector_0 和 insn-emit.c 中通过调用 gen_rtx_CONST_VECTOR 生成。

下面详细介绍各类结点。

1.2. TREE_COMMON 结点

此类结点定义所有树结点都具备的一些公共特性（并不是所有特性对所有类型的结点都有用），其结构定义为：

```
struct tree_common
{
  tree chain;
  tree type;
  ENUM_BITFIELD(tree_code) code : 8;
  unsigned side_effects_flag : 1;
  unsigned constant_flag : 1;
  unsigned addressable_flag : 1;
  unsigned volatile_flag : 1;
  unsigned readonly_flag : 1;
  unsigned unsigned_flag : 1;
  unsigned asm_written_flag: 1;
  unsigned unused_0 : 1;       /*表示第一个字节中还有一位没用到*/
  unsigned used_flag : 1;
  unsigned nothrow_flag : 1;   /*处理异常*/
  unsigned static_flag : 1;
```

```
  unsigned public_flag : 1;
  unsigned private_flag : 1;
  unsigned protected_flag : 1;
  unsigned bounded_flag : 1;
  unsigned deprecated_flag : 1;
  unsigned lang_flag_0 : 1;
  unsigned lang_flag_1 : 1;
  unsigned lang_flag_2 : 1;
  unsigned lang_flag_3 : 1;
  unsigned lang_flag_4 : 1;
  unsigned lang_flag_5 : 1;
  unsigned lang_flag_6 : 1;
  unsigned unused_1 : 1;
};
```

其中，各个域的含义如下：

chain

不同目的的多个结点通过 chain 域连接起来。例如，各种类型通过 chain 域连接起来以便调试器使用；同一作用域中的符号名通过 chain 域连接以记录此作用域的内容；相邻的语句结点通过 chain 域连接；TREE_LIST 中的结点通过 chain 域连接起来。对于 IDENTIFIER 结点，chain 域将具有相同 hash 值的标识符连接在一起。

通常用 TREE_CHAIN （node）宏来访问 chain 域。

type

在所有表达式结点中，此域指向表达式的类型。

在 POINTER_TYPE 结点中，此域指向指针所指类型。

在 ARRAY_TYPE 结点中，此域指向数组元素的类型。

通常用 TREE_TYPE（node）宏来访问 type 域。

code

结点编码，说明此结点属于何种结点，各种结点编码的介绍详见第二部分。通常用 TREE_CODE (node) 宏来访问 code 域，用宏 TREE_SET_CODE (node, value) 来为 code 定值。

对一个表达式树结点，在保证删除前后机器模式不变的情况下（即 TYPE_MODE (TREE_TYPE (exp)) = = TYPE_MODE (TREE_TYPE (TREE_OPERAND (exp , 0)))），用 STRIP_NOPS (exp) 宏来删掉 NOP_EXPRS、CONVERT_EXPR 和 NON_LVALUE_EXPR。用 STRIP_TYPE_NOPS (exp) 宏来删掉 NOP_EXPRS 、CONVERT_EXPR 和 NON_LVALUE_EXPR，但保证删除前后树结点的类型不变，即 TREE_TYPE (exp) = =TREE_TYPE (TREE_OPERAND (exp , 0))。用 INTEGRAL_TYPE_P (type)、FLOAT_TYPE_P (type)和 GGREGATE_TYPE_P (type) 宏来判断一个类型结点是否整型、实型或聚合类型（包括数组类型、记录类型、联合类型、集合类型和 QUAL_UVION_TYPE）。

以下是此结点中的一些位域，用于标记各种特性。

side_effect_flag

在任何表达式中，此域非零表示此表达式具有副作用，或表示对整个表达式重新估值将产生一个与以前不同的值。

若某一子表达式是一个函数调用，一个副作用或对一个临时变量的引用，则此域为 1。

在一个 tree_decl 结点中，只有当此符号的 volatile 域为 1 时，此域才设为 1。

用 TREE_SIDE_EFFECTS (node) 宏来访问此域。

constant_flag

此域为 1 表示表达式的值为常数，通常所有常数结点中此域为 1。在一个算术表达式、ADDR_EXPR 或 CONSTRUCTOR 结点中，若值为常数，则此域为 1。用 TREE_CONSTANT

(node) 宏来访问此域。

addressable_flag

在 VAR_DECL 结点中，此域非零表示需要此符号的地址，所以，此符号不能放在寄存器中。

在 FUNCTION_DECL 结点中，此域非零表示需要此函数名的地址，所以，既使它是一个内联函数，也需要编译。

在 CONSTRUCTOR 结点中，此域非零表示构造的对象必须存放于内存中。

在 LABEL_DECL 结点中，此域非零表示在嵌套结构之外有一转向此标号的 GOTO 语句。

在 TYPE 结点中，此域非零表示具有此类型的对象必须全部存放于内存，不能将对象的一部分放入参数寄存器。

在 IDENTIFIER 结点中，此域非零表示此符号名的某外部定义已赋予其地址。这与内联函数有关系。

用 TREE_ADDRESSABLE (node) 宏来访问此域。

volatile_flag

在表达式结点中，从 C 语言角度而言，此域非零表示此表达式是易变的：

表达式的地址必须是‘Volatile WHATEVER *’类型。此域用于 DECL 结点和 REF 结点。

在 TYPE 结点中，此域非零表示此类型是短暂存在的。

如果在一个表达式结点中，此域非零，则 side_effect_flag 域也非零.通常。用 TREE_THIS_VOLATILE (node) 宏来访问此域，但若 node 代表一个类型，则用 TYPE_VOLATILE 宏来替代 TREE_THIS_VOLATILE 宏。

readonly_flag

在 VAR_DECL、PARM_DECL、FIELD_DECL 或任一种结点中，此域非零表示此结点可能不是赋值语句的左值，在 TYPE 结点中，此域非零意味着这是一个常数类型。

通常用 TREE_READONLY (node) 宏来访问此域，但当结点是 TYPE 类型的结点时，用 TYPE_READONLY (node) 宏来替代 TREE_READONLY (node)。

unsigned_flag

在 INTEGER_TYPE 或 ENUMERAL_TYPR 结点中，此域非零表示一种无符号类型。

在 FIELD_DECL 结点中，此域非零表示无符号位域。

asm_written_flag

在 VAR_DECL 结点中，此域非零意味着已经写出了此符号的汇编代码。

在 FUNCTION_DECL 结点中，此域非零意味着此函数已经被编译。

在 RECORD_TYPE、UNION_TYPE、QUAL_UNION_TYPE 或 ENUMERAL_TYPE 结点中，此域非零表示有关此类型的 sdb 调试信息已经写入。

在 BLOCK 结点中，此域非零表示此 block 已被 reorder_blocks（为优化而将块重新排序）过程处理过了。

通常用 TREE_ASM_WRITTEN (node)宏来访问此域。

used_flag

在_DECL 结点中，此域非零表示此符号名在其作用域内被使用过。

在表达式结点中，此域非零表示当表达式的值未被使用时，不报警告信息。

在 IDENTIFIER_NODE 中，此域非零表示此符号名已被某一外部定义使用过。

nothrow_flag

(原来为 raises_flag，此域非零表示对此结点的估算能导致一个异常。2.7 版本中没有实现这一功能。)

在 FUNCTION_DECL 中，此域非零表示调用此函数的 CALL 语句不能 throw 一个异常。

在 CALL_EXPR 中，此域非零表示此调用语句不能 throw 一个异常？

在类型结点中，此域非零表示此类型的所有对象都由语言或前端来保证其正确的对齐方式？

通常用 TREE_NOTHROW（node）宏和 TYPE_ALIGN_OK(node) 宏来访问此域。

static_flag

在 VAR_DECL 结点中，此域非零表示应为此符号分配静态存储空间。

在 FUNCTION_DECL 结点中，非零表示此函数已被定义。

在 CONSTRUCTOR 结点中，非零表示为其分配静态存储空间。

通常用 TREE_STATIC (node) 宏来访问上述结点中的此域。

在 CONVERT_EXPR、NOP_EXPR 或 COMPOUND_EXPR 结点中，非零表示已隐式地创建此结点，而不导致一个“unused value”的警告信息。

用 TREE_NO_UNUSED_WARNING (node) 宏来访问上述结点中的此域。

在 TREE_LIST 或 TREE_VEC 结点中，此域非零表示结点链是通过‘virtual’定义而得到的。

用 TREE_VIA_VIRTUAL (node) 宏来访问上述结点中的此域。

在 INTEGER_CST、REAL_CST 或 COMPLEX_CST 结点中，此域非零表示在常数折迭优化中发生溢出。这与 TREE_OVERFLOW 不同，因为，标准 C 在常数表达式发生溢出时，需要一个诊断信息。用 TREE_CONSTANT_OVERFLOW (node) 宏来访问上述结点中的此域。

在 IDENTIFIER_NODE 中，此域非零表示以此字符串为参数调用 assemble_name。用 TREE_SYMBOL_REFERENCED (node) 宏来访问上述结点的此域。

public_flag

在 INTEGER_CST，REAL_CST 或 COMPLEX_CST 结点中，此域非零表示常数折迭优化中出现了溢出，且不产生相关的警告信息，用 TREE_OVERFLOW (node)宏来访问上述结点中的此域。TREE_OVERFLOW 隐含 TREE_CONSTANT_OVERFLOW，但反之不成立。

在 VAR_DECL 或 FUNCTION_DECL 结点中，此域非零表示能在外部模块中访问此符号名。

在 IDENTIFTER 结点中，此域非零表示对内层作用域中的标识符，已有一个能被外部模块访问的外部说明。

用 TREE_PUBLIC (node) 宏来访问上述结点中的此域。

在 TREE_LIST、TREE_VEC 结点中，此域非零表示指向基类的结点链是通过‘public’定义而得，这样就保证了基类的 public 域为真。用 TREE_VIA_PUBLIC (node) 宏来访问上述结点的此域。

private_flag

此域用于‘private’定义。用 TREE_VIA_PRIVATE (node) 宏来访问此域。

在 C++中，用 TREE_PRIVATE (node) 宏来访问此域。

protected_flag

在 TREE_LIST 结点中，此域非零表示指向基类的结点链是由‘protected’定义而得，这就保证了基类的 protected 域为真。

用 TREE_VIA_PROTECTED (node) 宏来访问此域。

在 C++中，用宏 TREE_PROTECTED (node) 访问此域。在 C++ 的 Block 结点中，此域等同于 HANDLER_BLOCK_FLAG。

bounded_flag

在…_TYPE 结点中，此域非零表示此类型的大小和 layout 会由于指针 bounds 的出现而被改变。

在 FUNCTION_TYPE 或 METHOD_TYPE 中，此域非零表示函数参数或返回值的大小和 layout 会由于指针的出现而被改变。

用 TYPE_BOUNDED(type)宏来访问…_TYPE 结点中的此域。TYPE_BOUNDED(type)并不意味着这是一个 bounded indirect type？而应该用 BOUNDED_POINTER_TYPE_P、BOUNDED_REFERENCE_TYPE_P 和
BOUNDED_INDIRECT_TYPE_P 宏来测试。

```
#define BOUNDED_INDIRECT_TYPE_P(TYPE) \
  (TREE_CODE (TYPE) == RECORD_TYPE && TREE_TYPE (TYPE))
/* Access the simple-pointer subtype of a bounded-pointer type.  */
```

#define TYPE_BOUNDED_SUBTYPE(TYPE) TREE_TYPE (TYPE_BOUNDED_VALUE (TYPE))
/* Access the field decls of a bounded-pointer type. */
#define TYPE_BOUNDED_VALUE(TYPE) TYPE_FIELDS (TYPE)
#define TYPE_BOUNDED_BASE(TYPE) TREE_CHAIN (TYPE_BOUNDED_VALUE (TYPE))
#define TYPE_BOUNDED_EXTENT(TYPE) TREE_CHAIN (TYPE_BOUNDED_BASE (TYPE))
#define TYPE_FIELDS(NODE) (TYPE_CHECK (NODE)->type.values)
#define BOUNDED_REFERENCE_TYPE_P(TYPE) \
(BOUNDED_INDIRECT_TYPE_P (TYPE) \
&& TREE_CODE (TYPE_BOUNDED_SUBTYPE (TYPE)) == REFERENCE_TYPE)

在表达式、VAR_DECL、 PARM_DECL、FIELD_DECL、 FUNCTION_DECL 和 IDENTIFIER_NODE 中，此域非零表示相应对象的大小和 layout 会由于指针的出现而被改变。

在 FUNCTION_DECL 中，此域非零表示函数参数或返回值的大小和 layout 会由于指针的出现而被改变。

用 TREE_BOUNDED(node) 宏来访问上述结点的此域。

在 FUNCTION_DECL 中，存在下面两种情况，使得 TREE_BOUNDED(fndecl)的值可能与 TYPE_BOUNDED(TREE_TYPE(fundecl))的值不相同。

1 函数被隐式说明？调用时实参中有指针。

2 函数定义成可变长参数，调用时实参中有指针。

在 VAR_DECL、PARM_DECL 和 FIELD_DECL 中, TREE_BOUNDED 的值和此…_DECL 中 type 的 BOUNDED_POINTER_TYPE_P 值相同。

在 CONSTRUCTOR 或其它表达式中，此域非零表示此表达式的值是一个 bounded pointer。因为允许将指针临时 CAST 为一个整数（为边界对齐），因而，BOUNDED_POINTER_TYPE_P (TREE_TYPE (EXP)) 不足以确定一个表达式的boundedness。

在 IDENTIFIER_NODE 中，此域非零表示此标示符 is prefixed with BP_PREFIX (see varasm.c) 。当函数的参数或返回值有 bounded pointer 时，此函数的 DECL_ASSEMBLER_NAME 中此域非零。

deprecated_flag

Nonzero in a IDENTIFIER_NODE if the use of the name is defined as a deprecated feature by __attribute__((deprecated)).

