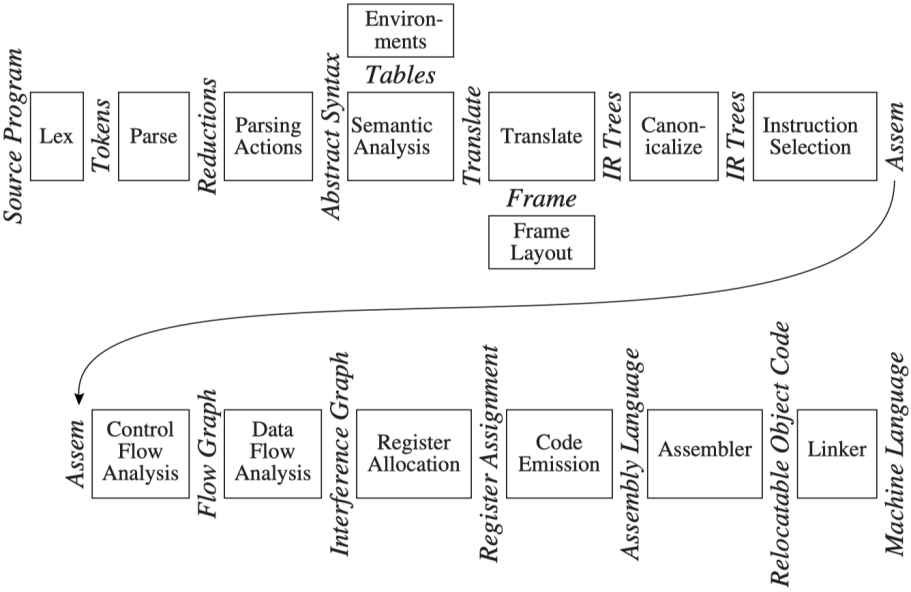
CH1



Parse阶段进行语法分析

Parse action阶段进行 abstract syntax tree 的构建

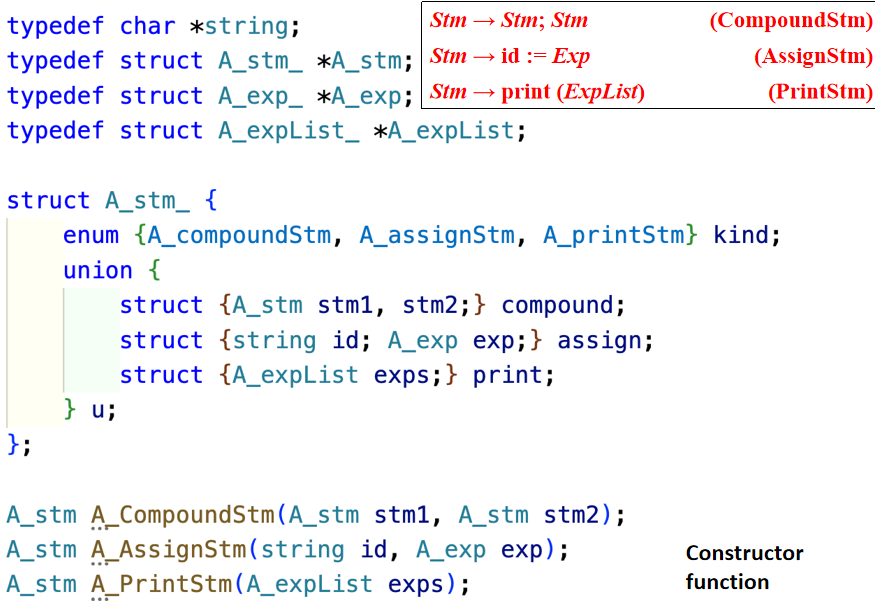
Translate阶段进行IR树的构建

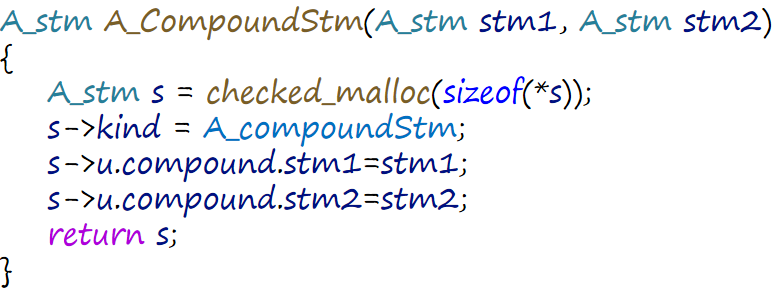
Canonicalize阶段进行规范化，消除语句副作用并减少不必要的分支

Control flow和data flow不同，control flow关注程序的控制结构流，data flow控制记录具体信息变量是否仍处liveness状态

Code emission阶段，将暂时使用的名称更换为machine register名称

类型定义





构造函数（在prefix之后的）首字母大写

在“虎书”中，The string type means a heap-allocated string that will **not be modified** after its initial creation；且暂不考虑调用free()函数

CH2

编译器的front-end执行analysis（分析），back-end执行synthesis（合成）

词法分析（lexical analysis）

注释、宏、预处理语句、空格、制表符、空行都不属于词法分析的token

Token必须要以letter开始，\_被看作一个letter，区分大小写

实现lexical analyzer可以借助正则语言与DFA构造（通过表导向的实现）

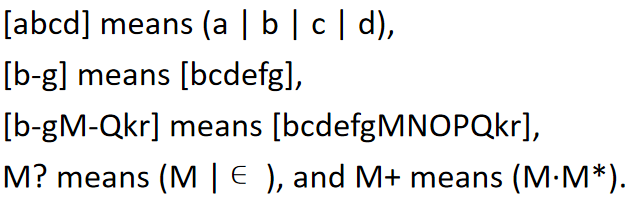
正则语言regular expression (RE)

A language is a set of strings

A string is a finite sequence of symbols

A symbol is taken from a finite alphabet

正则语言的优先关系：Kleene closure高于“连接”操作高于“或”操作；空也可以省略不写，如（a|）；



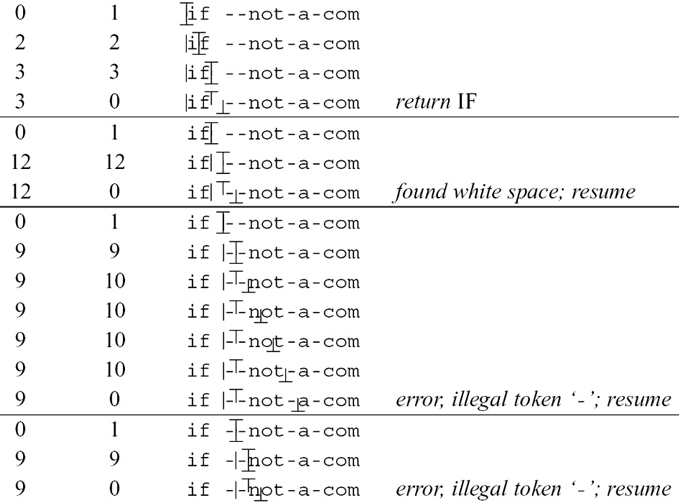
+表示至少有一个，？表示一个或为空，. 表示 any single character except newline；若加上“”则表示符号本身，如“.”表示 . 本身

默认注释要以“--”开始，并以“换行符”截止

正则表达式间可能存在匹配冲突，可以使用Longest match（token最长匹配）和Rule priority（rule优先级匹配，默认priority从上至下递减）两条规则

可以使用二维数组来维护状态与转移关系，需要添加一dead state（state 0）；并在最后添加final state到具体rule接收的对映

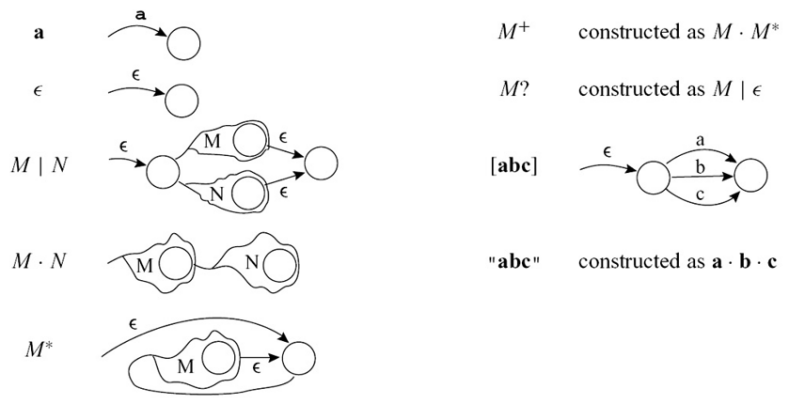
对于FA而言，在token匹配的运行过程中，需要记录string input起始位置，last final state记录位置，以及当前token遍历所处位置



正则语言转NFA规则

在lexical的RE与NFA的转换过程中，final state需要额外标注终止时对映的token类型

NFA到DFA的转变，合并“空”边集，合并同一symbol指向的不同state，直到没有新的state集合产生



The algorithm **does not visit unreachable states** of the DFA；在lexical过程中，需要根据rule priority标注final state对应的token

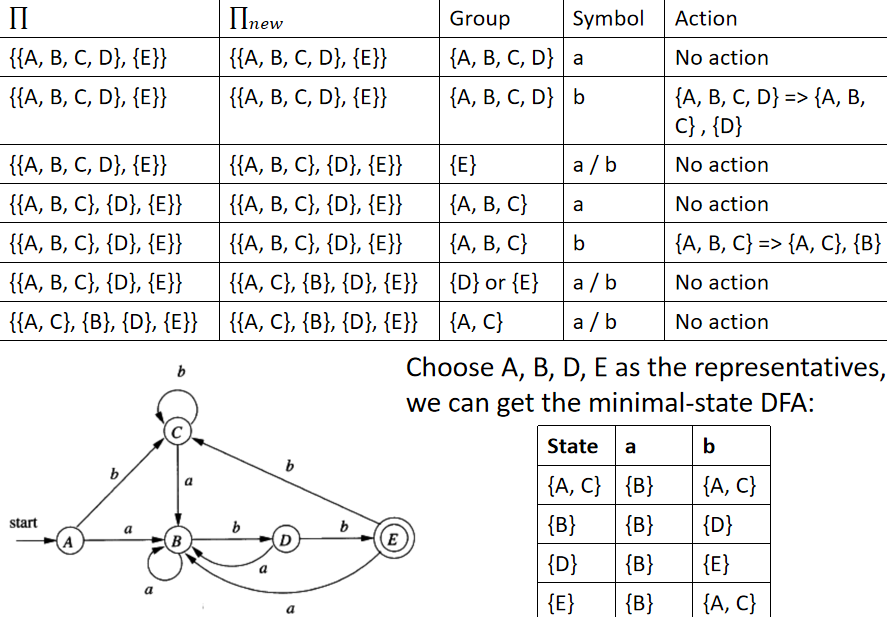
DFA的化简：寻找等价（两个state都是非final state或者final state，且out edge的symbol及对应的target state都一致）的state

