编译原理概览

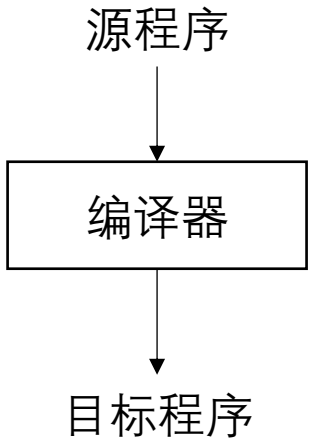
邢明杰
mingjie@iscas.ac.cn

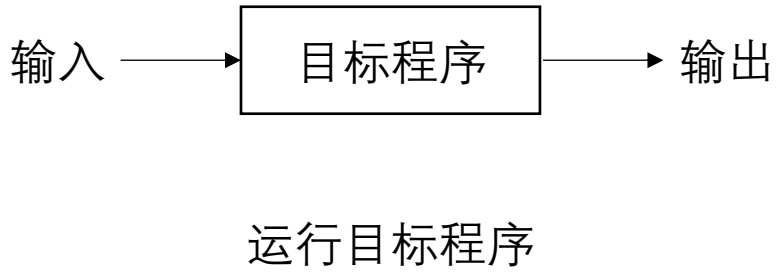
关键词

- 前端
  - 词法分析：正则表达式、正则定义、有限自动机
  - 语法分析：上下文无关文法、下推自动机、自顶向下语法分析、自底向上语法分析
  - 语义分析、中间代码生成：语法制导翻译
- 中间代码
  - 抽象语法树、三地址代码、有向无环图、静态单赋值
- 优化
  - 控制流分析：基本块、控制流图
  - 数据流分析：数据流迭代方程、格/半格、最大不动点、活跃变量分析
  - 优化：冗余消除、常量传播
- 后端
  - 指令选择：展开+窥孔优化、树模式匹配
  - 指令调度：流水线、列表调度、软件流水线
  - 寄存器分配：活跃范围、图着色、线性扫描

什么是编译器（compiler）？

- 编译器是一个语言处理程序，可以将高级语言（源语言）编写的程序，翻译成为等价的低级语言（目标语言）编写的程序。





A **compiler** was originally a program that "compiled" subroutines [a link-loader]. When in 1954 the combination "algebraic compiler" came into use, or rather into misuse, the meaning of the term had already shifted into the present one.

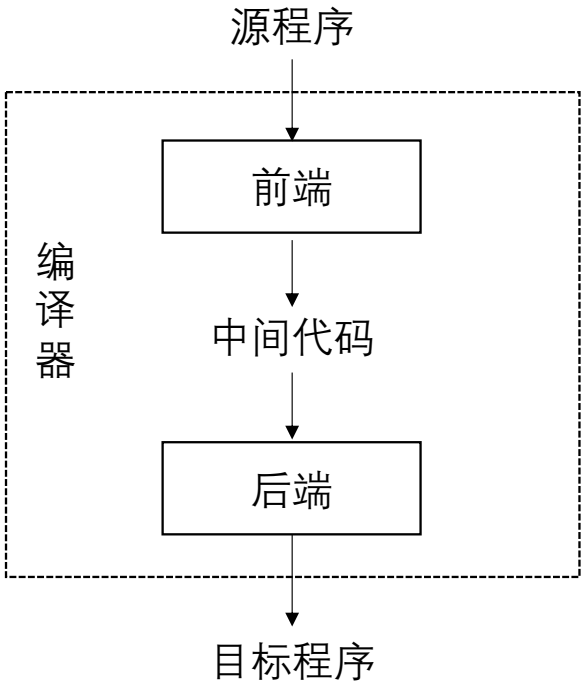
Bauer and Eickel [1975]

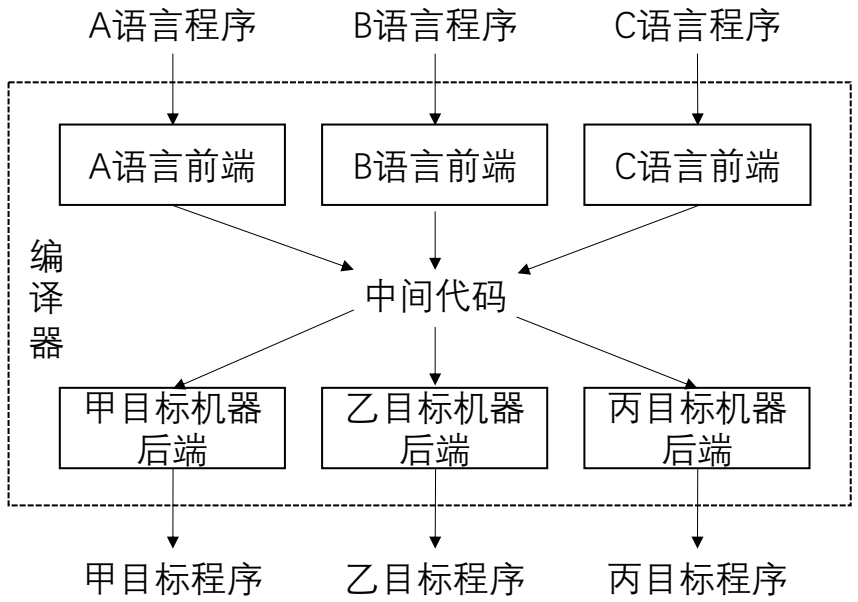
编译器（compiler）：原指一种将各个子程序装配组合到一起的程序 [连接-装配器]。当1954年出现了（确切地说是误用了）复合术语"代数编译器"（algebraic compiler）之后，这个术语的意思变成了现在的含义。

——《现代编译原理C语言描述》Andrew W. Appel等著，赵克佳等译

前端、后端

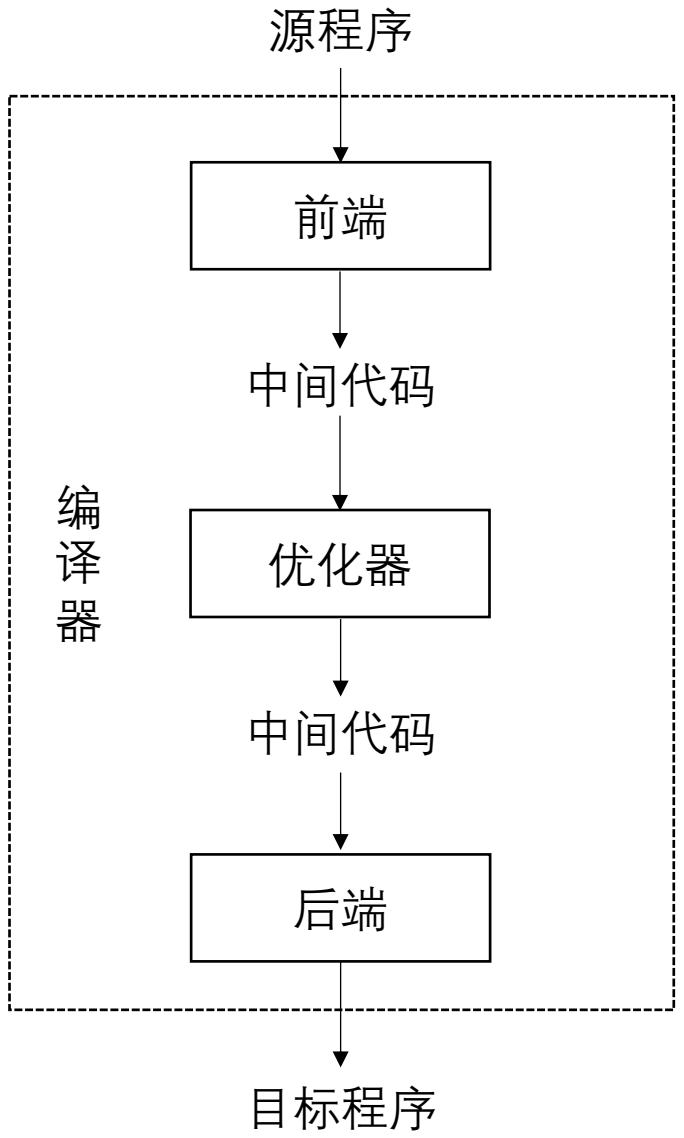
- 针对输入输出对编译器进行模块化分层设计
  - 前端（front-end）负责解析源程序
  - 后端（back-end）负责生成目标程序
  - 中间代码/中间表示（IR）是源程序的内部表示形式
- 当然，也可以在解析过程中（通过语法制导翻译技术）直接生成目标代码
- 分层的好处
  - 多语言、多目标支持
  - N+M工作量 vs. N*M工作量





更高的目标：优化

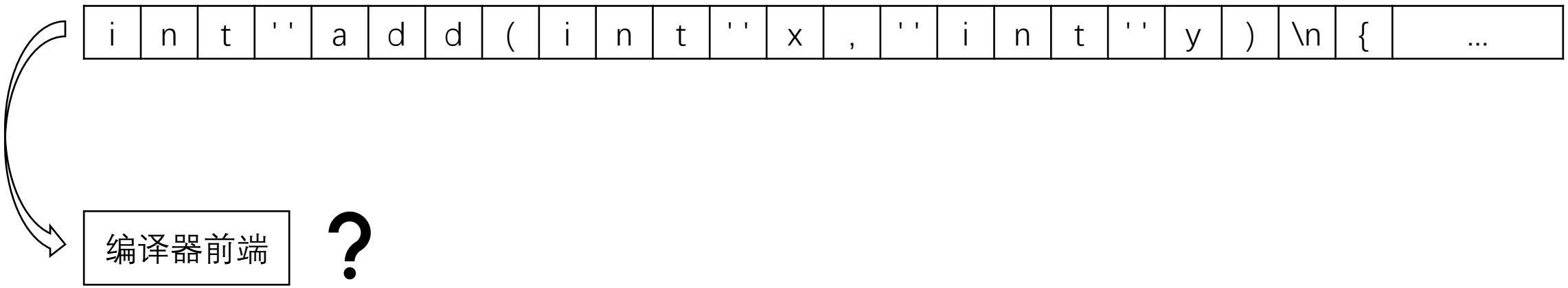
- 进一步分层
  - 前端（front-end）负责解析源程序
  - 后端（back-end）负责生成目标程序
  - 优化器（optimizer）负责（通常是体系结构无关的）代码优化
  - 中间代码/中间表示（IR）是源程序的内部表示形式
- 当然，在后端代码生成过程中也会有优化（通常是体系结构相关的）
- （没有所谓的中端，end：末端）



前端：如何解析源程序？

- 编译器读入的只是一个字符流
- 需要从字符流中识别出高级语言编写的程序

```
int add(int x, int y)
{
  return x + y;
}
```



欲解析之，先形式化描述之

- 自然语言（英文）中的一个句子由单词组成，并遵循一定的语法
- 类似的，程序语言也如此
  - 使用正则表达式来形式化描述一个单词或词法单元（token）
    - 可选/并 a | b，连接 ab，Kleene闭包 a^*，正闭包 a^+

letter_(letter_|digit)*　　id → letter_(letter_|digit)*　正则定义

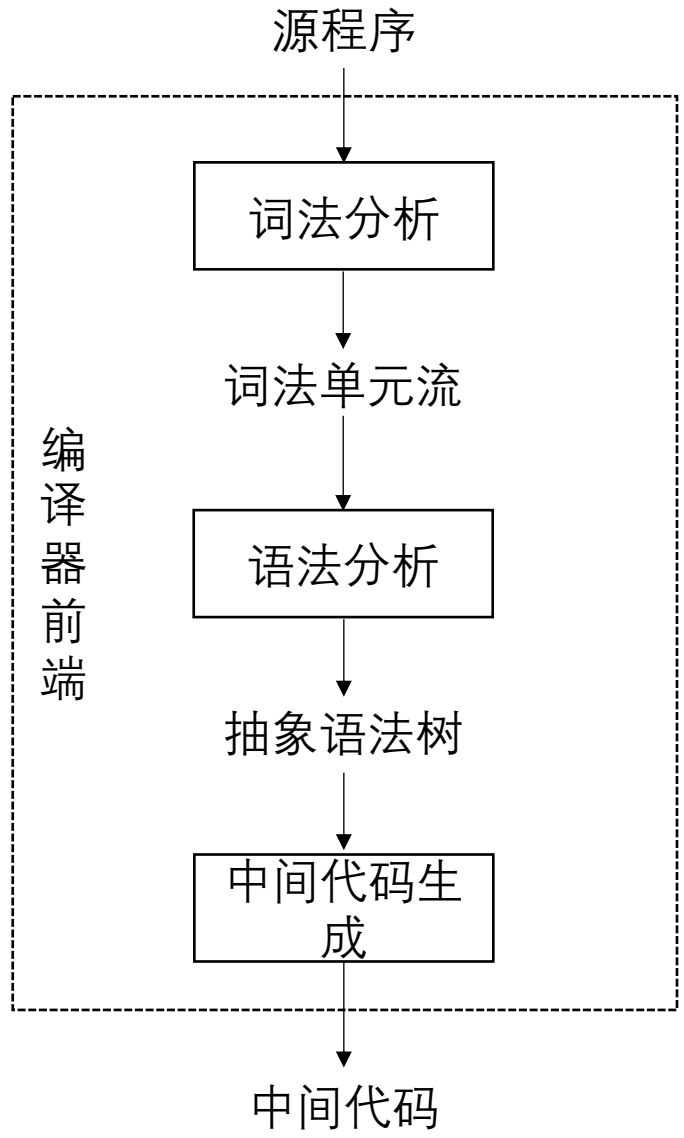
  - 使用上下文无关文法（context-free grammar，CFG）或BNF范式来形式化描述一个语言的语法
    - 产生式，终结符，非终结符，起始符号

