文章编号: 1001－9081(2004)03－0115－02

一种解析 GCC 抽象语法树的方法

石 峰, 刘 坚
(西安电子科技大学 软件工程研究所, 陕西 西安 710071)
(liujian@sei.xidian.edu.cn)

摘 要: 介绍了 GCC 抽象语法树的结构及在编译过程中的作用。给出解析抽象语法树所使用的词法规则和语法规则, 提出了解析 GCC 抽象语法树的方法。

关键词: 抽象语法树; 词法规则; 语法规则
中图分类号: TP314 文献标识码: A

A Method for Parsing GCC Abstract Syntax Tree

SHI Feng, LIU Jian
(*Software Engineering Institute, Xidian University, Xi'an Shaanxi* 710071, *China*)

Abstract: The structure of GCC abstract syntax tree and its functions in compiling are introduced. Lexical rules and syntax rules used for parsing abstract syntax trees are given, and a method for parsing GCC is proposed.

Key words: abstract syntax tree; lexical rule; syntax rule

1 引言

GCC 编译系统[1] 是由美国自由软件基金会(Free Software Foundation)开发, 可以编译多种程序设计语言, 支持多平台编译的编译程序集合, 是 Linux 下最稳定和成熟的编译器。GCC 编译系统主要由三部分组成: 与程序设计语言相关的前端、与程序设计语言无关的后端和与机器相关的机器描述部分。为了支持多平台编译, GCC 前端需要一种中间表示, 能够向上支撑多种程序设计语言到中间代码的映射, 向下支持跨平台代码转换并且适合在多种平台下进行优化。为此 GCC 编译系统使用两个层次的中间表示, 分别为抽象语法树(AST)层和寄存器转移语言(RTL)层。抽象语法树作为一种通用的中间表示, 不仅包含各种语言共有的语法结构, 某些特定类型的树节点还可以表示一些语言特有的语法结构。抽象语法树易于转换成寄存器转移语言, 而寄存器转移语言适合在不同平台下进行优化, 这使得 GCC 的两层中间表示具有良好的通用性。

作为一种良好的中间表示, 抽象语法树包含了完整的源程序信息。利用抽象语法树可以实现多种源程序处理工具, 比如智能编辑器、源程序浏览器等。此外, 抽象语法树的解析器也可以作为程序静态分析工具的前端, 为其提供一种便于分析的输入。因此, 解析抽象语法树具有重要意义。

2 抽象语法树的结构

GCC 编译系统以源程序的过程为单位生成抽象语法树, 抽象语法树与过程一一对应。使用 GCC 的编译参数-fdump-ast-original 可以使源程序生成名为 *.original 的文件, 其中 * 为源程序名, 该文件包含以文本形式输出的抽象语法树。

在抽象语法树文件中, 以";; Function" 开始的行代表该语法树的开始, 该行包含抽象语法树对应的函数名, 我们用 FUNCTION 代表该行。FUNCTION 下面是抽象语法树所有节点的列表。每个节点形如:

@1 function_decl name: @2 type: @3 srcp: stu.c:2
C extern body: @4

其中@1 称作节点标号, 用 NUMBER 表示, 它是抽象语法树上区分该节点的唯一标志, 也是访问该节点的索引。其后是节点标识符 IDENT 和节点标记的序列 TAG_LIST。节点标识符是该节点的名称, 代表了该节点的含义, @1 节点的节点标识符为 function_decl。其余部分为节点标记的列表, 每个节点标记形如: name: @2。

结点标记的列表记录了该节点连接到其他节点的所有分支, 每个节点标记对应一个分支。节点标记由标记标签 TAG_LABEL 和标记值 TAG_VALUE 组成, 标记值可以为空。标记标签是该分支的名称, 标记值是该分支连接的目标。@1 节点的第一个节点标记是 name: @2, name 为标记标签, @2 为标记值, 代表该节点第一个分支是 name 分支, 其指向目标为@2 节点。

存在两种特殊的节点标记: 一种是标记标签由两个单词组成的节点标记, 比如算术操作运算的左操作数由 op 和 0 两个单词组成; 另一种是标记值可以为空的标记, 比如上面节点的 extern 标记。整个抽象语法树由具有上述结构的树节点构成, 通常抽象语法树以函数声明节点作为根结点, 在根节点处逐渐展开, 形成完整树结构。

部分抽象语法树文件示例如下:

收稿日期: 2003－09－12 基金项目: 武器装备预研基金项目(51406070101DZ0151)
作者简介: 石峰(1978－), 男, 辽宁盘锦人, 硕士研究生, 主要研究方向: 程序分析、编译器构造; 刘坚(1954－), 女, 河南新县人, 教授, 主要研究方向: 软件开发环境、编译器构造.

```
;;Function int main() (main)
@1   function-decl   name : @2   type : @3   srcp : stu. c :2
                     C            extern      body : @4
@2   function-node   strg :main   Ingt :4
@3   function-type   size :@5     algn :64    retn : @6
                     prms : @7
@4   compound-stmt   line :6      body :@8    next :@9
... ...
```

