UNICORE32-MINIC 实习报告

作者:	ID
彭焯	00848xxx
华连盛	00848xxx
王衍	00848xxx
李春奇	00848xxx

2010 年 12 月 21 日

目录

第一章 词法、语法分析 **5**

1.1 词法分析, flex 5

1.2 语法分析, bison 5

1.2.1 MiniC 文法 5

1.2.2 利用 bison 进行文法分析 6

1.2.3 语法错误检查 7

1.3 谬误与陷阱 8

第二章 中间表示 **9**

2.1 抽象语法树 AST 9

2.1.1 生成 9

2.1.2 生成的语法树示例 10

2.1.3 符号表 10

2.1.4 符号表示例 12

2.1.5 类型检查 12

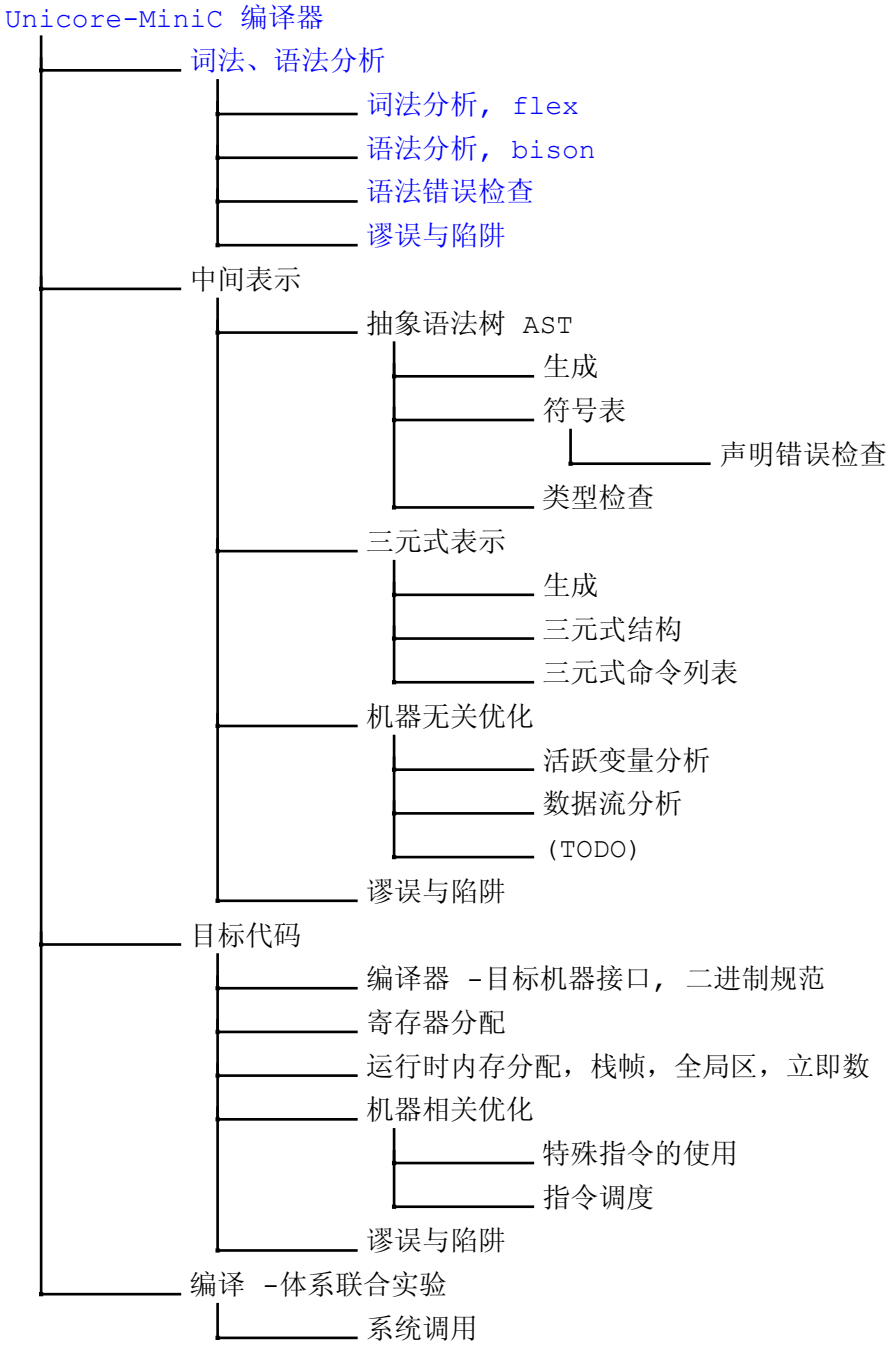
第三章 目标代码 **13**

第四章 编译 -体系联合实验 **14**

简介

如何阅读本文档

下面给出了本文档的结构，单击目录树上的节点便可以查看文档的相关部分。



本文档包括的内容

本文档的目标并不是用于展示 MiniC 冗杂的代码，而是提供一个概括的全局视角来了解这个实习项目。文档涵盖了如下的几个方面：

- Unicore-MiniC 编译器（以下在没有歧义的情况下简称之为 MiniC）的总体设计方案
- MiniC 的各个单元的设计方案，包括主要的数据结构和重要的算法
- MiniC 各个单元之间的协同关系
- 为相关代码的阅读提供的指引
- 在开发过程中遇到的困难，遭遇的陷阱和付出的代价[1]

MiniC In a Nutshell

