中国科学技术大学 University of Science and Technology of China

中间语言与中间代码生成

《编译原理和技术》

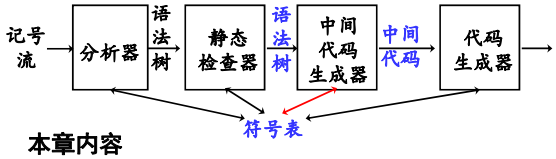
张昱

0551-63603804，yuzhang@ustc.edu.cn

中国科学技术大学

计算机科学与技术学院

本章内容



本章内容

- 中间语言：常用的中间表示(Intermediate Representation)
  - 后缀表示、图表示、三地址代码、LLVM IR
- 基本块和控制流图
- 中间代码的生成
  - 声明语句(=>更新符号表)
  - 表达式、赋值语句（=>产生临时变量、查符号表）
  - 布尔表达式、控制流语句（=>标号/回填技术、短路计算）

中国科学技术大学 University of Science and Technology of China

1. 中间语言

- 后缀形式、图形表示
- 三地址代码
- 静态单赋值

后缀表示

- 后缀表示不需要括号

  (8 – 5) + 2 的后缀表示是 8 5 –2 +
- 后缀表示的最大优点是便于计算机处理表达式

计算栈	输入串
	8 5 –2 +
8	5 –2 +
8 5	–2 +
3	2 +
3 2	+
5	

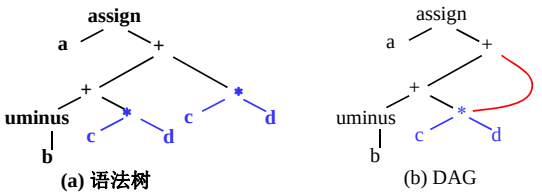
后缀表示

- 后缀表示不需要括号

  (8 – 5) + 2 的后缀表示是 8 5 –2 +
- 后缀表示的最大优点是便于计算机处理表达式
- 后缀表示的表达能力
  - 可以拓广到表示赋值语句和控制语句
  - 但很难用栈来描述控制语句的计算

图形表示

- 语法树是一种图形化的中间表示
- 有向无环图也是一种中间表示



(a) 语法树　(b) DAG

a = (–b + c*d) + c*d的图形表示

图形表示

构造赋值语句语法树的语法制导定义

修改构造结点的函数可生成有向无环图

产 生 式	语 义 规 则
$S \rightarrow$ id $=E$	$S.nptr = mkNode$('assign', $mkLeaf$ (id, id.$entry$), $E.nptr$)
$E \rightarrow E_1 + E_2$	$E.nptr = mkNode$('+', $E_1.nptr$, $E_2.nptr$)
$E \rightarrow E_1 * E_2$	$E.nptr = mkNode$('*', $E_1.nptr$, $E_2.nptr$)
$E \rightarrow -E_1$	$E.nptr = mkUNode$('uminus', $E_1.nptr$)
$E \rightarrow (E_1)$	$E.nptr = E_1.nptr$
$F \rightarrow$ id	$E.nptr = mkLeaf$ (id, id.$entry$)

三地址代码

- 三地址代码(three-address code)

一般形式：$x = y\ op\ z$

例 表达式 $x + y * z$ 翻译成的三地址语句序列是

$t_1 = y * z$

$t_2 = x + t_1$

三地址代码

- 三地址代码是语法树或DAG的一种线性表示

例 a = (–b + c*d) + c*d

语法树的代码

$t_1 = -b$

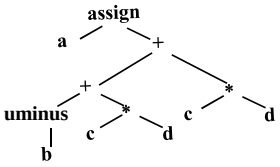
$t_2 = c * d$

$t_3 = t_1 + t_2$

$t_4 = c * d$

$t_5 = t_3 + t_4$

$a = t_5$

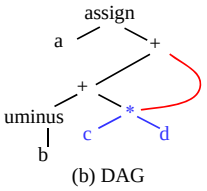


三地址代码

- 三地址代码是语法树或DAG的一种线性表示

例 a = (–b + c*d) + c*d

语法树的代码	DAG的代码
$t_1 = -b$	$t_1 = -b$
$t_2 = c * d$	$t_2 = c * d$
$t_3 = t_1 + t_2$	$t_3 = t_1 + t_2$
$t_4 = c * d$	$t_4 = t_3 + t_2$
$t_5 = t_3 + t_4$	$a = t_5$
$a = t_5$	



(b) DAG

三地址代码

- 常用的三地址语句
  - 赋值语句　$x = y\ op\ z$，　$x = op\ y$
  - 复写语句　$x = y$
  - 无条件转移　goto L
  - 条件转移　if x *relop* y goto L
  - 过程调用　param x 和call p , n
  - 过程返回　return y
  - 索引赋值　$x = y[i]$和 $x[i] = y$
  - 地址和指针赋值　$x = \&y$，$x = *y$ 和 $*x = y$

静态单赋值

- 静态单赋值形式(static single-assignment form)
  - 一种便于某些代码优化的中间表示
  - 和三地址代码的主要区别

所有赋值指令都是对不同名字的变量的赋值

三地址代码	静态单赋值形式
p = a +b	p_1 = a +b
q = p – c	$q_1 = p_1 - c$
p = q * d	$p_2 = q_1 * d$
p = e – p	$p_3 = e - p_2$
q = p + q	$q_2 = p_3 + q_1$

静态单赋值

- 静态单赋值形式(static single-assignment form)
  - 一种便于某些代码优化的中间表示
  - 和三地址代码的主要区别

    所有赋值指令都是对不同名字的变量的赋值

    一个变量在不同路径上都定值的解决办法

    if (flag) x = –1; else x = 1;

    y = x * a;