We say that string x **distinguish**es state s from state t if exactly one of the states reached from s and t by following the path with label x is an accepting state

最小化DFA的算法形式

Split distinguishable states into different group（最开始分为non-final state和final state）

遍历当前state支持的symbol转移edge，若存在edge指向不同的group（当前等价组），则继续split当前group直到group不再更新



在lexical analysis流程中使用DFA最小化算法，需要区分不同的final state（根据不同的token类型）

Flex的使用

%{ definitions（C语言，预处理语句 [ #include... ] 以及C语言处理语句）%}

digit [0-9]（为一些symbol作缩写声明，如digit 等价 [0-9]）

状态声明

%%

{ rules（正则表达式摆放区域，rule后 {} 内可以加入辅助运行代码，称为action，rule match后才执行） }

%%

auxiliary routines（C语言函数，可以放main函数）

yylex() 会自动读取下一个token；yyleng 记录当前token的长度；yytext记录当前记录的token；yyin指定输入源；yyout指定输出源；yylineno记录当前token对映行数（使用前需要键入 %option yylineno，表示启用这个保留字）；yylval对映Yacc中union定义的结构体类型，并与返回的标签值对映

跨行注释匹配（"/\*"[^\*]\*"\*"+([^\*/][^\*]\*"\*"+)\*"/"）；[^\*] 表示任意非\*字符，同理 [^\*/] 表示任意非 \*、/ 字符

CH3

语法分析，需要使用CFG语言

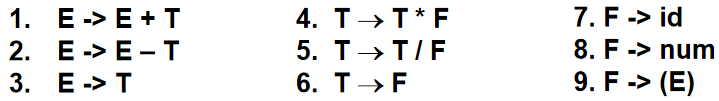
CFG对映多种derivation：left-most derivation，right-most derivation，随机derivation等，可能产生相同的parse tree；

对于parse tree而言，按顺序遍历叶子结点便是原始输入；parse tree还展现了符号的结合性

A derivation defines a parse tree，But one parse tree may have many derivations

A grammar is ambiguous if it can derive a string with two different parse trees.

加减乘除（含括号）的非二义性文法：

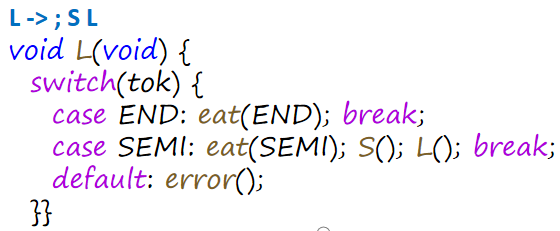


首先“左结合”保证了计算的左结合性，越左越下；符号优先级，结合性更强的符号更后出现

Recursive descent parsing（递归下降解析）

top-down parsing（要求不同rule中第一个token互不相同）

parse LL(1) grammars：Left-to-right parse（从左至右扫描）; Leftmost-derivation（最左推导优先）; 1 symbol look ahead（提前看一个token）



LL(k) 语法解析

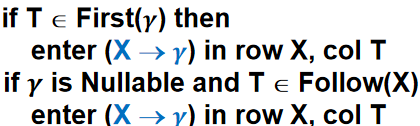
Nullable set，first set and follow set

First set的计算，注意可空non-terminal变量的忽略

Follow set的计算，直接出现在后侧的terminal symbol，注意右侧的follow set可以并上左侧的follow set（因为左侧一定可以转变为右侧项）

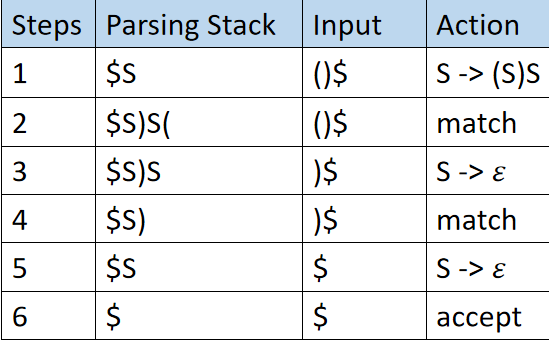
First set等可用于predictive parse过程中predictive parsing table的构建

构建规则：



如果在predictive parsing table的一格中有duplicate entries（冗余信息），则该文法不是LL(1)文法，不能用predictive parse解析

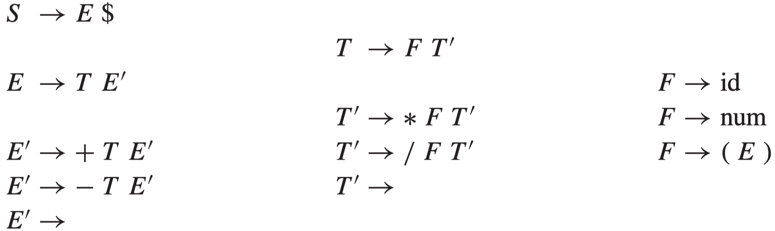
Predictive parse实际的运算过程



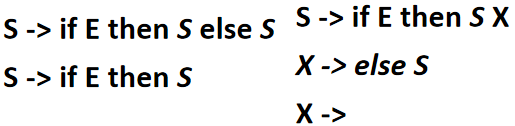
根据action在parsing stack中pop出“左侧non-terminal”，逆序填入“右侧产生的符号”

消除左结合（left association），左结合一定会使parse table产生冲突项

四则运算（含括号）修改为例



提取左公因式（left factoring）



Error recovery

Errors can be recovered by deleting, replacing, or inserting tokens

Inserting：pretend we have the token and return normally

Deleting：直接（连续）删除对映error token，直到下一个token出现在当前follow set中，更加安全

Bottom-up parse

LR-parsing（shift-reduce parsing）

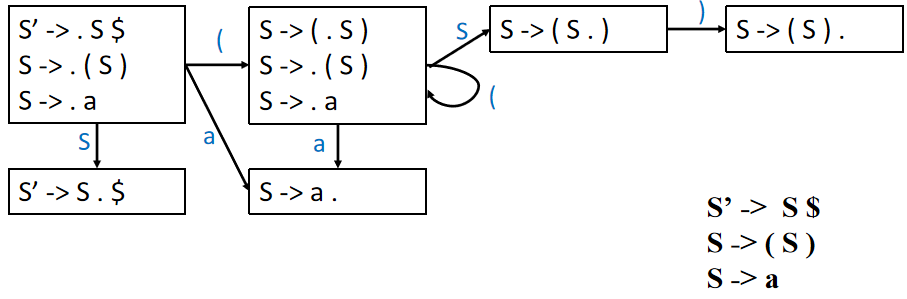
最右推导，从左向右遍历（读入token）

本质是reduce the string to the start symbol（最右reduce）；随着token的读入，选择reduce或者shift

需要keep a stack of terminal & non-terminal symbols

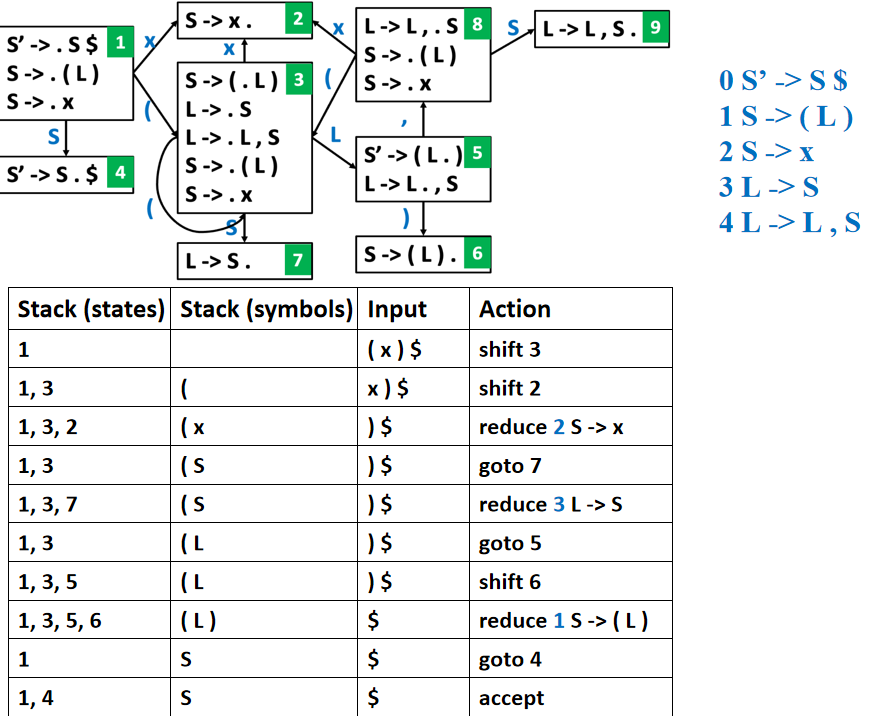
有下列四项动作：shift（读入token），reduce（归约rule，需要相应的操作stack），error，accept（reduce得到start symbol）

LR(0) 如下构造（注意递归non-terminal元素），从initial non-terminal开始，终止符不需要读入

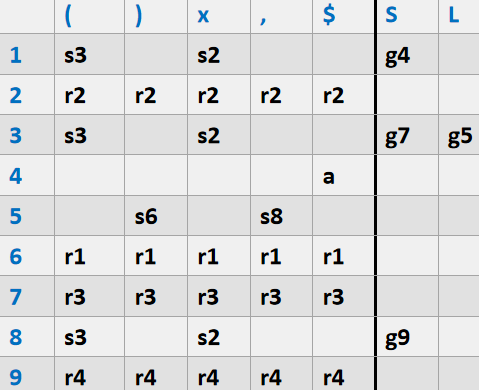


Stack需要记录state的变化情况以及symbol，方便回退；进行reduce操作时symbol stack和state stack需要同时进行pop，个数一一对应（symbol stack可能还需要push，rule左侧的non-terminal）

Reduce操作即pop symbol和state stack中相同个数元素，然后回退至state stack中top state处，并将reduce对映rule的non-terminal放入symbol stack中，并进行相应的state push（转移）操作



可以建立对映的table



Table中同一项可能又存在reduce操作，又存在shift操作（reduce-shift冲突），说明该文法不是LR(0) 的

SLR文法（Simple LR Parsing）：

在LR(0)文法中，我们将reduce操作写入了state对映的所有格中，在SLR文法中，我们仅将reduce操作写入可归约rule对映的follow set（左侧non-terminal的follow set）对映的symbol格中