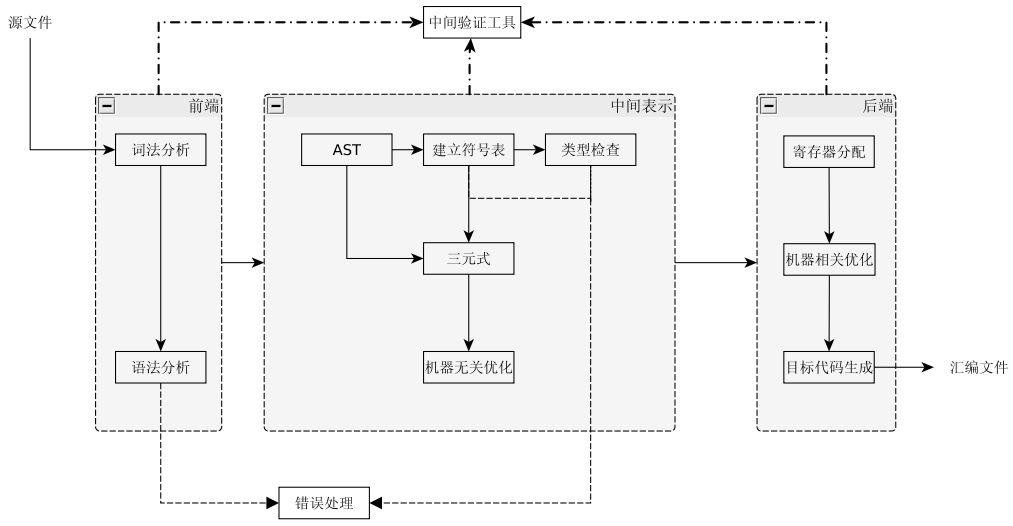
项目目标

- 输入 MiniC 源代码，输出 Unicore32 汇编代码
- 利用 Unicore32 二进制工具链中的链结器链接后的程序能在 Unicore32 实体机器上运行
- 链接后的程序能够在同时设计的 Unicore32 模拟器上运行
- 通过对模拟器相关参数、指令系统的修改，考察编译 -ISA 对性能的综合影响

整个项目将全部以 C 语言实现，用`gcc` 进行编译；以`GNU/Linux, i386` 为运行平台。使用`svn` 进行版本控制。项目主页：code.google.com/p/minic

项目结构

下面是一个 MiniC 编译器的总体结构图：



[1] 参见某些章中的“谬误与陷阱”一节

总体上看，MiniC 由三大模块构成：

1. 前端：接收输入文件，结合文法，负责语法、词法分析；将源文件的结构和内容信息提取到抽象语法树 (AST) 上
2. 中间表示：共有两层：AST 和三元式；语义分析在 AST 上完成后，将 AST 变换为三元式；机器无关优化在三元式表示上进行。
3. 后端：目标机器代码（汇编代码）生成；结合目标机器二进制规范和 ISA，生成目标代码；机器相关优化在目标代码上进行。

与此同时，还有错误处理模块，用于在源文件有误时产生出错提示；中间验证工具集，用于检查工程进展的每一个步骤的正确性。
在下面的章节中，我们将一一介绍以上模块，并且还将在最后一章展示 MiniC 编译器是如何与体系实习的模拟器项目配合，进行一系列编译 -体系相关的实验和评测。