  改成

    if (flag) x_1 = –1; else x_2 = 1;

    $x_3 = \phi(x_1, x_2)$; // 由flag的值决定用x_1还是x_2

2. 基本块和控制流图

- 基本块
- 流图

基本块

- 程序举例

```
prod = 0;
i = 1;
do {
   prod = prod + a[i] * b[i];
   i = i +1;
} while (i <= 20);
```

(1)prod = 0
(2) i = 1
(3) $t_1 = 4 * i$
(4) $t_2 = a[t_1]$
(5) $t_3 = 4 * i$
(6) $t_4 = b[t_3]$
(7) $t_5 = t_2 * t_4$
(8) $t_6 = prod + t_5$
(9) prod = t_6
(10) $t_7 = i + 1$
(11) i = t_7
(12) if i <= 20 goto (3)

基本块和流图

- 基本块

  连续的语句序列，控制流从它的开始进入，并从它的末尾离开，没有停止或分支的可能性（末尾除外）
- 流图(flow graph)

  用有向边表示基本块之间的控制流信息，基本块作为结点

(1)prod = 0
(2) i = 1
(3) $t_1 = 4 * i$
(4) $t_2 = a[t_1]$
(5) $t_3 = 4 * i$
(6) $t_4 = b[t_3]$
(7) $t_5 = t_2 * t_4$
(8) $t_6 = prod + t_5$
(9) prod = t_6
(10) $t_7 = i + 1$
(11) i = t_7
(12) if i <= 20 goto (3)

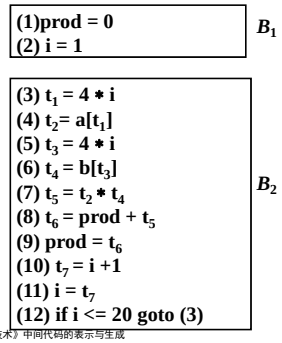
基本块的划分

- 划分方法
  - 首先确定所有入口语句
    - 序列的第一个语句
    - 能由(无)条件转移语句转到的语句
    - 紧跟在(无)条件转移语句后面的语句
  - 每个入口语句到下一个入口语句之前（或到程序结束）的语句序列构成一个基本块

(1)prod = 0
(2) i = 1
(3) $t_1 = 4 * i$
(4) $t_2 = a[t_1]$
(5) $t_3 = 4 * i$
(6) $t_4 = b[t_3]$
(7) $t_5 = t_2 * t_4$
(8) $t_6 = prod + t_5$
(9) prod = t_6
(10) $t_7 = i + 1$
(11) i = t_7
(12) if i <= 20 goto (3)

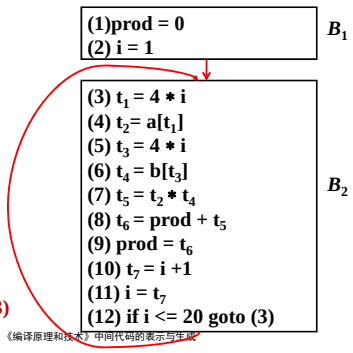
流图

(1)prod = 0
(2) i = 1
(3) $t_1 = 4 * i$
(4) $t_2 = a[t_1]$
(5) $t_3 = 4 * i$
(6) $t_4 = b[t_3]$
(7) $t_5 = t_2 * t_4$
(8) $t_6 = prod + t_5$
(9) prod = t_6
(10) $t_7 = i + 1$
(11) i = t_7
(12) if i <= 20 goto (3)



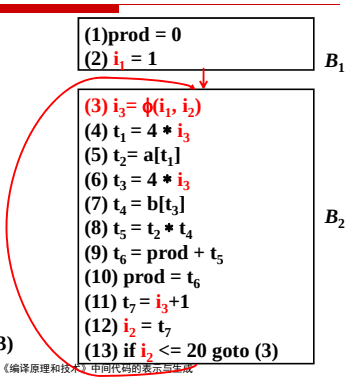
流图

(1)prod = 0
(2) i = 1
(3) t_1 = 4 * i
(4) t_2= a[t_1]
(5) t_3 = 4 * i
(6) t_4 = b[t_3]
(7) t_5 = t_2 * t_4
(8) t_6 = prod + t_5
(9) prod = t_6
(10) t_7 = i +1
(11) i = t_7
(12) if i <= 20 goto (3)



流图(变换成 SSA 格式)

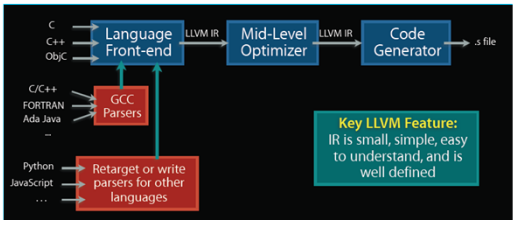
(1)prod = 0
(2) i_1= 1
(3) i_3= ϕ(i_1, i_2)
(4) t_1 = 4 * i_3
(5) t_2= a[t_1]
(6) t_3 = 4 * i_3
(7) t_4 = b[t_3]
(8) t_5 = t_2 * t_4
(9) t_6 = prod + t_5
(10) prod = t_6
(11) t_7 = i_3+1
(12) i_2 = t_7
(13) if i_2 <= 20 goto (3)

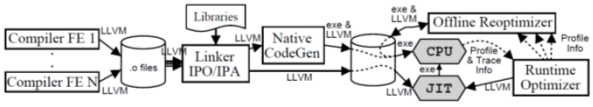


3. LLVM 编译器框架和基础设施

- 总体结构
- LLVM IR
- LLVM Pass Manager
- LLVM Tools

LLVM编译系统





LLVM IR

- 参考资料
  - LLVM IR参考手册 (http://llvm.org/docs/LangRef.html)
  - 教程(http://llvm.org/docs/tutorial/LangImpl03.html)
- 举例：bar(a) foo(a, 4.0) + bar(31337);

```
define double @bar(double %a) {
entry:
    %calltmp = call double @foo(double %a, double 4.000000e+00)
    %calltmp1 = call double @bar(double 3.133700e+04)
    %addtmp = fadd double %calltmp, %calltmp1
    ret double %addtmp
}
```