用 TREE_DEPRECATED(NODE)宏来访问此域。

lang_flag_0
lang_flag_1
lang_flag_2
lang_flag_3
lang_flag_4
lang_flag_5
lang_flag_6

这些域用于各种语言前端。

以下介绍的各种结点都包含一个 tree_common 类型的域，文中不再介绍。

1.3. TREE_INT_CST 结点

此类结点的结构定义为：

```
struct tree_int_cst
{
  struct tree_common common;
  rtx rtl;          /* acts as link to register transfer language
                                    (rtl) info */
  /* A sub-struct is necessary here because the function `const_hash'
       wants to scan both words as a unit and taking the address of the
       sub-struct yields the properly inclusive bounded pointer.   */
  struct {
    unsigned HOST_WIDE_INT low;
    HOST_WIDE_INT high;
  } int_cst;
};
```

其中，HOST——WIDE——INT 是与宿主机相关的宽整数。此类结点代表一个双字整数，若数据类型是有符号的，则值被符号扩展成两个字。若此整数为一个不足两个字的无符号常数，则多余的位为零。它的各个域的含义如下：

rtl 域

在 REAL_CST、STRING_CST、 COMPLEX_CST、VECTOR_CST 和 CONSTRUCTOR 节点中以及所有的符号常量（立即数除外），都用此域指向对应的 RTL。用 TREE_CST_RTL(NODE)宏来访问此域。

int_cst_low 域

与机器相关的宽整型，用于存放整数的低位字节。

用 TREE_INT_CST_LOW (node) 宏来访问此域。

int_cst_high 域

与机器相关的宽整型，用于存放整数的高位字节。

用 TREE_INT_CST_HIGH (node) 宏来访问此域。

另外，用宏 INT_CST_LT (A, B)和 INT_CST_LT_UNSIGNED (A, B)来判断 A 是否小于 B。

1.4. TREE_REAL_CST

tree_real_cst 结点的结构定义为：

```
struct tree_real_cst
{
  struct tree_common common;
  rtx rtl;          /* acts as link to register transfer language (rtl) info */
  REAL_VALUE_TYPE real_cst;
};
```

其中，REAL_VALUE_TYPE 是与机器相关的浮点类型。此类结点代表一个实数，它的各个域的含义如下：

rtl 域

指向 rtx 的指针，是树结点和 rtl 连接的钮带。

用 TREE_CST_RTL (node) 宏来访问此域。

real_cst 域

实数的内部表示形式。

用 TREE_REAL_CST (node) 宏来访问此域。

1.5. TREE_STRING

tree_string 结点的结构定义为：

```
struct tree_string
{
   struct tree_common common;
   rtx rtl;          /* acts as link to register transfer language (rtl) info */
   int length;
   const char *pointer;
};
```

此类结点代表一个字符串，它的各个域的含义如下：

rtl 域

指向 rtx 的指针。

用 TREE_CST_RTL (node) 宏来访问此域。

length

整型，表示串的长度。

用 TREE_STRING_LENGTH (node) 宏来访问此域。

pointer

字符指针类型，指向串的内容。

用 TREE_STRING_POINTER (node) 宏来访问此域。

1.6. TREE_COMPLEX

tree_complex 结点的结构定义为：

```
struct tree_complex
{
   struct tree_common common;
   rtx rtl;          /* acts as link to register transfer language (rtl) info */
   tree real;
   tree imag;
};
```

此类结点代表一个复数，它的各个域的含义如下：

rtl

指向 rtx 的指针。用 TREE_CST_RTL (node) 宏来访问此域。

real

代表复型数据的实部。

用 TREE_REALPART (node) 宏来访问此域。

imag

指向树结点的指针，代表复型数据的虚部。

用 TREE_IMAGPART (node) 宏来访问此域。

1.7. TREE_IDENTIFIER 结点

identifier 结点的结构定义如下：

```
struct tree_identifier
{
   struct tree_common common;
   struct ht_identifier id;
};
```

此类结点表示一个标识符，其中，id 指向 Hash table 中的字符串。

1.8. TREE_LIST 结点

tree_list 结点的结构定义为：

```
struct tree_list
{
   struct tree_common common;
   tree purpose;
   tree value;
 };
```

此类结点代表一串相关结点组成的表，其中：

purpose

指向树结点的指针

用 TREE_PURPOSE (node) 宏访问此域。

value

指向基类树结点的指针。

用 TREE_VALUE (node) 宏访问此域。

1.9. TREE_VEC 结点

tree_vec 结点的结构定义为：

```
struct tree_vec
{
   struct tree_common common;
   int length;
   tree a[1];
};
```

此类结点代表一组结点向量，其中：

length

整型，代表向量中结点的个数。用 TREE_VEC_LENGTH (node) 宏访问此域。

a

指向树结点的指针数组。

用 TREE_VEC_ELT (node, i) 宏来访问向量中的第 i 号结点。

用 TREE_VEC_END (node) 宏来获得向量中最后一个结点的地址。

1.10. TREE_EXP 结点

tree_exp 结点的结构定义为：

```
struct tree_exp
{
    struct tree_common common;
    int complexity;
    tree operands[1];
};
```

此类结点代表一个表达式，其中：

complexity

整型。

用 TREE_COMPLEXITY (node) 宏来访问此域。

operands

指向操作数树结点的指针数组。用 TREE_OPERAND (node, i) 来访问表达式的第 i 号操作数。

在 SAVE_EXPR 结点中，用 SAVE_EXPR_CONTEXT(node)宏来访问第 1 号操作数， 用 SAVE__EXPR_RTL(node) 宏来获得第 2 号操作数的相应的 rtx。在 RTL_EXPR 结点中，用 RTL_EXPR_SEQUENCE(node)宏获得第 0 号操作数相应的 rtx。用 RTL_EXPR_RTL(node) 宏获得第 1 号操作数相应的 rtx。

在 CALL_EXPR 结点中，用 CALL_EXPR_RTL（node）来获得第 2 号操作数相应的 rtx。

在 CONSTRUCTOR 结点中，用 CONSTRUCTOR_ELTS (node) 宏来访问第 1 号操作数。

1.11. TREE_BLOCK 结点

TREE_BLOCK 结点的结构定义为：

```
struct tree_block
{
    struct tree_common common;
    unsigned handler_block_flag : 1;
    unsigned abstract_flag : 1;
    unsigned block_num : 30;
    tree vars;
    tree subblocks;
    tree supercontext;
    tree abstract_origin;
    tree fragment_origin;
    tree fragment_chain;
};
```

一旦存在多重层次嵌套结构且其内容已被编译，则用 BLOCK 结点来表示此嵌套结构及其符号定义，其中。

vars

指向此 BLOCK 中定义的符号结点链

用 BLOCK_VARS (node) 宏访问。

subblocks

指向嵌套其内的下级 BLOCK 链，此 BLOCK 链通过 BLOCK_CHAIN 域链接起来。

用 BLOCK_SUBBLOCKS (node) 宏访问。

supercontext

指向包含此块的父级 BLOCK。对于一个函数作用域内的最外块，此域指向 FUNCTION_DECL 结点。用 BLOCK_SUPERCONTEXT (node) 宏访问。

abstract_origin

指向一个抽象树结点，此 BLOCK 是抽象树结点的一个实例。如果此域为空，说明此 BLOCK 不是任何其它树结点的实例，如果此域不为空，则它可能指向另一个 BLOCK 结点，也可能指向一个 FUNCTION_DECL 结点（例如当一个块代表一个特殊内联函数作用域内的最外层块时，此块的此域指向 FUNCTION_DECL 结点）。用 BLOCK_ABSTRACT_ORIGIN (node) 宏访问。

handler_block_flag

此域非零表示此块将处理列在 BLOCK_VAR 中的异常。

用 BLOCK_HANDLER_BLOCK (node) 宏访问。

abstract_flag

若此域非零表示此 BLOCK 代表一个块的抽象实例，用 BLOCK_ABSTRACT (node) 宏访问。

BLOCK_CHAIN (node) 宏用于访问同一级的下一 BLOCK。

Block_num

块的索引，用 BLOCK_NUMBER(NODE)宏访问此域。

Fragment_origin

Fragment_chain

若 block 重排将一个词法意义上的 BLOCK 分割成不连续的地址空间（将每个地址空间成为一个 fragment），则将原始 BLOCK 保留一个副本。在 fragment 中任选一段作为 ORIGIN，其它段通过 fragment_origin 指向 origin，origin 块本身的这个指针为空。段之间通过 fragment_chain 连接起来。用 LOCK_FRAGMENT_ORIGIN(NODE) 和 BLOCK_FRAGMENT_CHAIN(NODE)来访问这两个域。

1.12. tree_type

tree_type 结点的结构定义为：

```
struct tree_type
{
  struct tree_common common;
  tree values;
  tree size;
  tree size_unit;
  tree attributes;
  unsigned int uid;
  unsigned int precision : 9;
  ENUM_BITFIELD(machine_mode) mode : 7;
  unsigned string_flag : 1;
  unsigned no_force_blk_flag : 1;
  unsigned needs_constructing_flag : 1;
  unsigned transparent_union_flag : 1;
  unsigned packed_flag : 1;
  unsigned restrict_flag : 1;
  unsigned pointer_depth : 2;
  unsigned lang_flag_0 : 1;
  unsigned lang_flag_1 : 1;
  unsigned lang_flag_2 : 1;
```

```
    unsigned lang_flag_3 : 1;
    unsigned lang_flag_4 : 1;
    unsigned lang_flag_5 : 1;
    unsigned lang_flag_6 : 1;
    unsigned user_align : 1;
    unsigned int align;
    tree pointer_to;
    tree reference_to;
    union {int address; char *pointer; } symtab;
    tree name;
    tree minval;
    tree maxval;
    tree next_variant;
    tree main_variant;
    tree binfo;
    tree context;
    HOST_WIDE_INT alias_set;
    /* Points to a structure whose details depend on the language in use.   */
    struct lang_type *lang_specific;
};
```

此类结点记录各种数据类型的信息，其中：

values

用于 C 语言的枚举类型。在枚举类型中，此域指向一条链，链中每个元素的 PURPOSE 域代表一个枚举常量的名字，VALUE 域指向此枚举常量 (用 INTEGER_CST 结点表示)。

用 TYPE_VALUES (node) 宏访问。

size

每一种类型树结点都有此域，它指向一个树结点，此树结点是一个以位为单位表示此种类型大小的表达式。

用 TYPE_SIZE (node) 宏访问。

size_unit

表示此种类型的以字节为单位的大小的表达式。

attributes

指向一 IDENTIFIER 结点链，此链中的每一结点代表这种类型的各种属性。

用 TYPE_ATTRIBUTES (node) 宏访问。

uid

无符号整型，标识树。结点的一个唯一的标号，用 TYPE_UID (node) 宏访问。

precision

表示此数据类型所占的位数，是一个无符号字符型，用 TYPE_PRECISION (node) 宏访问。

mode

整型或枚举类型，包含具有此数据类型的值所对应的机器模式。

用 TYPE_MODE (node) 宏访问。

align

整型，其值以位为单位，表示此种类型的对象必须具备的对齐要求。

用 TYPE_ALIGN (node) 宏访问。

pointer_to

指向本类型的指针类型，可能为空。

用 TYPE_POINTER_TO (node) 宏访问。

reference_to

用于 C++，类似于一个指针，但它被强制成与它所指的值一致。

用 TYPE_REFERENCE_TO (node) 宏访问。

name

此域包含程序中所使用的此类型的名字的有关信息，（GDB 符号表输出时需用到），它或指向一个 TYPE_DECL 结点，或指向一个 IDENTIFIER_NODE 结点，前者适合于用 typedef 定义的类型；后者适合于结构、联合或枚举类型中有一个标志但无具体名字的类型。

用 TYPE_NAME (node) 宏访问。

minval

说明此种类型的变量所能具有的最小值。

用 TYPE_MIN_VALUE (node) 宏访问。

maxval

说明此种类型的变量所能具有的最大值。

用 TYPE_MAX_VALUE (node) 宏访问。

next_variant

将由 const、volatile 等定义的类型变量串起来。

用 TYPE_NEXT_VARIANT (node) 宏访问。

main_variant

存在于 next_variant 链中的每一成员中，指向链的开始处。

用 TYPE_MAIN_VARIANT (node) 宏访问。

binfo

用于 C++中的基类型用途描述：

用 TYPE_BINFO (node) 宏访问。

context

存在于任何一种有名或含有有名成员的类型中，此域指向一个说明此类型作用域的结点。如果此类型具有文件范围的作用域，则此域为空。对大多数类型而言，此域将指向一个 BLOCK 结点或 FUNCTION_DECL 结点。但当某些类型的作用域，局限于某一函数类型说明的形式参数表时，此域指向一个 FUNCTION_TYPE 结点；对 C++的“成员”类型，此域指向一个 RECORD_TYPE、UNION_TYPE 或 QUAL_UNION_TYPE 结点。