致谢

- 感谢刘先华老师对我们小组的指导。他总能及时、耐心地解答我们的问题，与此同时又鼓励我们积极思考和大胆尝试，使我们从这个实习项目中收获了知识和快乐。
- 感谢刘锋老师在工程进展和规划、模拟器 -编译器结合方面提供的帮助。
- 感谢与我们一同选择 MiniC 项目的另外两个小组的成员，你们的帮助和启发使我们受益匪浅。
- 当然，最需要感谢的是这个项目本身：我们通过它同时体验了 CPU 设计人员、模拟器设计人员、甚至是（一部分）操作系统设计人员在设计一个大型的、可靠的系统时所面临的挑战，以及克服这些挑战后所获得的成就感。

下面，我们将从头开始，介绍 UNICORE-MINIC编译器。

第一章　词法、语法分析

1.1　词法分析, flex

词法分析的目的是将输入的源文件中的符号识别为语法分析器能够接受的 tokens。其基本原理是利用正则表达式 pattern 来匹配、分割源文件中的符号。同时，词法分析还需要识别并保存一些名字和常量，比如函数名、变量名和字符串常量。
MiniC 的词法规则同标准 C 的对应规则基本相同：

1. 标识符必须以下划线或大小写字母开头，由下划线、大小写字母和数字组成
2. 字符常量需要放在单引号'' 中，字符串常量需要放在双引号"" 中
3. 有如下保留字不能当作标识符：`extern`, `register`, `void`, `int`, `char`, `if`, `else`, `for`, `while`, `return`
4. 特殊符号包括：`{}`, `()`, `[]`, `+`, `-`, `*`, `!`, `&`, `=`, `|`, `>`, `<`

由于语法分析器 bison 实际上不能获得AST 叶节点上的信息，因此生成 AST 叶节点的工作交由 flex 完成。

⚓*flex 源文件请参阅*：`minic.l`

1.2　语法分析, bison

语法分析的目的是根据语言的 BNF 范式，将词法分析器提交的 token 流进行规约，在规约的同时应用一些语法规则。语法分析的结果有两个：

1. 将语法正确的源文件转换为语法规则所要求的中间形式，这一中间形式不在具有语言的语法特性，或
2. 发现语法错误，如果错误能够暂时恢复，就继续语法分析，在完成后提示错误信息；否则，直接停止语法分析，报告错误。

MiniC 项目利用 bison 辅助进行语法分析，分析完成后，将建立一棵 AST。
在介绍对 bison 的利用前，由于我们的项目修改了给定的 MiniC 文法，所以首先要对修改后的文法进行说明。

1.2.1　MiniC 文法

我们实现的 MiniC 文法和实习项目开始时给定的有所不同，最大的不同之处在表达式文法部分。以下是原表达式文法：

```
unary_expr := unary_op unary_expr | postfix_expr
unary_op := ! | + | - | ++ | -- | & | *
postfix_expr := postfix_expr ( expression )
| ident ( argumentlist )
| ident ( )
| postfix_expr ++
| postfix_expr --
| primary_expr
```

注意到这种文法允许诸如下述的语法：

- `++ ++ a;`（等价于`a=a+1=a+1`）
- `!a = 1;`
- `f()=1;`

而这些语法实际上是语言所不能处理的，因为“计算结果”和“代表一个存储位置的符号”是完全不同的概念，前者没有地方存储也因此不能被赋值。
我们修改后的表达式文法基于“左值”——代表一个存储区域的符号和“右值”——计算结果，并且是二义性文法：

```
expression := rvalue | assignment_expression
assignment_expression:= lvalue = assignment_expression
| lvalue = rvalue
lvalue := * rvalue | IDENT | IDENT [ expression ]
rvalue := lvalue
| rvalue + rvalue
| rvalue - rvalue
| rvalue * rvalue
| rvalue op rvalue
| ( rvalue )
| + rvalue
| - rvalue
| ! rvalue
| & lvalue
| IDENT ( argument_list )
| IDENT ( )
...
```

这样定义的文法在表达的意义上更清晰，同时也避免了上述的错误。
⚓ 有关原文法漏洞的详细说明，请参阅：`BNF_leaks.pdf`
⚓ 本项目的全部 *BNF* 范式请参阅：`BNF.pdf`

我们对原 MiniC 文法的改动主要在其表达式文法部分。

1.2.2 利用 bison 进行文法分析

bison 是一个能够自动构建 $LALR(1)$（或 $LR(K)$）语法分析器的软件。它的输入是一个文法描述，输出是一个能够直接编译的 C 源文件，只需调用`yyparse()`函数，就可以开始按照文法进行规约，同时执