全局标识符　局部标识符

LLVM IR

- 主要特征
  - RISC风格的三地址代码
  - SSA格式、无限的虚拟寄存器
  - 简单、低级的控制流结构
  - Load/store指令带类型化指针
- IR的格式：text(.ll)、binary(.bc)、in-memory

```
for (i = 0; i < N;
     ++i)
  Sum(&A[i], &P);
```

```
bb:          ; preds = %bb, %entry
  %i.1 = phi i32 [ 0, %entry ], [ %i.2, %bb ]
  %AiAddr = getelementptr float* %A, i32 %i.1
  call void @Sum( float* %AiAddr, %pair* %P )
  %i.2 = add i32 %i.1, 1
  %exitcond = icmp eq i32 %i.2, %N
  br i1 %exitcond, label %return, label %bb
```

LLVM IR速览

C program language	*LLVM IR*
• Scope: *file, function*	*module, function*
• Type: *bool, char, int, struct{int, char}*	*i1, i8, i32, {i32, i8}*
• A statement with multiple expressions	A sequence of instructions each of which is in a form of "x = y *op* z".
• Data-flow: a sequence of reads/writes on variables	1. load the values of memory addresses (variables) to registers; 2. compute the values in registers; 3. store the values of registers to memory addresses * each register must be assigned exactly once (SSA)
• Control-flow in a function: if, for, while, do while, switch-case,…	A set of basic blocks each of which ends with a conditional jump (or return)

LLVM类型系统

- **类型系统的组成**
  - **Primitives: integer, floating point, label, void**
    - no "signed" integer types
    - arbitrary bitwidth integers (i32, i64, i1)
  - **Derived: pointer, array, structure, function, vector,…**

No high-level types: type-system is language neutral!

- **Type system allows arbitrary casts:**
  - **Allows expressing weakly-typed languages, like C**
  - ***Front-ends can implement safe languages***
  - ***Also easy to define a type-safe subset of LLVM***

LLVM程序结构

- **模块Module：包含函数和全局变量**
  - 是编译/分析/优化的基本单位
- **函数Function：包含基本块/参数**
- **基本块BasicBlock: 指令序列**
- **指令Instruction: opcode+vector of operands**
  - 所有操作数operands都有类型
  - 指令结果是类型化的

示例：C 编译到LLVM

```
int callee(const int *X) {
  return *X+1;   // load
}
int caller() {
  int T;         // on stack
  T = 4;         // store
  return callee(&T);
}
```

Stack allocation is explicit in LLVM

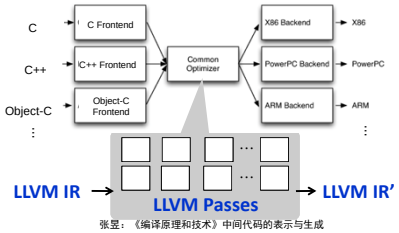
All loads/stores are explicit in the LLVM representation

```
define internal i32 @callee(i32* %X) {
entry:
  %tmp2 = load i32* %X
  %tmp3 = add i32 %tmp2, 1
  ret i32 %tmp3
}

define internal i32 @caller() {
entry:
  %T = alloca i32
  store i32 4, i32* %T
  %tmp1 = call i32 @callee( i32* %T )
  ret i32 %tmp1
}
```


LLVM Passes

- **LLVM提供108+ Passes** http://llvm.org/docs/Passes.html
  - 分析器：别名分析、调用图构造、依赖分析等
  - 变换器：死代码消除、函数内联、常量传播、循环展开等
  - 实用组件：CFG viewer、基本块提取器等



LLVM Pass Manager

- **编译器组织成一系列的passes**
  - 每个pass是一个分析或变换
- **Pass的类型**
  - **ModulePass: general interprocedural pass**
  - **CallGraphSCCPass: bottom-up on the call graph**
  - **FunctionPass: process a function at a time**
  - **LoopPass: process a natural loop at a time**
  - **BasicBlockPass: process a basic block at a time**
- **施加的约束 (e.g. FunctionPass):**
  - FunctionPass 只能查看当前函数
  - 不能维护跨函数之间的状态

LLVM工具

- 基础工具
  - llvm-as: Convert from .ll (text) to .bc (binary)
  - llvm-dis: Convert from .bc (binary) to .ll (text)
  - llvm-link: Link multiple .bc files together
  - llvm-prof: Print profile output to human readers
  - llvmc: Configurable compiler driver
- 集成工具
  - bugpoint: automatic compiler debugger
  - llvm-gcc/llvm-g++: C/C++ compilers

4. 中间代码生成概述

- 方法和关键问题
- 符号表结构的变化

中间代码生成的方法

- 边解析边生成中间代码
  - 语法制导的翻译方案
  - 难点：理解分析器的运转机制、继承属性的处理
- 基于树访问的中间代码生成
  - 重点：树结构的设计、访问者模式、enter/exit接口及实现
  - 如实验2 的任务

本节将以基于树访问的中间代码生成方法为主来讲解，这是现代编译器使用的主流方法。

中间代码生成的关键问题

假设采取的中间语言类似三地址代码

- 类型与符号表的变化
  - 多样化类型 => 整型(字节、字)、浮点型、类型符号表
  - 1个某类型的数据 => m 个字节(m为类型对应的字宽)
- 语句的翻译
  - 声明语句：不生成指令，但会更新符号表（作用域，字宽及存放的相对地址）
  - 赋值语句：引入临时变量、数组/记录元素的地址计算、类型转换
  - 控制流语句：跳转目标的确定(引入标号或使用回填技术)、短路计算