对无标志的类型（non_tagged_types〕 ，此域不指向任何东西。

用 TYPE_CONTEXT (node) 宏访问。

alias_set

关于此类型的 type-based alias set（language-specific）。若两个对象的 TYPE_ALIAS_SET 不同，则两个对象不能相互别名。若 TYPE_ALIAS_SET 为-1，则说明此类型还没有 alias set；否则，若为 0，则此类型的对象可与任一类型的对象别名。用 TYPE_ALIAS_SET(NODE) 宏来访问此域。

lang_specific

指向一个结构，其具体内容依赖于所使用的语言。

用 TYPE_LANG_SPECIFIC (node) 宏访问。

string_flag

在 ARRAY_TYPE 中，对字符与字符数组有区别的语言而言，此位域非零标识一字符串（而非一个字符数组）；在 SET_TYPE 结点中，此位域非零意味着一个位串类型。

用 TYPE_STRING_FLAG (node) 宏访问。

no_force_blk_flag

在 RECORD_TYPE、UNION_TYPE 或 QUAL_UNION_TYPE 中，此位域非零表示因为此类型没有对其大小的对齐要求，所以只能是 BLKmode。

用 TYPE_NO_FORCE_BLK (node) 宏访问。

needs_constructing_flag

此位域非零意味着在产生具有此类型的对象时，必须调一个函数对其初始化。

用 TYPE_NEEDS_CONSTRUCTING (node) 宏访问。

transparent_union_flag

此位域非零表示当传递此联合类型的对象时，其传递方式必须与第一个联合对象的传递方式一致。

用 TYPE_TRANSPARENT_UNION (node) 宏访问。

restrict_flag

标示带 restrict 说明的类型说明。

pointer_depth

depth=0 时表明此类型是标量，或是一个只包含 depth=0 类型的 aggregate，或是只包含 depth=0 类型哑参和返回值的函数。

depth=1 时表示此类型是一个指向 depth=0 类型的指针，或一个只包含 depth=0 和 depth=1 类型的 aggregate，或一个只包含 depth=0 和 depth=1 类型哑参和返回值的函数。

user_align

值为 1 时表示此类型根据 aligned 属性的要求来对齐；值为 0 表示按缺省方式对齐。

lang_flag_0
lang_flag_1
lang_flag_2
lang_flag_3
lang_flag_4
lang_flag_5
lang_flag_6

对每一种语言前端，上述各位域的意义是不同的。

用 TYPE_LANG_FLAG_i (node) 宏分别访问上述位域。

（其中 0≤i≤6）。

对 C++中的基类型和类型继承，分别定义了以下宏：

一个“基类型”意味着一种数据类型用于另一种类型继承的特殊用途。

每一种基类型都有自己的 binfo 对象来描述它，binfo 对象是一个 TREE_VEC 结点。

继承是通过为给定类型分配 binfo 结点来表示。例如，给定类型 C 和 D，其中，D 被 C 继承；则必须分配 3 个 binfo 结点：一个描述类型 C 的 binfo 特性，一个描述类型 D 的 binfo 特性，还有一个描述 D 作为 C 的基类型的 binfo 特性。所以，给定一个指向 C 类的指针，可以通过 C 的 binfo 基类型而得到作为 C 的基类型的 D 的 binfo。

·BINFO_TYPE (node)即 TREE_TYPE (node)

访问此基类中被继承的实际数据类型结点。

·TYPE_BINFO_OFFSET (node)即 BINFO_OFFSET (TYPE_BINFO (node))

·BINFO_OFFSET (node) 即 TREE_VEC_ELT ((node)，1)

表示从整个对象的开始处到为代表此类型而分配的对象的开始处的偏移量（以字节为单位）。通常，此偏移量为 0，但有多重继承时，此偏移量不为零。

·BINFO_VTABLE (node)即 TREE_VEC_ELT ((node), 2)

·TYPE_BINFO_VTABLE (node)即 BINFO_VTABLE (TYPE_BINFO (node))

访问属于此基类型的虚函数表。虚函数表提供一个机制使得运行时进行方法调度，一个虚函数表的入口依语言不同而不同。

·BINFO_VIRTUALS (node) 即 TREE_VEC_ELT ((node), 3)

·TYPE_BINFO_VIRTUALS (node)即 BINFO_VIRTUALS (TYPE_BINFO (node))

虚函数表中的虚函数，指向 TREE_LIST 结点，此虚函数作为建立关于此基类型的虚函数表的初始近似值。

·BINFO_BASETYPES (node)即 TREE_VEC_EIT(TYPE_BINFO (node), 4)

·TYPE_BINFO_BASETYPES (node)即 TREE_VEC_ELT (TYPE_BINTO (node), 4)

描述被此基类型所继承的类型的一组额外的 binfos。如果此基类型用于描述在 C 中被继承的 D 的类型，并且 D 的基类型是 E 和 F，则 BINFO_BASETYPES (node) 向量包含关于 E 和 F 被 C 继承的有关信息。

·BINFO_VPTR_FIELD (node)即 TREE_VEC_ELT ((node), 5)

对一个描述继承的 BINFO 记录而言，此宏产生一个指向 FIELD_DECL 结点的指针，此 FIELD_DECL 结点包含给定继承的“虚基类指针”。

·BINFO_BASETYPE (node, n)即 TREE_VEC_ELT (BINFO_BASETYPE (node), (n))

访问基类型中第 n 个 basetype.

·BINFO_INHERITANCE_CHAIN (node)即 TREE_VEC_ELT ((node), 0)

指向一条表示继承的链。

例：如果 x 由 y 而得，y 由 z 而得，则此域用于连接 x 的 binfo 结点和 x 中 y 的 binfo 结点以表示从 x 到 y 的继承性。同样地，x 中 y 的 binfo 结点可以指向 z。从而，我们可以利用 binfo 结点本身来表示和遍历继承。是否保留此信息取决于具体语言前端。

1.13. TREE_DECL 结点

tree_decl 结点的结构定义如下：

```
struct tree_decl
{
  struct tree_common common;
  const char *filename;
  int linenum;
  unsigned int uid;
  tree size;
  ENUM_BITFIELD(machine_mode) mode : 8;
  unsigned external_flag : 1;
  unsigned nonlocal_flag : 1;
  unsigned regdecl_flag : 1;
  unsigned inline_flag : 1;
  unsigned bit_field_flag : 1;
  unsigned virtual_flag : 1;
  unsigned ignored_flag : 1;
  unsigned abstract_flag : 1;
  unsigned in_system_header_flag : 1;
  unsigned common_flag : 1;
  unsigned defer_output : 1;
  unsigned transparent_union : 1;
  unsigned static_ctor_flag : 1;
  unsigned static_dtor_flag : 1;
  unsigned artificial_flag : 1;
  unsigned weak_flag : 1;
  unsigned non_addr_const_p : 1;
  unsigned no_instrument_function_entry_exit : 1;
  unsigned comdat_flag : 1;
  unsigned malloc_flag : 1;
  unsigned no_limit_stack : 1;
  ENUM_BITFIELD(built_in_class) built_in_class : 2;
  unsigned pure_flag : 1;
  unsigned pointer_depth : 2;
  unsigned non_addressable : 1;
  unsigned user_align : 1;
  unsigned uninlinable : 1;
```

```
  /* Three unused bits.    */
  unsigned lang_flag_0 : 1;
  unsigned lang_flag_1 : 1;
  unsigned lang_flag_2 : 1;
  unsigned lang_flag_3 : 1;
  unsigned lang_flag_4 : 1;
  unsigned lang_flag_5 : 1;
  unsigned lang_flag_6 : 1;
  unsigned lang_flag_7 : 1;
  union {
    /* In a FUNCTION_DECL for which DECL_BUILT_IN holds, this is
       DECL_FUNCTION_CODE.    */
    enum built_in_function f;
    /* In a FUNCITON_DECL for which DECL_BUILT_IN does not hold, this
       is used by language-dependent code.    */
    HOST_WIDE_INT i;
    /* DECL_ALIGN and DECL_OFFSET_ALIGN.    (These are not used for
       FUNCTION_DECLs).    */
    struct {unsigned int align : 24; unsigned int off_align : 8;} a;
  } u1;
  tree size_unit;
  tree name;
  tree context;
  tree arguments;             /* Also used for DECL_FIELD_OFFSET */
  tree result;    /* Also used for DECL_BIT_FIELD_TYPE */
  tree initial; /* Also used for DECL_QUALIFIER */
  tree abstract_origin;
  tree assembler_name;
  tree section_name;
  tree attributes;
  rtx rtl;           /* RTL representation for object.    */
  rtx live_range_rtl;
  /* In FUNCTION_DECL, if it is inline, holds the saved insn chain.
      In FIELD_DECL, is DECL_FIELD_BIT_OFFSET.
      In PARM_DECL, holds an RTL for the stack slot
      of register where the data was actually passed.
      Used by Chill and Java in LABEL_DECL and by C++ and Java in VAR_DECL.    */
  union {
    struct function *f;
    rtx r;
    tree t;
    int i;
  } u2;
  /* In a FUNCTION_DECL, this is DECL_SAVED_TREE.    */
  tree saved_tree;
  /* In a FUNCTION_DECL, these are function data which is to be kept
      as long as FUNCTION_DECL is kept.    */
  tree inlined_fns;
  tree vindex;
  HOST_WIDE_INT pointer_alias_set;
  /* Points to a structure whose details depend on the language in use.    */
  struct lang_decl *lang_specific;
};
```

此类结点代表被说明的符号名。下面依次介绍各个域。

filename

字符指针， 说明此符号名在哪个源程序中被说明。用 DECL_SOURCE_FILE(node)宏访问此域。

linenum

整型。说明此符号名在源程序的哪一行中被说明。用 DECL_SOURCE_LINE(node) 宏访问此域。

size

指向表达式树结点的指针，说明符号名数据的大小。

用 DECL_SIZE(node) 宏访问此域。

uid

无符号整型，每个树结点都有一个唯一的号码。

用 DECL_UID(node)宏访问此域。

mode

整型或枚举类型，代表所说明变量的相应的机器模式，用 DECL_MODE(node)宏访问。除了 FIELD_DECL 结点外，此宏等同于 TYPE_MODE (TREE_TYPE (decl))宏。

name

指向树结点的指针，代表用户所写对象的名字，指向 IDENTIFIER_NODE 树结点。用 DECL_NAME (node) 宏访问此域。

context

在 FIELD_DECLS 结点中，此域指向 RECORD_TYPE 、UNION_TYPE 或 QUAL_UNION_TYPE 结点，说明此 FIELD 是上述类型中的一个组成成员。

在 VAR_DECL、PARM_DECL、FUNCTION_DECL、LABEL_DECL 和 CONST_DECL 结点中，此域指向包含此符号名的 FUNCTION_DECL 结点，若此符号名具有“文件范围”的作用域，则此域为 NULL_TREE。

用 DECL_CONTEXT (node)宏和 DECL_FIELD_CONTEXT (node) 宏来访问此域。

arguments

在 FUNCTION_DECL 结点中，此域指向一串 DECL 结点，这些 DECL 结点均为此函数的参数。

在 VAR_DECL 和 PARM_DECL 结点中，保留此参数以作与具体语言相关的其它用途。

用 DECL_ARGUMENTS (node) 宏来访问此域。

在 FIELD_DECL 中，代表位域的位置，即从结构开始处到此 FIELD 处所隔的位数，此时用 DECL_FIELD_BITPOS (node)宏来访问此域。

result

在 FIELD_DECL 结点中，此域指明此 FIELD 是否为一个位域，如果是，指向最初被说明的类型。用 DECL_BIT_FIELD_TYPE (node) 宏来访问此域。

在 FUNCTION_DECL 结点中，指向返回值的符号名。用 DECL_RESULT(node)宏访问此域。

在 PARM_DECL 结点中，代表程序员所写的哑参的类型。

用 DECL_ARG_TYPE_AS_WRITTEN (node) 宏访问此域。

initial

在 VAR_DECL 中，指向变量的初值。

用 DECL_INITIAL (node) 宏来访问上述结点中所述意义的域。

在 PARM_DECL 中，此域记录的是函数传递的此参数的类型（地址或值的类型），此类型可能与程序中定义的不一致。哑参对应的初值记录在函数的实参结点树中。

用 DECL_ARG_TYPE (node) 宏来访问。

在 QUAL_UNION_TYPE 类型的 FIELD_DECL 结点中，记录一个表达式，当此表达式非零时，表示当前的 QUAL_UNION_TYPE 被此 FIELD_DECL 填充。