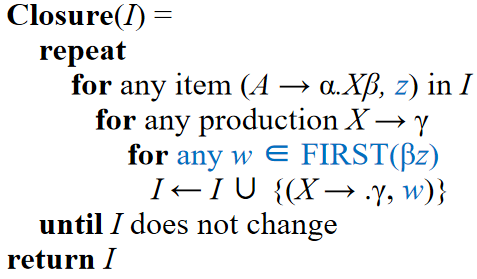
仍然可能存在reduce-shift冲突

LR(1)文法

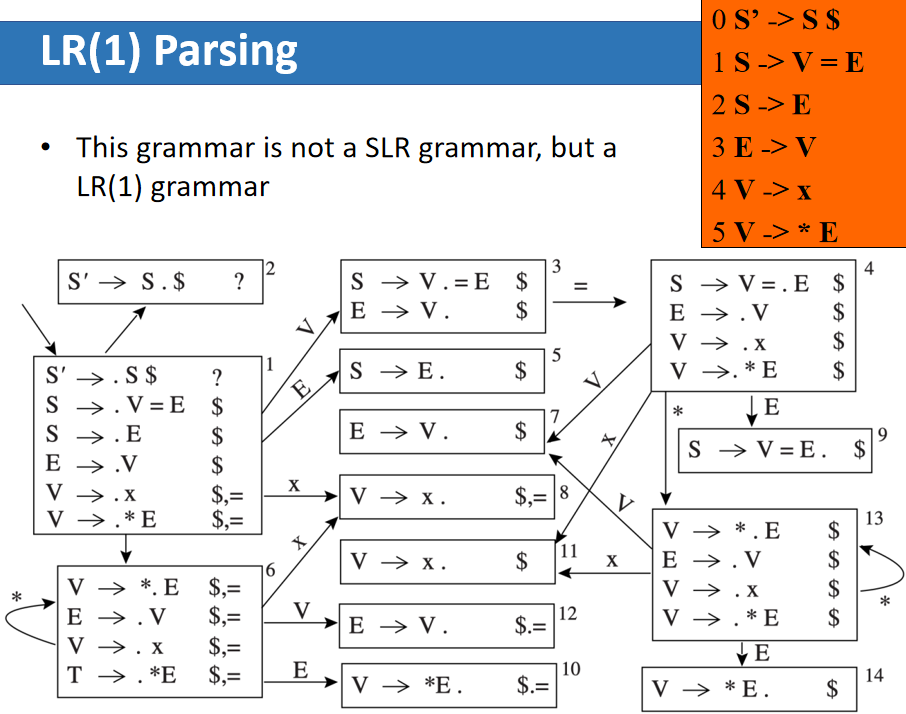
The look ahead token of LR(1) is more accurate than that of SLR

需要计算first集合，作为额外的信息元素

LR(1) parsing tables can be very large, with many states（因为提前看的symbol不同而认为是不同的state）



LR(1)对映DFA示例图，table构造要求同SLR

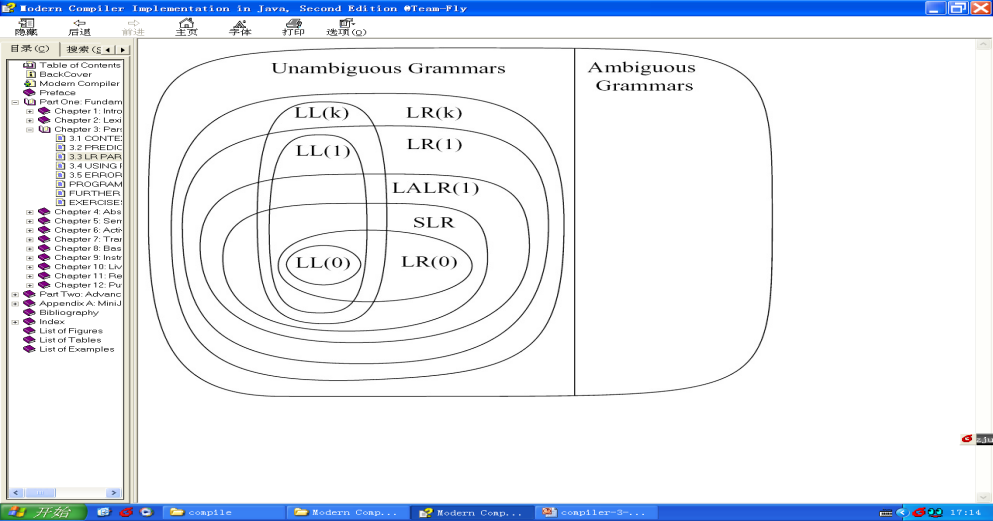


LALR parsing，the basis for parser for most modern programming languages；implemented in tools such as Yacc

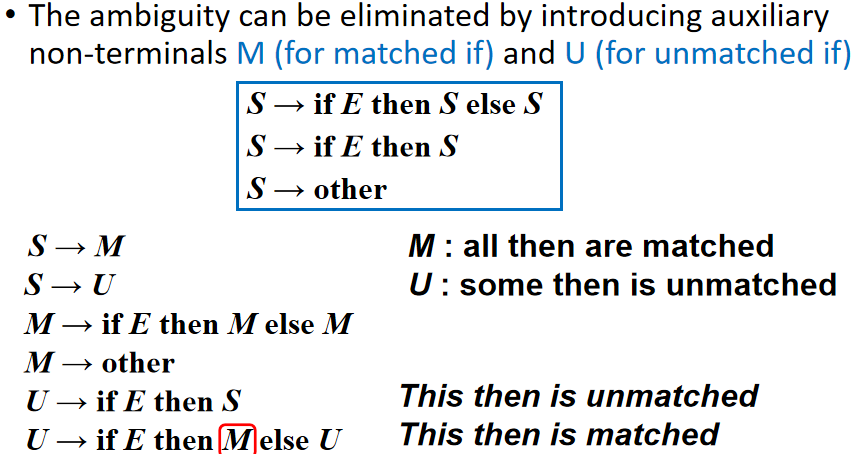
LALR(1) parsing: the parsing table is made by merging any two states whose items are identical **except for look ahead sets** in the LR(1) parsing table

LALR(1) 可能存在reduce-reduce冲突（因为state的合并而导致）

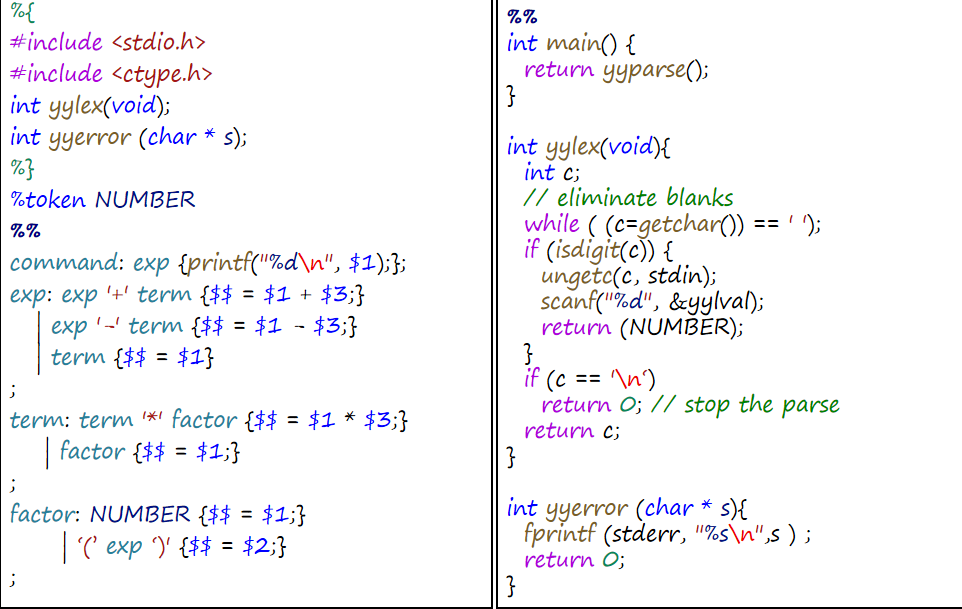
LALR(1) parsing table requires less memory to represent than the LR(1) table



依然存在if-else的选择匹配问题（shift-reduce conflict）



Yacc的使用



%{ ... %} 最开始包含预处理信息及函数声明

使用 %union定义token值可能的类型（若不定义union，则non-terminal的类型与YYSTYPE类型相同）

%token <tree> INT "int"，定义token，<>内说明token类型，INT为flex返回标签，“int”为等价替换；

%type <tree> INT\_ID 定义非终结符类型

%start Calc 定义起始字符（未定义则默认第一个）

%nonassoc EQ NEQ（非结合性，不能连等）

%left ADD SUB （符号优先级定义，越向下优先级越高；%left表示符号运算使用左结合）

%left MUL DIV MOD

%right NOT

%left A（可以搭配%prec A使用，%prec A可视作占位符，A不会作为token被返回）

（在priority方面，The priority of shifting a token is given by this token；The priority of a rule is given by the last token occurring on the right-hand side of that rule

If the token and the rule have equal priority: %left favors reducing；%right favors shifting；%nonassoc yields an error action）

%%

CFG法则，{}中内容为匹配后执行内容（每个|之后都可以有附加动作），$$表示左侧non-terminal，$x表示右侧第x个符号

%%

yyerror函数可以自行定义

yyparse为yacc相关调用函数（会自动调用yylex函数，返回token类型，并用yylval返回附加值），若语法分析成功则返回0，否则返回1

Yacc支持Embedded Actions，即在rule内部额外引入“辅助操作”（注意，额外的{} 辅助操作也被视为符号，即后续 $2 会因为额外 {} 的引入转变为 $3）

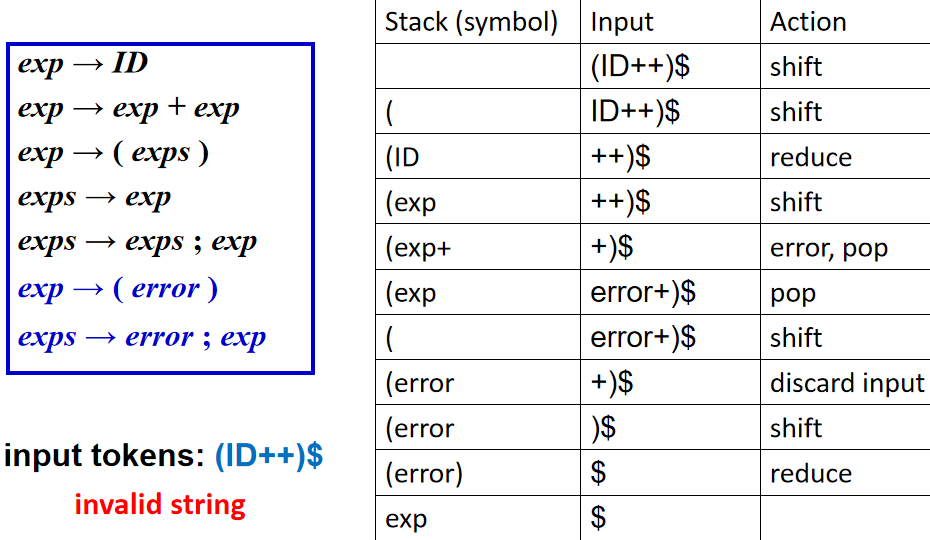
Yacc在reduce-shift冲突时采纳shift，在reduce-reduce冲突时采纳rule中更靠前项