符号表的设计

- 类型检查后的符号表
  - 符号表条目：（标识符、存储类别、类型信息）
  - 存储类别：extern, static, register, …
  - 类型信息：(类别标识, 该类别关联的其他信息)
    - 如数组(Array，(len, elemtype))
- 本章符号表的变化
  - 作用域 =>多个符号表
  - 变量：字宽、存储的相对地址（以字节为单位）
  - 记录类型：用符号表管理各个成员的字宽、相对地址

5. 声明语句

- 分配存储单元，更新符号表
- 作用域的管理
- 记录类型的管理

声明语句的翻译

- 主要任务
  - 为局部名字分配存储单元

符号表条目：名字、类型、宽度、偏移

  - 作用域信息的保存
  - 记录类型的管理

不产生中间代码指令，但是要更新符号表

块中无变量声明时的翻译

计算被声明名字的类型和相对地址

$P \rightarrow \{offset = 0\}\ D; S$ 相对地址初始化为0

$D \rightarrow D\ ;\ D$

$D \rightarrow \text{id} : T$ $\{enter(\text{id}.lexeme, T.type, offset);$

$offset = offset + T.width\}$ 更新符号表信息

$T \rightarrow \text{integer}$ $\{T.type = integer;\ T.width = 4\}$

$T \rightarrow \text{real}$ $\{T.type = real;\ T.width = 8\}$

$T \rightarrow \text{array}[\text{num}]\ \text{of}\ T_1$ 类型=>字宽

$\{T.type = array(\text{num}.val, T_1.type);$

$T.width = \text{num}.val \times T_1.width\}$

$T \rightarrow \uparrow T_1$ $\{T.type = pointer(T_1.type);\ T.width = 4\}$

仅有主过程时的翻译

基于树访问的代码生成 enterP时处理

$P \rightarrow \{offset = 0\}\ D; S$

$D \rightarrow D\ ;\ D$

$D \rightarrow \text{id} : T$ $\{enter(\text{id}.lexeme, T.type, offset);$

$offset = offset + T.width\}$ exitD时处理

$T \rightarrow \text{integer}$ $\{T.type = integer;\ T.width = 4\}$

$T \rightarrow \text{real}$ $\{T.type = real;\ T.width = 8\}$

$T \rightarrow \text{array}[\text{num}]\ \text{of}\ T_1$ exitT时处理

$\{T.type = array(\text{num}.val, T_1.type);$

$T.width = \text{num}.val \times T_1.width\}$

$T \rightarrow \uparrow T_1$ $\{T.type = pointer(T_1.type);\ T.width = 4\}$

visitD时处理 (只有访问D时才知道D是哪种结构)

visitT时处理 (只有访问T时才知道T是哪种结构)

允许自定义过程时的翻译

- 所讨论的语言的文法

$P \rightarrow D; S$

$D \rightarrow D\ ;\ D \mid \text{id} : T \mid$

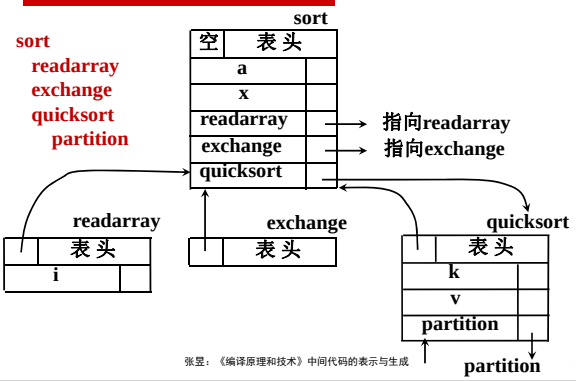
$\text{proc id}\ ;\ D\ ;\ S$

- 管理作用域
  - 每个过程内声明的符号要置于该过程的符号表中
  - 方便地找到子过程和父过程对应的符号表

```
sort
  var a:…; x:…;
  readarray
    var i:…;
  exchange
  quicksort
    var k, v:…;
    partition
      var i, j:…;
```

P186,图6.14 (过程参数被略去)

各过程的符号表



符号表的组织与管理

- 相关的数据结构设计
  - 符号表：哈希表
  - 符号表之间的连接(双向链)

父→子：过程中包含哪些子过程定义；

子→父：分析完子过程后继续分析父过程

  - 一遍分析时，需要维护符号表栈
- 本章使用的符号表相关的函数

mkTable(*previous*)

enter(*table, name, type, offset*)

addWidth(*table, width*)

enterProc(*table, name, newtable*)

```
sort
  var a:…; x:…;
  readarray
    var i:…;
  exchange
  quicksort
    var k, v:…;
    partition
      var i, j:…;
```

P186,图6.14 (过程参数被略去)

声明语句的处理

$P \rightarrow D; S$

$D \rightarrow D\ ;\ D \mid \text{id} : T \mid$
proc id ; D ; S

tblptr：符号表栈

offset：偏移量栈

enterP：$t = mkTable(nil)$； $push(t, tblptr)$; $push(0, offset)$

visitD：

1) id : T：更新符号表中id对应的条目
   $enter(top(tblptr), \text{id}.lexeme, T.type, top(offset))$;
   $top(offset) = top(offset) + T.width$