E → E + E | E * E | (E) | **id**

- 对源程序的解析，相当于识别其是否为程序语言的一个合法句子
  - 从起始符号开始，根据产生式推导出该句子：自顶向下（构建语法分析树）语法分析
  - 根据产生式将句子规约成起始符号：自底向上（构建语法分析树）语法分析

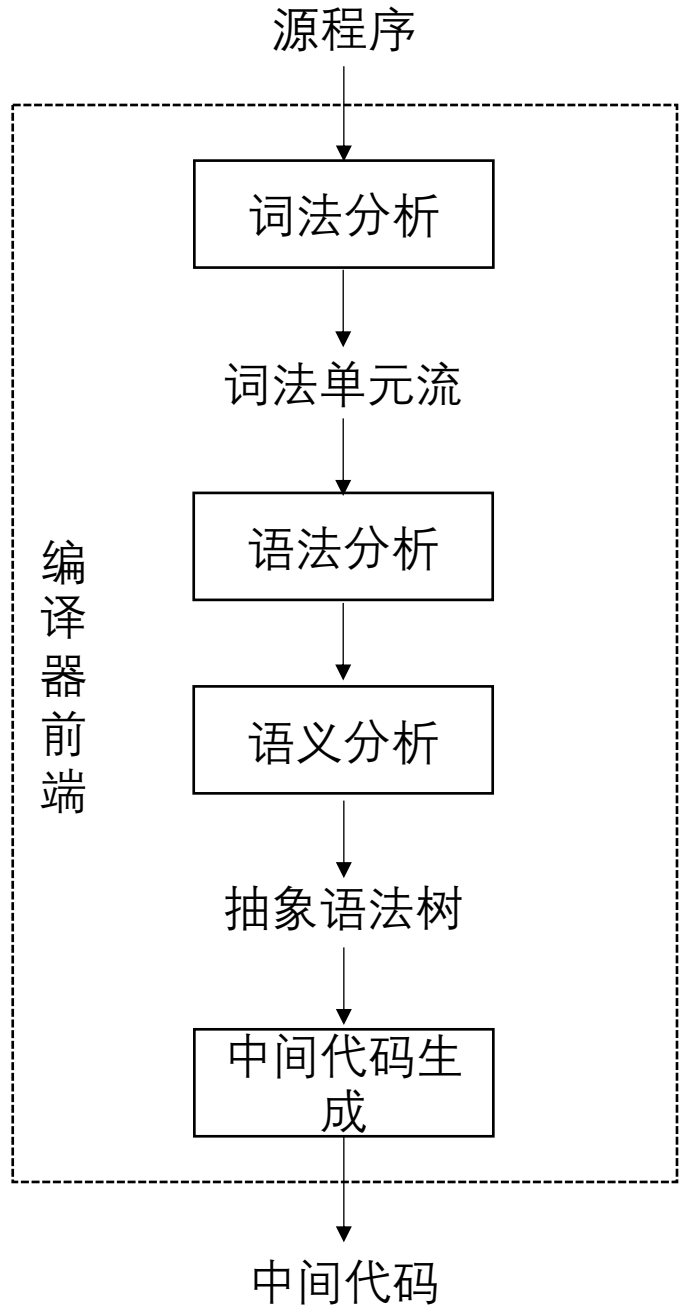
词法分析、语法分析

- 可以将识别过程拆分为两步：
  - 识别单词：扫描器（scanner）负责词法分析，将输入的字符流识别为词法单元流
  - 识别句子：解析器（parser）负责语法分析，对词法单元流进行解析（并创建抽象语法树）
  - 中间代码生成负责生成IR
- 当然，也可以在解析过程中（通过语法制导翻译技术）直接生成中间代码或机器代码



语义分析

- 符合语法的语言未必有意义，有些语言规范无法用上下文无关文法描述
  - 例如先定义后使用，函数调用实参与形参一致，数据类型一致等
- 进一步划分
  - 词法分析或扫描器（scanner）负责将输入的字符流识别为词法单元流
  - 语法分析或解析器（parser）负责对词法单元流进行解析（并创建抽象语法树）
  - 语义分析负责检查源程序是否符合语义规范
  - 中间代码生成负责生成IR
- 实践中，可以在创建AST之前进行语义分析



词法分析：如何识别一个单词？

- 根据正则表达式手工编写识别代码

识别关键字和标识符

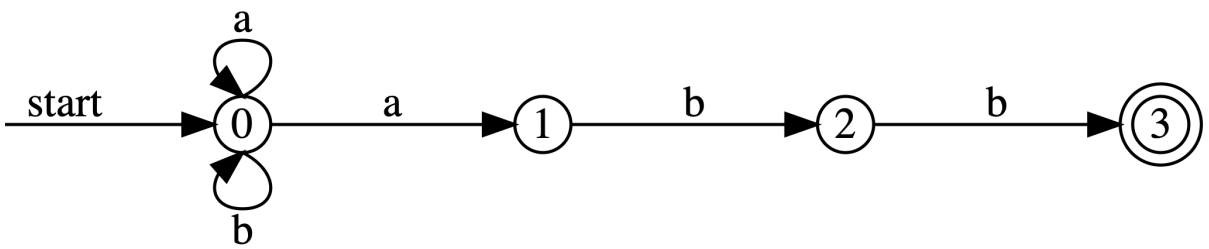
id → letter_(letter_|digit)*

```
if (isalpha(peek) || peek == '_') {
  std::string s;
  do {
    s.push_back(peek);
    readch();
  } while(isalnum(peek) || peek == '_');

  if (words.find(s) != words.end())
    return words[s];
  Token w = Token::Word(s.c_str(), Tag::ID);
  words[s] = w;
  return w;
}
```

词法分析：如何识别一个单词？

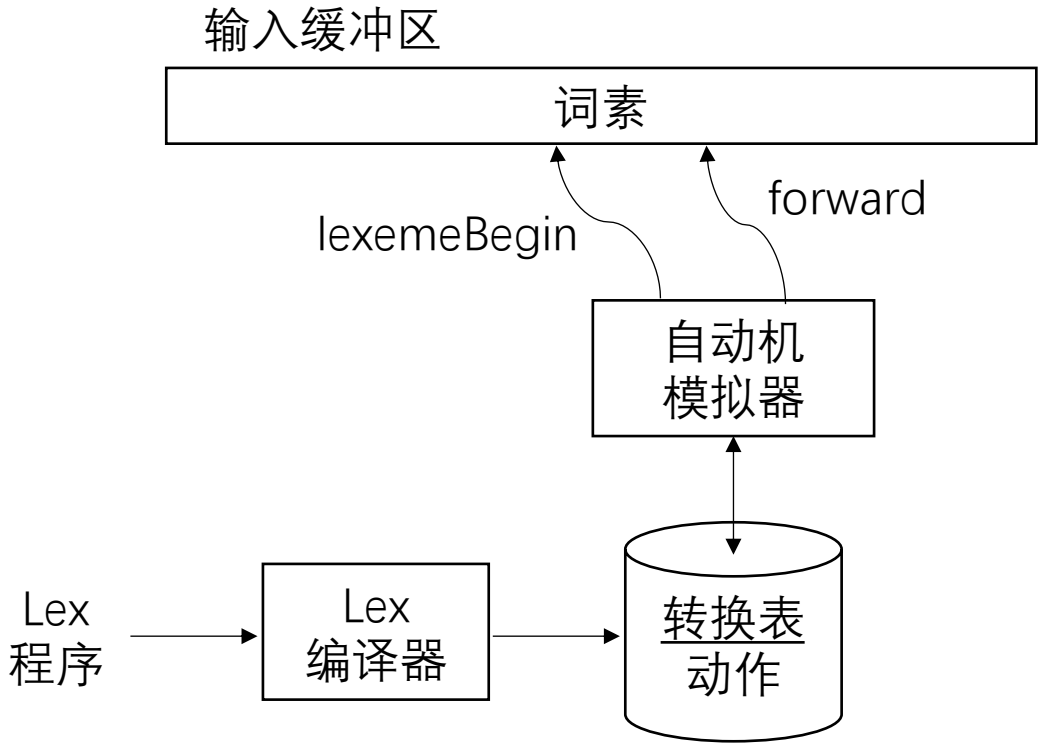
- 将正则表达式转换成有限自动机（NFA或DFA），自动生成词法分析器
  - 正则表达式与NFA/DFA等价



接受(a|b)*abb的NFA

状态	a	b	ϵ
0	{0, 1}	{0}	Ø
1	Ø	{2}	Ø
2	Ø	{3}	Ø
3	Ø	Ø	Ø

对应的转换表



生成的词法分析器的结构

语法分析：如何识别一个句子？

- 根据上下文无关文法手工编写递归下降识别代码
- 实践中，先转换成EBNF等增强描述形式，更容易手工实现

给定文法：

1) S → cAd
2) A → ab
3) A → a

增强形式文法（类似EBNF）：

1) S → cAd
2) A → a {b}

大括号表示可选

手动构造的语法分析程序

```
void S() {
    if (当前输入符号==c)
        读入下一个输入符号
    else
        发生错误

    调用过程 A();

    if (当前输入符号==d)
        成功完成
    else
        发生错误
}

void A() {
    if (当前输入符号==a)
        读入下一个输入符号
    else
        发生错误

    if (当前输入符号==b)
        读入下一个输入符号
}
```

语法分析：如何识别一个句子？

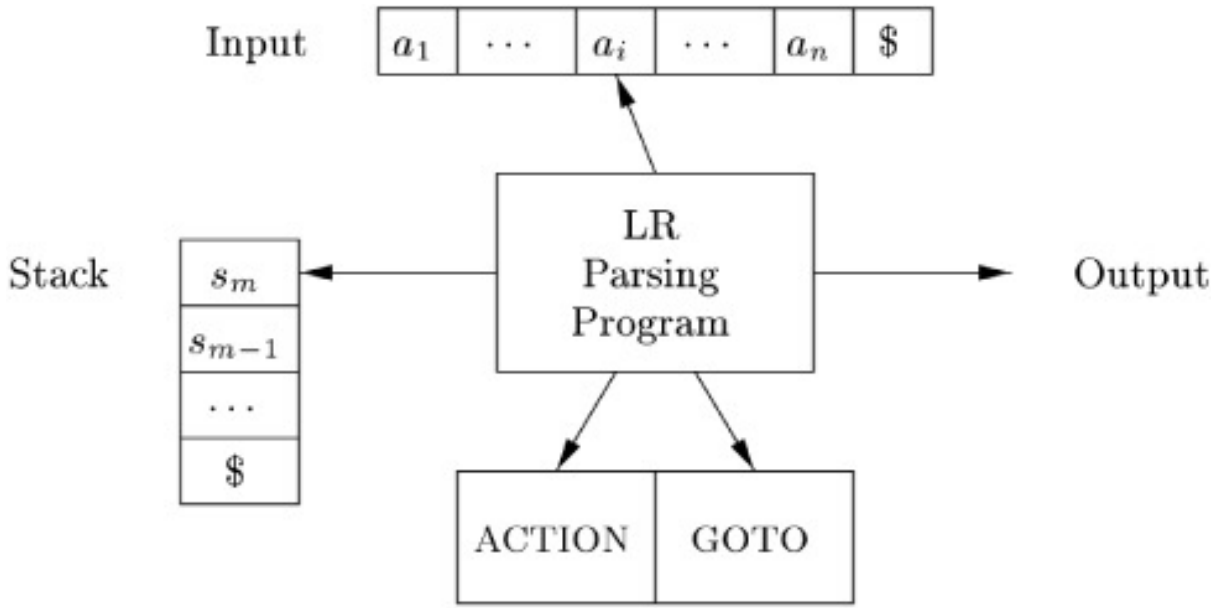
- 将上下文无关文法转换成下推自动机（PDA），自动生成语法分析器
  - 上下文无关文法与PDA等价

带有编号的产生式：

(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to F$
(5) $F \to (E)$
(6) $F \to \mathbf{id}$