用 DECL_QUALIFIER (node)宏访问。

abstract_origin

在任何 DECL 结点中，此域指向一个原始的（抽象的）符号结点，使得当前的 DECL 结点是抽象的符号名节点的一个实例。此域为空表示当前 DECL 不是任何其它符号名的实例。

DECL_ABSTRACT_ORIGIN 宏访问。

assembler_name

指向一个 IDENTIFIER 结点，表示一个对象在汇编代码中的表示形式，通常此名字和 DECL_NAME 一致。

用 DECL_ASSEMBLER_NAME (node)宏来访问。

section_name

记录段属性中的段名。用于将段名从 decl_attributes 过程传送给 make_function_rtl 和 make_decl_rtl 两过程。

用 DECL_SECTION_NAME (node) 宏访问。

rtl

指向变量值、或函数值对应的 RTL 表达式。

如果定义了 PROMOTED_MODE，则此 RTL 表达式的模式可能不同于 DECL_MODE。在那种情况下，DECL_MODE 包含变量数据类型所对应的模式，而 DECL_RTL 则表示此数据的真正的模式。

用 DECL_RTL (node) 宏访问。

frame_size

此域是一个联合。在 FUNCTION_DECL 中，如果是内联函数（inline），此域表示此函数所需帧大小。用 DECL_FRAME_SIZE (node)宏访问。如果是内部函数（built_in），此域指明是哪一个内部函数。用 DECL_FUNCTION_CODE (node)宏访问。

在其它任何结点中，此域表示数据的对齐要求，用 DECL_ALIGN (node) 宏访问。

saved_insns

此域是一个联合。

在 PARM_DECL 结点中，包括实际用于数据传送的栈或寄存器的 RTL。

用 DECL_INCOMING_RTL (node)宏访问。

在 FUNCTION_DECL 结点中，如果是内联函数，此域是一条保护指令链。

用 DECL_SAVED_INSNS (node)宏访问。

在 FIELD_DECL 结点中，此域指明结构或联合中成员的大小。

用 DECL_FIELD_SIZE (node)宏访问。

vindex

在 FIELD_DECL 结点中，DECL_FCONTEXT 指向一个基类，在此基类中，此 FIELD_DECL 第一次被定义，当输出 C++中的 vfield decl 和 vbase decl 的调试信息时需要上述信息。

用 DECL_FCONTEXT (node) 宏访问。

在 FUNCTION_DECL 结点中，此域有两种使用法：

①在包含 FUNCTION_DECL 的结构输出以前，DECL_VINDEX 可能指向基类中的一个函数，基类中的此函数将作为一个虚函数被 FUNCTION_DECL 所替代。

②当包含 FUNCTION_DECL 的结构输出以后，此域指向一个 INTEGER_CST 结点，此整数结点适合于作为虚函数表的索引。

用 DECL_VINDEX (node)宏来访问。

lang_specific

指向一个结构，此结构的具体内容依赖于所使用的语言。

用 DECL_LANG_SPECIFIC (node) 宏访问。

external_flag

在 VAR_DECL 或 FUNCTION_DECL 中，此域非零表示外部引用：不分配存贮空间，而是指向其在另一个地方的定义。

在 TYPE_DECL 结点中，此域非零表示有关此类型的详细信息没有输出到 stabs 中。相反，

它将产生一个对名字的交叉引用。

用 TYPE_DECL_SUPPRESS_DEBUG (node)宏访问。

nonlocal_flag

在 VAR_DECL、PARM_DECL 和 FUNCTION_DECL 结点中，此域非零表示此变量在嵌套函数中被引用。

在 LABEL_DECL 结点中，如果允许非局部 goto 转移到该标号，则此域非零。

用 DECL_NONLOCAL (node)宏来访问。

regdecl_flag

在 VAR_DECL 和 PARM_DECL 结点中，此域非零表示此变量被说明成“register”。

在 LABEL_DECL 结点中，此域非零表示已产生一个关于由外层跳转到内层该标号的错误信息。

用 DECL_REGISTER (node)宏访问。

在 FIELD_DECL 结点中，此域非零表示此 field 将被位填充。

用 DECL_PACKED (node)宏访问。

inline_flag

在 FUNCTION_DECL 结点中，此域非零表示此函数将嵌入在其被调用处。

用 DECL_INLINE (node)宏访问。

bit_field_flag

在 FIELD_DECL 结点中，此域非零表示它是一个位域，必须以特殊的方式来访问它。用 DECL_BIT_FIELD (node) 宏访问此位域。

在 LABEL_DECL 结点中，此域非零表示此标号在一个符号名层次嵌套结构中被定义，此嵌套结构已恢复了一个栈并且已经退出。

用 DECL_TOO_LATE (node)宏访问。

在 FUNCTION_DECL 结点中，此域非零表示一个内部（built_in）函数。

用 DECL_BUILT_IN (node) 宏访问。

在静态的 VAR_DECL 结点中，如果此变量的空间在 text 段，则此域非零。用 DECL_IN_TEXT_SECTION (node) 宏访问。

virtual_flag

在 VAR_DECL 结点中，此域非零表示变量是一个 vtable。

在 FIELD_DECL 结点中，此域非零表示 vtable pointer。

用 DECL_VIRTUAL_P (node)宏访问。

ignored_flag

在 DECL 结点中，此域非零表示在符号调试中将忽略此说明的名字。

用 DECL_IGNORED_P (node)宏访问。

abstract_flag

在一个给定的 DECL 结点中，此域非零表示此 decl 结点代表一个给定说明的“抽象实例”（例如，内联函数的原始说明）。当产生符号调试信息时，不必产生关于标志为“抽象实例”的结点的任何地址信息，因为实际上，我们并不产生此实例的任何代码，也不给其分配数据空间。

用 DECL_ABSTRACT (node)宏访问。

in_system_header_flag

在 DECL 结点中，此域非零表示因为此 decl 没被使用，所以不产生有关它的警告信息。用 DECL_IN_SYSTEM_HEADER (node) 宏访问。

common_flag

在 DECL 结点中，此域非零表示若该变量无初值，应置于·common 区，否则，若 DECL_INITIAL 不为空且也不是 error_mark_node，则该变量不能放在·common 区。用 DECL_COMMON (node)宏访问。

defer_output

表示此 DECL 的连接状态未知，不能马上输出。

用 DECL_DEFER_OUTPUT (node)宏访问。

transparent_union

用于其类型是联合的 PARM_DECL 结点，说明将以与第一个联合参数传递方式相同的方式来传递此参数。

用 DECL_TRANSPARENT_UNION (node)宏访问。

non_addr_const_p

表明指向此 DECL 的指针不能当成一个地址常量。

no_instrument_function_entry_exit

在 FUNCTION_DECL 中，此域非零表示函数入口和出口 should be instrumented with calls to support routines。

comdat_flag

此域是对后端的一个暗示，由前端设置，非零时表示释放某些数据是安全的。

malloc_flag

在 FUNCTION_DECL 中，此域非零表示将此函数看成 malloc，即意味着它返回一个不是别名的指针。

no_limit_stack

在 FUNCTION_DECL 中，此域非零表示此函数中不能用 limit_stack。

built_in_class

classify which part of the compiler has defined a given builtin function。

pure_flag

在 FUNCTION_DECL 中，此域非零表示此函数是 PURE 函数（类似于 CONST 函数，但能读全局内存）。

pointer_depth

其值往往等于 decl's type 结点中的 TYPE_POINTER_DEPTH，但对函数，此值可能大于 TYPE_POINTER_DEPTH。例如，当函数定义成接收一个 void 类型（depth=1）的参数，但实际调用时传递一个具有 foo**类型(DEPTH=2)的参数时，函数 TYPE 将获得形参的 depth，但函数 DECL 将获得实参的 depth。

non_addressable

用于 FIELD_DECL 中，暗示不能形成此 component 的地址。

User_align

值为 1 时表示此类型根据 aligned 属性的要求来对齐；值为 0 表示按缺省方式对齐。

Uninlinable

在 FUNCTION_DECL 中，此域非零表示此函数不能 inline。

Size_unit

表示以字节为单位的大小。

Saved_tree

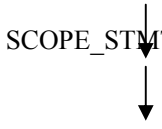
在 FUNCTION_DECL 中，此域用来保存整个函数体的语法树，通常是一个 COMPOUND_STMT。但在 C++中，也可能是 RETURN_INIT、 CTOR_INITIALIZER 或 TRY_BLOCK。

编译器中 saved_tree 的形成过程如下：

1　对每种复合语句，编译器都保存有完整的语法树，如对 FOR 语句，一个树结点的第 0 个操作数表示 FOR 变量的初始化语句；第 1 个操作数记录循环终止判断条件；第 2 个操作数记录条件表达式；第 3 个操作数记录循环体。

2　对一个函数，其保存的语法树结构一般为：

COMPOUND_STMT->exp.operands[0]



SCOPE_STMT->common.chain

Stmt1->common.chain

Stmt2->common.chain

……

3　在 expand_stmt 时，将整棵语法树扩展成 rtx，然后删除此语法树。

Live_range_rtl

Holds an INSN_LIST of all of the live ranges in which the variable has been moved to a possibly different register.

Inlined_fns

List of FUNCTION_DECLs inlined into this function's body.

Pointer_alias_set

Used to indicate an alias set for the memory pointed to by this particular FIELD_DECL, PARM_DECL, or VAR_DECL, which must have pointer (or reference) type.

U2.r

For PARM_DECL, holds an RTL for the stack slot or register where the data was actually passed

U2.f

For FUNCTION_DECL, if it is inline, holds the saved insn chain

U2.t

In a FIELD_DECL, this is the offset, in bits, of the first bit of the field from DECL_FIELD_OFFSET

U1.f

For FUNCTION_DECL, if it is built-in, this identifies which built-in operation it is.

lang_flag_0
lang_flag_1
lang_flag_2
lang_flag_3
lang_flag_4
lang_flag_5
lang_flag_6
lang_flag_7

上述 8 个位留待具体语言使用，用 DECL_LANG_FLAG_i (node)宏访问。（其中 0≤i≤7）。

2. 结点代码

树结构中共有 147(126)种结点代码，分别标识各种结构，下面一一例举：

2.1. 特殊结点代码

ERROR_MARK

标识任何出错的结构。在后续分析过程中，此类结点在任何上下文中被无错误接收，以避免对同一错误产生多个错误信息。除了 TREE_CODE 域外，这类结点不使用其它域。这是一类特殊结点。

IDENTIFIER_NODE

表示一个名字。从内部看，它类似于 STRING_CST 结点。对任一特定名字，只产生一个对应的 IDENTIFIER_NODE。用 get_identifier 访问它，当第一次用 get_identifier 时，创建一个 IDENTIFIER_NODE 结点。这是一类特殊结点。

TREE_LIST

这是一类特殊结点。有 TREE_VALUE 和 TREE_PURPOSE 两个域。这些结点通过 TREE_CHAIN 域相联。链中的元素存在于 TREE_VALUE 域，TREE_PURPOSE 域只在某些情况下使用。

TREE_VEC

这是一类特殊结点，包含一个结点数组。

BLOCK

代表一个词法块，表示一个符号名层次块。

2.2. 类型结点代码

VOID_TYPE

属于 TYPE 结点类，表示 C 语言中的 void 类型。

INTEGER_TYPE

属于 TYPE 结点类，表示所有语言中的整型，包括 C 中的字符型，也用于其它离散类型的子范围类型。包含 TYPE_MIN_VALUE、TYPE_MAX_VALUE 和 TYPE_PRECISION 三个域。

对 PASCAL 中的子类型，TREE_TYPE 将指向其超类型（另一个 INTEGER_TYPE、ENUMERAL_TYPE 或 BOOLEAN_TYPE）。否则 TREE_TYPE 为空。

REAL_TYPE

属于 TYPE 结点类，表示 C 语言中的浮点和双精度类型，不同的浮点类型由机器模式、TYPE_SIZE 和 TYPE_PRECISION 区分。

COMPLEX_TYPE

属于 TYPE 结点类，表示复数类型，TREE_TYPE 域指向实部和虚部的类型。

VECTOR_TYPE

表示向量类型，类型的 TREE_TYPE 域指向向量元素的数据类型。

ENUMERAL_TYPE

属于 TYPE 结点类，表示 C 中的枚举类型，类似于 INTEGER_TYPE 结点。表示枚举值的符号用 CONST_DECL 结点定义，但 TREE_TYPE 并不指向它们，这类结点的 TREE_VALUE 域指向一由 TREE_LIST 结点组成的链表，其中每个元素的 TREE_PURPOSE 域表示枚举名而 TREE_VALUE 域表示其值（为 INTEGER_CST 结点）。

如果还没有定义枚举名，但已有对它的前向引用，则其 TYPE_SIZE 为空。标签名存在 TYPE_NAME 域。如果在以后定义了此类型，则这些规定的域将被填充。对 RECORD_TYPE、UNION_TYPE 和 QUAL_UNION_TYPE 的前向引用也作相似处理。