3 抽象语法树解析方法

抽象语法树结构比较简单,其对应的词法规则和语法规则[2,3]易于构造,使用 flex 和 bison 工具生成的解析器能够有效地对抽象语法树进行解析。解析器由三部分组成,分别是 flex 生成的词法分析器、bison 生成的语法分析器和手工编写的驱动程序。词法分析器识别抽象语法树文件的记号流,提供给语法分析器;语法分析器利用嵌入其中的语义动作识别语法树节点,完成解析任务;驱动程序负责为词法分析器和语法分析器提供一个调用接口,并提供解析所需的数据结构和函数的实现。

为使解析后的抽象语法树便于遍历,使用哈希表[4]作为抽象语法树的存储结构,每个哈希表节点对应一个抽象语法树节点,树节点编号作为哈希表的关键字,采用直接定址法构造哈希函数。哈希表节点存储树节点的节点编号、节点标识符和指向节点标记列表的指针。节点标记列表采用链表存储,每个哈希表节点包含一个指向该链表的指针。解析过程主要使用链表、哈希表等数据结构,因此 Linux 下实现该解析器,可以使用 glib 库提供的链表、哈希表及其操作函数作为解析器所需的数据结构和函数。

词法分析器使用的词法规则如下(&b& 代表空格):

```
FUNCTION = ";;Function". * $
NUMBER = "@"[0-9]+
IDENT = [a-zA-Z]+
TAGLABEL = [a-zA-Z]+(":"| "&b&:"| "&b&&b&:")
TAGVALUE = [a-zA-Z0-9<>.:-]+| \"(.)*\"
```

对于其他记号,词法分析器不做任何动作。

语法分析器使用的语法规则及嵌入的语义动作如下:

```
asg :asg node
    |
    ;
node : NUMBER IDENT tags 狚add-node( $1, $2) 狅
     | FUNCTION 狚new-asg ( $1) 狅
     ;
tags : doublet tags
     | IDENT tags
     狚add-tag( $1, NULL) 狅
     | IDENT TAGVALUE NUMBER tags
     狚add-tag( create-tag( $1, $2), $3 );狅
     |
     ;
doublet :TAGLABEL tag-value 狚add-tag( $1, $2);狅
        ;
tag-value : NUMBER 狚 $ $ = $1;狅
          | TAGVALUE 狚 $ $ = $1;狅
          | IDENT 狚 $ $ = $1;狅
          | 狚 $ $ = NULL;狅
          ;
```

词法分析器识别 FUNCTION、NUMBER、IDENT、TAGLABEL 和 TAGVALUE 五种记号,词法分析器通过使用 bison 变量 yylval 将这五种记号对应的文本传递给语法分析器。语法分析器的语法规则包含五种非终结符,分别为 asg、node、tags、doublet 和 tag-value。语法规则的第一个非终结符为 asg,它代表一个抽象语法树分析的开始;node 代表一个抽象语法树节点;tags 代表节点标记的列表;doublet 代表标记标签由单个单词组成,标记值非空的标记;tag-value 代表节点标记值。语义动作中各函数使用的伪变量是词法分析器所识别记号的文本。

语义动作使用了四个函数:

1) new-asg($1) 为抽象语法树新建名为 $1 的空哈希表,作为抽象语法树的存储结构;

2) add-node($1, $2) 在哈希表中增加一个关键字为 $1,标识符值为 $2 的节点;

3) add-tag($1, $2) 在当前节点的标记序列中增加一个标记标签为 $1,标记值为 $2 的标记;

4) create-tag($1, $2) 将 $1 和 $2 存储的字符串组合,并返回该组合。

对第 2 部分中源程序产生的文本形式的抽象语法树进行解析,得到如图 1 所示的解析结果。

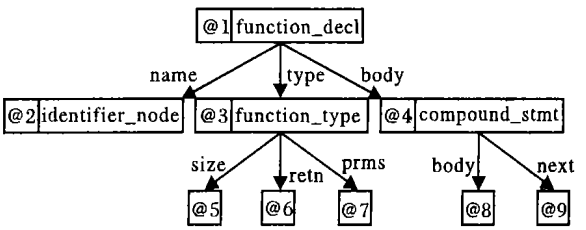

图 1 部分抽象语法树解析结果

从图 1 可以看到,抽象语法树从函数声明节点逐渐展开,生成完整的树结构。抽象语法树的结点代表了语言不同的语法结构,从树的根结点开始对抽象语法树进行遍历的过程中,可以访问到所关心语法结构对应的树节点。遍历过程中加入特定的语义动作对树节点进行处理,就可以实现源程序浏览器、程序分析器等工具。

4 结束语

利用 flex 和 bison 工具生成的解析器能够快速有效地解析 GCC 产生的抽象语法树文件,使用哈希表作为存储抽象语法树的存储载体可以快速遍历抽象语法树,便于对解析后的抽象语法树进行处理。我们在 Linux 环境下实现的解析器,能够成功解析出 GCC 生成的抽象语法树。

参考文献

[1] Stallman RM. Using and Porting the GNU Compiler Collection [EB/OL]. http://gcc.gnu.org/onlinedocs/, 2003-05-11/2003-07-10.

[2] Aho AV, Sothi R, Ullman JD. Compilers-Principles, Techniques, and Tools [M]. Addison-Wesley Publishing Company, 1986.

[3] Watt DA. Programming Language Syntax and Semantics [M]. Prentice Hall Inc, 1991.

[4] 严蔚敏, 吴伟民. 数据结构 [M]. 北京:清华大学出版社, 1997.