声明语句的处理

$P \rightarrow D; S$

$D \rightarrow D\ ;\ D \mid \text{id} : T \mid$
proc id ; D ; S

tblptr：符号表栈

offset：偏移量栈

enterP：$t = mkTable(nil)$； $push(t, tblptr)$; $push(0, offset)$

visitD：

1) id : T：更新符号表中id对应的条目
2) proc id ; D; S:
   访问D前：新建该过程的符号表，进入该过程的作用域
   t=$mkTable(top(tblptr))$; $push(t,tblptr)$;$push(0, offset)$

声明语句的处理

$P \rightarrow D; S$

$D \rightarrow D\ ;\ D \mid \text{id} : T \mid$
proc id ; D ; S

tblptr：符号表栈

offset：偏移量栈

enterP：$t = mkTable(nil)$； $push(t, tblptr)$; $push(0, offset)$

visitD：

1) id : T：更新符号表中id对应的条目
2) proc id ; D; S:
   访问D前：新建该过程的符号表，进入该过程的作用域
   访问S后：将该过程符号信息插入到父符号表，退出作用域
   $t = top(tblptr)$; $addWidth(t, top(offset))$;
   $pop(tblptr)$; $pop(offset)$; $enterProc(top(tblptr), \text{id}.lexeme, t)$

如果S中存在对该过程的递归调用,则在分析S前将该过程名插入符号表

声明语句的处理

$P \rightarrow D; S$

$D \rightarrow D\ ;\ D \mid \text{id} : T \mid$
proc id ; D ; S

tblptr：符号表栈

offset：偏移量栈

enterP：$t = mkTable(nil)$； $push(t, tblptr)$; $push(0, offset)$

visitD：

1) id : T：更新符号表中id对应的条目
2) proc id ; D; S:

exitP：
$addWidth(top(tblptr), top(offset))$; $pop(tblptr)$; $pop(offset)$

记录的域名管理

- 关联的文法

$T \rightarrow$ record D end

记录类型单独建符号表(类型表达式),域相对地址从0开始

visitT: record D end

访问D之前：建立符号表，进入作用域

$t = mkTable(nil)$; $push(t, tblptr)$; $push(0, offset)$

结尾：设置记录的类型表达式和宽度，退出作用域

$T.type = record(top(tblptr))$;

$T.width = top(offset)$; $pop(tblptr)$; $pop(offset)$

6. 赋值语句

- 分配临时变量，存储表达式计算的中间结果
- 数组元素的地址计算
- 类型转换

赋值语句的翻译

- 主要任务
  - 复杂的表达式 => 多条计算指令组成的序列
  - 分配临时变量保存中间结果
  - id：查符号表获得其存储的场所
  - 数组元素：元素地址计算
    - 符号表中保存数组的基址和用于地址计算的常量表达式的值
    - 数组元素在中间代码指令中表示为“基址[偏移]”
  - 可以进行一些语义检查
    - 类型检查、变量未定义/重复定义/未初始化
  - 类型转换：因为目标机器的运算指令是区分类型的

赋值语句的中间代码生成

- 关联的文法

$S \rightarrow \text{id} := E \qquad E \rightarrow E_1 + E_2 \mid -E_1 \mid (E_1) \mid \text{id}$

visitS: id := E

结尾：获取id的地址和存放E结果的场所，发射赋值指令

p = *lookup*(id.*lexeme*);

if *p* != *nil* then *emit* (*p*, ‘=’, *E.place*) else *error*

visitE:

$E \rightarrow E_1 + E_2$ 结尾：发射加法指令

E.place = *newTemp*();

emit (*E.place*, ‘=’, E_1.*place*, ‘+’, E_2.*place*)

赋值语句的中间代码生成

- 关联的文法

$S \rightarrow \text{id} := E \qquad E \rightarrow E_1 + E_2 \mid -E_1 \mid (E_1) \mid \text{id}$

visitE:

$E \rightarrow E_1 + E_2$ 结尾：发射加法指令

$E \rightarrow - E_1$ 结尾：发射负号运算指令

E.place = *newTemp*();

emit (*E.place*, ‘=’, uminus, E_1.*place*)

$E \rightarrow (E_1)$ 结尾：*E.place* = E_1.*place*;

$E \rightarrow \text{id}$ 结尾：获取id的地址并作为E的场所

p = *lookup*(id.*lexeme*);

if *p* != *nil* then *E.place* = *p* else *error*

数组元素的地址计算

- 一维数组元素的地址计算

A的第*i*个元素的地址： $base + (i - low) \times w$

变换成： $i \times w + (base - low \times w)$

$low \times w$是常量，编译时计算，减少运行时计算

- 二维数组元素的地址计算
  - 列为主序(列优先)？行为主序？

行为主序时： $base + ((i_1 - low_1) \times n_2 + (i_2 - low_2)) \times w$

（A[i_1, i_2]的地址，其中$n_2 = high_2 - low_2 + 1$）

变换成： $((i_1 \times n_2) + i_2) \times w +$

$(base - ((low_1 \times n_2) + low_2) \times w)$

数组元素的地址计算

- 多维数组元素的地址计算
  - 以行为主序

下标变量A[i_1, i_2, ..., i_k]的地址表达式

$((\ldots ((i_1 \times n_2 + i_2) \times n_3 + i_3) \ldots) \times n_k + i_k) \times w$

$+ base - ((\ldots ((low_1 \times n_2 + low_2) \times n_3 + low_3) \ldots) \times n_k + low_k) \times w$

- 翻译的主要任务
  - 发射地址计算的指令
  - “基址[偏移]”相关的中间指令：t =b[o]，b[o]=t

数组元素的访问处理

- 关联的文法

$S \rightarrow L := E \qquad L \rightarrow \text{id} [\textit{Elist}] \mid \text{id}$

$\textit{Elist} \rightarrow \textit{Elist}, E \mid E \qquad E \rightarrow L \mid \ldots$