BOOLEAN_TYPE

属 TYPE 结点类。表示 PASCAL 中的布尔类型，只有 true 和 false 两个值，不需使用特殊域

CHAR_TYPE

属 TYPE 结点类，表示 PASCAL 中的 CHAR 类型，不在 C 中使用。不需使用特殊域。

POINTER_TYPE

属于 TYPE 结点类，所有指针类型都有此代码。TREE_TYPE 指向指针所指类型。

OFFSET_TYPE

属于 TYPE 结点类。表示相对于某一对象的一个偏移指针。TREE_TYPE 域指向在偏移处的对象的类型，TYPE_OFFSET_BASETYPE 指向相对结点的类型。

REFERENCE_TYPE

属于 TYPE 结点类，类似于指针类型，与指针类型不同的是，它自动与所指向值一致，用

于 C++。

METHOD_TYPE

属于 TYPE 结点类，表示一种函数类型。此函数为其自己占据一个额外的参数作为第一个参数，且此参数不出现在程序所定义的参数链中。

TREE_TYPE 指向返回类型，TYPE_METHOD_BASETYPE 指向额外参数的类型。

TYPE_ARG_TYPES 表示真正的参数链，此链中包含自身隐含的第一个参数。

FILE_TYPE

属于 TYPE 结点类，在 PASCAL 中使用，具体内客现在还没定。

ARRAY_TYPE

属于 TYPE 结点类，表示 C 或 PASCAL 中的数组类型。此类型有几个特殊的域。

TREE_TYPE 指向数组元素的类型。

TYPE_DOMAIN 指向用于检索的类型，其值的范围说明了数组的长度。

TYPE_POINTER_TO（TREE_TYPE (array_type)）通常不为空。

TYPE_STRING_FLAG 指明一个字符串（而不是一个字符数组）。

TYPE_SEP 指向从一个元素到下一个元素的单位数表达式。

TYPE_SEP_UNIT 指明前一单位中的位数。

SET_TYPE

属于 TYPE 结点类，表示 PASCAL 中的集合类型。

RECORD_TYPE

属于 TYPE 结点类，表示 C 中的结构、PASCAL 中的记录，特殊的域有：

TYPE_FIELDS 链, 此链以 FIELD_DECLS 结点形式连接结构中的各个域。

其前向引用处理同枚举类型。

UNION_TYPE

属于 TYPE 结点类，表示 C 中的联合，类似于结构类型但域间的偏移为 0，其前向引用处理同枚举类型。

QUAL_UNION_TYPE

属于 TYPE 结点类，类似于 UNION_TYPE，但每一个 FIELD_DECL 中的 DECL_QUALIFIER 域中的表达式决定了联合中的内容，第一个 DECL_QUALIFIER 表达式为真的域将是整个联合的值。

FUNCTION_TYPE

属于 TYPE 结点类，表示函数类型，特殊域有

TREE_TYPE 指向返回值类型

TYPE_ARG_TYPES 指向参数类型表，由 TREE_LIST 结点组成。

若语言中过程与函数有差别， 表示过程的类型也具有 FUNCTION_TYPE 代码，但 TREE_TYPE 为空。

LANG_TYPE

属于 TYPE 结点类，是与语言相关的一种类型，其具体意义依赖于语言前端，layout_type 过程不知道怎样处理这种类型，所以前端必须手工处理。

2.3. 表达式结点代码

2.3.1. 表示常数的树结点编码

INTEGER_CST

表示整常数，其内容在 TREE_INT_CST_LOW 和 TREE_INT_CST_HIGH 域中，每个域占 32 位，所以能表示 64 位整常数。

PASCAL 语言中的 char 类型的常数也表示为 INTEGER_CST，PASCAL 和 C 中的指针常量如 NIL 和 NULL 也表示为 INTEGER_CST。C 中的‘(int *)’也产生一个 INTEGER_CST。

REAL_CST

表示实常数，其内容在 TREE_REAL_CST 域中。

COMPLEX_CST

表示复型常数，其内容在 TREE_REALPART 和 TREE_IMAGPART 域中。

上述两个域指向其它常数结点。

VECTOR_CST

表示向量常数，其内容在 TREE_VECTOR_CST_ELTS 域中。

STRING_CST

表示字符串常数，其内容在 TREE_STRING_LENGTH 和 TREE_STRING_POINTER 域中。

2.3.2. 表示名字的树结点编码

所有对名字的引用皆表示为…_DECL 结点，同一层次的名字通过 TREE_CHAIN 域串起来。每个 DECL 有一个 DECL_NAME 域指向一个 IDENTIFIER 结点（对有些名字，如标号，DECL_NAME 为空）。

FUNCTION_DECL

表示一个函数名，有四个特殊域。

DECL_ARGUMENTS 指向一串表示参数的 PARM_DECL。

DECL_RESULT 指向表示函数返回值的 RESULT_DECL。

DECL_RESULT_TYPE 指向函数返回的类型，这通常和 DECL_RESULT 的类型一致。但⑴它可能是一种比 DECL_RESULT 类型宽的整型。⑵为方便内联函数，它可能在此函数编译完成时仍有效。

DECL_FUNCTION_CODE 是一个代码数，非零表示内部函数。其值是内部函数的一个枚举值，此值代表相应的内部函数。

LABEL_DECL

表示一个标号名。

CONST_DECL

表示一个常数名。

TYPE_DECL

表示一个类型名。

VAR_DECL

表示一个变量名。

PARM_DECL

表示一个哑参名。有一个特殊域：

DECL_ARG_TYPE 指向实际传递的参数类型，此类型有可能与程序中定义的类型不一致。

RESUCL_DECL

表示返回值名。

FIELD_DECL

表示域名。

NAMESPACE_DECL

表示名字空间说明，名字空间出现在其它_DECL 的 DECL_CONTEXT 域中，提供一个名字的层次结构。

2.3.3. 表示内存引用的结点编码

COMPONENT_REF

表示对结构或联合中的一个分量的引用。参数 0 表示结构或联合，参数 1 表示一个域（一个 FIELD_DECL 结点）。

BIT_FIELD_REF

表示访问一个对象内的一组位域。类似于 COMPONENT_REF，与 OMPONENT_REF 不同的是位域的位置是明确给出的，而不是通过 FIELD_DECL 给出。参数 0 是结构或联合。参数 1 是一棵树，给出被访问的位数。参数 2 是一棵树，给出访问的第 1 位的位置。被访问的域可以是有符号的，也可以是无符号的，由 TREE_UNSIGNED 域决定。

INDIRECT_REF

表示 C 语言中的一元操作‘*’或 PASCAL 语言中的‘ ^ ’，有一个关于指针的表达式参数。

BUFFER_REF

PASCAL 语言中对文件的‘ ^ ’操作，有一个关于文件的表达式参数。

ARRAY_REF

数组访问，参数 0 表示数组，参数 1 是一串关于索引下标的 TREE_LIST 结点。

ARRAY_RANGE_REF

类似于 ARRAY_REF，不同之处在于其表示一个数组片段(slice) 。参数 0 表示数组，参数 1 表示索引起始点，片段的大小从表达式的类型中获取。

VTABLE_REF

Vtable 索引，此数据主要用于 vtable 的垃圾收集。

操作数 0 表示一个数组引用或等价的表达式。

操作数 1 表示 vtable 的基址 (必须是一个 var_decl)。

操作数 2 表示在 vtable 中的索引(必须是一个 integer_cst)。

CONSTRUCTOR

返回一组由指定分量组成的集合值。

在 C 中，这只用于结构和数组初始化。

参数 1 是一个指向 RTL 的指针，且仅对常数 CONSTRUCTOR 有效。

参数 2 是一串由 TREE_LIST 结点组成的分量值。

2.3.4. 表示表达式的结点编码

COMPOUND_EXPR

包含待计算的两个表达式，第一个表达式的值被忽略，第二个表达式的值被利用。

MODIFY_EXPR

赋值表达式，操作数 0 表示左值，操作数 1 表示右值。

INIT_EXPR

初始化表达式，操作数 0 表示被初始化的变量，操作数 1 表示初始值。

TARGET_EXPR

操作数 0 表示一个初始化目标，操作数 1 表示此目标的初值，如果有的话操作数 2 表示此结点的 cleanup。

COND_EXPR

条件表达式，操作数 0 表示条件，操作数 1 表示 THEN 值，操作数 2 表示 ELSE 值。

BIND_EXPR

用于去分配时，其值需要被清除的表达式，C++使用。

局部变量说明，包括 RTL 生成和空间分配。

操作数 0 是变量表的 VAR_DECL 链。

操作数 1 是要计算的表达式（表达式中含变量表中的变量）。操作数 1 的值即 BIND_EXPR 的值。

操作数 2 是一个与此层次相对应的 BLOCK，用于调试。如果此 BIND_EXPR 被 扩展，则置 BLOCK 结点中的 TREE_USED 标志。

BIND_EXPR 不负责将这些变量的信息通知扫描器。如果表达式串来自于一个输入文件，则产生 BIND_EXPR 的代码负责将这些变量的信息告诉扫描器。

一旦 BIND_EXPR 被扩展，则 TREE_USED 标志位为 1。如果调试需用到 BIND_EXPR，但它又未被扩展，则手工设置 TREE_USED 域。

为使 BIND_EXPR 全程可见，产生它的代码必须将它作为一个 subblock 而存入函数的 BLOCK 结点中。

CALL_EXPR

函数调用。操作数 0 指向函数，操作数 1 是一个参数链（由一串 TREE_LIST 结点组成）。无操作数 2，操作数 2 的域被 CALL_EXPR_RTL 宏使用。

METHOD_CALL_EXPR

调用一个方法。操作数 0 是调用的方法（其类型为 METHOD_TYPE）

操作数 1 是代表方法自身的表达式。操作数 2 是一串参数。

WITH_CLEANUP_EXPR

指定一个要伴随它对应的清除动作同时计算的值。操作数 0 表示其值需清除的表达式，操作数 1 是一个最终代表该值的 RTL_EXPR，操作数 2 是此对象的清除表达式。清除表达式中含 RTL_EXPR，表示此表达式怎样作用于要清除的值。清除动作是由第一个封闭 CLEANUP_POINT_EXPR（如果存在的话）来执行的，否则，若需要时，调用者有责任手工调用 EXPAND_CLEANUPS_TO 函数。

CLEANUP_POINT_EXPR

定义一个清除点。操作数 0 是确保有 cleanups 的被清除的表达式。

PLACEHOLDER_EXPR

代表一个记录，当估算此表达式时，提供一个 WITH_RECORD_EXPR 来替代此记录 。表达式的类型用于寻找替代此表达式的记录。

WITH_RECORD_EXPR

提供一个表达式，此表达式引用一个用以替代 PLACEHOLDER_EXPR 的记录。操作数 1 包含用来替代的记录，其类型与操作数 0 中的 PLACEHOLDER_EXPR 的类型一致。

上述两代码用于有“自引用”类型的语言, 如 ADA 语言。

2.3.5. 表示简单算术运算的结点代码

算术运算的操作数必须有相同的机器模式，结果具有同样的机器模式。

PLUS_EXPR

加操作，有两个操作数。

MINUS_EXPR

减操作，有两个操作数。

MULT_EXPR

乘操作。

TRUNC_DIV_EXP

整除法，其结果的尾数部分被截除。理论上，操作数可为实型，但目前还不支持。其结果通常为定点数，和操作数具有相同类型。

CEIL_DIV_EXPR

整除法，其结果向上取整。

FLOOR_DIV_EXPR

整除法，其结果向下取整。

ROUND_DIV_EXPR

整除法，其结果做四舍五入。

TRUNC_MOD_EXPR

对应于 TRUNC_DIV_EXPR 的求余操作。

CEIL_MOD_EXPR

对应于 CEIL_DIV_EXP 的求余操作。

FLOOR_MOD_EXPR

对应于 FLOOR_DIV_EXPR 的求余操作

ROUND_MOD_EXP

对应于 ROUND_DIV_EXP 的求余操作。

RDIV_EXPR

实数除法，所得结果为实数。两个操作数必须具有相同的类型。理论上，操作数可为整数，但目前仅支持实型操作数。结果的类型必须与操作数类型一致。

EXACT_DIV_EXPR

不需四舍五入的除法，用于 C 中的指针减操作。

上述各操作都有两个操作数。

下面是四种不同截断方式的将实数转换为定点数的操作。CONVERT_EXPR 也可用于将一个实数转换成整数。

FIX_TRUNC_EXPR
FIX_CEIL_EXPR
FIX_FLOOR_EXPR
FIX_ROUND_EXPR

其转换含义同四种整数除法。上述四种转换操作只有 1 个操作数。

FLOAT_EXPR

将整数转为实数，只有 1 个操作数。

NEGATE_EXPR

一元求反操作。结果与操作数具有同一类型。

MIN_EXPR

求最小值操作，有两个操作数。

MAX_EXPR

求最大值操作，有两个操作数。

ABS_EXPR

求绝对值操作，有一个操作数。

FFS_EXPR

FFS 操作，有一个操作数。

LSHIFT_EXPR
RSHIFT_EXPR

分别表示左移和右移操作，有两个操作数。如果操作数为无符号类型，则为逻辑移位；否则为算术移位。第二个操作数是需移动的位数，必须是 SImode，结果的模式与第一个操作数一致。