行相应的文法规则。
使用 bison 需要注意几个问题：

1. LR 分析允许文法中有左递归
2. bison 允许使用二义性文法，但是需要指名一个规约规则，否则会提示规约 -规约冲突。例如，为了使文法设计简单，MiniC 使用的表达式文法就是二义性文法，但是利用 bison 提供的“定义规约优先级”和“定义结合律”功能，就可以实现按照指定的算符优先级来进行规约。
3. 每个终结符和非终结符都有同样的“属性”结构，因此，在声明这个结构时需要考虑到属性文法中所要用到的所有属性。下面是 MiniC 语法分析器所用的属性结构：

```
%union{
    iattr int_attr; //可能具有的整数属性
    cattr char_attr; //可能具有的字符属性
    sattr string_attr; //可能具有的字符串属性
    AST_NODE* ptr; //指向子节点的指针
}
```

只有叶子节点才会使用上面三种属性，因为它们可能包含标识符名称或常量。

⚓ 本项目的 *bison* 源文件请参阅：`minic.y`

1.2.3　语法错误检查

bison 提供了完善的错误检查功能，当发现一个语法错误时，它会暂停当前的规约过程，并在当前规约到的表达式后添加一个”error” 终结符，然后开始在规约栈中从error 以下弹出符号，直到找到一个带有”error” 终结符的产生式，便用它规约，并应用相应的文法规则（一般来说是错误处理代码）；然后文法分析恢复正常运行。如果找不到用于处理错误的文法，那么分析器报错退出。
与此同时，bison 还提供了追踪终结符/非终结符在源文件中的位置的机制，只需要词法分析器在一个名为yylloc 的结构中填入信息，这个结构中记录了终结符的开始行列编号和结束行列编号。bison 将在规约的过程中自动维护每个文法符号的yylloc。
MiniC 添加了简单的错误处理/错误代码追踪功能。我们在lvalue 和rvalue 产生式集中各添加了一个error 单一产生式，并且加入了报错的代码，因此，一旦 MiniC 源文件中表达式出现错误，语法分析器就会指示其位置。
例如，下面代码第 4 行少了一个分号：

```
int main()
{
    int i;
    i = i + 1
}
```

编译器会提示：

```
Syntax error
Parsing error: line 4.0-5.1: bad lvalue
 Parsing error: line 4.0-5.1: bad lvalue
 Parser: terminated, 2 error(s).
```

1.3　谬误与陷阱

谬误 1： 文法的设计可以草率为之，将问题留给语义检查会比考虑一个正确的文法更方便。
文法具有其数学上的严谨性，一条有漏洞的文法可能会导致允许的语言集合过大或过小。最为危险的是如果这个集合过大，那么一类或几类错误的语言集合就会顺利通过语法分析：这类集合在文法上应当具有相似的结构，但是在语义上就可能很不相同，如果在语义分析时再检查这些错误，将造成复杂的代码逻辑和冗长的分支。

谬误 2： 语法所支持的就是语言应支持的。
通过了语法分析仅仅以为着符合语法的规范，确不意味着语义的正确。例如：`void` 关键字可以修饰一个变量，这是符合 MiniC 语法的，然而，一个`void` 形的变量既不能赋值又不能引用，如果允许它出现，那么在生成三元式，生成目标代码时将会不知道如何去处理它。所以我们发现了这个问题后，就 MiniC 的语义分析中，禁用了这种表示。

第二章　中间表示

2.1　抽象语法树 AST

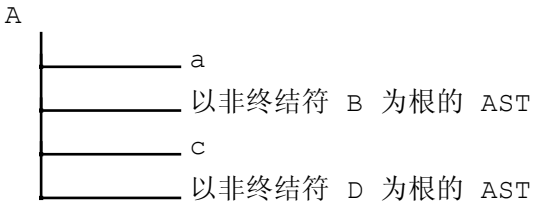
AST 是 MiniC 采用的第一重中间表示，它由在词法/语法分析阶段生成。由于我们在 AST 上完成了符号表的生成和类型检查，因此这两部分也在本章介绍。

