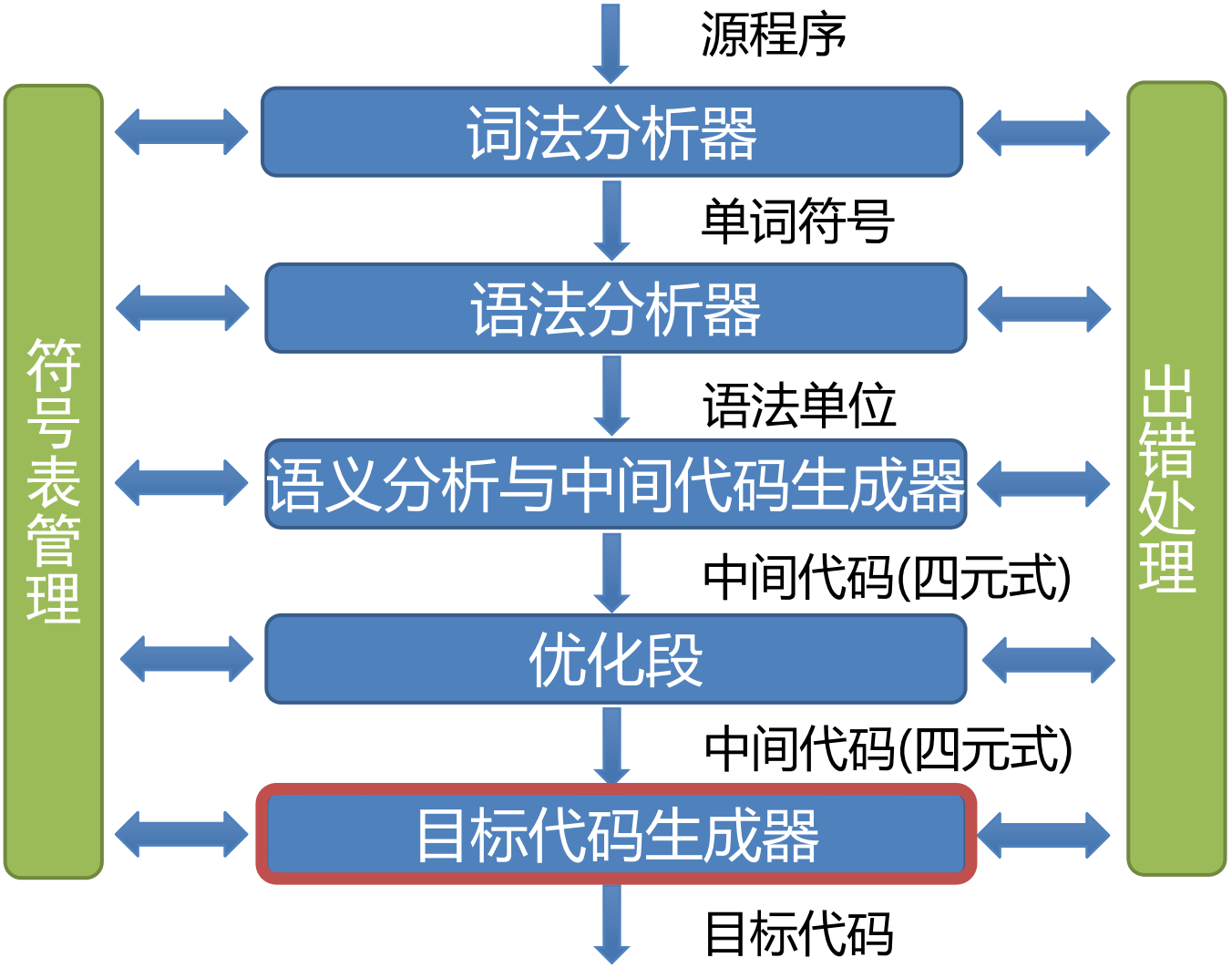
编译原理

目标代码生成

编译程序总框


源程序
词法分析器
单词符号
语法分析器
语法单位
语义分析与中间代码生成器
中间代码(四元式)
优化段
中间代码(四元式)
目标代码生成器
目标代码
符号表管理
出错处理

目标代码生成器

- 任务
  - 把分析、翻译、优化后的中间代码变换成目标代码
- 输入
  - 源程序的中间表示，以及符号表中的信息
  - 类型检查

x := y + i*j
其中x、y为实型；i、j为整型

T_1 := i int* j
T_3 := inttoreal T_1
T_2 := y real+ T_3
x := T_2

目标代码生成器

- 输出
  - 绝对指令代码：能够立即执行的机器语言代码，所有地址已经定位
  - 可重新定位指令代码：待装配的机器语言模块，执行时，由连接装配程序把它们和某些运行程序连接起来，转换成能执行的机器语言代码
  - 汇编指令代码：需要经过汇编程序转换成可执行的机器语言代码