LROTATE_EXPR
RROTATE_EXPR

分别表示循环左移和循环右移，具体说明同上。

BIT_IOR_EXPR

位同或操作，即 A|B（A、B 代表两个操作）。

BIT_XOR_EXPR

位异或操作，即 A ^ B。

BIT_AND_EXPR

位与操作，即 A&B。

BIT_ANDTC_EXPR

位与非操作，即 A& (～B)。

BIT_NOT_EXPR

位非操作。

TRUTH_ANDIF_EXPR

TRUTH_ORIT_EXPR

表示逻辑与和逻辑或操作。如果能从第一个操作数的值获得些表达式的值，则不计算第二个操作数的值。

TRUTH_AND_EXPR

TRUTH_OR_EXPR

TRUTH_XOR_EXPR

TRUTH_NOT_EXPR

分别表示逻辑与、或、异或和非操作，不管是否需要第二个操作数的值，总是计算第二个操作数。

这些表达式除 TRUTH_NOT_EXPR 只有一个操作数外，其余都有两个操作数，操作数为布尔值或只取零或非零的整数值。

LT_EXPR

LE_EXPR

GT_EXPR

GE_EXPR

EQ_EXPR

NE_EXPR

分别表示两个操作数之间的小于、小于等于、大于、大于等于、等于和不等于操作。

其中 EQ_EXPR 和 NE_EXPR 允许对任何类型进行操作，其它只允许对整型、指针、枚举或实型进行操作。所有情况下，操作数都必须具有相同类型。结果是布尔型。

ORDERED_EXPR

UNORDERED_EXPR

Additional relational operators for floating point unordered

UNEQ_EXPR

UNGE_EXPR

UNGT_EXPR

UNLT_EXPR

UNLE_EXPR

These are equivalent to unordered or These are equivalent to unordered

IN_EXPR

SET_LE_EXPR

CARD_EXPR

RANGE_EXPR

对 PASCAL 中的集合进行的操作，目前还没使用。

CONVERT_EXPR

对一个值的类型进行转换。所有的转换，包括隐式的转换，都必须表示成 CONVERT_EXPR，只有一个操作数。

NOP_EXPR

表示一个不产生任何代码的转换。只有一个操作数。

NON_LVALUE_EXPR

表示一个与其参数相同的值，但保证此值不为左值。只有一个操作数。

VIEW_CONVERT_EXPR

表示将具有某种类型的对象看成具有另一种类型。此类结点所表示的含义对应于 ADA 语言中的"Unchecked Conversion"，大致对应于 C 语言中的*(type2 *)&X 操作。其唯一的操作数是将被看成具有另一种类型的那个值。如果输入数和表达式的类型大小不同，则此操作无定义（It is undefined if the type of the input and of the expression have different sizes）。此结点代码也可以作为 MODIFY_EXPR 的 LHS，此时并没有真正的数据赋值动作发生。在这种情况下应该设置 TREE_ADDRESSABLE。

SAVE_EXPR

表示计算一次但多次用到的表达式。第一个操作数即是此表达式；第二个操作数指向一个函数名，其中 SAVE_EXPR 在此函数中产生；第三个操作数为表达式对应的 RTL，只有当此表达式被计算以后，RTL 域才不为空。

UNSAVE_EXPR

UNSAVE_EXPR 表达式的操作数 0 代表 unsave 的那个值。我们可以将所有的_EXPR 结点如 TARGET_EXPRs、 SAVE_EXPRs、 CALL_EXPRs 和 RTL_EXPRs 等这些仅估值一次（不允许多次估值）的表达式重新设置为 UNSAVE_EXPR，使得对这些表达式的一次新的 expand_expr 调用能导致对这些表达式的重新估值。当我们想在不同的地方重用一棵树但有必须重新扩展时，此表达式非常有用。

RTL_EXPR

表示一个其 RTL 已被扩展成一个序列，且当表达式被扩展时，应当流出该序列的一个表达式。

第一个操作数指向将被流出的 RTL，它是 insn 序列中的第一项。 第二个操作指向表示结果的 RTL 表达式。

ADDR_EXPR

C 中的 & 操作，其值是操作数值所在单位的地址，操作数可具有任一模式，结果模式为 Pmode。

REFERENCE_EXPR

表示一个非左值引用或指向一个对象的指针。

ENTRY_VALUE_EXPR

仅在那些需要静态链的语言中使用。操作数是一个函数常量；结果是一具有 Epmode 的函数变量值。

FDESC_EXPR

操作数 0 函数常量，结果是 part N of a function descriptor of type ptr_mode

COMPLEX_EXPR

给定两个相同类型的实型或整型操作数，返回一个具有相应复数类型的复数值。

CONJ_EXPR

取操作数的复共轭，仅对复类型操作，值与操作数具有相同的类型。

REALPART_EXPR

IMAGPART_EXPR

分别表示复数的实部和虚部，只对复型数据进行操作。

PREDECREMENT_EXPR

PREINCREMENT_EXPR

POSTDECREMENT_EXPR

POSTINCREMENT_EXPR

分别对应于 C 中的++和－－操作。有两个操作数，第二个操作数说明每次增减的程度，对指针而言，第二个操作数为所指对象的大小。

VA_ARG_EXPR

用于实现 `va_arg'。

TRY_CATCH_EXPR

对操作数 1 进行估值，当且仅当在此估值过程中 throw 一个异常时，才对操作数 2 进行估值。这与 WITH_CLEANUP_EXPR 不同的地方在于除非 throw 一个异常，否则永远不对操作数 2 进行估值。

TRY_FINALLY_EXPR

对操作数 1 进行估值，操作数 2 是一个 cleanup 表达式，在从此表达式退出（正常、异常或跳出）之前被估值。此操作类似于 CLEANUP_POINT_EXPR 和 WITH_CLEANUP_EXPR 的结合，但 CLEANUP_POINT_EXPR 和 WITH_CLEANUP_EXPR 常常将 cleanup 表达式拷贝到所需的地。而 TRY_FINALLY_EXPR 则产生一个跳转，转到一个 cleanup 过程。当 cleanup 是当前函数中的实际语句（如用户想在此设置断点）时，应该使用 TRY_FINALLY_EXPR。

GOTO_SUBROUTINE_EXPR

用于实现 TRY_FINALLY_EXPR 中的 cleanups。它由 expand_expr 而非前端产生。操作数 0 是代表我们将要调用的过程的开始处的 rtx，操作数 1 代表存储过程返回地址的变量的 rtx。

下面这些表达式都无有用的值，但都有副作用。

LABEL_EXPR

封装成语句的一个标号定义，唯一的一个操作数为 LABEL_DECL 结点。此表达式的类型必须为空，其值被忽略。

GOTO_EXPR

表示 GOTO 语句。唯一的一个操作数为 LABEL_DECL 结点。表达式的类型必须为空，其值被忽略。

RETURN_EXPR

表示 return 语句，对唯一的一个操作数进行估值，然后从当前函数返回。表达式的类型必须为空，其值被忽略。

EXIT_EXPR

表示从内层循环中条件退出。唯一的一个操作数为退出的条件。表达式的类型必须为空，其值被忽略。

LOOP_EXPR

表示一个循环。唯一的一个操作数表示循环的体。此表达式必须包含 EXIT_EXPR，否则，表示它是一个无穷循环。

表达式的类型必须为空，其值被忽略。

LABELED_BLOCK_EXPR

表示一个带标号的块，操作数 0 是标示块结束的标号，操作数 1 表示块。

EXIT_BLOCK_EXPR

表示退出一个带标号的块，可能会返回一个值。操作数 0 是一个欲退出的 LABELED_BLOCK_EXPR，操作数 1 是返回值，返回值可能为空。

EXPR_WITH_FILE_LOCATION

给一个树结点（通常是一个表达式）注释上源程序位置信息：文件名字(EXPR_WFL_FILENAME)；行数(EXPR_WFL_LINENO)和列数(EXPR_WFL_COLNO)。其扩展动作同 EXPR_WFL_NODE。如果有 EXPR_WFL_EMIT_LINE_NOTE，则必须首先 emit 一个行注释信息。第三个操作数仅在 JAVA 前端中使用。

SWITCH_EXPR

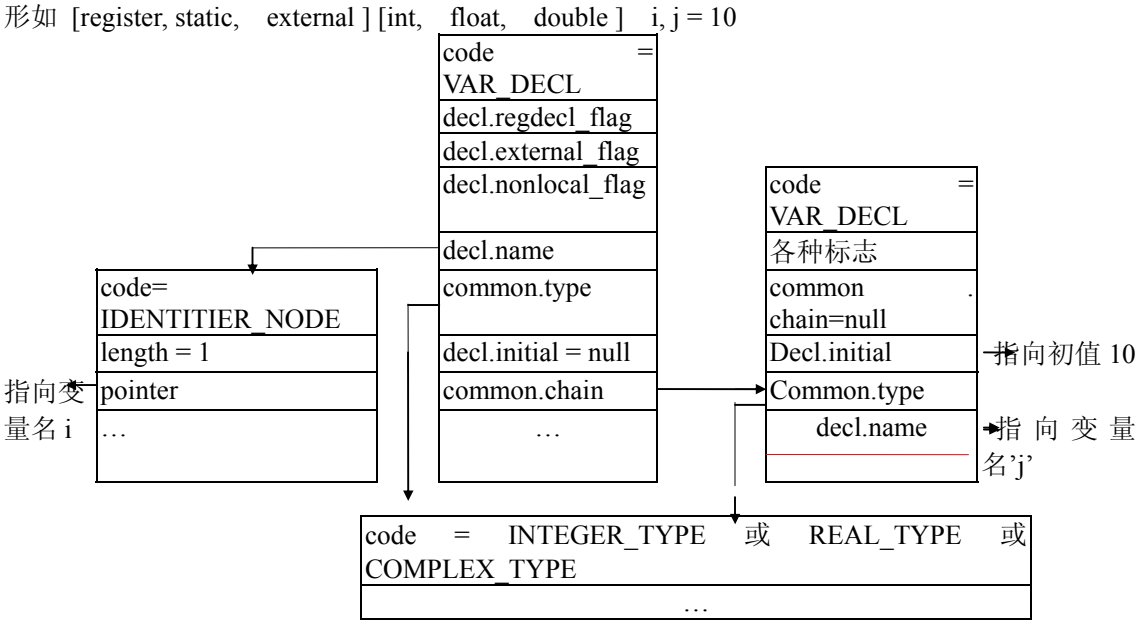
Switch 表达式，操作数 0 是用于执行分枝的表达式，操作数 1 包含 CASE 值，其组织方式和前端实现相关。