Error recovery

Local error recovery

额外添加接收error信息的CFG文法语句，当error出现时，跳过error继续parse过程，用于阻隔error的；等符号，被称为“synchronizing tokens”（同步token）

Error处理样例过程（这样的处理可能会引入语义上的错误）



Global error repair

finds the **smallest** set of insertions and deletions that would turn the source string into a syntactically correct string（即使真实出错点不在报错点上）

Burke-Fisher Error Repair

tries every possible single-token insertion, deletion, or replacement at every point that occurs no earlier than K tokens before the point where the parser reported the error（向前K遍历尝试insertion、deletion and replacement）

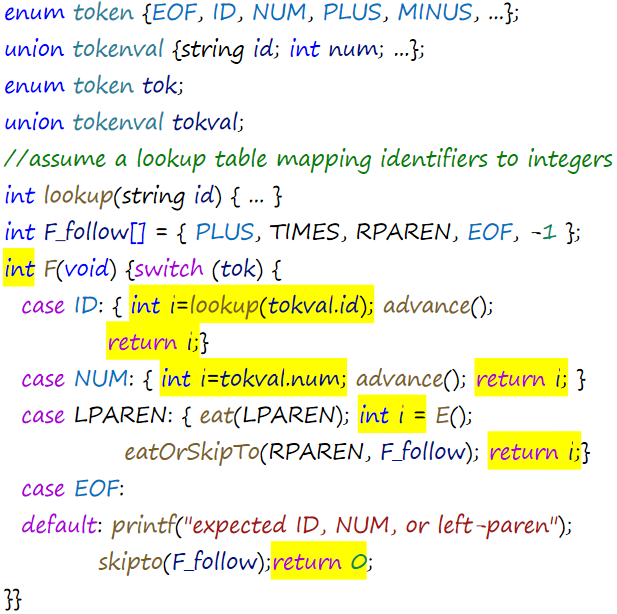
维护老栈 + 单词队列 + 当前栈（记录队列中symbol）

Shift获得新单词时，需要push近当前栈，并把当前栈最底部的元素push至老栈中，并照常尽心reduce动作

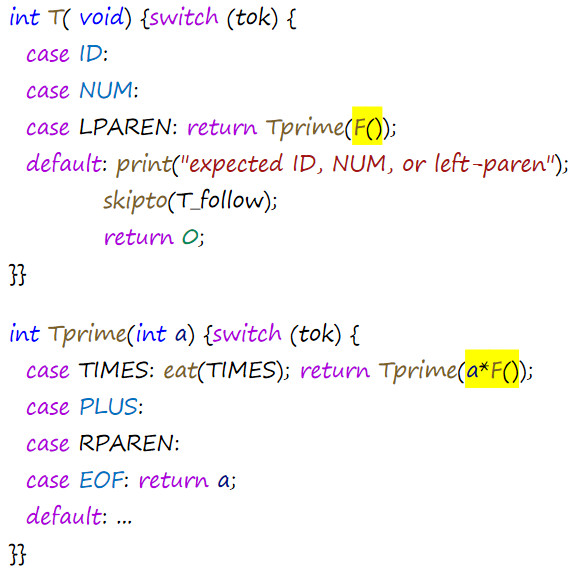
遇到error时，在symbol queue中的每一个位置尝试所有可能单词，并从老栈开始进行修复，如果修复使parse能连续进行3 - 4个单词，则认为修复成功

CH4 抽象语法

Recursive descent pars过程中引入semantic analysis



对于一些rule，可能无法直接获得所有的语义内容，需要依赖函数的参数传递



Abstract Parse Tree

Post-order的遍历即可获得semantic数值

a parse tree has exactly **one leaf for each token of the input** and **one internal node for each grammar rule** reduced during the parse；被称为concrete parse tree（具体语法树）

具体语法树会饱含大量冗余信息，并极度依赖grammar的实现

abstract syntax tree（抽象语法树）

可以解析具体语法，构建抽象语法树

CH5

semantic analysis

使变量的定义与使用相连；检查表达式的类型；将抽象数据结构转变为易于产生机器码的表达

Symbol table（environment）

将变量名与类型绑定

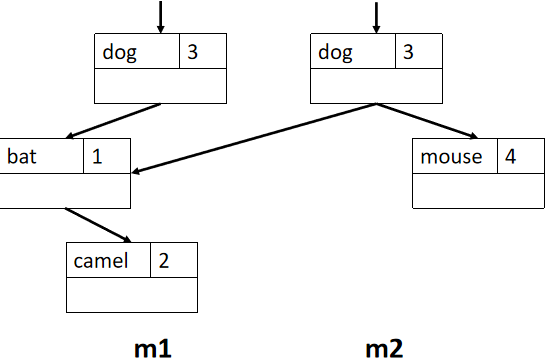
Environment相加时，右侧的table会覆盖左侧的table（X + Y for tables is not the same as Y + X）；在作用域结束后，相应的environment也需要被删除

Functional style（函数式）

在添加新环境时旧环境不发生改变

新作用域产生时，为新环境创造新表（如果复制整个hash table，内存消耗过大）

使用binary search tree结构



如图式更新，仅复制需要插入节点到root路径上的节点，保证旧环境没有发生改变，开销大约为log(n)

Imperative Style（命令式）

调节旧环境使其变成新环境，新环境存在时无法查看旧环境，新环境作用域到底后，需要返回为旧环境

使用hash table + bucket结构实现

仅用一张hash表记录，新变量作用域产生时加入hash-bucket结构，作用域消失时更新删除hash-bucket结构中对映内容

String间的比较开销较大

将string转变为symbol类型进行比较

Symbol便于产生hash-key

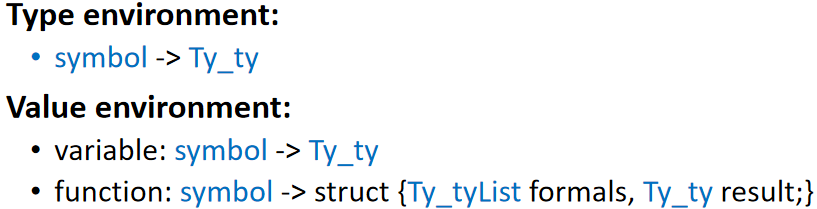
Symbol间比较便捷迅速（相等/大小比较）

不同类型的symbol table

type bindings for types（type与名称的对映）

value bindings for variables and functions（变量、函数与名称的对映）

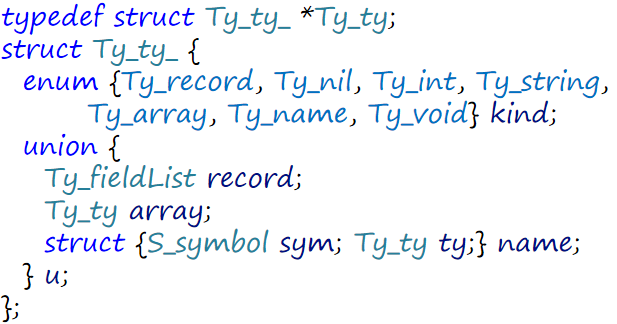
类型，变量，函数名的symbol与类型对映关系



类型a和变量a或者函数a可以同时存在

即使是相同的结构，在tiger中也被认为是不同的类型

Tiger的类型定义



Tiger使用imperative style的symbol table（也称为destructive-update environments）

通过添加标记（在stack中加入mark）来显示表示scope的begin and end；binder中可以记录所属的stack top

Type check

Expression的type check

An l-value is a location whose value may be read or assigned

Declaration的type check

类型Ty\_Nil的初始化表达式必须受到Ty-Record类型的约束

函数定义先构造Ty\_ty类型的result\_ty，Ty\_tylist类型的参数列表；记录函数后，再打开函数参数列表，建立新的环境

递归声明克服

对于type list = {first: int, rest: list}（递归定义），可以先使用S\_enter(tenv, list, Ty\_Name(list, NULL)) 函数进行占位

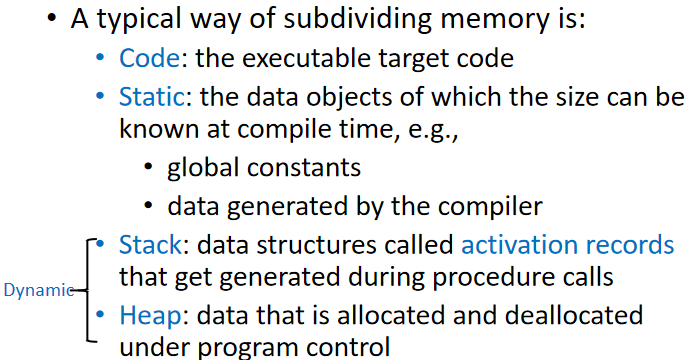
一组相互递归的类型声明中的每个循环都必须至少含有一个record或array类型声明

相互递归函数调用处理

第一遍遍历（代码）先记录所有函数的名称、参数以及返回值，body部分等第二遍遍历（代码）时再创建新的环境进行处理

CH6

Activation Records（活跃记录）

  
 activation record (sometimes called a frame)

如果一种语言既支持函数内函数定义，又支持返回值为函数，则不能用stack来存储函数活动

All locations beyond the stack pointer are considered to be garbage and all locations before the stack pointer are considered to be allocated

A function’s **activation record** or **stack frame**: The area on the stack devoted to the local variables, parameters, return address, and other temporaries for this function

caller-save register and callee-save register的概念

Some architectures have register windows, so that each function invocation can allocate a fresh set of registers without memory traffic.

函数调用时参数需要写入内存的几个条件

the variable will be passed by reference（引用）

嵌套的内层函数可以访问的外层函数的变量

过大的变量，无法放入register中

Array数组变量

逃逸变量（escape variable）if

it is passed by reference

its address is taken (using C’s & operator)

it is accessed from a nested function

Static link 静态链接

**Block Structure**: In languages allowing nested function declarations, the inner functions may use variables declared in outer functions.