- 采用语法制导的翻译方案时存在的问题

$\textit{Elist} \rightarrow \textit{Elist}, E \mid E$ 由*Elist*的结构只能得到各维的下标值，但无法获得数组的信息（如各维的长度）

需要改写文法为： $L \rightarrow \textit{Elist}] \mid \text{id} \quad \textit{Elist} \rightarrow \text{id} [E \mid \textit{Elist}, E$

$\textit{Elist} \rightarrow \text{id} [E$ 由这个定义可以获得数组的信息，并从左到右传播下去，达到边分析边计算的目的

数组元素的访问处理

- 关联的文法

基于树来生成会简单多了，不用改写文法

$S \to L := E$　　$L \to \text{id}\ [\ Elist\ ] \mid \text{id}$　　$Elist \to Elist, E \mid E$

visitL: $L \to \text{id}\ [\ E_1, E_2, \ldots, E_n\ ]$

访问E_1之后: $ndim = 1$; $place = E_1.place$; // 局部变量

每次访问E_i之后计算: $t = newTemp()$; $ndim$ ++;

emit (*t*, '=', *place*, '*', *limit*(id.*place*, *ndim*));

emit (*t*, '=', *t*, '+', E_i.*place*);　*place* = *t*;

结尾: *L.place* = *newTemp*();

emit (*L.place*, '=', *base*(id.*place*), '–', *invariant* (id.*place*));

L.offset = *newTemp*();

emit (*L.offset*, '=', *place*, '*', *width*(id.*place*)));

数组元素的访问处理

- 关联的文法

$S \to L := E$　　$E \to L$

visitE: $E \to L$

结尾: if (*L.offset* == null) /* 简单变量 */ *E.place* = *L.place*

else { *E.place* = *newTemp*();

emit (*E.place*, '=', *L.place*, '[', *L.offset*, ']'); }

visitS: $S \to L := E$

结尾: if (*L.offset* == null)　*emit* (*L.place*, '= ', *E.place*);

else　*emit* (*L.place* , '[', *L.offset*, ']', '=', *E.place*);

类型转换

例　x = y + i * j

（x和y的类型是real，i和j的类型是integer）

中间代码

t_1 = i int× j

t_2 = inttoreal t_1

t_3 = y real+ t_2

x = t_3

目标机器的运算指令是区分整型和浮点型的

高级语言中的重载算符=>中间语言中的多种具体算符

类型转换的处理

- 以$E \to E_1 + E_2$为例说明

visitE: $E \to E_1 + E_2$

结尾：判断E_1和E_2的类型，看是否要进行类型转换;若需要，则分配存放转换结果的临时变量并发射类型转换指令

E.place = *newTemp*( );

if (*E*$_1$.*type* == *integer* && *E*$_2$.*type* == *integer*) {

emit (*E.place*, '=', *E*$_1$.*place*, 'int+', *E*$_2$.*place*);

E.type = integer;

} else if (*E*$_1$.*type* == *integer* && *E*$_2$.*type* == *real*) {

u = *newTemp*( );　　*emit* (*u*, '=', 'inttoreal', *E*$_1$.*place*);

emit (*E.place*, '=', *u*, 'real+', *E*$_2$.*place*);　　*E.type* = *real*;

}

7. 布尔表达式和控制流语句

- 布尔表达式：短路计算
- 控制流语句的翻译: 标号、回填技术
- switch的翻译优化
- 过程调用的中间代码格式与翻译

中间代码生成的主要任务

- 主要任务
  - 布尔表达式的计算：完全计算、短路计算
  - 控制流语句
    - 分支结构(if、switch)、循环结构、过程/函数的调用
  - 各子结构的布局+无条件或有条件转移指令
  - 跳转目标的两种处理方法
    - 标号技术：新建标号，跳转到标号
    - 回填技术：先构造待回填的指令链表，待跳转目标确定时再回填链表中各指令缺失的目标信息

布尔表达式

- 布尔表达式的作用
  - 计算逻辑值
  - 作为控制流语句中的条件
- 本节关联的布尔表达式文法

  $B \rightarrow B$ or $B \mid B$ and $B \mid$ not $B \mid (B) \mid E$ relop $E \mid$ true $\mid$ false
- 布尔表达式的计算
  - 完全计算：各子表达式都要被计算
  - 短路计算：B_1 or B_2 定义成 if B_1 then true else B_2

    B_1 and B_2 定义成 if B_1 then B_2 else false

控制流语句的翻译

- 关联的控制流语句

$S \rightarrow$ if B then S_1

| if B then S_1 else S_2

| while B do S_1

| switch E begin case V_1: S_1 ...

case V_{n-1}: S_{n-1}

default: S_n

end

| call id (*Elist*)

| S_1; S_2

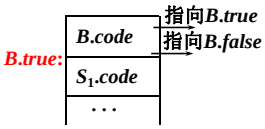
$Elist \rightarrow Elist, E \mid E$

if 语句的中间代码布局

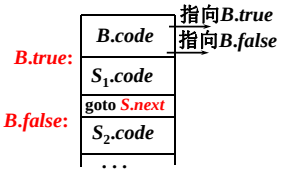
- 问题与对策
  - B的短路计算中，需要知道其为真或假时的跳转目标
  - B、S_1、S_2分别会发射多少条指令是不确定的

引入标号：先确定标号，在目标确定时发射标号指令

可调用newLabel( ) 产生新标号，每条语句有*next* 标号



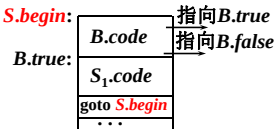
(a) if-then



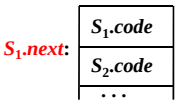
(b) if-then-else

while语句和顺序结构

- while循环语句的中间代码
  - 引入开始标号*S.begin*,作为循环的跳转目标
- 顺序结构
  - 为每一语句S_1引入其后的下一条语句的标号S_1.*next*