目标代码生成需要考虑的问题

- 如何充分利用计算机的指令系统的特点
- a:=a+1

```
INC a
```

```
LD    R0, a
ADD   R0, #1
ST    R0, a
```

目标代码生成需要考虑的问题

- 如何充分利用计算机的寄存器，减少目标代码中访问存贮单元的次数
  - 在寄存器分配期间，为程序的某一点选择驻留在寄存器中的一组变量
  - 在随后的寄存器指派阶段，挑出变量将要驻留的具体寄存器

代码生成

- 目标机器模型
- 一个简单代码生成器

编译原理

目标机器模型

一个抽象的计算机模型

- 具有多个通用寄存器，可用作累加器和变址器
- 运算必须在某个寄存器中进行
- 含有四种类型的指令形式

一个抽象的计算机模型

类型	指令形式	意义
直接地址型	op R_i, M	(R_i) op (M) $\Rightarrow$ R_i
寄存器型	op R_i, R_j	(R_i) op (R_j) $\Rightarrow$ R_i
变址型	op R_i, c(R_j)	(R_i) op ((R_j)+c) $\Rightarrow$ R_i
间接型	op R_i, *M op R_i, *R_j op R_i, *c(R_j)	(R_i) op ((M)) $\Rightarrow$ R_i (R_i) op ((R_j)) $\Rightarrow$ R_i (R_i) op ((((R_j)+c)) $\Rightarrow$ R_i

- op 可以是常见的二目运算符
  - 如 ADD(加)、SUB(减)、MUL(乘)、DIV(除)
- op也可以是一目运行符
  - op R_i, M 的意义为：op（M）$\Rightarrow$ R_i

一个抽象的计算机模型

- 其它指令

指 令	意 义
LD R_i, B	把B单元的内容取到寄存器R，即(B)⇒ R_i
ST R_i, B	把寄存器R_i的内容存到B单元，即(R_i)⇒ B
J X	无条件转向X单元
CMP A, B	比较A单元和B单元的值，根据比较情况设置内部二位特征寄存器CT的值：根据A<B或A=B或A>B分别置CT为0或1或2。
J< X	若CT=0，转X单元
J≤ X	若CT=0或CT=1，转X单元
J=X	若CT=1，转X单元
J≠ X	若CT≠1，转X单元
J>X	若CT=2，转X单元
J≥ X	若CT=2或CT=1，转X单元

编译原理

一个简单代码生成器

最简单的代码生成

- 不考虑代码的执行效率，目标代码生成不难

源程序: A:=(B+C)*D+E

中间代码(四元式):

T_1:=B+C

T_2:=T_1*D

T_3:=T_2+E

A:=T_3

最简单的代码生成

- 假设只有一个寄存器R_0可供使用

四元式	目标代码
$T_1:=B+C$	LD R_0, B ADD R_0, C
	ST R_0, T_1
$T_2:=T_1*D$	LD R_0, T_1 MUL R_0, D
	ST R_0, T_2
$T_3:=T_2+E$	LD R_0, T_2 ADD R_0, E
	ST R_0, T_3
$A:=T_3$	LD R_0, T_3 ST R_0, A

假设T1，T2，T3在基本块之后不再引用

```
LD   R0, B
ADD  R0, C
MUL  R0, D
ADD  R0, E
ST   R0, A
```

带寄存器分配优化的代码生成

- 以基本块为单位生成目标代码
  - 依次把四元式的中间代码变换成目标代码
  - 在基本块的范围内考虑如何充分利用寄存器
  - 进入基本块时，所有寄存器空闲
  - 离开基本块时，把存在寄存器中的现行的值存回主存中，释放所有寄存器
  - 不特别说明，所有说明变量在基本块出口之后均为非活跃变量

带寄存器分配优化的代码生成

- 在一个基本块的范围内考虑充分利用寄存器

要做到...	要知道...
尽可能留：在生成计算某变量值的目标代码时，尽可能让该变量保留在寄存器中	四元式指令：每条指令中各变量在将来会被使用的情况
尽可能用：后续的目标代码尽可能引用变量在寄存器中的值，而不访问内存	变量：每个变量现行值的存放位置
及时腾空：在离开基本块时，把存在寄存器中的现行的值放到主存中	寄存器：每个寄存器当前的使用状况