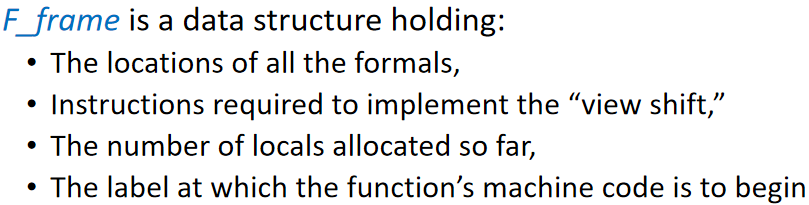
每次调用函数时，都会传递一个指针（current FP），用于指向最近活跃的函数帧栈 [ 一般观察函数定义处的调用关系 ]

使用静态链接后，就可以实现内层函数对外层函数的调用或对外层函数已定义变量的使用

全局array：display，在位置i中包含一个指针，指向静态嵌套深度为i的最近输入过程的帧

lambda lifting：需要被内层函数使用的变量直接作为参数传递给内层函数

调用函数时，可能会发生“shift of view”，因而可能需要额外的代码来克服这个问题（参数在callee看来在register、stack的位置同caller视角）



在当前阶段（tiger中）有

Temps: abstract names for local variables

Labels: abstract names for static memory addresses

使用frame.h抽象分离具体源语言和机器相关的栈帧布局（一些源语言不支持nested function declaration）；使用translate.h分离类型检查和语义转换（非必要存在），translate模块需要去管理static link

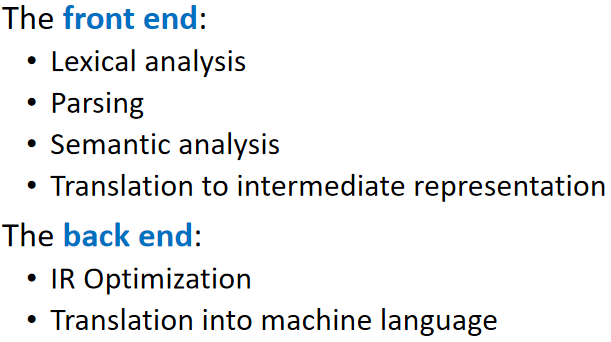
Tiger具体的实现函数暂未记录

CH7

Translate to Intermediate Code

中间code表示有利于将具体实现的源语言与具体使用的机器相隔离

Compiler的前后端



三地址编码（three-address code）

最多出现三个地址及一个operator（可以是四则运算也可以是比较符）

三地址编码好像不能让同一个register既当作源寄存器又当作目的寄存器

三地址码的实现有时使用四元组的形式，不足项使用empty or null占位（一般operator放在四元组最前面，后面跟三个操作数）

Expression（有返回值）

CONST(i)：表示立即数i

NAME(n)：符号常量n，汇编语言标签

TEMP(t)：表示寄存器n

BINOP(o, e1, e2)：四则运算、逻辑运算、逻辑左右移、算术右移操作

MEM(e)：地址为e的内容，左侧表示写入，右侧表示读出

CALL(f, l)：调用函数f（常用NAME标签），输入参数列表l

ESEQ(s, e)：先执行s的副作用，再计算e的值作为返回值

Statement（无返回值）

MOVE(TEMP t, e)：计算e并把结果放入寄存器t中

MOVE(MEM(e1), e2)：计算e2并把结果放入e1地址区域中

EXP(e)：计算e并忽略返回值

JUMP(e, labs) ：跳转至地址e或者一个name标签位置

CJUMP(o, e1, e2, t, f)：以条件o比较e1，e2的结果，并根据比较结果跳转

SEQ(s1, s2)：连续执行两个statement

LABEL(n)：将名称n的常数值定义为当前机器代码地址

一个条件判断语句，如果已经有了跳转的目的地以及跳转条件，就不需要赋予语句返回值

因为在CJUMP表达式处，我们可能还未知true or false的跳转地址，可以建立对映的true or false目的地列表，等待后续填充

对IR树的任何操作都应该由Translate完成

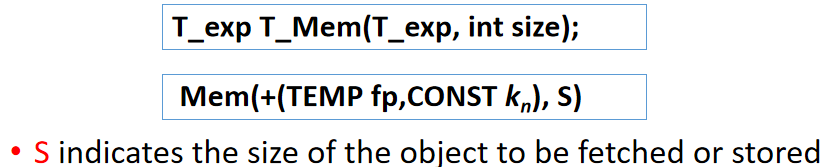
Operator可以化简：



Tiger中数组变量仅是地址，因而支持数组名之间的直接赋值（等价于地址赋值）

对于数组变量，未取地址时表示左值；取地址后表示右值；存储过程中的实现不需要MEM操作

注意这种用法：



Subscripting（下标） and Field Selection（字段选择）

注意，tree language没有二元操作符，如 -n需要表示为 0 - n 的三元样式

Unary floating-point negation cannot be implemented as subtraction from zero

If语句也有返回值，返回值为执行的then/else代码段的最后运行结果

调用函数时，需要将static link作为参数（一般作为第一个）进行传递

ch8

Basic block and trace

在生成IR之后对IR进行一定程度的化简

A tree is rewritten into a list of canonical trees without SEQ or ESEQ nodes

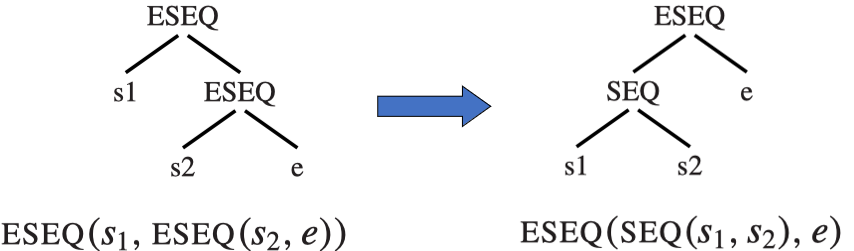
Linearize，需要remove ESEQ操作，并将CALL移至顶部

要求，tree language中没有SEQ和ESEQ操作，CALL的parent只能是EXP 或MOVE操作

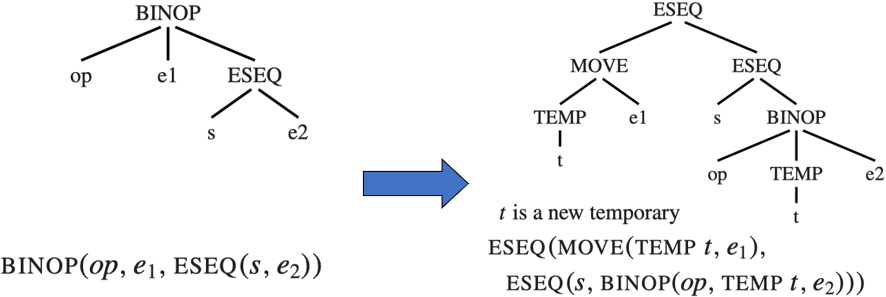
性质：每一棵canonical tree（标准化tree）只能有一个statement；一棵tree最多只能支持一个CALL操作，因为CALL的parent只能是EXP or MOVE

消除ESEQ操作（先置顶ESEQ操作）

注意不同statement间之心顺序不能发生改变，转换操作类似于下图



若statement提前可能会引起statement与expression的冲突，则可以先运算expression并将结果保存，再操作ESEQ的提前



Commute交换，因为存在可能的先后关系，我们保守的使用交换操作；常数、空语句一定支持交换

Reorder函数，将参数传递链表指针指向的所有操作分离，提取所有statement语句并合并，消除ESEQ的存在，并保持statement语句和expression语句的相对顺序不变

Reorder函数中SEQ操作也可化简，若seq的一部分涉及常数操作，则可直接化简

Move CALL操作至tree language顶部

不同的CALL连续调用可能会覆盖register中需要使用的有效值

直接将每一个CALL的返回值立即记录在temp t中即可

消除SEQ操作

反复使用以下规则，逐渐将seq操作后移，最后直接将一连串SEQ(s1, SEQ(s2, ..., SEQ(sn-1,sn)...))视作普通序列s1, s2, s3...即可



This list is grouped into a set of basic blocks, which contain no internal jumps or labels

基本块内部没有label标签和jump跳转操作，基本块以LABEL开头，以JUMP or CJUMP操作结尾（没有的话需要额外添加）

Control flow（控制流）instruction在程序中的运行顺序；忽略变量的具体value，从而假设jump true or false的目的地都有可能被跳转

Trace（迹）a sequence of statements that could be **consecutively executed** during the execution of the program and can include conditional branches.

一条trace可能无法覆盖整个程序的运行，我们希望使用的trace尽可能的少（这样可以减少不必要的在不同trace之间的跳转操作）

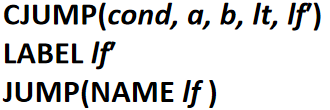
每一条trace的第一个block可以任意选取（一般选择未marked block）

The basic blocks are ordered into a set of traces in which every CJUMP is immediately followed by its false label.

traceSchedule：CJUMP 后紧紧跟随false的跳转目标

为方便后续进程，一些额外的操作：

每一个CJUMP操作后都跟随false的跳转目标，若只能跟随true的跳转目标，我们则反转跳转条件，并互换true / false的跳转目标，使CJUMP后跟随的是false的跳转目标；若CJUMP后没有跟随任何的true or false标签，则使用如下方法进行强行标注与跳转



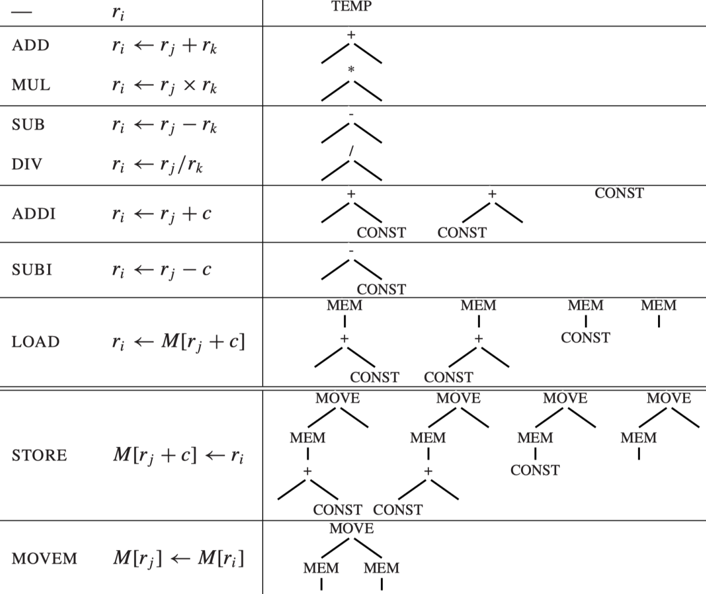
频繁一起执行的序列最好处于同一轨迹上

CH9

Tree language可以被看成很多个部分的组合，称为tree pattern

Instruction selection的目的：用最少的tree pattern覆盖整棵树（tree pattern之间不能互有重叠），覆盖过程称为tiling

Jouette architecture（上六条属于expression，下两条属于statement）



注意ADDI操作可以处理仅有const（立即数）的情况，ADDI操作和SUBI操作有一定的不同；乘法除法没有直接乘立即数的操作

Tiger语言中数组名对映的内存地址空间存放的是数组开始的地址，所以取 a[0] 需要两个MEM操作

Best tiling：转化后有最短instruction序列，program有最短的执行时间

**Optimum** tiling：the one whose tiles sum to the lowest possible value（代价最小）

**Optimal** tiling: the one where no two adjacent tiles can be combined into a single tile of lower cost（没有两个tile可以进行合并了）