(c) while-do



(d) S_1; S_2

if 语句的中间代码生成

- 问题与对策
  - B的短路计算中，需要知道其为真或假时的跳转目标
  - B、S_1、S_2分别会发射多少条指令是不确定的

引入标号：先确定标号，在目标确定时发射标号指令

可调用newLabel( ) 产生新标号

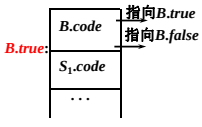
visitIf-then：

访问B前：*B.true* = *newLabel*();

B.false = *S.next*; // 继承属性

进入S_1前：S_1.*next* = *S.next*;

访问S_1后：*S.code* = *B.code* || *gen*(*B.true*, ':') || S_1.*code*



(a) if-then

if 语句的中间代码生成

回填：仅使用综合属性

- 把跳转到同一个标号的指令放到同一张指令表中

  如，为B引入综合属性*truelist* 和 *falselist*分别收集要回填的跳转指令，为S引入*nextlist* 收集要回填的跳转指令
- 等目的标号确定时，再把它填到表中的各条指令中

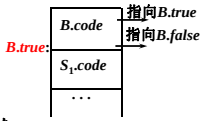
visitIf-then：$S \rightarrow$ if B then S_1

准备访问S_1前：*instr* = *nextinstr*;

访问S_1后：

backPatch(*B.truelist*, *instr*); //回填

S.nextlist = *merge*(*B.falselist*, S_1.*nextlist*);



(a) if-then

if 语句的中间代码生成

$S \rightarrow$ if B then S_1 else S_2 （标号技术）

访问B前：$B.true = newLabel();\ B.false = newLabel();$

进入S_1前：$S_1.next = S.next;$

进入S_2前：$S_2.next = S.next;$

访问S_2后：$S.code = B.code \| gen(B.true, ':') \| S_1.code \| gen('goto', S.next) \| gen(B.false, ':') \| S_2.code$

if 语句的中间代码生成

$S \rightarrow$ if B then S_1 else S_2 （标号技术）

访问B前：$B.true = newLabel();\ B.false = newLabel();$

进入S_1前：$S_1.next = S.next;$

进入S_2前：$S_2.next = S.next;$

访问S_2后：$S.code = B.code \| gen(B.true, ':') \| S_1.code \| gen('goto', S.next) \| gen(B.false, ':') \| S_2.code$

回填

进入S_1前：$instr1 = nextinstr;$

访问S_1后：$nextlist = makeList(nextinstr);\ emit('goto\_');$

进入S_2前：$instr2 = nextinstr;$

访问S_2后：$backPatch(B.truelist, instr1);$

$backPatch(B.falselist, instr2);$

$S.nextlist = merge(merge(S_1.nextlist, nextlist), S_2.nextlist);$

while语句的中间代码生成

$S \rightarrow$ while B do S_1

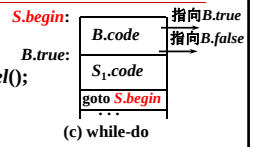
访问while前：$S.begin = newLabel();$

访问B前：$B.true = newLabel();$

$B.false = S.next;$

进入S_1前：$S_1.next = S.begin;$

访问S_1后：$S.code = gen(S.begin, ':') \| B.code \| gen(B.true, ':') \| S_1.code \| gen('goto', S.begin)$



(c) while-do

while语句的中间代码生成

$S \rightarrow$ while B do S_1

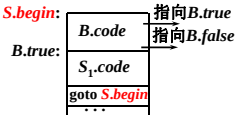
访问while前：$S.begin = newLabel();$

访问B前：$B.true = newLabel();$

$B.false = S.next;$

进入S_1前：$S_1.next = S.begin;$

访问S_1后：$S.code = gen(S.begin, ':') \| B.code \| gen(B.true, ':') \| S_1.code \| gen('goto', S.begin)$



(c) while-do

回填 进入B前：$instr1 = nextinstr;$

进入S_1前：$instr2 = nextinstr;$

访问S_1后：$backPatch(S_1.nextlist, instr1);$

$backPatch(B.truelist, instr2);$

$S.nextlist = B.falselist;\ emit('goto', instr1);$

布尔表达式的控制流翻译

如果B是$a < b$的形式，

那么代码是：

if $a < b$ goto $B.true$

goto $B.false$

例 表达式

$a < b$ or $c < d$ and $e < f$

的代码是：

if $a < b$ goto L_{true}

goto L_1

L_1: if $c < d$ goto L_2

goto L_{false}

L_2: if $e < f$ goto L_{true}

goto L_{false}

布尔表达式的翻译

$B \rightarrow B_1$ or B_2 （标号技术）

访问B_1前：$B_1.true = B.true;\ B_1.false = newLabel();$

访问B_2前：$B_2.true = B.true;\ B_2.false = B_1.false;$

访问B_2后：$B.code = B_1.code \| gen(B_1.false, ':') \| B_2.code$

$B \rightarrow$ not B_1 （标号技术）

访问not前：$B_1.true = B.false;\ B_1.false = B.true;$

访问B_1后：$B.code = B_1.code$

布尔表达式的翻译

$B \rightarrow B_1$ and B_2（标号技术）

访问B_1前：$B_1.true = newLabel(); B_1.false = B.false;$

访问B_2前：$B_2.true = B.true; B_2.false = B.false;$

访问B_2后：$B.code = B_1.code \| gen(B_1.true, ':') \| B_2.code$

$B \rightarrow (B_1)$（标号技术）

访问(前：$B_1.true = B.true; B_1.false = B.false;$

访问)后：$B.code = B_1.code$

布尔表达式的翻译

$B \rightarrow E_1$ relop E_2（标号技术）

访问E_2后：$B.code = E_1.code \| E_2.code \|$
$gen('if', E_1.place, relop.op, E_2.place, 'goto', B.true) \| gen('goto', B.false)$