2.1.1　生成

AST 的结构是根据上下文无关文法的产生式定义的：一个终结符节点是一棵 AST；一棵以产生式左部非终结符为根的 AST 的子节点按从左到右顺序分别是其右部分法符号所产生的 AST。
比如有如下产生式：

$$A := aBcD$$

其中大写字母代表非终结符，小写字母代表终结符，那么以非终结符 A 为根的 AST 就是如下形式：



注意到 AST 的定义实际上也给出了其产生方法：只要在做 LR 分析时，随着规约的进行，将子树与根进行连接即可。
在实现时，AST 的叶节点在词法分析时生成，内部节点在语法分析时在不同的产生式的语法规则指导下生成，并在规约时连接到其父节点上。
下面给出一个 MiniC 的 AST 节点所包含的内容：

```
typedef struct AST_NODE{
    int nodeType;//节点类型，即节点代表了哪个文法符号
    int nodeLevel;//节点深度
    union node_content content; //节点包含的内容，只有叶子节点才不为空，在词法分析时添加
    AST_NODE * father;//父节点
    AST_NODE * leftChild;//最左子节点
    AST_NODE * rightSibling;//右兄弟节点

    struct symtbl_hdr* symtbl;//文法符号所在范围的符号表
    AST_NODE* double_list;
}AST_NODE;
```

⚓ 有关节点类型的详细定义，请参阅：AST.h
⚓ 有关建立 *AST* 树的相应过程，请参阅：AST_operation.c, minic.y

2.1.2　生成的语法树示例

我们利用dot 生成了下面代码的 AST：

```
1 int main()
2 {
3     int i;
4     i = i + 1;
5 }
```

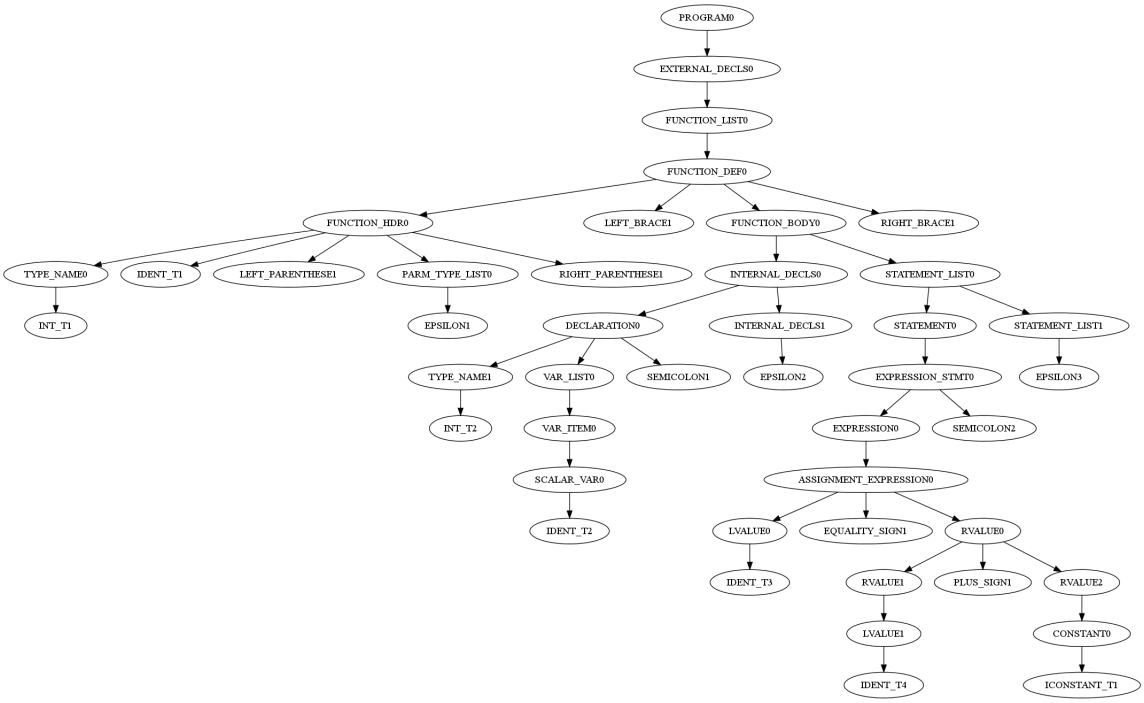


图 2.1: AST 示例

↝*MiniC* 的验证工具中提供了两种方式查看输入源文件的语法树：dot 和 *ASCII ART*，请参阅 *MiniC* 使用手册

2.1.3　符号表

符号表对于一个编译程序而言是最为重要的一个部分，从它创建以后开始的每一个步骤中，“标识符”（函数名、变量名）的出现，就意味着要在符号表中找到对应的表项，提取要使用的信息。