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

LR语法分析表



一个LR语法分析器的模型

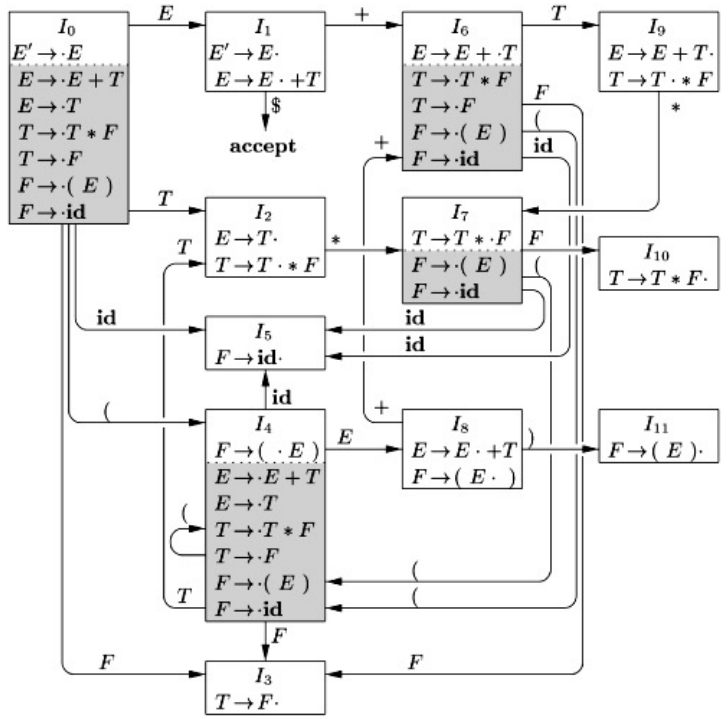
编译原理为什么这么难？

- 背后深奥的自动机理论，各种文法和相应的语法分析技术；但这些主要是用于自动构造，体现在flex、bison这些工具中
- 龙书前端理论介绍的非常详细，唯有表达式的语法分析技术比较落后；更高效的表达式语法分析技术（在llvm中使用）：
  - https://eli.thegreenplace.net/2012/08/02/parsing-expressions-by-precedence-climbing

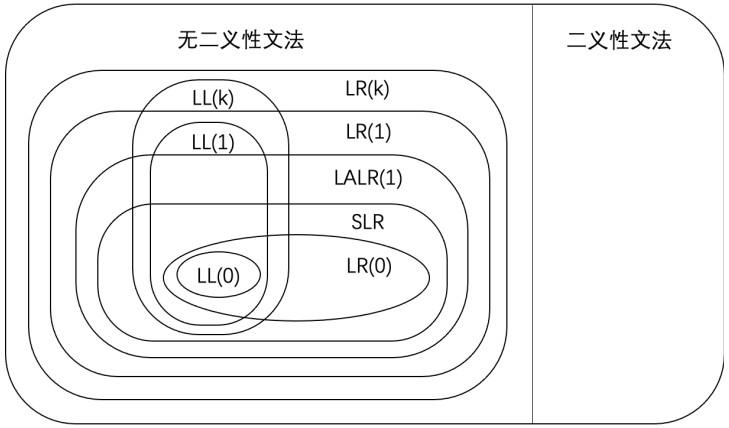
$$
\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +\ TE' \mid \epsilon \\
T &\rightarrow FT' \\
T' &\rightarrow *\ FT' \mid \epsilon \\
F &\rightarrow (\ E\ ) \mid \textbf{id}
\end{aligned}
$$

L: left-to-right scanning
L: leftmost derivation
R: rightmost derivation in reverse

无左递归版本，可用于自顶向下的语法分析



表达式文法的LR(0)自动机



各类文法的层次

不仅是识别，更重要的是翻译

- 自动机仅用于识别一个单词或者句子是否符合形式化描述
  - 只回答Yes or No
- 需要将代码生成等动作嵌入到识别过程中
  - 上下文无关文法和属性及规则的结合，形成语法制导定义（SDD）
  - 在产生式体中嵌入程序片段（称为语义动作），形成语法制导的翻译方案（syntax-directed translation scheme, SDT）

产生式	语义规则
1) L → E **n**	L.val = E.val
2) E → E_1 + T	E.val = E_1.val + T.val
3) E → T	E.val = T.val
4) T → T_1 * F	T.val = T_1.val * F.val
5) T → F	T.val = F.val
6) F → (E)	F.val = E.val
7) F → **digit**	F.val = **digit**.lexval

计算器的语法制导定义

1) L → E **n**	{ print(E.val); }
2) E → E_1 + T	{ E.val = E_1.val + T.val; }
3) E → T	{ E.val = T.val; }
4) T → T_1 * F	{ T.val = T_1.val * F.val; }
5) T → F	{ T.val = F.val; }
6) F → (E)	{ F.val = E.val; }
7) F → **digit**	{ F.val = **digit**.lexval; }

计算器的语法制导翻译方案

语法制导的翻译

- 将SDD转换为SDT，根据SDT手动编写代码或者用于自动构造代码中

产生式	语义规则
S → **while** (C) S_1	L1 = new(); L2 = new(); S_1.next = L1; C.false = S.next; C.true = L2; S.code = **label** || L1 || C.code || **label** || L2 || S_1.code

while语句的SDD

S → **while** ({ L1 = new(); L2 = new(); C.false = S.next; C.true = L2; print("label", L1); }
C) { S_1.next = L1; print("label", L2);}
S_1

while 语句的 SDT

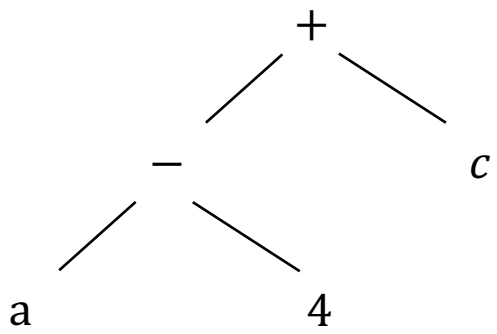
```
string S(label next) {
    label L1, L2; /* 局部标号 */
    if (当前输入 == 词法单元while) {
        读取输入；
        检查'('是下一个输入符号，并读取输入；
        L1 = new();
        L2 = new();
        print("label", L1);
        C(next, L2);
        检查')'是下一个输入符号，并读取输入；
        print("label", L2);
        S(L1);
    }
    else /* 其他语句类型 */
}
```

在递归下降语法分析过程中进行翻译

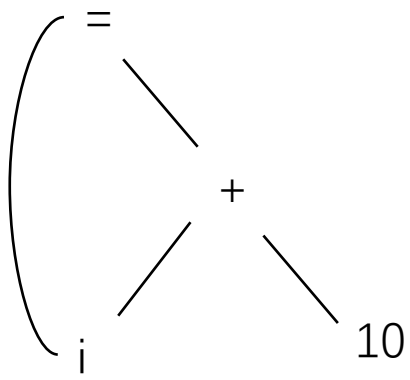
- SDD/SDT可用于语义分析、生成抽象语法树、中间代码或机器代码

中间代码

- 抽象语法树反映了输入程序的语言层次结构
- 三地址代码更加接近目标机器代码，反映了程序执行的操作流
- 有向无环图（DAG）不仅复用树中的公共部分，还可以用于多种优化
- 静态单赋值（SSA）将数据流信息编码在中间表示中，使得优化分析更高效



抽象语法树



i=i+10的DAG表示

$t = b - c$

$t = a * t$

$t = a + t$

$t = t * d$

三地址代码

$t_1 = b - c$

$t_2 = a * t_1$

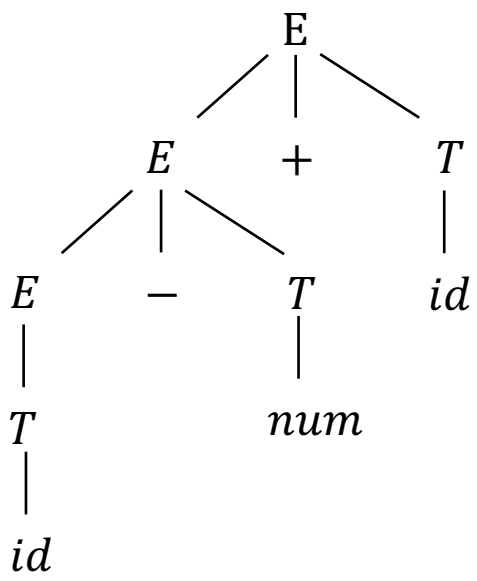
$t_3 = a + t_2$

$t_4 = t_1 * d$

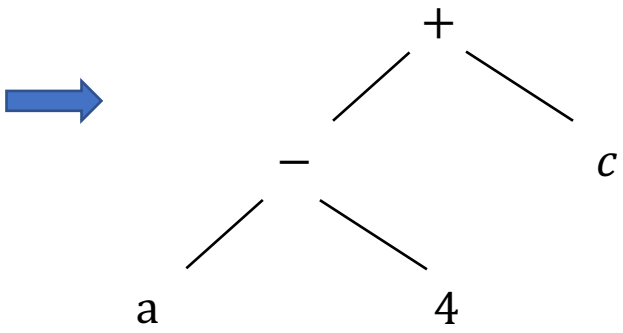
SSA

语法分析树和抽象语法树

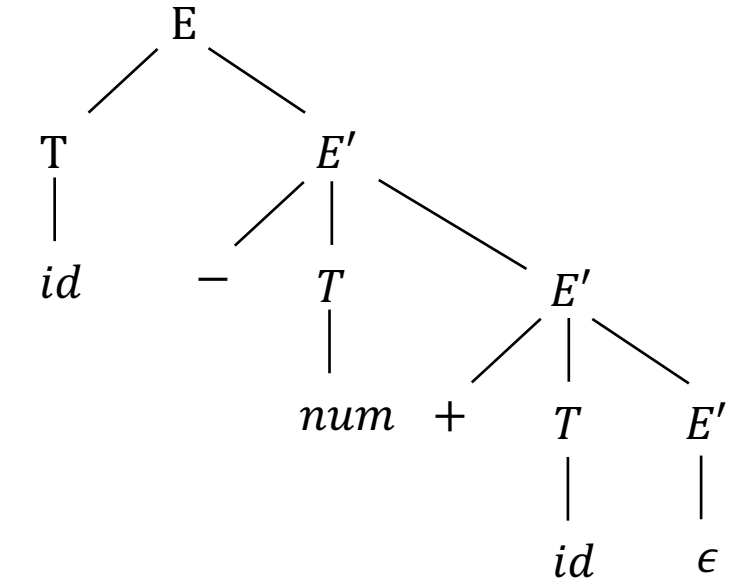
- 语法分析树（parse tree），是推导的图形化表示，结点为文法符号
- 抽象语法树（abstract syntax tree），是程序的中间表示，结点为程序的构造
- 同一个程序，使用不同的文法描述，对应有不同的分析树，但会生成同一个抽象语法树



a – 4 + c 采用LR文法的语法分析树



对应的抽象语法树



LL文法对应的语法分析树

有什么可优化的？以性能为例

- 冗余代码，无用代码，以及死代码
  - 从高层语言到低层中间代码的编译过程中引入了冗余（例如重复的加载/存储）
  - 源代码中的冗余（例如，重复计算）
  - 源代码中的死代码（例如，未打开的调试代码）
  - 优化变换引入的（复制传播经常会把一些复制语句变成死代码）
- 充分利用硬件的并行性
  - 指令级并行（流水线、软件流水线）
  - 数据级并行（SIMD、自动向量化）
  - 任务级并行（OpenMP）
- 其他硬件相关的机会
  - 缓存局部性（预取，循环分块）
  - 硬件加速指令（DSP特有指令）