编译原理

待用信息和活跃信息

待用信息

- 如果在一个基本块内，四元式i对A定值，四元式j要引用A值，而从i到j之间没有A的其他定值，那么，我们称j是四元式i的变量A的待用信息，即下一个引用点

i: A:=B op C

…

j: D:=A op E

- 变量的符号表登记项中含有记录待用信息和活跃信息的栏

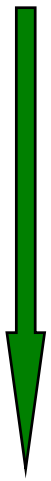
待用信息和活跃信息的表示

- 二元组(x, x)表示变量的待用信息和活跃信息
  - 第1元：i表示待用信息， ^表示非待用
  - 第2元：y表示活跃，^表示非活跃
- 待用信息和活跃信息的变化
  - (x，x)→(x，x)，用后者更新前者

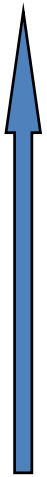
变量名	初始状态→信息链
T	(^,y)→(3,y)→(^,^)
A	(^,^)→(2,y)→(1,y)
B	(^,^)→(1,y)
C	(^,^)→(2,y)
U	(^,^)→(4,y)→(3,y)→(^,^)
V	(^,^)→(4,y)→(^,^)
W	(^,y)→(^,^)

待用信息和活跃信息的计算

- 例：基本块
  1. T:=A-B
  2. U:=A-C
  3. V:=T+U
  4. W:=V+U
- 设W是基本块出口之后的活跃变量。



序号	四元式	左值	左操作数	右操作数
(1)	T:=A-B	(3, y)	(2, y)	(^,^)
(2)	U:=A-C	(3, y)	(^, ^)	(^,^)
(3)	V:=T+U	(4, y)	(^, ^)	(4, y)
(4)	W:=V+U	(^, y)	(^, ^)	(^,^)



计算待用信息和活跃信息的算法

1. 把基本块中各变量的符号表中的待用信息栏填为“非待用”，并根据该变量在基本块出口之后是不是活跃的，把其中的活跃信息栏填为“活跃”或“非活跃”；
2. 从基本块出口到入口由后向前依次处理各个四元式i:A:=B op C：
   1) 把符号表中变量A的待用信息和活跃信息附加到四元式i上；
   2) 把符号表中A的待用信息和活跃信息分别置为“非待用”和“非活跃”；
   3) 把符号表中变量B和C的待用信息和活跃信息附加到四元式i上；
   4) 把符号表中B和C的待用信息均置为i，活跃信息均置为“活跃”；

变量名	初始状态→信息链
T	(^,y)
A	(^,^) →(n,y)
B	(^,^)
C	(^,^)

序号	四元式	左值	左操作数	右操作数
j	A:=B op T			
…	…	…	…	…
i	A:=B op C			
…	…	…	…	…

待用信息和活

- 例：基本块
  1. T:=A-B
  2. U:=A-C
  3. V:=T+U
  4. W:=V+U
- 设W是基本块出口之后的活跃变量。

变量名	初始状态→信息链
T	(^,^) → (3,y) → (^,^)
A	(^,^) → (2,y) → (1,y)
B	(^,^) → (1,y)
C	(^,^) → (2,y)
U	(^,^) → (4,y) → (3,y) → (^,^)
V	(^,^) → (4,y) → (^,^)
W	(^,y) → (^,^)

序号	四元式	左值	左操作数	右操作数
(1)	T:=A-B	(3,y)	(2,y)	(^,^)
(2)	U:=A-C	(3,y)	(^,^)	(^,^)
(3)	V:=T+U	(4,y)	(^,^)	(4,y)
(4)	W:=V+U	(^,y)	(^,^)	(^,^)

编译原理

变量地址描述和寄存器描述

变量地址描述和寄存器描述

要知道...

四元式指令：每条指令中各变量在将来会被使用的情况

变量：每个变量现行值的存放位置

寄存器：每个寄存器当前的使用状况

变量地址描述和寄存器描述

- 变量地址描述数组AVALUE
  - 动态记录各变量现行值的存放位置
  - AVALUE[A]={R_1, R_2, A}
- 寄存器描述数组RVALUE
  - 动态记录各寄存器的使用信息
  - RVALUE[R]={A,B}

变量地址描述和寄存器描述

- 对于四元式A:=B
  - 如果B的现行值在某寄存器R_i中，则无须生成目标代码
  - 只须在RVALUE(R_i)中增加一个A，(即把R_i同时分配给B和A)，并把AVALUE(A)改为R_i