符号表的结构　符号表的层次结构需要根据语言的特性来决定。例如在 MiniC 中，由于存在着全局变量、函数声明、语句块等特点，因此符号表需要组织成树状结构。
符号表的具体底层数据结构可以有很多种选择，常见的有哈希表、链表和线性表，在 MiniC 中，由于考虑到输入源文件的规模都较小，所以采用可扩张的线性表来实现[1]。在这种实现下，插入一个表项的均摊开销是 $O(1)$，检索一个表项的开销是 $O(n)$， n 为表项总数目。
MiniC 的符号表结构如下：

1. 每个作用域（函数、复合语句）一张符号表：

[1]这种线性表在空间不足时会另外申请一块两倍的空间并拷贝自己

```
struct symtbl_hdr
{
    symtbl_hdr* parent_tbl; //父表
    symtbl_hdr* leftChild_tbl; //最左子表
    symtbl_hdr* rightSibling_tbl;    //右兄弟子表

    //ret_type, ret_star, para_num, func are useful only for function's symtbl
    char* func_name; //函数名，若该表属于复合语句则该项为空
    int ret_type; //函数返回值基类型
    int ret_star; //函数返回值是否为指针类型
    int para_num; //该函数的参数个数
    int func_def; //该函数是否有定义
    int item_num; //符号表表项数目
    int maxSize; //全部表项占用的内存大小，字节
    symtbl_item* item; //表项列表
}
```

2. 每张符号表都有一个表项列表：

```
struct symtbl_item
{
    int isGlobal; //是否为全局变量
    int type; //表项的基类型
    int star_num; //表项是否为指针
    int writable; //表项是否为只读类型
    char* name; //符号名
    int size; //数组大小，非数组为0
    int func_off; //
    int offset; //在内存分配中的偏移
};
```

注意到在一个函数的符号表中，函数的参数和它的局部变量是存放在一起的，这种将参数和局部变量视作同类的安排能够为以后的内存分配、寄存器分配等提供一些方便。

在 AST 上生成符号表　如前所述，AST 上包含了源文件中所有的信息，因此只需要扫描 AST 的某些子树，提取符号名称、类型、作用域等信息，就可以生成符号表。

具体的做法是：先序深度优先周游 AST，找到作用域入口（FUNCTION_DEF 和COMPOUND_STMT）并将当前作用于压入作用域栈，然后在该作用域节点下寻找类型为EXTERNAL_DECLS 和INTERNAL_DECLS 的节点，分情况处理该节点的函数和变量的声明，将名称和类型（包括函数的返回类型，参数、参数类型）加入符号表，作用域从作用域栈顶取得。在作用域出口（周游函数返回时）弹出作用域栈顶。

⚓ 有关符号表生成的相关代码，请参阅：gen_symtbl.h

在符号表上查询符号　由于符号表是树状组织，因此查询时要从给定的一张符号表开始，在表中查找，如果未找到符号，就在当前表的父表中查找，直到找到该符号或者在最高级的Global Scope 也没有找到返回空。

⚓ 有关符号表查询的代码，请参阅：symtbl_operation.h

符号表生成与语义检查　在符号表生成的过程中，实际上还要进行一项语义检查：同一作用域的多重定义问题。即在向当前符号表中添加符号时，如果该符号已经出现在了当前表中，就要报告错误，停止建立符号表。

2.1.4　符号表示例

下图展示了以下代码的符号表:

```
1  int i,j;
2  void f(int k)
3  {
4      int i,j;
5  }
6  int main()
7  {
8      int i,j;
9      {
10         int k;
11     }
12 }
```

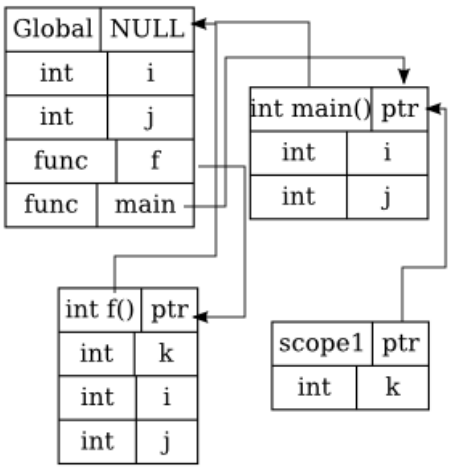


图 2.2: 符号表示例

↝*MiniC* 的验证工具中提供了查看源文件生成的符号表的方法：请参阅 *MiniC* 使用手册

2.1.5　类型检查

第三章　目标代码

第四章　编译 -体系联合实验