优化器：如何优化？

- 优化器读入的（通常）只是一个（三地址代码）指令流
- 需要从指令流中找到优化机会，获取相关信息

```
 1)  i = 1
 2)  j = 1
 3)  t1 = 10 * i
 4)  t2 = t1 + j
 5)  t3 = 8 * t2
 6)  t4 = t3 - 88
 7)  a[t4] = 0.0
 8)  j = j + 1
 9)  if j <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```

优化器

?

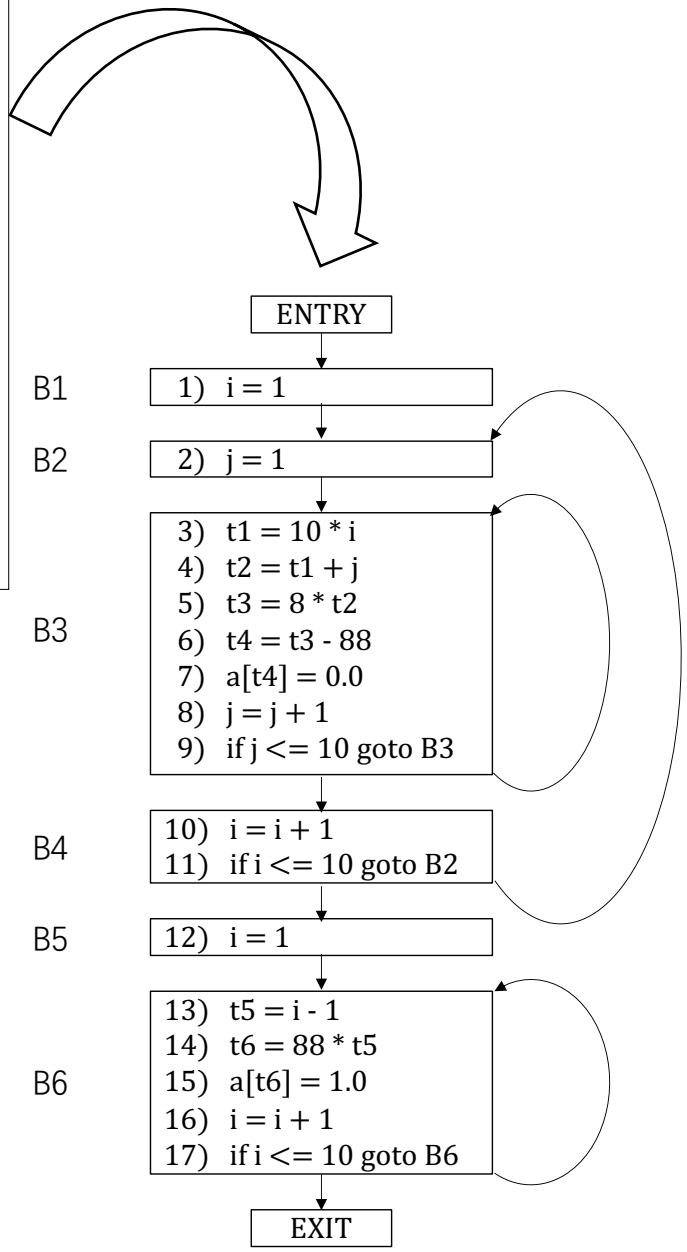
欲做优化，先做分析

- 分析是优化变换的基础
  - 编译器需要对程序的结构和相关信息有所了解
  - 用来确保优化变换是安全的、高效的；分析越精确，优化越高效
- 控制流分析
  - 用来确定过程内的控制流层次结构，例如
    - 基本块、控制流图：是其他分析的基础
    - 支配结点、支配树：可用于加速数据流迭代方程收敛，构造SSA
    - 循环：循环变换的基础
    - 调用图：过程间优化
- 数据流分析
  - 用来确定与数据处理有关的全局信息，例如
    - 到达定值：探测未定值先使用
    - 活跃变量：寄存器分配的基础
    - 可用表达式：可用于公共子表达式删除

控制流图构造算法

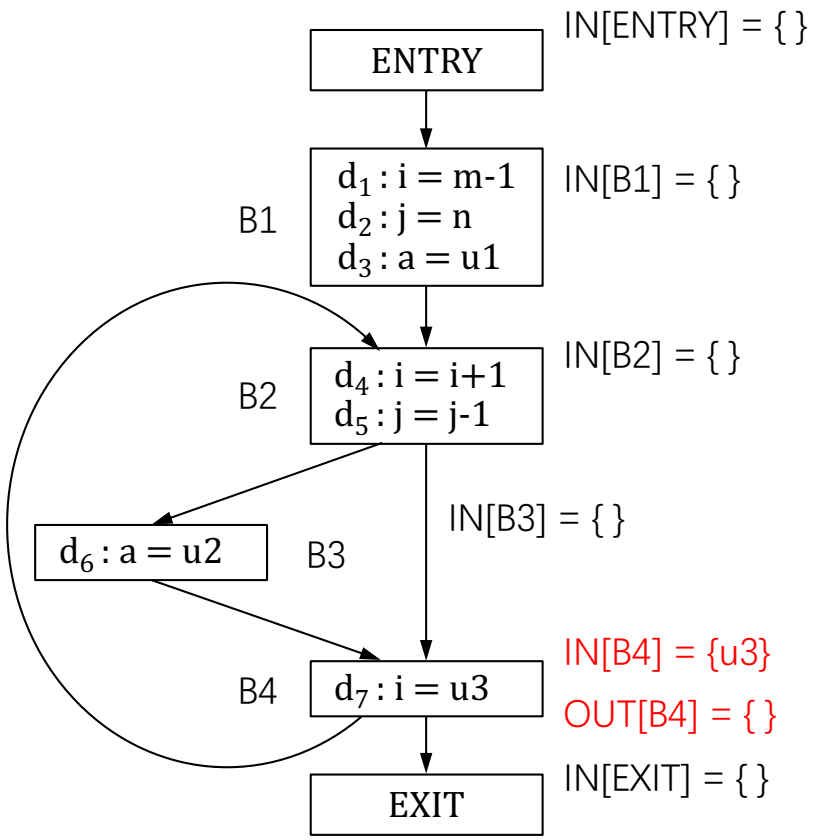
```
 1)  i = 1
 2)  j = 1
 3)  t1 = 10 * i
 4)  t2 = t1 + j
 5)  t3 = 8 * t2
 6)  t4 = t3 - 88
 7)  a[t4] = 0.0
 8)  j = j + 1
 9)  if j <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```

- 首先，确定基本块的首指令
  - 第一个三地址指令
  - 分支跳转的目标指令
  - 紧跟在分支跳转指令之后的指令
- 其次，确定基本块
  - 每个首指令对应于一个基本块：从首指令开始到下一个首指令
- 然后，确定基本块的边
  - B的结尾指令是一条跳转到C的开头的条件/无条件语句
  - C紧跟在B之后，且B的结尾不是无条件跳转语句
- 最后，添加入口和出口结点
  - 入口到第一条指令有一条边
  - 任何可能最后执行的基本块到出口有一条边

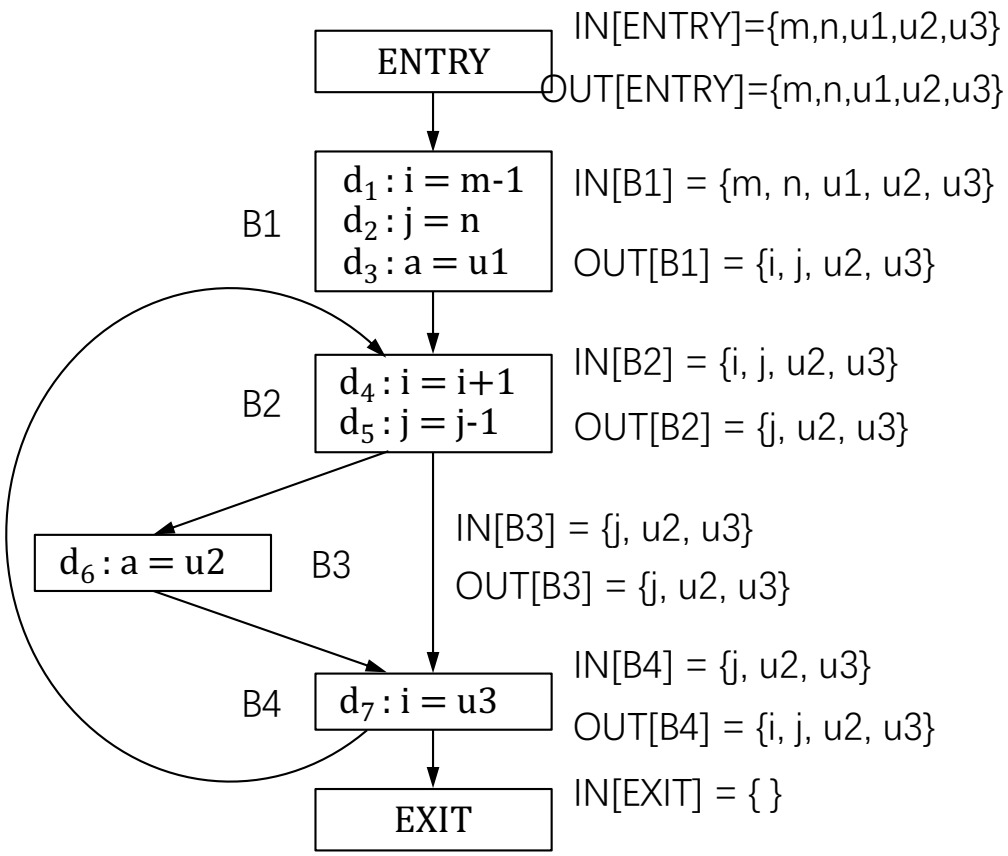


数据流分析

- 用于获取有关数据沿着程序执行路径流动的相关信息
  - 给定程序点上的初始值，沿着所有可能执行路径进行传播，直到达到一个不动点



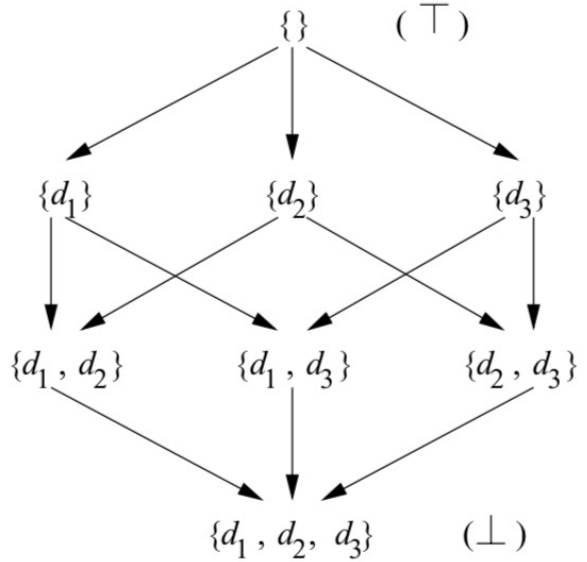
活跃变量分析



数据流迭代方程及背后的理论

- 如果框架的半格是单调的，且高度有穷，则迭代算法必定收敛；如果算法收敛，其结果就是数据流方程组的一个解
- 如果框架是单调的：$f(x \wedge y) \leq f(x) \wedge f(y)$，则找到的解就是数据流方程组的最大不动点（最精确的）

交半格：
集合V，交汇运算Λ，
满足等幂、可交换、
结合律。



格图

```
1)  OUT[ENTRY] = v_ENTRY;
2)  for (each basic block B other than ENTRY) OUT[B] = ⊤;
3)  while (changes to any OUT occur)
4)          for (each basic block B other than ENTRY) {
5)                  IN[B] = ⋀_{P a predecessor of B} OUT[P];
6)                  OUT[B] = f_B(IN[B]);
            }
```

前向数据流问题的迭代算法

```
1)  IN[EXIT] = v_EXIT;
2)  for (each basic block B other than EXIT) IN[B] = ⊤;
3)  while (changes to any IN occur)
4)          for (each basic block B other than EXIT) {
5)                  OUT[B] = ⋀_{S a successor of B} IN[S];
6)                  IN[B] = f_B(OUT[B]);
            }
```