EXC_PTR_EXPR

The exception object from the runtime

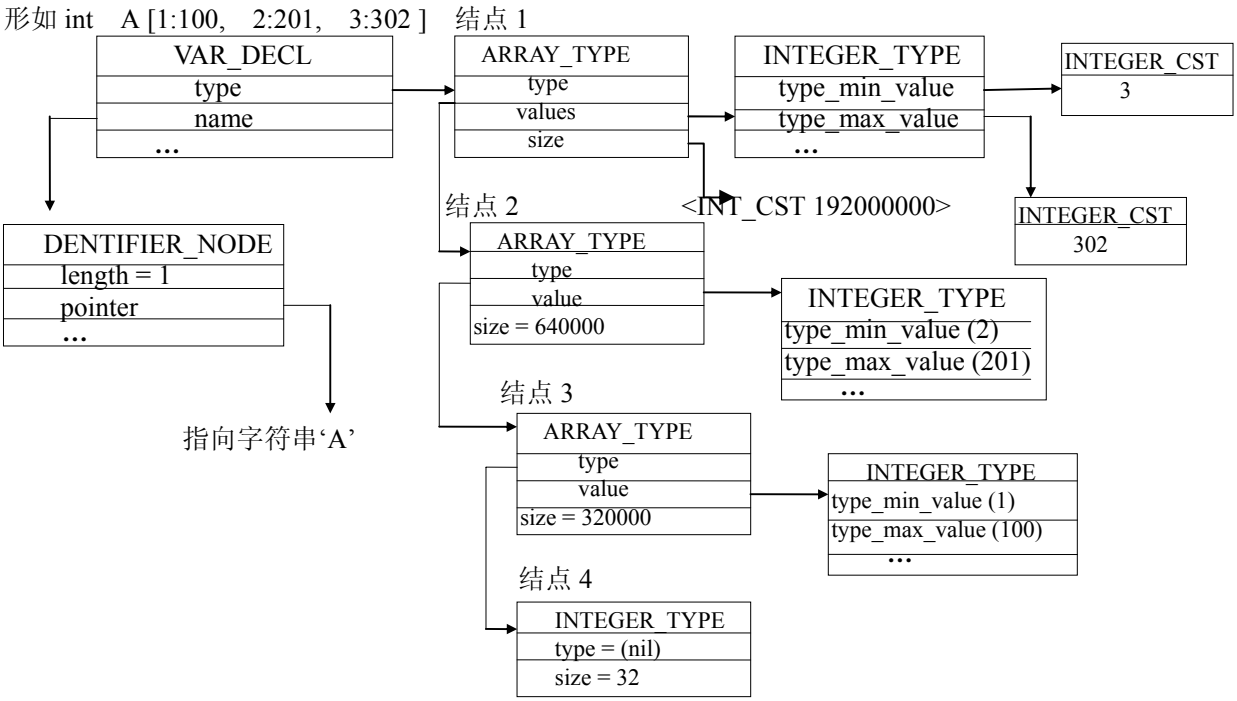
3. 各种语句对应的语法树

3.1. 变量说明



图一变量说明

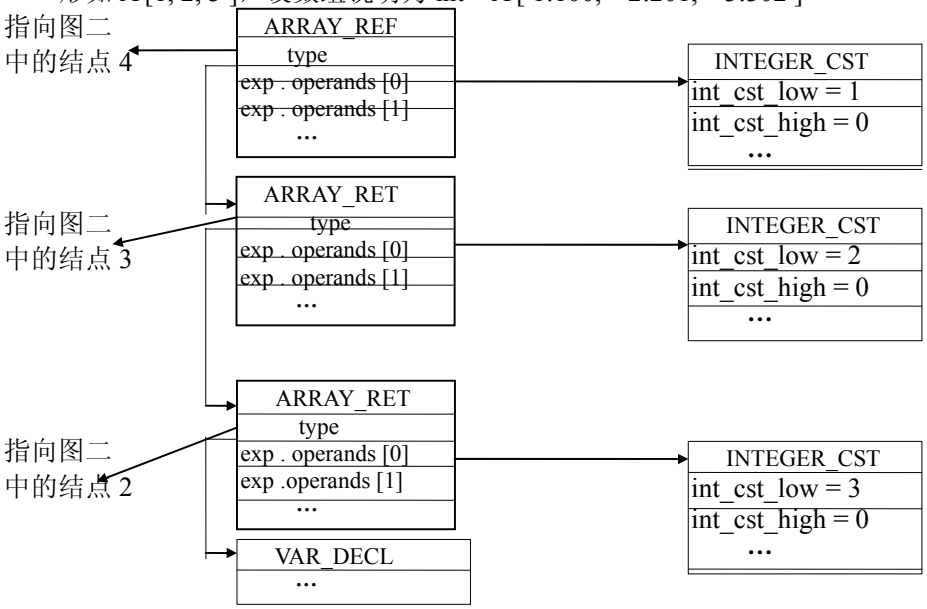
3.2. 数组说明



图二数组说明

3.3. 数组元素引用

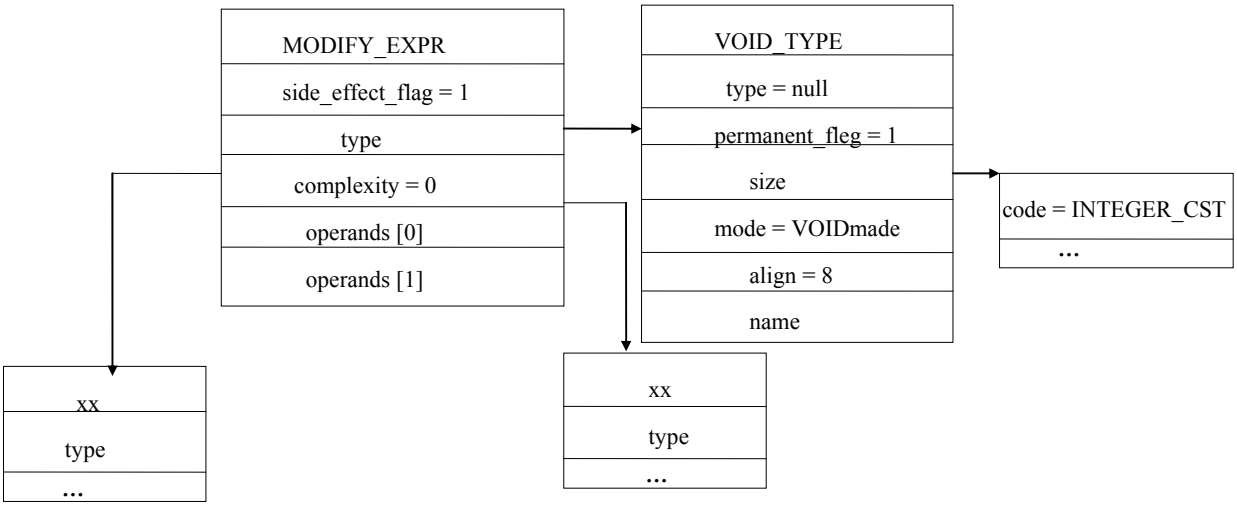
形如 A [1, 2, 3]，设数组说明为 int　A [1:100,　2:201,　3:302]



图三　　数组元素引用

3.4. 赋值语句

形如 A=B，其中 A、B 可为变量名，数组元素引用，表达式常量等。



图四　　赋值语句

上图中的 xx 代表一个变化的结点代码。若 A 或 B 为变量，则 xx = VAR_DECL 且树中对应的 decl 各域被填充)，common . type 指向变量的类型。若 A 或 B 为数组引用，则 xx = ARRAY_REF，

common . type 指向数组元素类型，若 A 或 B 为常数，则 xx = INTEGER_CST、REAL_CST 等，common . type 指向常数的类型。
若 A 或 B 表达式，则 xx 为相应的算术表达式代码。

3.5. 过程（函数）调用

形如call　sub (a1,　a2,　a3)　　(Fortran 形式)

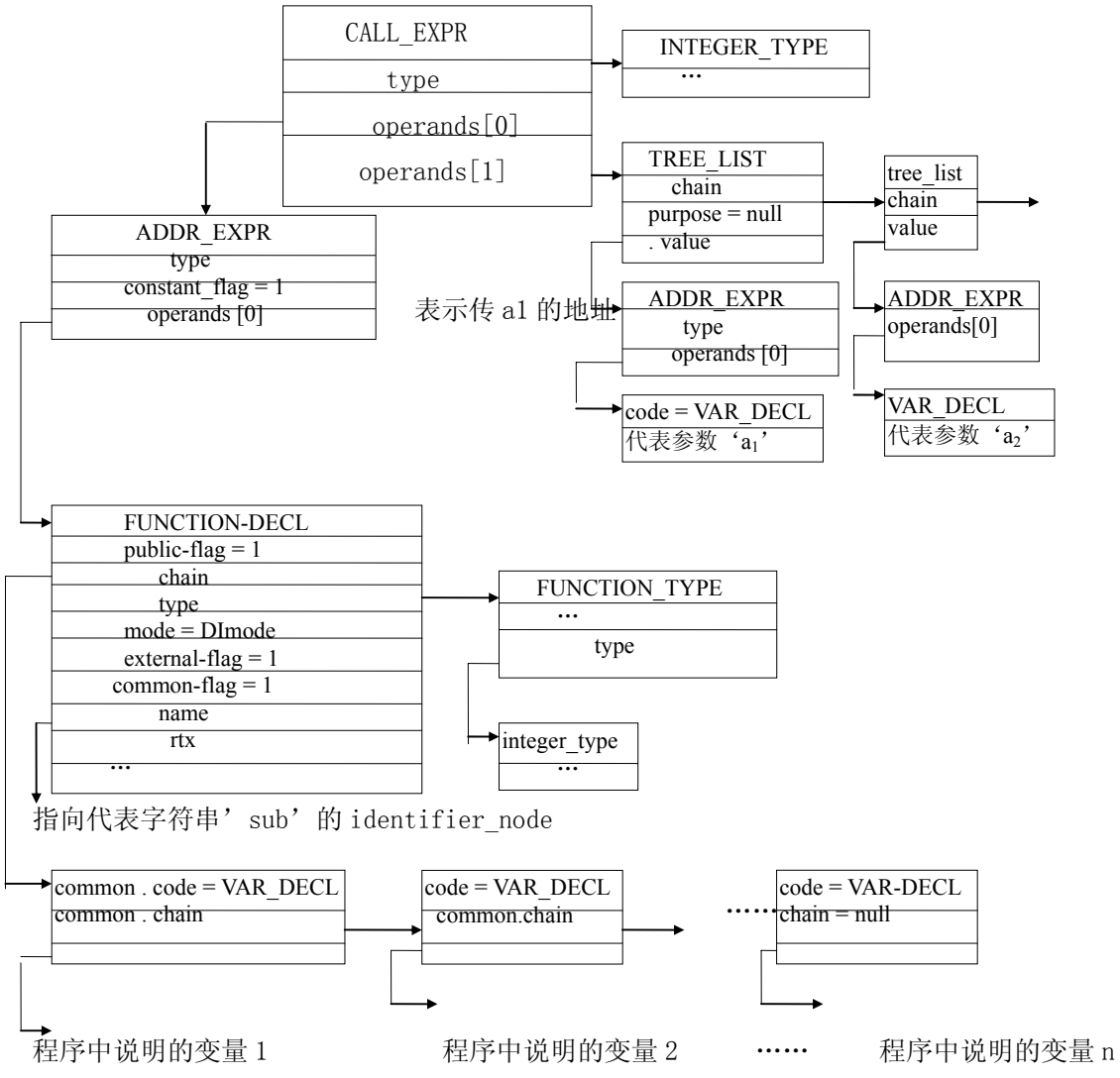


图　五　　过程调用

3.6. 条件表达式

形如exp1 ? exp2 : exp3

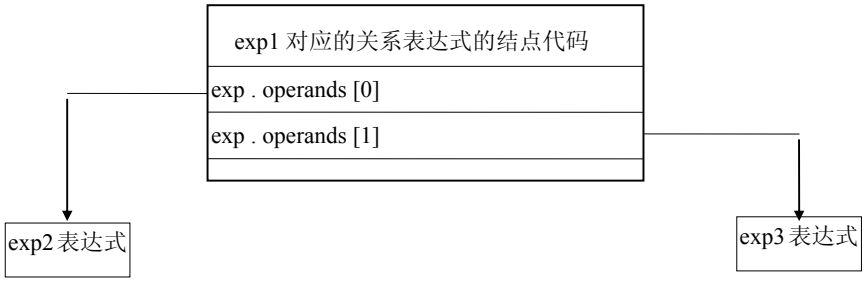


图 六 条件表达式

4. 与 C 和 C++相关的树结点介绍

SRCLOC
记住源代码位置的一个结点。
SIZEOF_EXPR
ARROW_EXPR
ALIGNOF_EXPR
EXPR_STMT
表示一个表达式语句，用 EXPR_STMT_EXPR 来获取表达式。
COMPOUND_STMT
表示一个由大括符括起来的块，操作数为 COMPOUND_BODY。
DECL_STMT
表示局部说明语句，操作数为 DECL_STMT_DECL。
IF_STMT
表示 IF 语句，操作数分别为 IF_COND、THEN_CLAUSE 和 ELSE_CLAUSE。
FOR_STMT
表示 for 语句，操作数分别为 FOR_INIT_STMT、FOR_COND、FOR_EXPR 和 FOR_BODY。
WHILE_STMT
表示 while 语句，操作数分别为 WHILE_COND 和 WHILE_BODY。
DO_STMT
表示 do 语句，操作数分别为 DO_BODY 和 DO_COND。
RETURN_STMT
表示 return 语句，操作数为 RETURN_EXPR。
BREAK_STMT
表示 break 语句。
CONTINUE_STMT
表示 continue 语句。
SWITCH_STMT
表示 switch 语句，操作数分别为 SWITCH_COND、SWITCH_BODY 和 SWITCH_TYPE。

GOTO_STMT

表示 goto 语句，操作数为 GOTO_DESTINATION。

LABEL_STMT

表示 label 语句，操作数为 LABEL_DECL，可通过宏 LABEL_STMT_LABEL 来访问

ASM_STMT

表示一条 inline 汇编语句。

SCOPE_STMT

标示一个 scope 的开始或结束。如果 SCOPE_BEGIN_P holds，则表示 scope 的开始，如果 SCOPE_END_P holds，则表示 scope 的结束。如果 SCOPE_NULLIFIED_P holds ，则表示此 scope 中没有变量。SCOPE_STMT_BLOCK 表示包含此 scope 中定义的变量的 BLOCK。

FILE_STMT

标示一个 spot where a function changes files。它没有其它语义，FILE_STMT_FILENAME 表示文件的名字。

CASE_LABEL

表示 CASE 中的标号，操作数分别为 CASE_LOW 和 CASE_HIGH，如果 CASE_LOW 为 NULL_TREE，表示标号为 default，如果 CASE_HIGH 为 NULL_TREE，表示标号是一个正常的 case 标号。CASE_LABEL_DECL 代表此结点的 LABEL_DECL。

STMT_EXPR

表示一个语句表达式，STMT_EXPR_STMT 表示由表达式给出的语句。

COMPOUND_LITERAL_EXPR

表示 C99 compound literal， COMPOUND_LITERAL_EXPR_DECL_STMT is the a DECL_STMT containing the decl for the anonymous object represented by the COMPOUND_LITERAL; the DECL_INITIAL of that decl is the CONSTRUCTOR that initializes the compound literal.