Every optimum tiling is also optimal, but not vice versa（不允许两个相对较小的patter value之和小于它们合并得到较大的pattern的value）

为optimal tiling设计算法大大方便于为optimum tiling设计算法

Maximal Munch算法

从 root开始向下寻找，每次都使用最大的tree pattern进行覆盖（同等大小tree pattern随机选择覆盖）

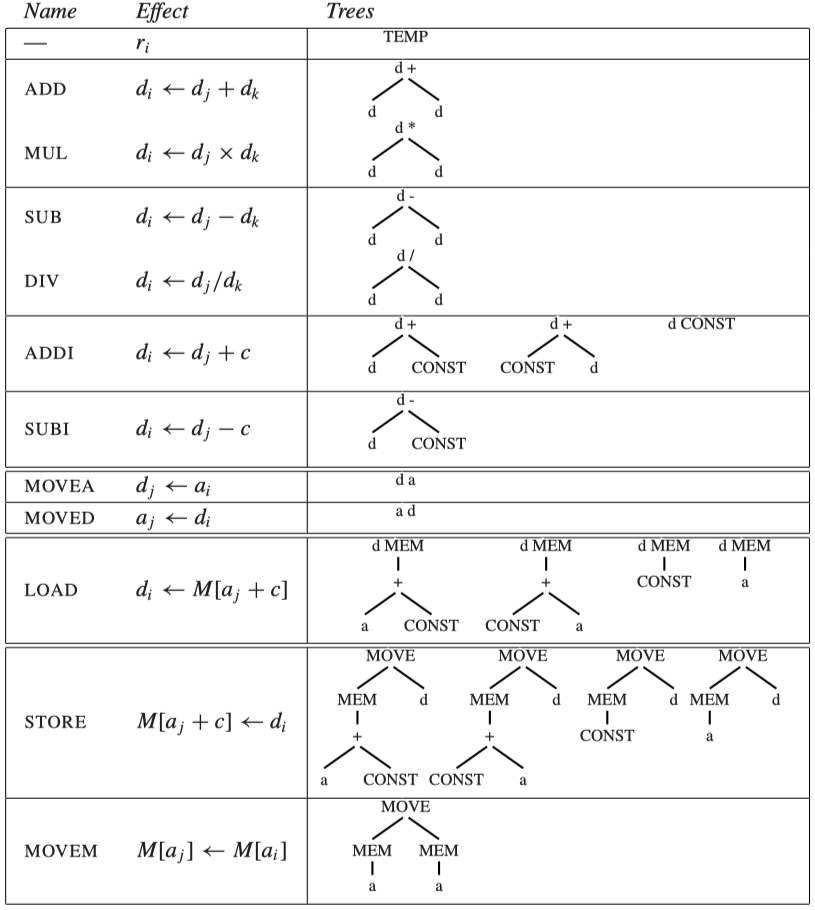
在生成具体的instruction时，maximal munch算法进行反向操作，从底部向上开始生成代码

Dynamic programming

Works bottom-up，可以找到optimum解

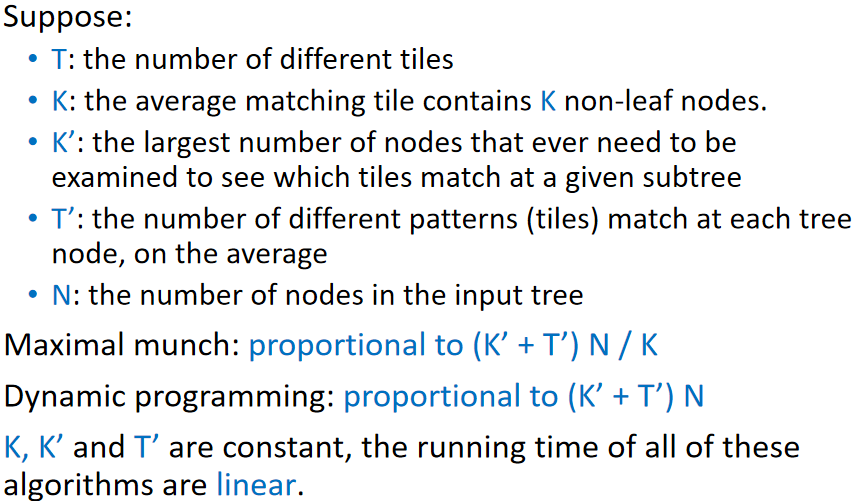
为每一个node添加一个cost，cost表示tile以node为root的sub-tree的最小cost，叶子节点的cost就是它本身

如果address register和value register需要区别，则tree pattern需要额外说明与跟踪



快速匹配，使用标号来快速抉择所用覆盖tile的类型

时间消耗



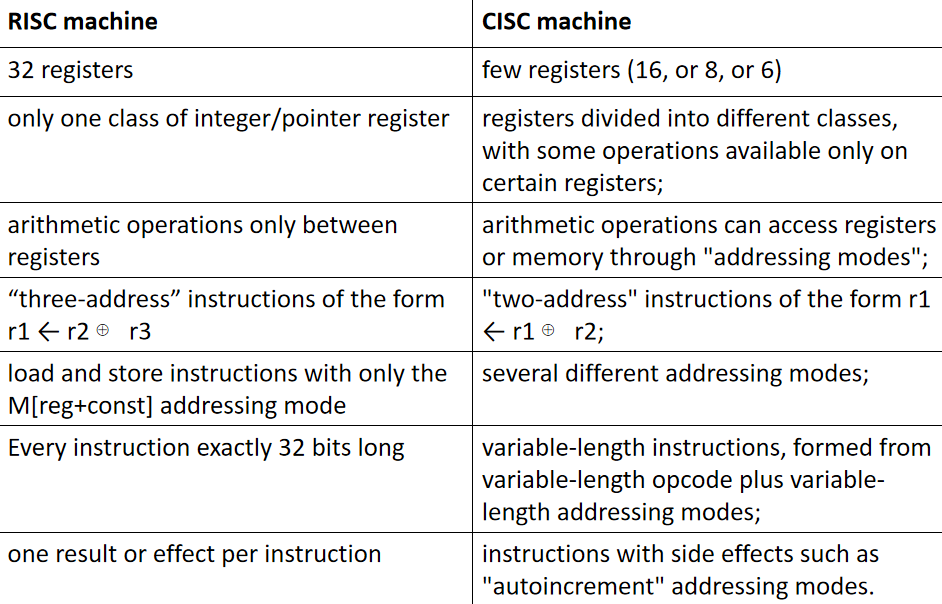
T表示不同的瓦片，K表示平均每个瓦片的非叶子节点数，K’表示子树为确定使用何种tile所需检查的最大节点个数（近似于最大瓦片大小），T’为每一个树节点都可以与多少种tile匹配；

Maximal munch的时间复杂度: proportional to (K’ + T’) N / K，每个tile占用K个节点的前提下，N个节点只需要遍历N/K个node，且快速匹配告诉我们每次匹配只需要查看最大的node点数和tile类型个数之和（使用switch结构快速匹配）

Dynamic programming：proportional to (K’ + T’) N，K’+T’同maximal munch的快速匹配，另外在DP过程中，每个节点都要计算cost（都要进行一遍操作）

CISC表示复杂指令集的意思；RISC精简指令集，在RISC中，一般optimum tiling和optimal tiling没有区别

CISC和RISC对比



在tiger编译器中，自动丢弃计算得到的高位数据

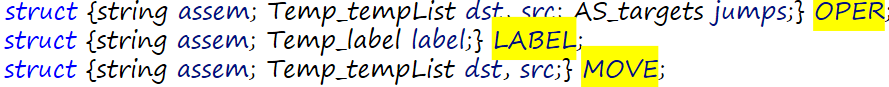
Two-address instruction中，the first source argument必须和target argument一致

CISC中地址中数据可以直接与register中数据相加（如add [ebp - 8], ecx）

We will do register allocation after instruction selection.

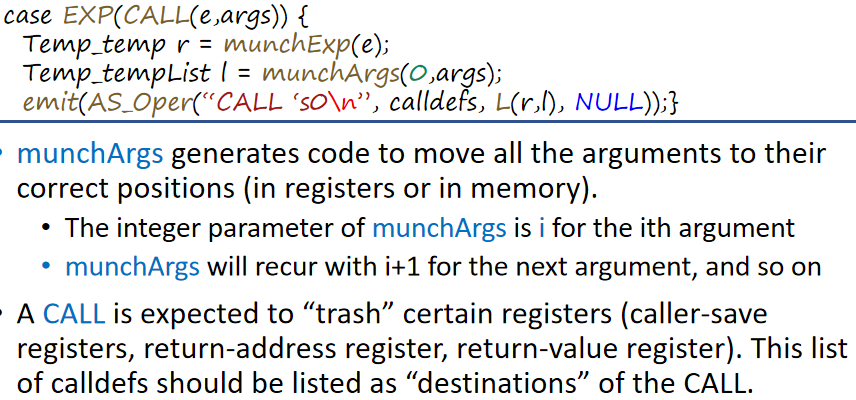
因而在IR指令转换为machine 指令这一步，我们仅分配abstract register

对于目标寄存器一定要是第一个源寄存器要求的指令集，我们忽略使用abstract register时忽略s0，注明d0；tiger中间过程三种类型结构如下：



Procedure call：过程调用，无返回值；Function call：函数调用，有返回值

过程调用的处理



注意其中munchArg函数和emit中calldefs的说明

CH10

Liveness analysis

A variable is **live** if it holds a value that may be needed in the future（在下次使用之前不会经过定义）；因为考虑变量的未来使用情况，一般采用Backward Analysis

先将program构造为control flow

构建Data-flow

Out-edges：flow去向

In-edges：flow来源

Pred[n]：all the predecessors of node n

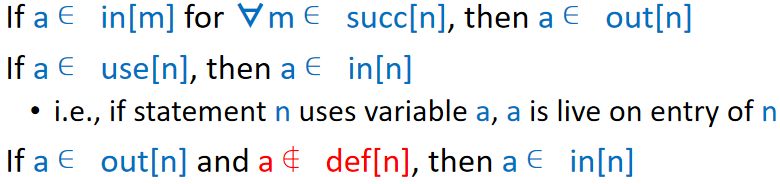
Succ[n]：the set of successors of node n

Def(3/a)：在节点3定义了哪些变量，变量a在哪些节点定义了

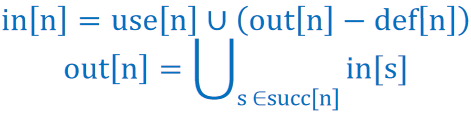
Use(3/a)：在节点3使用了哪些变量，变量a在哪些节点使用了

在所有in-edge上都活跃的node（live-in），在所有out-edge上都活跃的node（live-out）

In[n] and out[n]，性质如下



递归计算结论（得到的一定是最小不动点）：



后续也可以将node扩展为basic block，讨论basic block之间in/out关系，减少需要计算的节点数量；计算out/in集合时，也可以一个一个变量关注