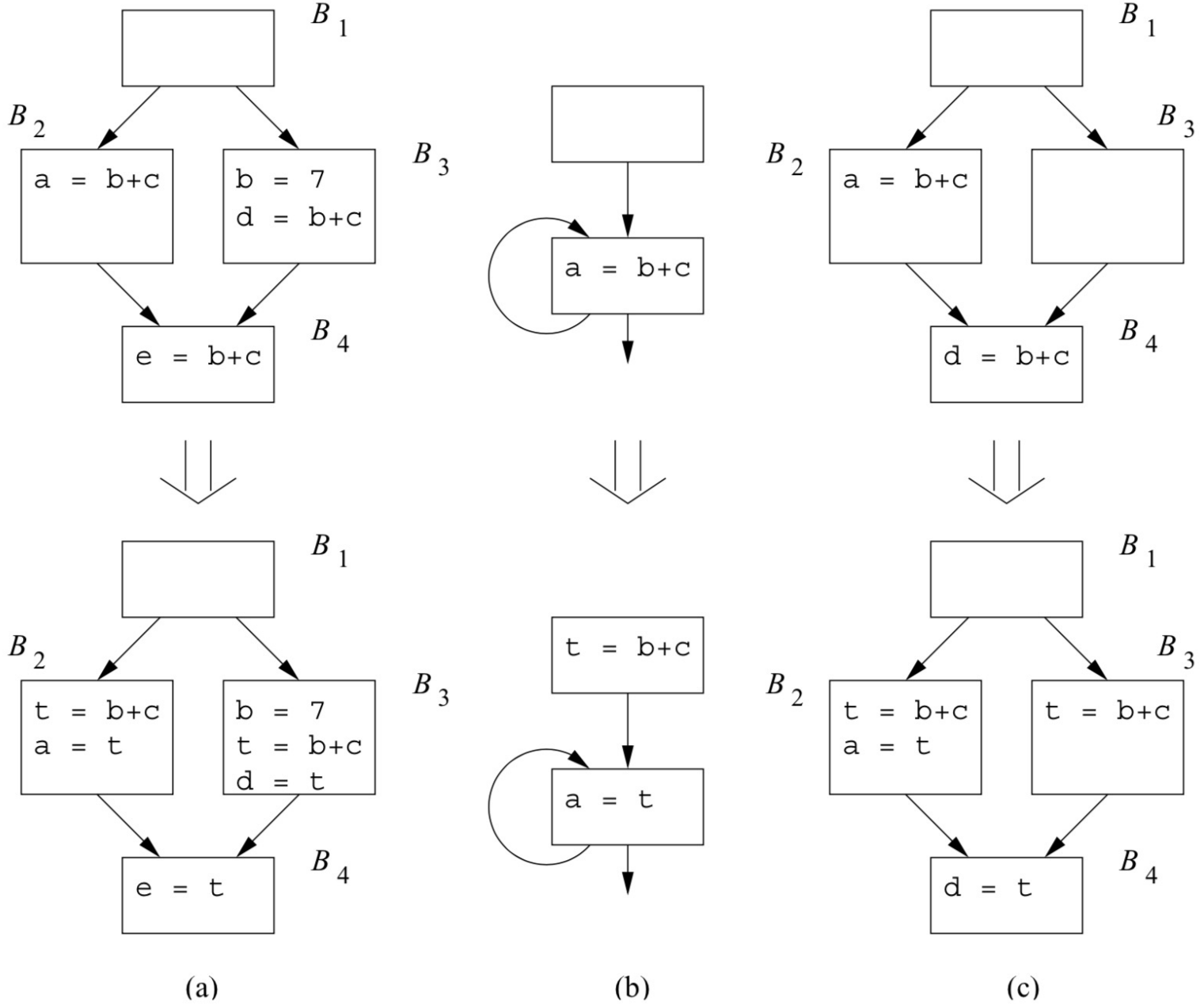
逆向数据流问题的迭代算法

冗余消除

a. 公共子表达式
   - B4中的表达式b+c在B1到B4的所有路径上都冗余

b. 循环不变表达式
   - 假设变量b和c在循环中没有被重新定值

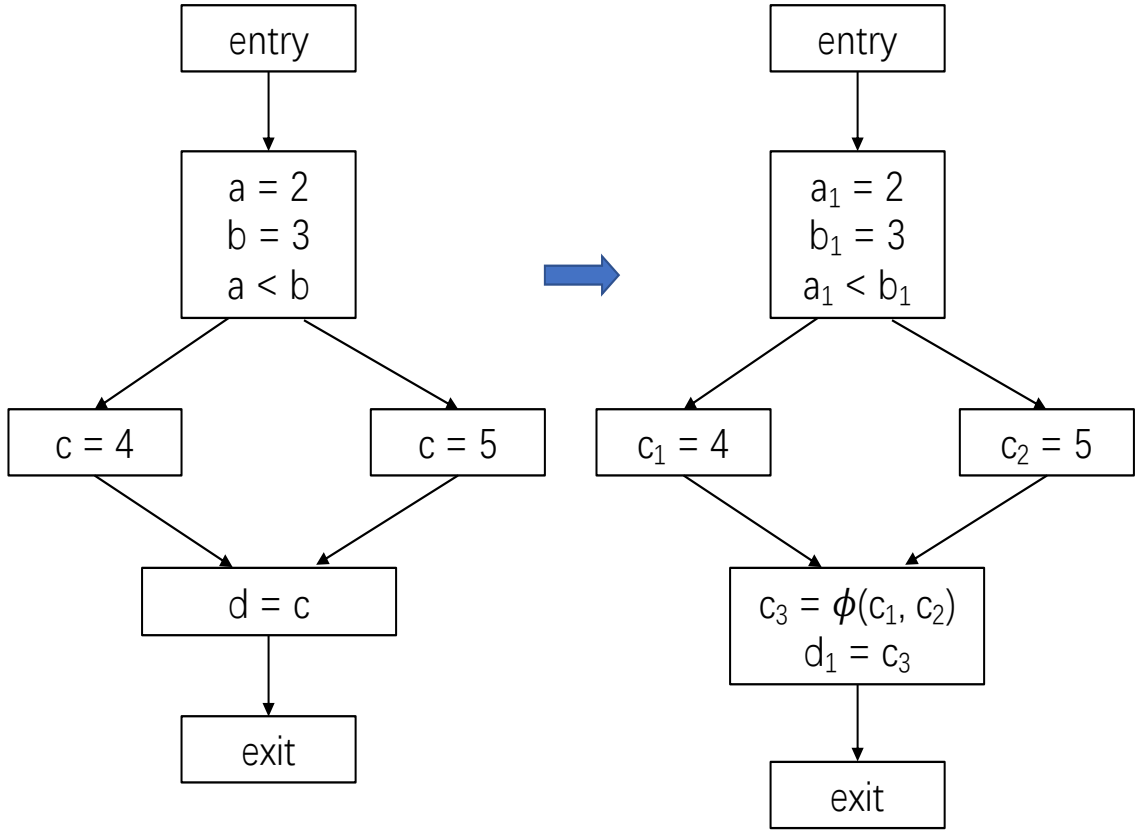
c. 部分冗余
   - B4中的表达式b+c只在B1 → B2 → B4的路径上冗余



SSA带来的改变

龙书中数据流分析介绍的非常详细，但现代编译器通常会将中间代码转换成SSA形式，基于SSA进行各种分析、变换；虎书在这方面有一些介绍

- SSA的优点
  - 将数据流信息直接编码在IR中，减少保存数据流信息的存储空间
    - 如果一个变量有N个使用和M个定值，则def-use链需要的空间和N*M成正比
    - 而通常情况下，SSA形式的程序大小和原始程序成线性关系
  - 简化变量的使用关系
    - 源程序中同一变量不相关的使用在SSA中变成了不同的变量，从而删除了它们之间不必要的关系
  - 使得数据流分析和优化算法变得更加简单



三地址代码　　静态单赋值形式

基于SSA的优化

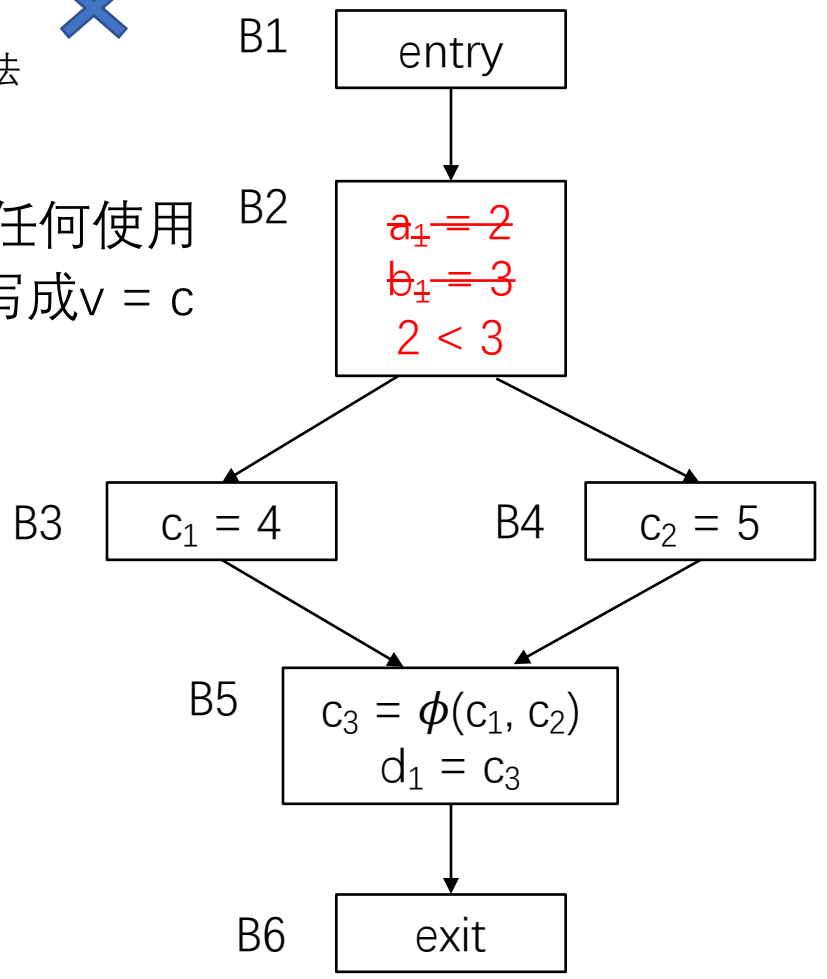
```
1)  OUT[ENTRY] = v_ENTRY;
2)  for (each basic block B other than ENTRY) OUT[B] = ⊤;
3)  while (changes to any OUT occur)
4)      for (each basic block B other than ENTRY) {
5)          IN[B] = ⋀_{P a predecessor of B} OUT[P];
6)          OUT[B] = f_B(IN[B]);
        }
```

前向数据流问题的迭代算法

- 稀疏简单常量传播
  - 只要有形如v = c的语句，其中c为常数，就可以用c替换v的任何使用
  - 对形如v = $\phi(c_1, c_2, ..., c_n)$的ϕ函数，若c_i都等于c，则可以改写成v = c

```
W ← SSA程序中所有语句的列表
while W非空
    从W中删除某条语句S
    if S 是形如v = φ(c, c, ..., c)的语句，其中c是常数
        用v = c替换S
    if S 是形如v = c的语句，其中c是常数
        从程序中删除S
        for 使用了v的每条语句T
            用c替换T中的v
            W ← W U { T }
```

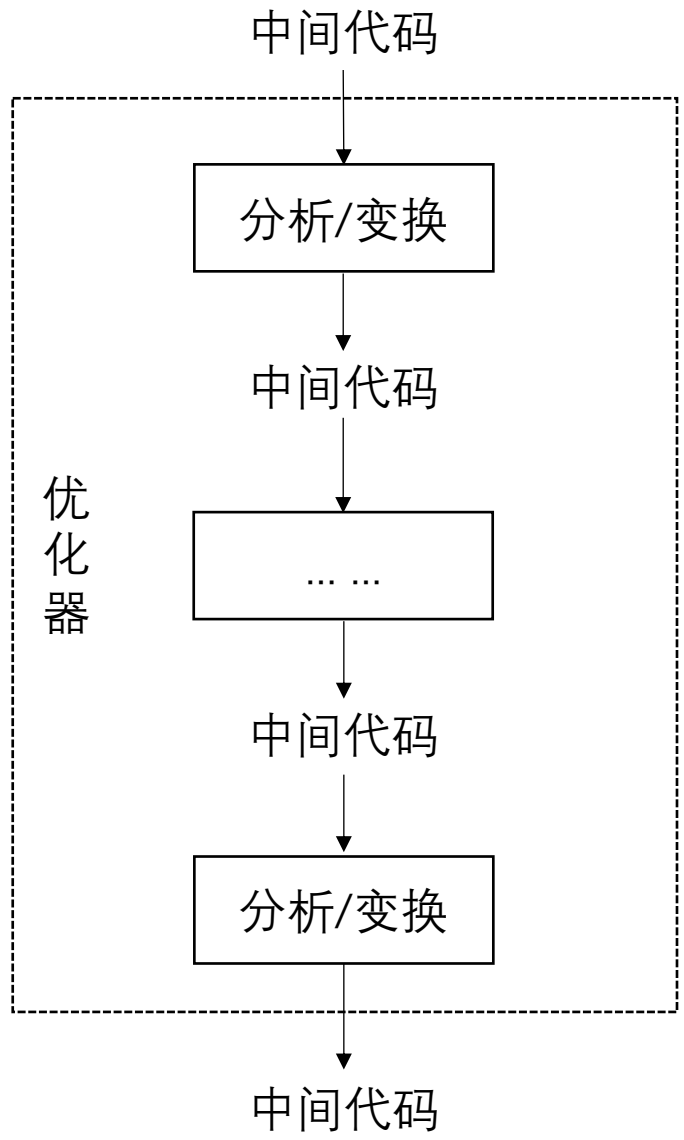
使用工作表算法来传播常数



基于SSA的常量传播的中间过程

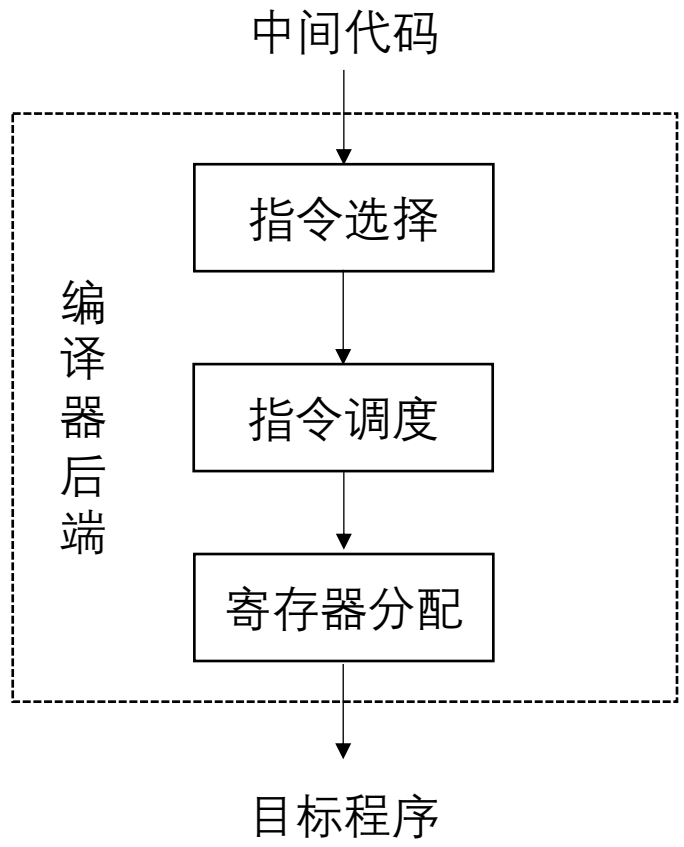
优化器的组成

- 对优化器进行模块化划分
  - 优化器由各种各样的分析和变换组成
  - 每个分析或变换都是对中间代码的一次处理过程（Pass）
  - 不同的优化级别（O1、O2、O3）对应不同的处理过程组合
  - 处理过程之间的顺序是根据经验来设置的
- 每种优化的背后都可能有一套专门的理论