编译原理

代码生成算法

代码生成算法

- 对每个四元式: i: A:=B op C，依次执行：

1. 以四元式: i: A:=B op C 为参数，调用函数过程GETREG(i: A:=B op C)，返回一个寄存器R，用作存放A的寄存器。
2. 利用AVALUE[B]和AVALUE[C]，确定B和C现行值的存放位置B′和C′ 。如果其现行值在寄存器中，则把寄存器取作B′ 和C′
3. 如果B′ ≠R，则生成目标代码：

```
LD  R, B′
op  R, C′
```

否则生成目标代码 op R, C′

如果B′ 或C′ 为R，则删除AVALUE[B]或AVALUE[C]中的R。

4. 令AVALUE[A]={R}, RVALUE[R]={A}。
5. 若B或C的现行值在基本块中不再被引用，也不是基本块出口之后的活跃变量，且其现行值在某寄存器R_k中，则删除RVALUE[R_k]中的B或C以及AVALUE[B]或AVALUE[C] 中的R_k ，使得该寄存器不再为B或C占用。

寄存器分配算法

- 寄存器分配：GETREG(i: A:=B op C) 返回一个用来存放A的值的寄存器

1. 尽可能用B独占的寄存器
2. 尽可能用空闲寄存器
3. 抢占非空闲寄存器

寄存器分配算法

1. 尽可能用B独占的寄存器
2. 尽可能用空闲寄存器
3. 抢占非空闲寄存器

- 寄存器分配：GETREG(i: A:=B op C) 返回一个用来存放A的值的寄存器

1. 如果B的现行值在某个寄存器R_i中，RVALUE[R_i]中只包含B，此外，或者B与A是同一个标识符，或者B的现行值在执行四元式A:=B op C之后不会再引用，则选取R_i为所需要的寄存器R，并转4；
2. 如果有尚未分配的寄存器，则从中选取一个R_i为所需要的寄存器R，并转4；
3. 从已分配的寄存器中选取一个R_i为所需要的寄存器R。最好使得R_i满足以下条件：占用R_i的变量的值也同时存放在该变量的贮存单元中，或者在基本块中要在最远的将来才会引用到或不会引用到。为R_i中的变量生成必要的存数指令。
4. 给出R，返回。

生成存数指令

- 对RVALUE[R_i]中每一变量M，如果M不是A，或者如果M是A又是C，但不是B并且B也不在RVALUE[R_i]中，则

  (1) 如果AVALUE[M]不包含M，则生成目标代码 ST R_i, M

  (2) 如果M是B，或者M是C但同时B也在RVALUE[R_i]中，则令AVALUE[M]={M， R_i}，否则令AVALUE[M]={M}

  (3) 删除RVALUE[R_i]中的M

为基本块生成代码示例

- 例：基本块
  1. T:=A-B
  2. U:=A-C
  3. V:=T+U
  4. W:=V+U
- 设W是基本块出口之后的活跃变量，只有R0和R1是可用寄存器。

为基本块生成代码示例

- 例：基本块
  1. T:=A-B
  2. U:=A-C
  3. V:=T+U
  4. W:=V+U
- 设W是基本块出口之后的活跃变量，只有R0和R1是可用寄存器。

序号	四元式	左值	左操作数	右操作数
(1)	T:=A-B	(3,y)	(2,y)	(^,^)
(2)	U:=A-C	(3,y)	(^,^)	(^,^)
(3)	V:=T+U	(4,y)	(^,^)	(4,y)
(4)	W:=V+U	(^,y)	(^,^)	(^,^)

为基本块生

序号	四元式	左值	左操作数	右操作数
(1)	T:=A-B	(3,y)	(2,y)	(^,^)
(2)	U:=A-C	(3,y)	(^,^)	(^,^)
(3)	V:=T+U	(4,y)	(^,^)	(4,y)
(4)	W:=V+U	(^,y)	(^,^)	(^,^)

中间代码	目标代码	RVALUE	AVALUE
T:=A - B	LD R_0, A SUB R_0, B	R_0含有T	T在R_0中
U:=A - C	LD R_1, A SUB R_1, C	R_0含有T R_1含有U	T在R_0中 U在R_1中
V:=T + U	ADD R_0, R_1	R_0含有V R_1含有U	V在R_0中 U在R_1中
W:=V + U	ADD R_0, R_1 ST R_0, W	R_0含有W	W在R_0中

小结

- 目标代码生成的概念
- 代码生成着重考虑的问题
- 一个简单代码生成算法
  - 待用信息，活跃信息
  - 寄存器描述信息
  - 变量地址描述信息
  - 寄存器分配算法