Out/in集合可以依赖bit array或sorted list实现，便于进行union操作（sorted list可能更快）

计算的时间复杂度：

N个node，N个变量，每一轮遍历需要考虑N个node每个node union操作可能要O(N) 操作，时间开销O(N2) ，元素个数最多为2N2，in/out两个集合，每个集合最多N个元素，共有N个node，所以worst-time是O(N4)，In practice ，it between O(N) and O(N^2)

我们一般考虑计算in/out集合的最小不动点（least fixed point，被所有其他解都包含），但执行保守的近似（宁可认为变量live也不认为变量已经dead）

No compiler can ever fully understand how all the control flow in every program will work（类似停机问题），可能知道一部分label（代码）是否会在某次运行中被执行，但不总能预测知晓

Dynamic liveness: a variable a is dynamically live at node n if **some execution** of the program goes from n to a use of a without going through any definition of a（关注于运行时是否经过，动态活跃一定静态活跃）

Static liveness: a variable a is statically live at node n if there is **some path of control-flow edges** from n to some use of a that does not go though a definition of a（仅关注control flow静态分析）

寄存器分配

Interference，prevents a and b from being allocated to the same register

两种类型，变量存活时间重叠；某变量与某寄存器间存有冲突

特别注意MOVE指令，可能无需引入额外的冲突

Interference的添加

Out set中变量和当前语句被定义的变量a之间

在MOVE语句中，仅添加MOVE左侧和live out set（近似于out set）间的冲突

即使是仅定义而未使用的变量，也最好进行计算（可能会带来一些副作用）并放入register中

CH11

Register allocator（NP完全问题）

Coloring近似算法

Build：Construct the interference graphs

每个node代表一个临时的寄存器（变量）

Move指令变量间用虚线（不算已经冲突）

Simplify：存储与一个stack中

如果有顶点度小于K（寄存器的数量），则直接在图中删除这些点

Spill：如果simplify之后仍然留有度大于等于K的点

选择一些点，使其存储于stack中（不再与其他点interference）；Spill结束后，（如果有actual spill）重写program（因为每次取spill node都要从内存中操作），重新构造interference graphs后再进行simplify操作

Select：从stack中不断取出node，并进行color

如果spill node的邻居确实拥有K中颜色，则是actual spill；否则，为该spill node上色（称为optimistic coloring）

Coalescing（聚集）

如果move指令的target和source间没有实线相连，则两个顶点间可以合并

合并这些点之后，原本可以K color的可能无法被K color

两种保证合并安全的算法

**Briggs**：if the resulting node will have fewer than K neighbors of significant degree（度大于等于K的node邻居少于K个）

**George**：for every neighbor t of a, either t already interferes with b or t is of insignificant degree（共同邻居或邻居度小于K）

Build过程不变，建立interference graphs

simplify过程中，与MOVE相关的节点暂时不能simplify

coalesce阶段进行保守的合并，和simplify阶段往复进行（结点可能从MOVE有关变为MOVE无关）

Freeze阶段，如果simplify和coalesce都不能继续进行，将MOVE有关的节点转变为MOVE无关，继续simplify和coalesce过程

Spill and select，在select阶段，成功coalesce的多个node color同样的颜色

如果有actual spill才重构graph

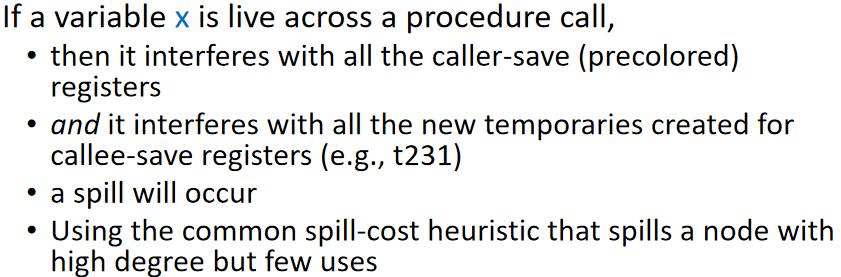
Precolored Nodes

特定暴露的register可能需要precolor，需要预着色的register之间相互interference

We cannot simplify a precoloared node.

We should not spill precolored nodes to memory

跨越函数调用变量需要被调用者来保存



尽量spill那些node度数较高，且不太被使用的变量

Spill priority = （use + def）/ degree ；spill priority越低越先spill

Select完成之后，将color的寄存器填入原程序中，并化简

CH13

Garbage Collection

Garbage: Allocated but no longer used storage

Heap-allocated records that are not reachable by any chain of pointers from program variables are garbage（garbage也有可能被指向，额外考虑循环point）

Garbage collection: The process of reclamation of allocated but no longer used storage without an explicit call to free

program variables是进行这些操作的root

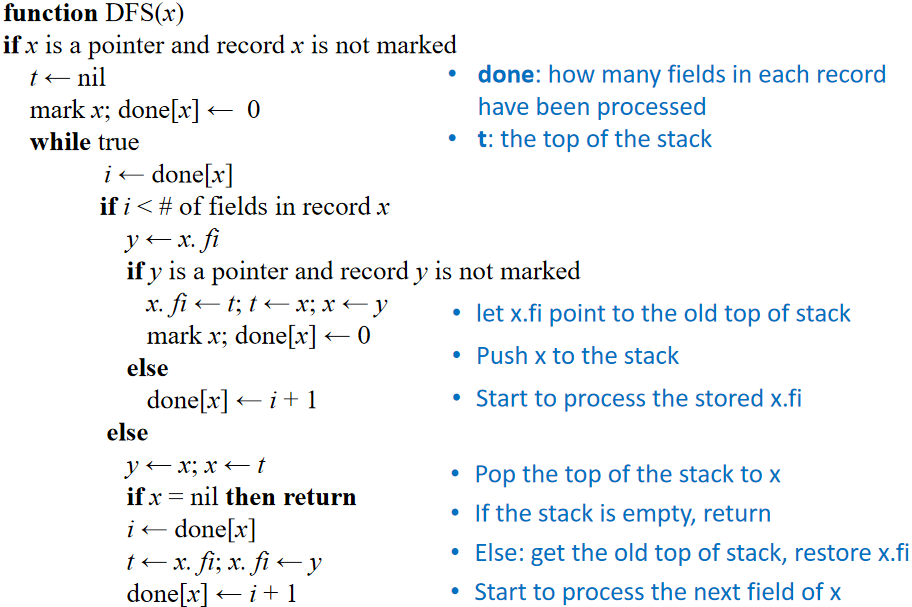
Mark-and-Sweep Collection：

使用DFS算法，从可访问pointer触发，遍历并mark所有可以reach的memory

Mark完成之后，遍历整个heap，清除没有mark的元素（可以使用free-list收集这些空间）；注意，遍历过程中需要将已mark的memory的mark清除掉

如果递归使用DFS算法，不停递归的stack开销可能大于整个heap的大小；可以考虑显示的使用stack来实现DFS算法，但辅助的存储开销可能近似Heap的大小

使用pointer Reversal方法，更加节省空间开销（下图特别重要）



Fragmentation（碎片）

External fragmentation（外部碎片）：连续空闲memory大小小于要求memory大小，从而称为“碎片”

Internal fragmentation（内部碎片）：The program uses a too-large record without splitting it分配空间大于要求memory空间，内部memory无法使用

Reference Counts（引用计数）

记录records被多少其他pointer引用（将其保存于对映record区域中）

当record的reference count减少至0时，需要回收这个record（指针引用，所以改变指针指向前可以操作指向目标的reference count）

虽然容易实现，但有以下缺陷

增加reference count会引入大量辅助代码

减少至0的操作不立即递归进行，直到从free-list中被再次分配时，才触发递归减少reference count，加速程序运行并方便管理（递归计算减少reference count会引入大量代码）

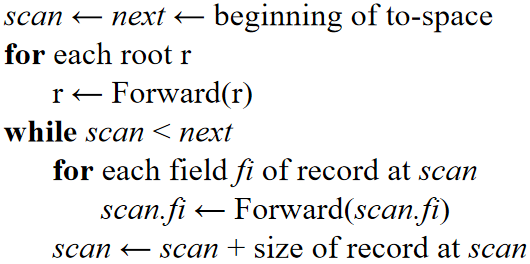
循环引用的record无法被清除

Copying collection

将heap分为两份，一般为from space，正常进行memory的分配；一般为to space，当from space中无法进行memory的分配时，将from space的record搬移到to space，且在 to space中连续存放

Cheney’s algorithm: a collection algorithm using breadth-first search to traverse the reachable data使用BFS进行data的迁移

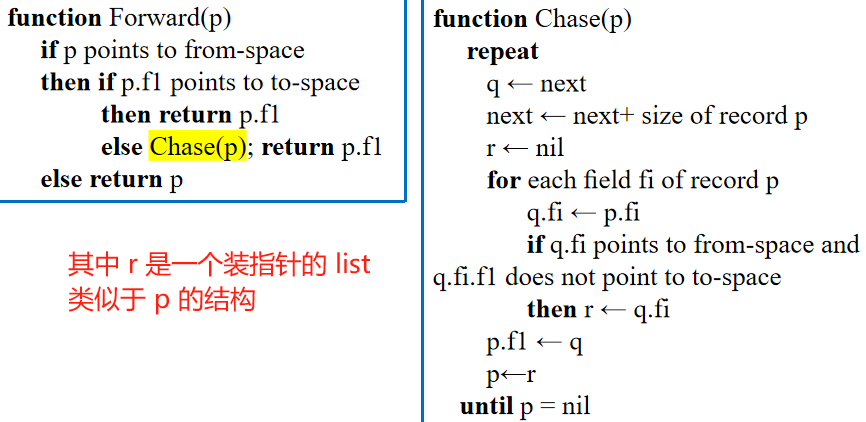
先将所有root复制，再在to space中进行宽度优先的搜寻



宽度优先copy在空间局部性上有不小的缺陷，但使用深度优先copy需要使用pointer-reversal，相对而言不方便且运行速度较慢

Hybrid Algorithm

partly depth-first and partly breadth-first



分配memory这一过程大约需要四条instruction

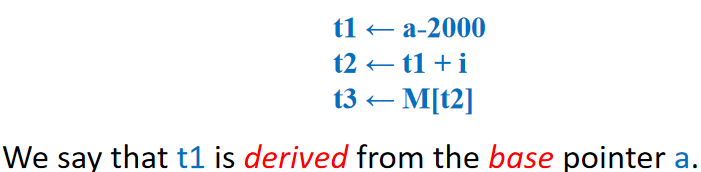
Tiger中，the first word of every object point to a special type- or class-descriptor record（指明当前object的total size，指明当前object中the location of each pointer field）

对于面向对象的语言而言，它们本身就有这样的记录信息

使用pointer map记录存放指针的是临时变量还是局部变量，并指出存放这些指针的区域（在寄存器还是stack中）；在每个可能产生新一轮的垃圾收集的点都需要进行描述