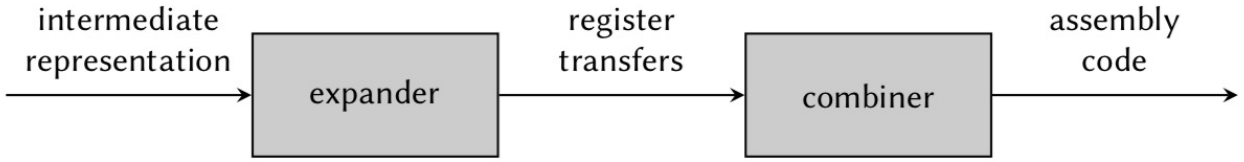
后端：如何生成目标程序？

- 目标机器的指令集架构（ISA）
  - 主要有指令和寄存器组成
- 代码生成就是将IR映射到ISA
  - 将IR映射到ISA中的指令：指令选择
  - 保持指令之间的数据正确流动，充分利用处理器的指令级并行：指令调度
  - 充分利用处理器的寄存器资源：寄存器分配
- 指令调度和寄存器分配也可视为优化
- 实践中，寄存器分配前后都可以有指令调度

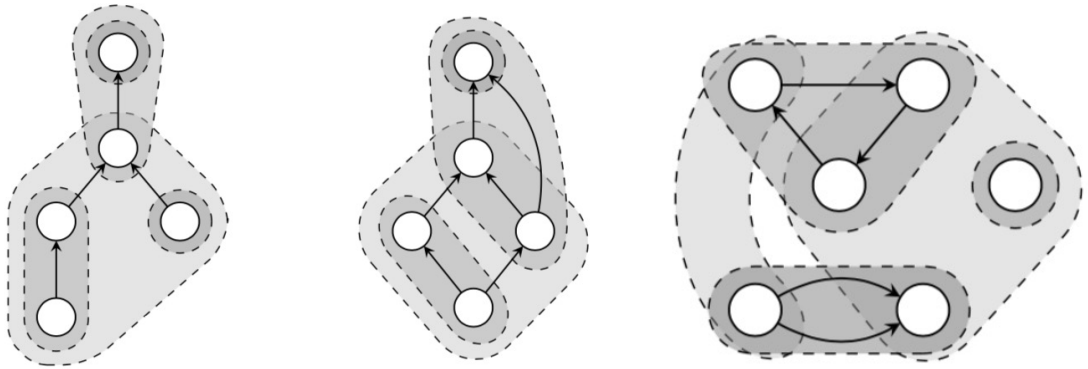


指令选择

- 1对1、1对多：展开+窥孔优化
  - 指令选择由两部分组成
    - Expander：将中间代码依次展开成RTL
    - Combiner：通过窥孔优化的方式将多个RTL合并成更大的RTL
- 多对多：模式匹配
  - 树覆盖（Tree Covering）
    - 只有1个根节点
    - 自底向上、自顶向下
  - DAG覆盖（DAG Covering）
    - 能有效处理公共子表达式和多输出指令



Davidson-Fraser approach

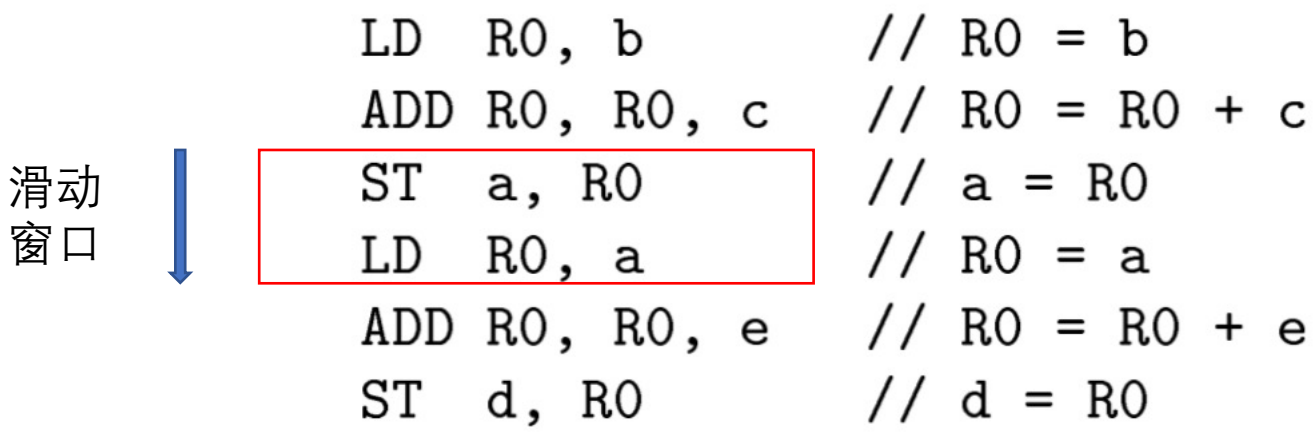


树覆盖　　DAG覆盖　　基于图的方法

Gabriel S. Hjort Blindell, "Survey on Instruction Selection : An Extensive and Modern Literature Review", October 4, 2013

通过窥孔优化改进代码质量

- 窥孔优化（peephole optimization）
  - 一个简单却有效的、用于局部改进目标代码的技术
  - 在优化时仅检查滑动窗口（即窥孔）中的目标指令，用更快或更短的指令来替换窗口中的指令序列
  - 也可用于中间代码，来提高中间表示的质量
- 常见窥孔优化
  - 冗余指令消除
  - 控制流优化
  - 代数化简
  - 机器特有指令的使用



GCC中的指令选择

- GCC中的机器描述
  - 机器描述用define_insn等RTL表达式来定义指令模式（instruction pattern）

gcc中RISC-V加法指令的机器描述

GCC中的指令选择

- GCC采用了类似的技术
  - 将三地址中间代码（GIMPLE）转换成RTL
  - 然后进行优化合并，生成汇编

```
int add(int x, int y)
{
  return x + y;
}
```

riscv32-unknown-elf-gcc -O -S -fdump-rtl-all add.c

有冗余的代码

```
;;
;; Full RTL generated for this function:
;;
(note 1 0 5 NOTE_INSN_DELETED)
(note 5 1 2 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
(insn 2 5 3 2 (set (reg/v:SI 73 [ x ])
        (reg:SI 10 a0 [ x ])) "add.c":2:1 -1
     (nil))
(insn 3 2 4 2 (set (reg/v:SI 74 [ y ])
        (reg:SI 11 a1 [ y ])) "add.c":2:1 -1
     (nil))
(note 4 3 7 2 NOTE_INSN_FUNCTION_BEG)
(insn 7 4 8 2 (set (reg:SI 75)
        (plus:SI (reg/v:SI 73 [ x ])
            (reg/v:SI 74 [ y ]))) "add.c":3:12 -1
     (nil))
(insn 8 7 12 2 (set (reg:SI 72 [ <retval> ])
        (reg:SI 75)) "add.c":3:12 -1
     (nil))
(insn 12 8 13 2 (set (reg/i:SI 10 a0)
        (reg:SI 72 [ <retval> ])) "add.c":4:1 -1
     (nil))
(insn 13 12 0 2 (use (reg/i:SI 10 a0)) "add.c":4:1 -1
     (nil))
```

add.c.245r.expand

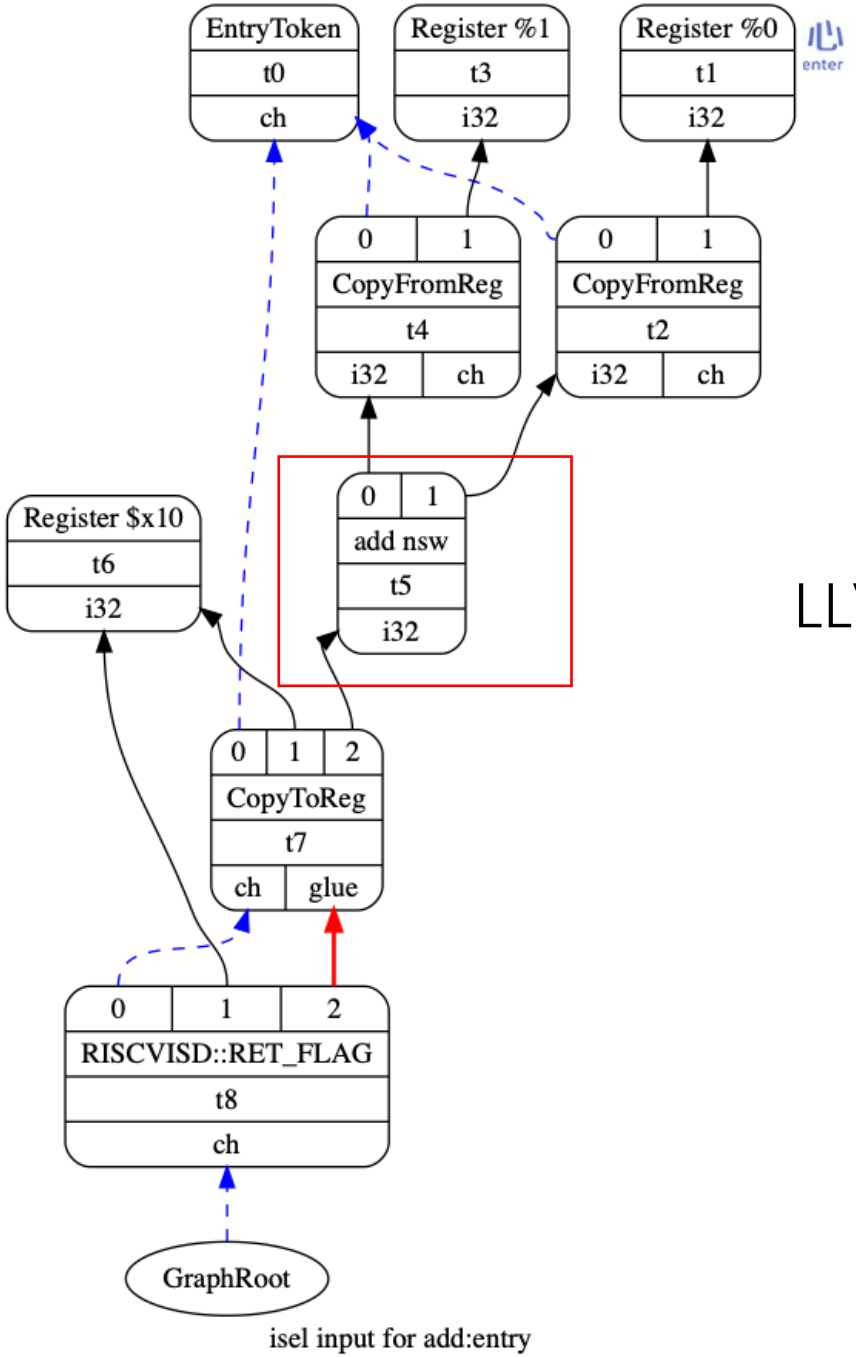
```
;;
;; Full RTL generated for this function:
;;
(note 1 0 5 NOTE_INSN_DELETED)
(note 5 1 2 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
(insn 2 5 3 2 (set (reg/v:SI 73 [ x ])
        (reg:SI 10 a0 [ x ])) "add.c":2:1 -1
     (nil))
(insn 3 2 4 2 (set (reg/v:SI 74 [ y ])
        (reg:SI 11 a1 [ y ])) "add.c":2:1 -1
     (nil))
(note 4 3 7 2 NOTE_INSN_FUNCTION_BEG)
(insn 7 4 8 2 (set (reg:SI 75)
        (plus:SI (reg/v:SI 73 [ x ])
            (reg/v:SI 74 [ y ]))) "add.c":3:12 -1
     (nil))
(insn 8 7 12 2 (set (reg:SI 72 [ <retval> ])
        (reg:SI 75)) "add.c":3:12 -1
     (nil))
(insn 12 8 13 2 (set (reg/i:SI 10 a0)
        (reg:SI 72 [ <retval> ])) "add.c":4:1 -1
     (nil))
(insn 13 12 0 2 (use (reg/i:SI 10 a0)) "add.c":4:1 -1
     (nil))
```