$B \rightarrow$ true（标号技术）

访问true后：$B.code = gen('goto', B.true)$

$B \rightarrow$ false（标号技术）

访问false后：$B.code = gen('goto', B.false)$

布尔表达式的翻译(回填)

$B \rightarrow B_1$ or M B_2 { $backPatch(B_1.falselist, M.instr)$;
$B.falselist = B_2.falselist$;
$B.truelist = merge(B1.truelist, B2.truelist)$;}

$M \rightarrow \varepsilon$ { $M.instr = nextinstr$; }

$B \rightarrow B_1$ and M B_2 { $backPatch(B_1.truelist, M.instr)$;
$B.truelist = B_2.truelist$;
$B.falselist = merge(B_1.falselist, B_2.falselist)$; }

$B \rightarrow$ not B_1 { $B.truelist = B_1.falselist$;
$B.falselist = B_1.truelist$; }

布尔表达式的翻译(回填)

$B \rightarrow (B_1)$ { $B.truelist = B_1.truelist$;
$B.falselist = B_1.falselist$; }

$B \rightarrow E_1$ relop E_2 { $B.truelist = makeList(nextinstr)$;
$B.falselist = makeList(nextinstr+1)$;
$emit('if', E_1.place, relop.op, E_2.place, 'goto \_')$;
$emit('goto \_')$; }

$B \rightarrow$ true { $B.truelist = makeList(nextinstr)$;
$B.falselist$ = null; $emit('goto \_')$; }

$B \rightarrow$ false { $B.falselist = makeList(nextinstr)$;
$B.truelist$ = null; $emit('goto \_')$; }

switch语句的翻译

switch E
begin
case V_1: S_1
case V_2: S_2
. . .
case V_{n-1}: S_{n-1}
default: S_n
end

分支数较少时

t = E的代码
if t != V_1 goto L_1
S_1的代码
goto next
L_1: if t != V_2 goto L_2
S_2的代码
goto next
L_2: . . .
L_{n-2}: if t != V_{n-1} goto L_{n-1}
S_{n-1}的代码
goto next
L_{n-1}: S_n的代码
next:

switch语句的翻译

分支较多时，将分支测试代码集中在一起，便于生成较好的分支测试代码

t = E的代码	L_n: S_n的代码
goto test	goto next
L_1: S_1的代码	test: if t == V_1 goto L_1
goto next	if t == V_2 goto L_2
L_2: S_2的代码	. . .
goto next	if t == V_{n-1} goto L_{n-1}
. . .	goto L_n
L_{n-1}: S_{n-1}的代码	next:
goto next	

switch语句的翻译

中间代码增加一种case语句，便于代码生成器对它进行特别处理

```
test:   case V1      L1
        case V2      L2
        ...
        case Vn-1    Ln-1
        case  t      Ln
next:
```

代码生成器可做两种优化：
- 用二分查找确定该执行的分支
- 建立入口地址表，直接找到该执行的分支

（例子见第244页习题8.8）

过程调用的翻译

$S \rightarrow$ call id ($Elist$)	过程调用id(E_1, E_2, …, E_n)的中间代码结构
$Elist \rightarrow Elist, E$	$E_1.place = E_1$的代码
$Elist \rightarrow E$	$E_2.place = E_2$的代码
	...
	$E_n.place = E_n$的代码
	param $E_1.place$
	param $E_2.place$
	...
	param $E_n.place$
	call id.$place$, n

过程调用的翻译

$S \rightarrow$ call id ($Elist$)

结尾：

为长度为n的队列中的每个$E.place$，$emit$('param', $E.place$);

$emit$('call', id.$place$, n);

$Elist \rightarrow Elist, E$

结尾：把$E.place$放入队列末尾

$Elist \rightarrow E$

结尾：将队列初始化，并让它仅含$E.place$

例题1

Pascal语言的标准将for语句

for v := initial to final do stmt

能否定义成和下面的代码序列有同样的含义？

```
begin
        t1 := initial; t2 := final;
        v := t1;
        while v <= t2 do begin
                stmt; v := succ(v)
        end
end
```


例题1

Pascal语言的标准将for语句

for v := initial to final do stmt

能否定义成和下面的代码序列有同样的含义？

```
begin
        t1 := initial; t2 := final;
        v := t1;
        while v <= t2 do begin
                stmt; v := succ(v)
        end
end
```

final为最大整数时，succ(final)会导致越界错误

例题1

Pascal语言的标准将for语句

for v := initial to final do stmt

的中间代码结构如下：

```
        t1 = initial
        t2 = final
        if t1 > t2 goto L1
        v = t1
L2:     stmt
        if v == t2 goto L1
        v = v + 1
        goto L2
L1:
```

例题2

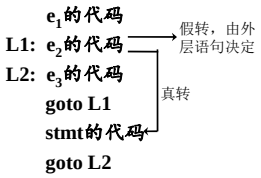
C语言的for语句有下列形式

for (e_1;e_2;e_3) stmt

它和

e_1;
while (e_2)do begin
    stmt;
    e_3;
end

有同样的语义



例题3

- **Pascal语言**

  var a,b : array[1..100] of integer;
  a:=b // 允许数组之间赋值
  也允许在相同类型的记录之间赋值

- **C语言**

  数组类型不行，结构体类型可以

为这种赋值选用或设计一种三地址语句，它要便于生成目标代码

答：仍然用中间代码复写语句 x = y，在生成目标代码时，必须根据它们类型的size，产生一连串的值传送指令