CLEANUP_STMT

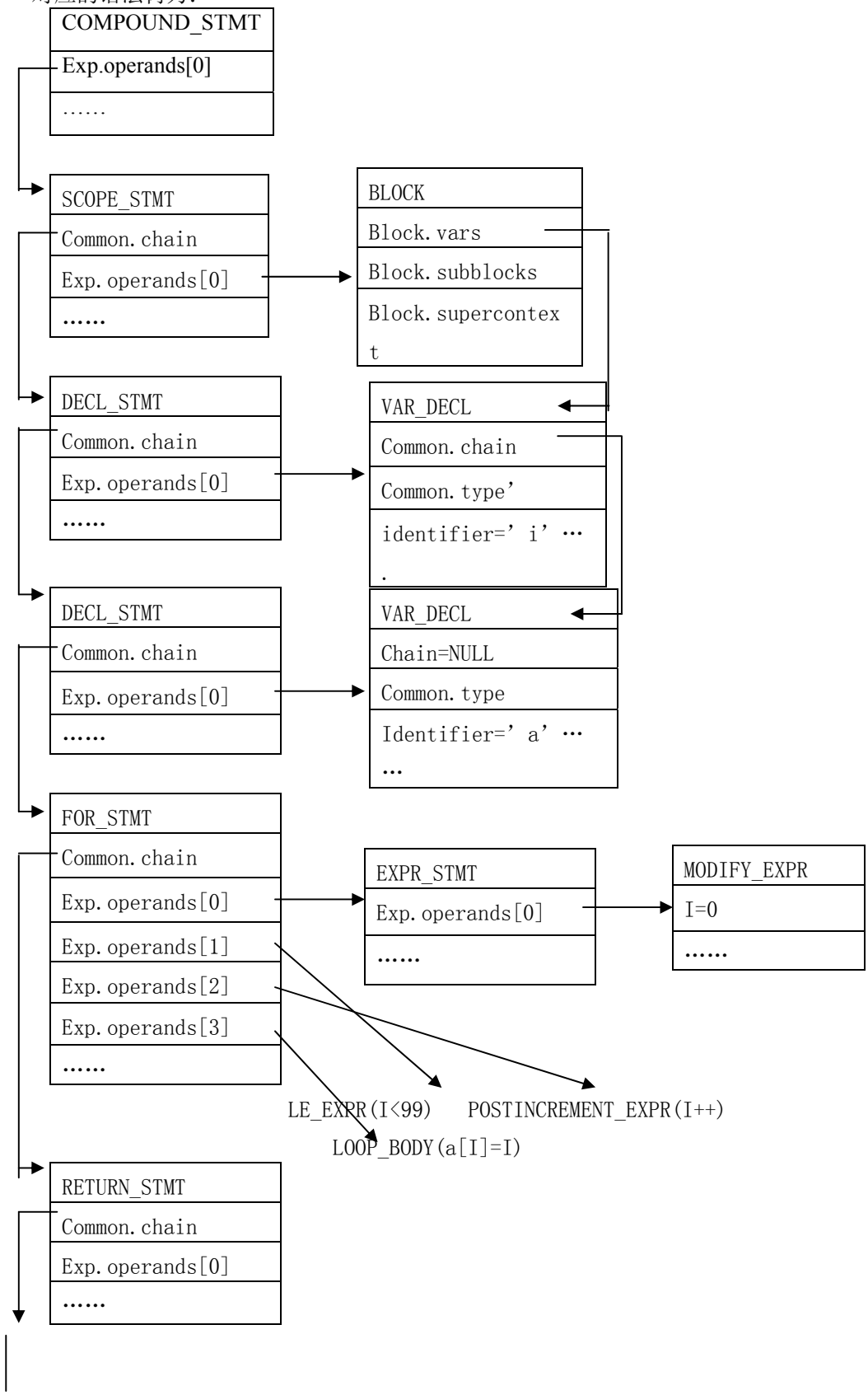
A CLEANUP_STMT marks the point at which a declaration is fully constructed. If, after this point, the CLEANUP_DECL goes out of scope, the CLEANUP_EXPR must be run.

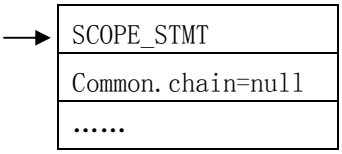
5. 举例

程序代码如下所示：

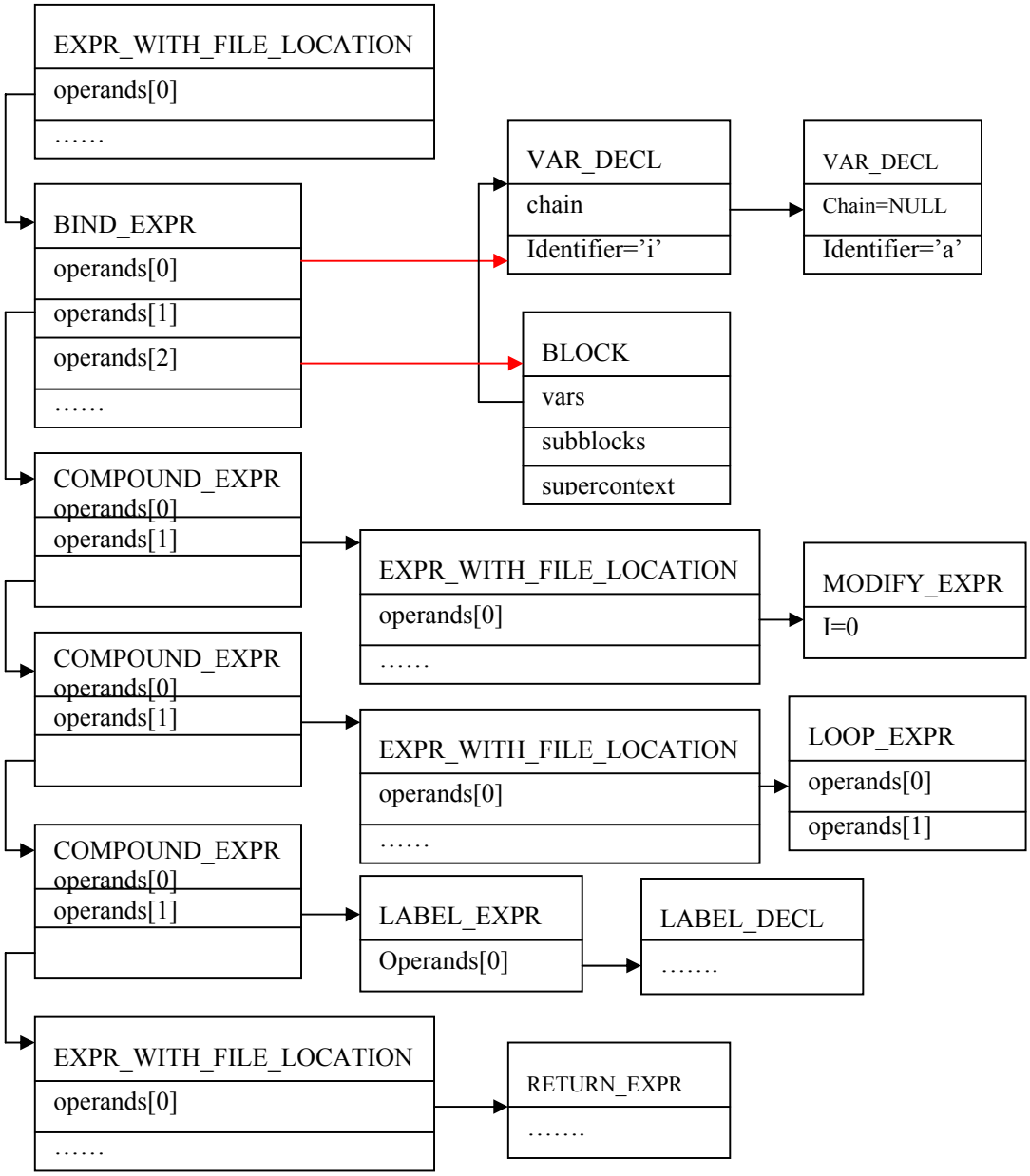
```
Int fun()
{
    int I,a[100];
    for(I=0;I<99;I++)
        a[I]=I;
    return I;
}
```

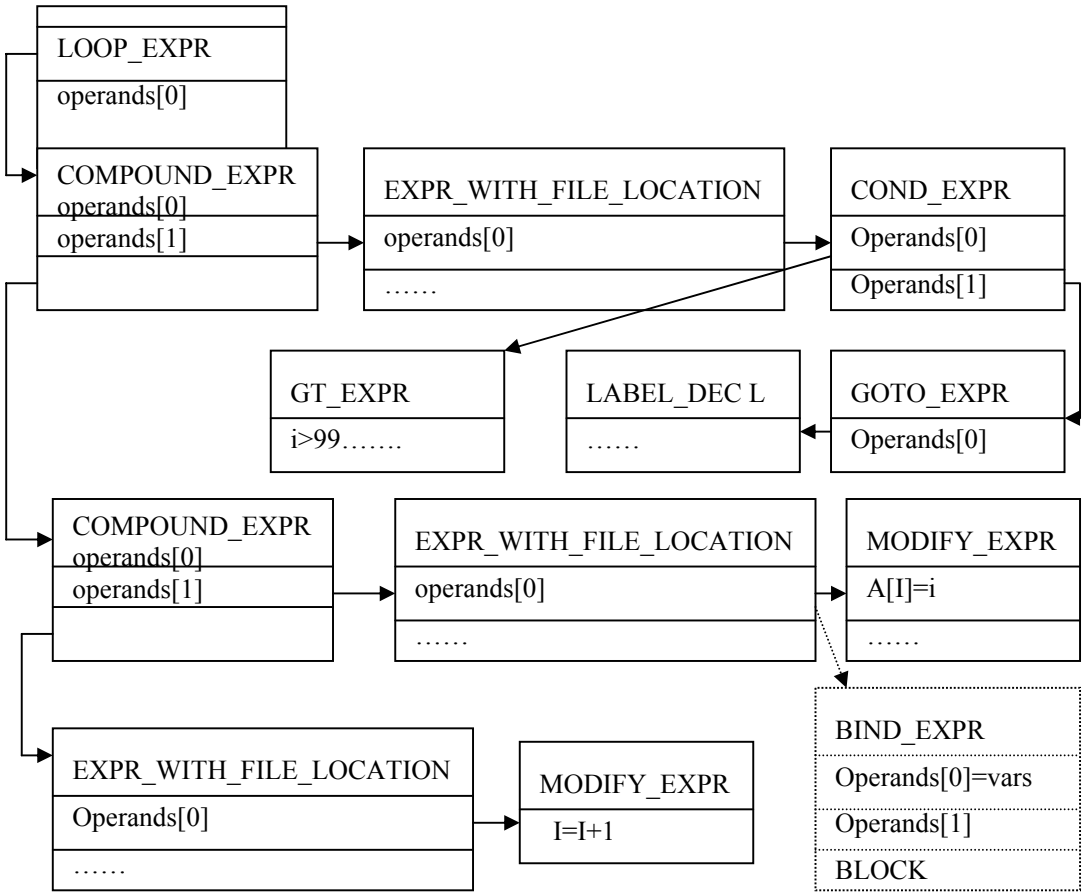
对应的语法树为：


COMPOUND_STMT
Exp.operands[0]
……
SCOPE_STMT
Common.chain
Exp.operands[0]
……
BLOCK
Block.vars
Block.subblocks
Block.supercontex
t
DECL_STMT
Common.chain
Exp.operands[0]
……
VAR_DECL
Common.chain
Common.type’
identifier=’ i’ …
.
DECL_STMT
Common.chain
Exp.operands[0]
……
VAR_DECL
Chain=NULL
Common.type
Identifier=’ a’ …
…
FOR_STMT
Common.chain
Exp.operands[0]
Exp.operands[1]
Exp.operands[2]
Exp.operands[3]
……
EXPR_STMT
Exp.operands[0]
……
MODIFY_EXPR
I=0
……
LE_EXPR(I<99)
POSTINCREMENT_EXPR(I++)
LOOP_BODY(a[I]=I)
RETURN_STMT
Common.chain
Exp.operands[0]
……


SCOPE_STMT
Common.chain=null
……

简化后的树结构如下：


EXPR_WITH_FILE_LOCATION
operands[0]
……
BIND_EXPR
operands[0]
operands[1]
operands[2]
……
VAR_DECL
chain
Identifier='i'
VAR_DECL
Chain=NULL
Identifier='a'
BLOCK
vars
subblocks
supercontext
COMPOUND_EXPR
operands[0]
operands[1]
EXPR_WITH_FILE_LOCATION
operands[0]
……
MODIFY_EXPR
I=0
COMPOUND_EXPR
operands[0]
operands[1]
EXPR_WITH_FILE_LOCATION
operands[0]
……
LOOP_EXPR
operands[0]
operands[1]
COMPOUND_EXPR
operands[0]
operands[1]
LABEL_EXPR
Operands[0]
LABEL_DECL
…….
EXPR_WITH_FILE_LOCATION
operands[0]
……
RETURN_EXPR
…….



上述语法树对应如下代码序列：

```
        I=0
L1：
        If(I>99) goto L2
        A[I]=I
        I=I+1
        Goto L1
L2:
       Return I
```