add.c.245r.expand

在展开之后通过合并等优化可以消除掉冗余的代码

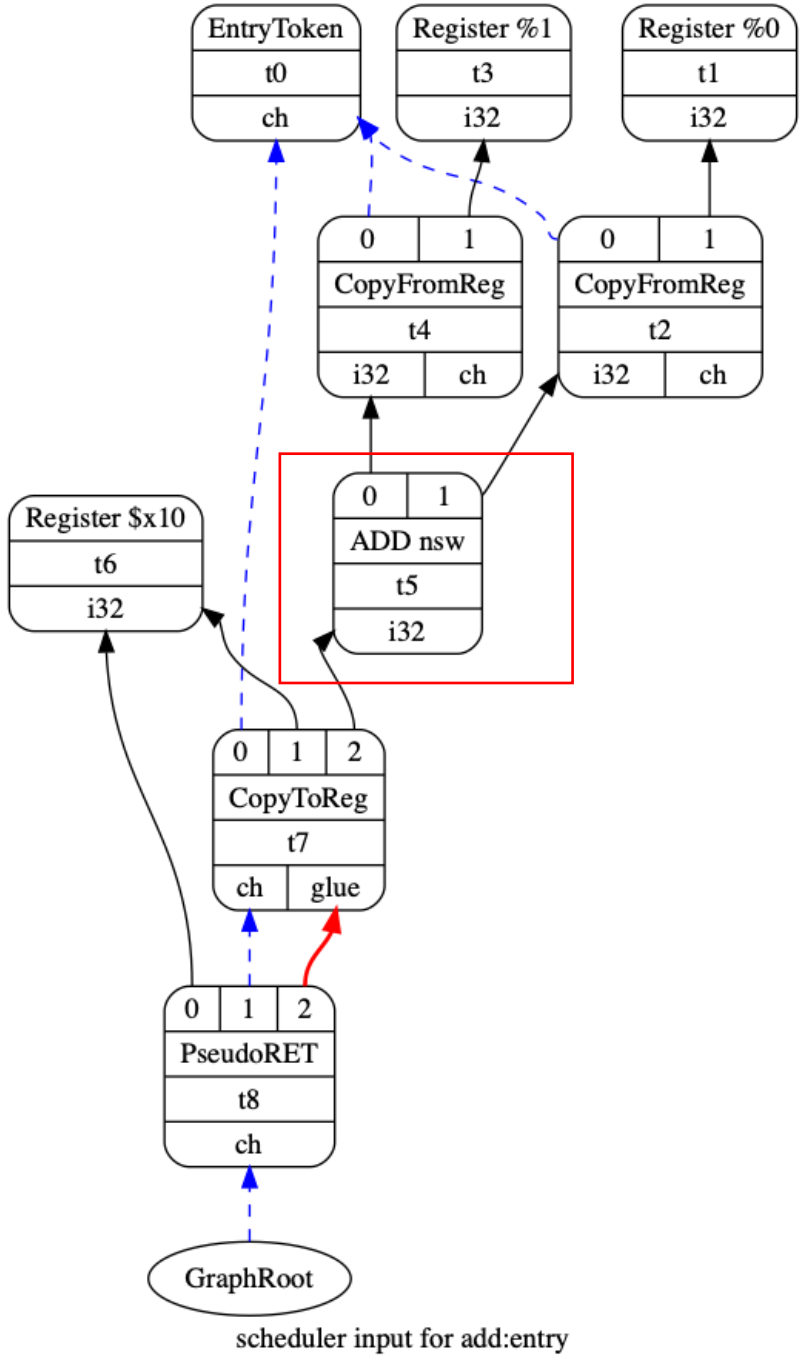
```
(note 5 0 15 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
(insn 15 5 2 2 (set (reg:SI 76)
        (reg:SI 10 a0 [ x ])) "add.c":2:1 -1
     (expr_list:REG_DEAD (reg:SI 10 a0 [ x ])
        (nil)))
(note 2 15 16 2 NOTE_INSN_DELETED)
(insn 16 2 3 2 (set (reg:SI 77)
        (reg:SI 11 a1 [ y ])) "add.c":2:1 -1
     (expr_list:REG_DEAD (reg:SI 11 a1 [ y ])
        (nil)))
(note 3 16 4 2 NOTE_INSN_DELETED)
(note 4 3 7 2 NOTE_INSN_FUNCTION_BEG)
(note 7 4 12 2 NOTE_INSN_DELETED)
(insn 12 7 13 2 (set (reg/i:SI 10 a0)
        (plus:SI (reg:SI 76)
            (reg:SI 77))) "add.c":4:1 3 {addsi3}
     (expr_list:REG_DEAD (reg:SI 76)
        (expr_list:REG_DEAD (reg:SI 77)
            (nil))))
(insn 13 12 0 2 (use (reg/i:SI 10 a0)) "add.c":4:1 -1
     (nil))
```

add.c.277r.combine



isel input for add:entry

LLVM的指令选择



scheduler input for add:entry

```
llc --filetype=asm -view-isel-dags add.ll
```

```
llc --filetype=asm -view-sched-dags add.ll
```

指令调度

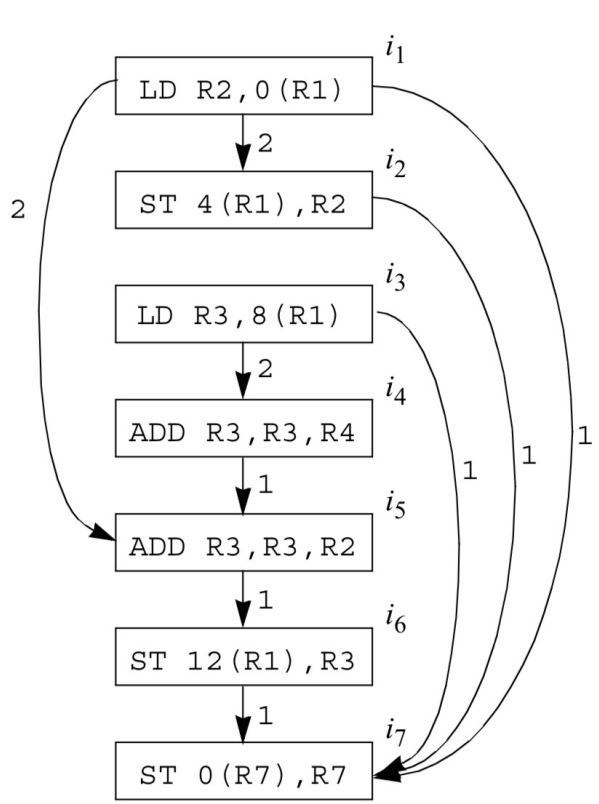
- 硬件的流水线（pipelining）
  - 将指令的处理过程划分为多个独立阶段
  - 即使每个时钟周期发送一条指令，仍然能够获得指令级并行性
- 延迟（latency）
  - 每条指令从开始执行，到结果可用，所需要的时钟周期
- 指令调度的作用
  - 通过重排指令执行顺序来掩盖指令延迟，从而减少代码执行时间
  - 充分利用硬件的多个功能单元，让处理器忙碌起来，减少空闲时间

	i	i + 1	i + 2	i + 3	i + 4
1.	IF				
2.	ID	IF			
3.	EX	ID	IF		
4.	MEM	EX	ID	IF	
5.	WB	MEM	EX	ID	IF
6.		WB	MEM	EX	ID
7.			WB	MEM	EX
8.				WB	MEM
9.					WB

简单的5级流水：取指、译码、执行、访存、写回

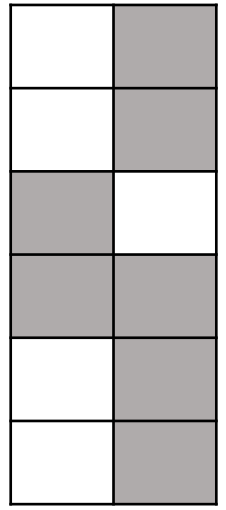
列表调度

- 输入：一个机器-资源向量R =[r_1, r_2, ...]；一个数据依赖图G = (N, E)
- 输出：一个调度方案S，将N中的每个运算映射到时间槽中
- 列表调度过程
  1. 按优先级拓扑排序访问每个结点
  2. 根据每个结点和之前已调度的结点之间的数据依赖约束，计算出能够执行该结点的最早时间槽
  3. 根据资源预约表来检查该结点所需要的资源是否得到满足
  4. 该结点被安排在最早能够获得足够资源的时间槽上
  5. 更新资源预约表



	i_3: LD R3, 8(R1)
	i_1: LD R2, 0(R1)
i_4: ADD R3, R3, R4	
i_5: ADD R3, R3, R2	i_2: ST 4(R1), R2
	i_6: ST 12(R1), R3
	i_7: ST 0(R7), R7

调度方案



循环的软件流水线化

- 假设循环至少有4个迭代，则可以使用左图的代码，来简洁地编码

行号						
1)		LD				
2)		LD				
3)		MUL	LD			
4)			LD			
5)			MUL	LD		
6)		ADD		LD		
7)	L:			MUL	LD	
8)		ST	ADD		LD	BL(L)
9)					MUL	
10)			ST	ADD		
11)						
12)				ST	ADD	
13)						
14)					ST	

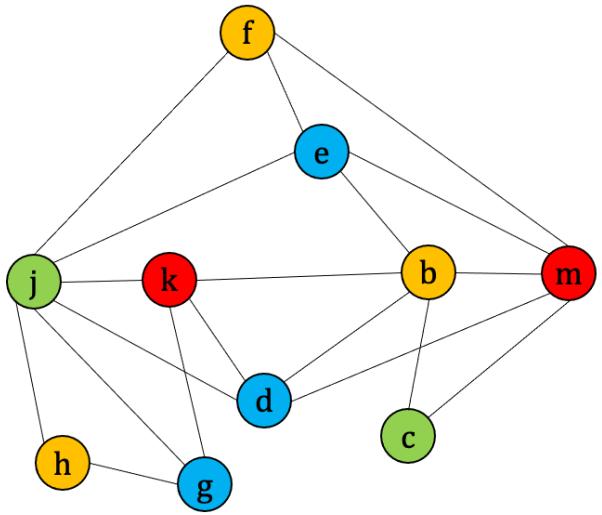
经过软件流水线化的代码

时钟	i = 0	i = 1	i = 2	i = 3	i = 4
1	LD				
2	LD				
3	MUL	LD			
4		LD			
5		MUL	LD		
6	ADD		LD		
7			MUL	LD	
8	ST	ADD		LD	
9				MUL	LD
10		ST	ADD		LD
11					MUL
12			ST	ADD	
13					
14				ST	ADD

展开5次，并且每个迭代采用相同的调度方案

寄存器分配

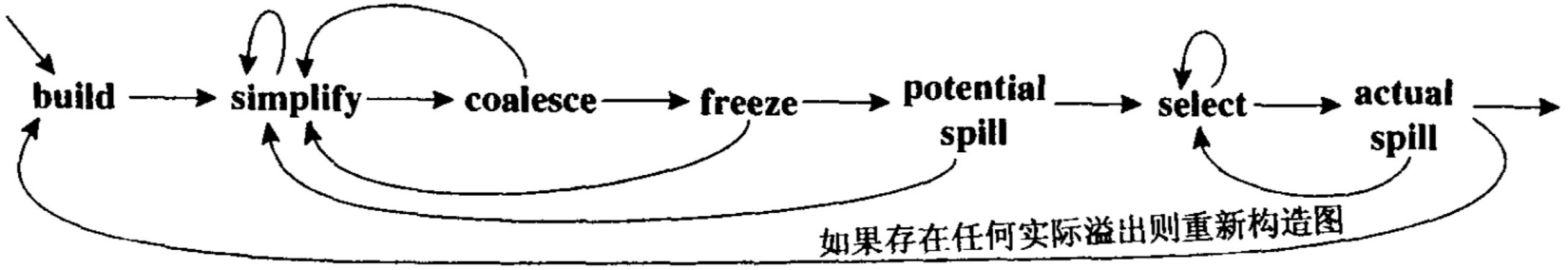
- 将寄存器分配问题转换成对干涉图着色问题
  - 一种颜色对应一个可分配的寄存器
  - 相邻结点使用不同的颜色
- 干涉图/冲突图（Interference Graph）
  - 结点表示变量
  - 边表示不能放在同一个寄存器中的变量对
    - 例如，同一个程序点，同时活跃的变量不能放在同一个寄存器中
    - 其他约束
- 使用k种颜色的着色方案称为k-着色（k-coloring）
  - 表示有k个寄存器用于分配



干涉图

Gregory J. Chaitin, Mark A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. Computer Languages, 6:47–57, 1981

带合并的图着色



- 各个处理阶段
  - 构造
    - 构造干涉图，并区分传送相关的（move-related）结点
  - 简化
    - 从图中删除低度数的（度<K）且与传送无关的结点
  - 合并
    - 对简化图实施保守的合并
  - 冻结
    - 如果简化和合并都无法进行，则冻结一个度数较低的传送相关的结点，将其看作传送无关的，从而使得有更多的结点可以简化
  - 溢出
    - 如果没有低度数的结点，则选择潜在可能溢出的高度数结点并将其压栈
  - 选择
    - 弹出整个栈并指派颜色

Lal George and Andrew W. Appel. 1996. Iterated register coalescing. ACM Trans. Program. Lang. Syst. 18, 3 (May 1996), 300–324. https://doi.org/10.1145/229542.229546

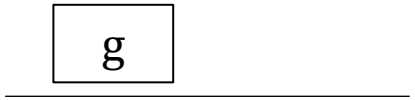
简化

可用于分配的寄存器，k=4

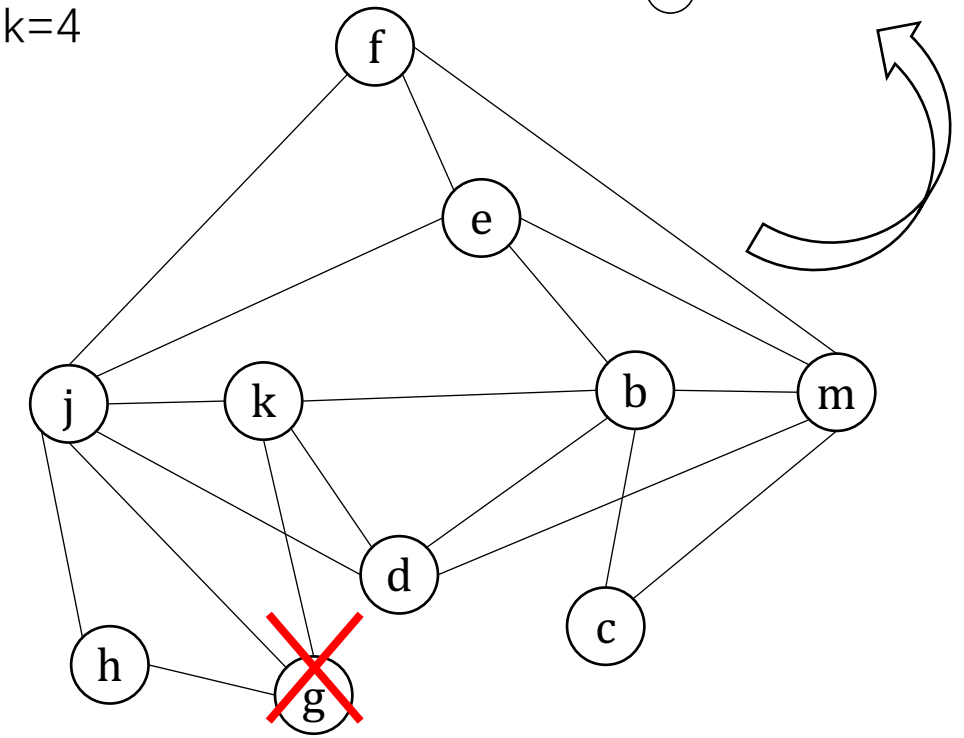
用一个简单的启发式对干涉图进行简化：

从图G中删除一个度数小于k的结点n，以及相临的边，得到一个子图G'。如果G'可以被k着色，则G也可以。

将删除的结点压入栈中。



栈



干涉图

线性扫描

参与分配的寄存器：
分别对应不同的颜色

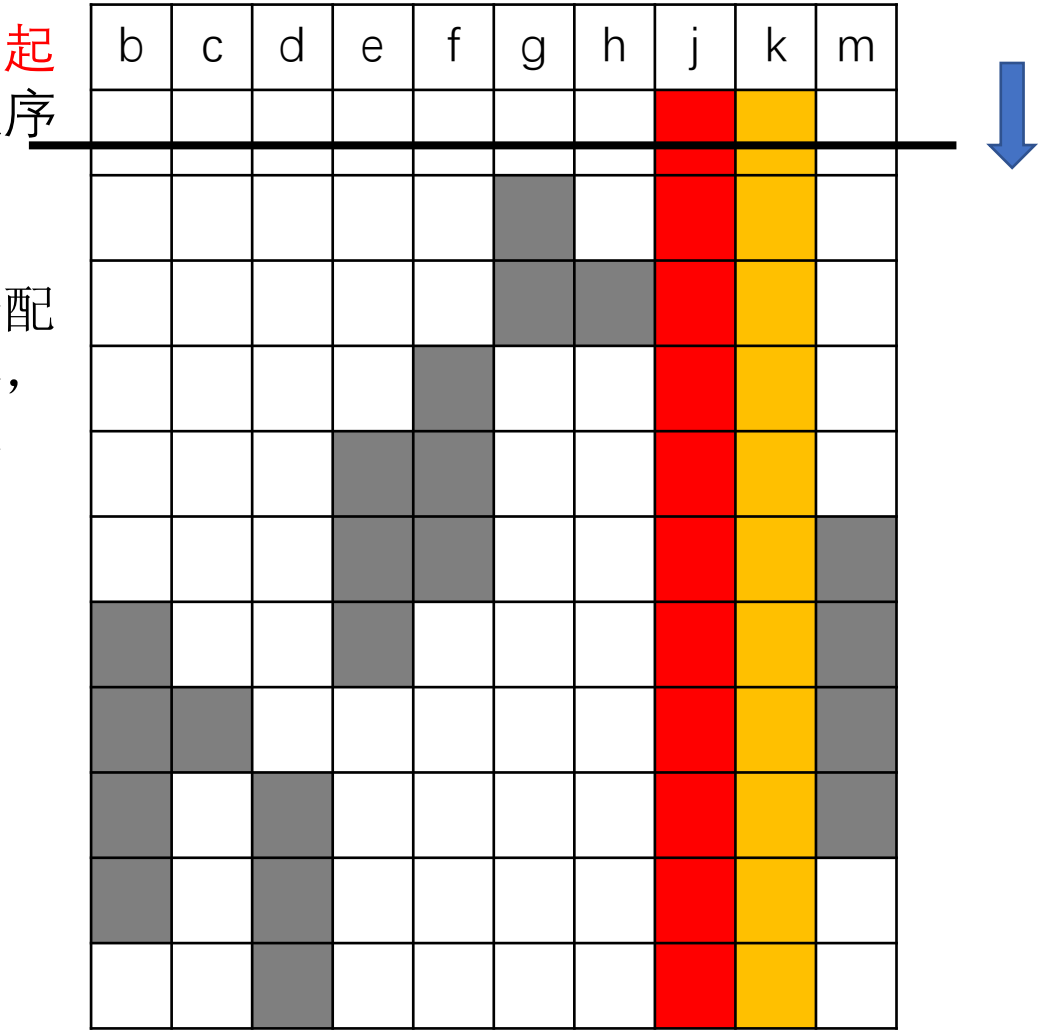
寄存器的使用状态：
灰色表示已经被分配

活动列表：j、k

存放与当前程序点重叠，已经分配在寄存器中的活跃间隔；按照活跃间隔的结束点进行递增排序

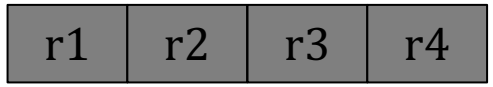
按照活跃间隔起始点的递增顺序扫描活跃间隔

为活跃间隔分配空闲的寄存器，更新活动列表



线性扫描

参与分配的寄存器：
分别对应不同的颜色



寄存器的使用状态：
灰色表示已经被分配

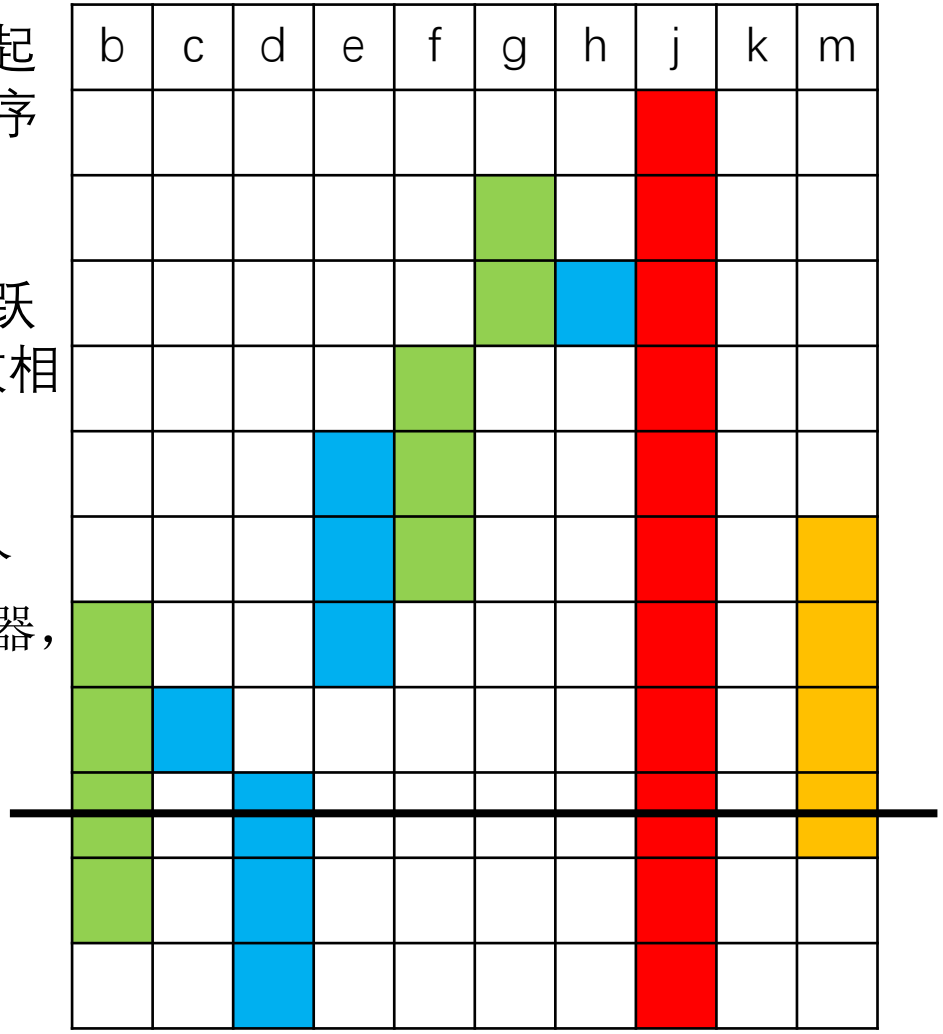
活动列表：m、b、d、j

存放与当前程序点重叠，已经分配在寄存器中的活跃间隔；按照活跃间隔的结束点进行递增排序

按照活跃间隔起始点的递增顺序扫描活跃间隔

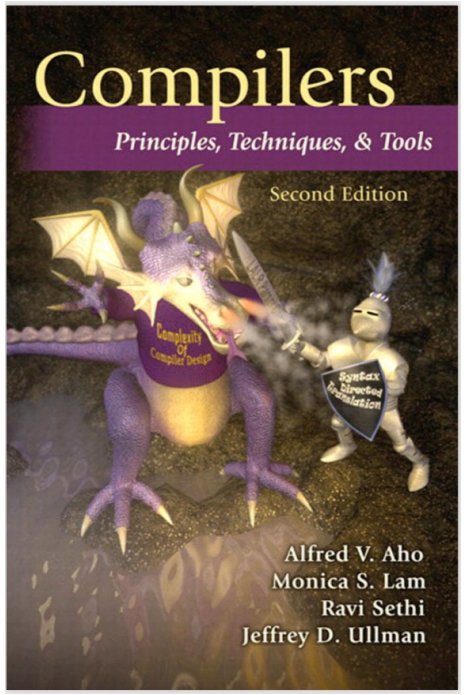
移除过期的活跃间隔c，并释放相应的寄存器

为活跃间隔d分配空闲的寄存器，更新活动列表

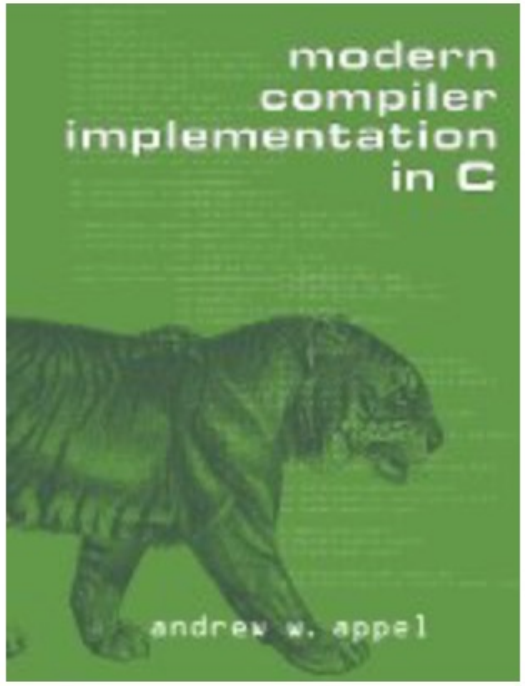


参考资料

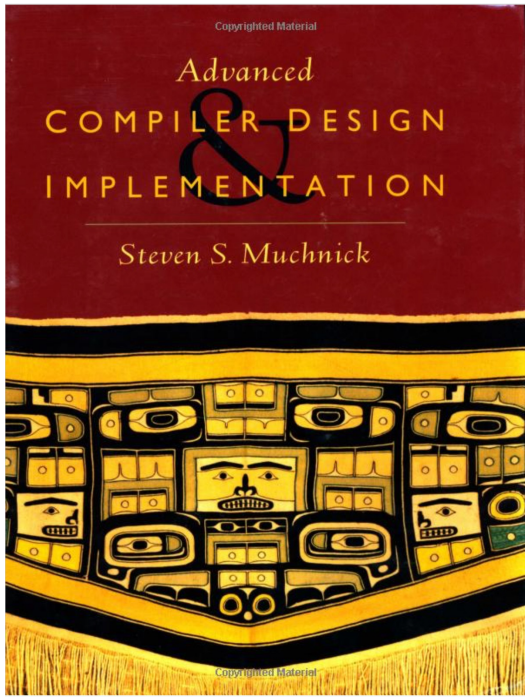
- 本幻灯片内容参考了龙书、虎书、鲸书
- 部分内容来自杭高院编译课程课件（依然是源于上面这些书籍）



龙书



虎书



鲸书

感谢大家观看！

如果有问题，欢迎随时讨论。