Pointer map一般也用位图记录，且对于程序不同的程序点而言，pointer map可能存在多张；pointer map一般以return address作为索引，记录了不同函数新建栈帧中哪些变量是指针，

Derived Pointer导出指针



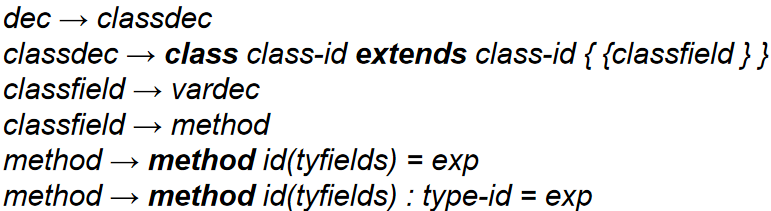
Pointer map同样必须记录这些导出指针并记录它们的base pointer

CH14

面向对象的语言（将tiger扩展）

面向对象有利于模块的封装（encapsulation）和信息隐藏；扩展（extension）和继承（inheritance）

扩展tiger支持class（extend关键词表示继承类的关系）



Tiger自带一个预先定义的class object（没有任何属性和方法），任何类的方法自带一个self参数

Data field的单一继承

Class之间只能单一继承

当B继承A时，我们及拿过A的field放置于B的field之前，从而满足类对继承类属性的访问

对于method而言，每一个类都会维护一个class descriptor指向当前class支持的函数list以及parent class

Static method，跟随parent层级向上找对应function list，直到首次找到为止

Dynamical method，The class descriptor must contain a vector with a method instance for each (nonstatic) method name

Method类似于field记录一般，在记录当前类method之前记录parent类对映method（层级传递，可能越来越多）

多继承（一个class可以继承多个class）

使用graph-coloring的方法，分析不同class不同field同时出现的可能性；根据着色情况对field进行不同offset的排布（可能需要全局记录各个field的offset）

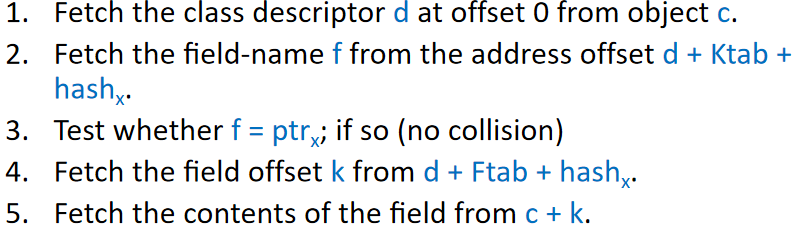
在一些sub-class的field中，可能存在不少 empty slot

可以借助class descriptor，将field进行color位置与offset的映射，在class本身的记录中，field就可以直接连续存储了

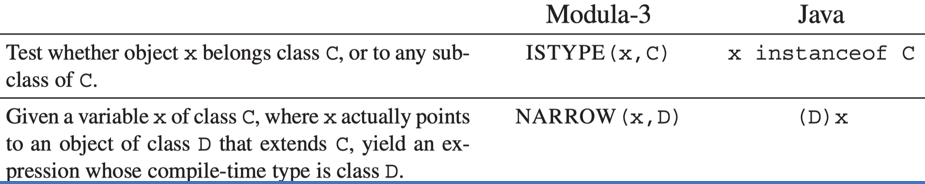
Method的寻找过程也类似使用graph-coloring的方法（和data field一起进行graph-coloring的操作）

Graph-coloring可以在link-time时完成，对于不支持动态增加class的语言而言

Class descriptor的具体实现依赖Ftab（域位移表，真实存储数据）和Ktab（键值表，用于判别冲突的情况）；hash冲突时，Ftab表可以依赖Ktab的存储进行具体定位（两者记录的偏移量一致）



Class test函数



对于ISTYPE函数而言，如果层级向上寻找，效率十分低下；可以显示的维护一个display用于表现nested class关系

NARROW即强制类型转换（保护安全性的前提下）

Display数组从0开始记录，记录当前class的顶层super类型（object），第j层记录自身的类别，大于j的display索引均置0

此时，我们只需要知道目标class在对应关系中为display的索引号，就可以直接判断（在nested class关系中，display的索引号一致保持不变）

父类可以接受子类的赋值定义

可以用switch结构来执行安全的test and narrow类型转换

私有域和私有函数

一般而言，在对映class的symbol table中为field和method的offset后添加一个boolean flag记录该对象是否为私有使用

不同类型的隐私与保护

定义class才可访问

定义class及其sub-class可访问

定义class同模块class可访问

定义class外class仅读，但可用method可写入

CH18

Loop Optimization

Loop: a series of instructions that is repeated until a terminating condition is reached

作为loop的一组点，h能到每一个点，每一个点有一条指向h的边，没有外部的边直接访问当前set中非h的点

A **loop entry** node is one with predecessor outside the loop

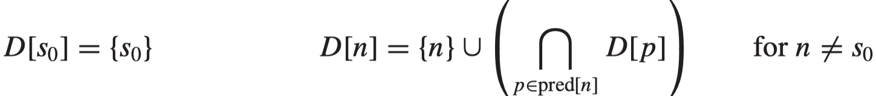
A **loop exit** node is one with a successor outside the loop（本身均属于loop中node）

Dominator支配者

A node d dominates a node n if every path of directed edges from s0（控制流图的start node） to n must go through d（到达某node一定要经过的node d）

每一个node都支配自己；一个node可以被多个node dominate

可以通过迭代的方式计算某一个node的dominate集合



初始化时支配集合仅有自身，如果所有入边的支配集合都包括node A，则当前node的支配集合也包括node A

Theorem: In a connected graph, suppose d dominates n, and e dominates n. Then it must be that either d dominates e or e dominates d（dominate一定是顺序结构）

immediate dominator（ idom(n) ）

Node n的支配节点，且不是node n，且不支配任何node n的支配者（离node n最近的node n支配者）

Dominator Tree，在原图的基础上，仅连接node n和n的immediate dominator

**Back Edge**: a flow-graph edge from a node n to a node h that dominates n

The **natural loop** of a back edge n → h is the set of nodes x such that:

h dominates x and there is a path from x to n not containing h

一般都说the natural loop of back edge......

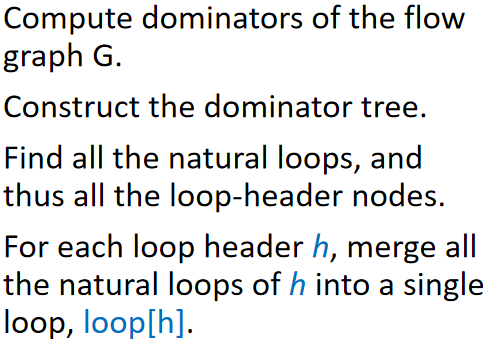
**Nested Loops**:

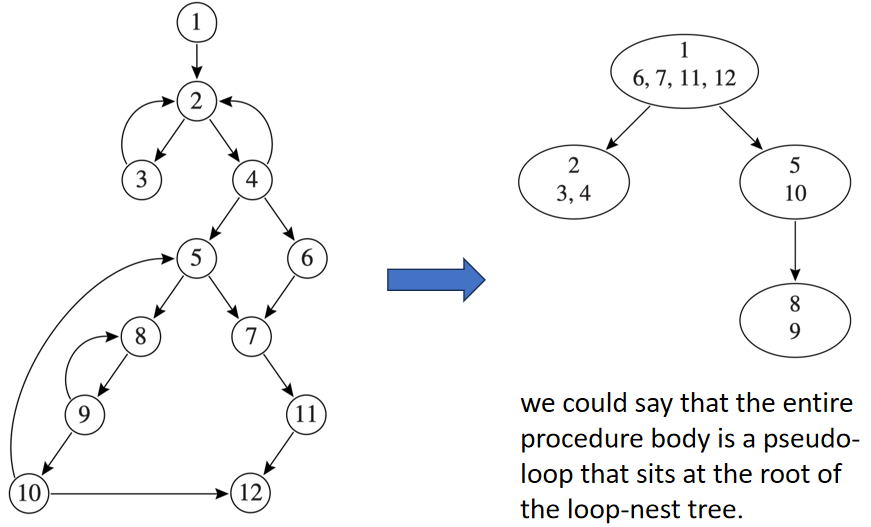
If A and B are loops with headers a and b, such that a ≠ b and b is in A, then the nodes of B are a proper subset of the nodes of A.

We say that loop B is nested within A, or that B is the inner loop

Loop-Nest tree

循环包含解析树，其中叶子节点就是原instruction最内层循坏；在loop-nest tree的node中，第一行放置该loop的header阶段，第二行放置loop中包含的节点；不同的loop如果header一致则可合并；并列loop在图中node为并列关系，包含loop在图中为parent-child关系；另将原图视作一棵“伪树”





Loop-Invariant Computations循环不变计算

根据不变性，可见循环内的部分语句提出至循环外进行而不影响循环进行的一致性

参与运算项是阐述

支配Node d的定义参与运算项outside the loop

可以将循环内操作（t的计算instruction d）提前的限制是

d dominates all loop exits at which t is live-out（有时会带来较大的限制）

there is only one definition of t in the loop

t is not live-out of the loop preheader

Induction Variables归纳变量

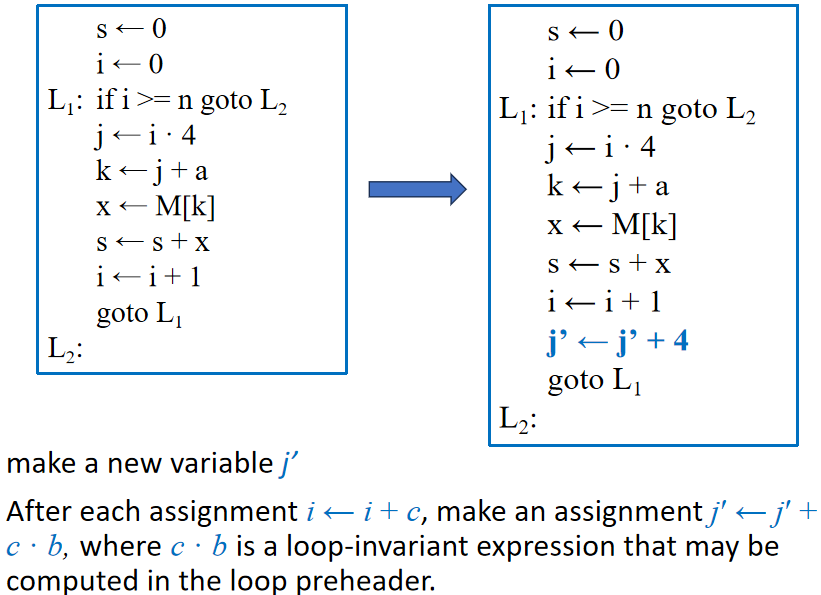
Basic induction variables

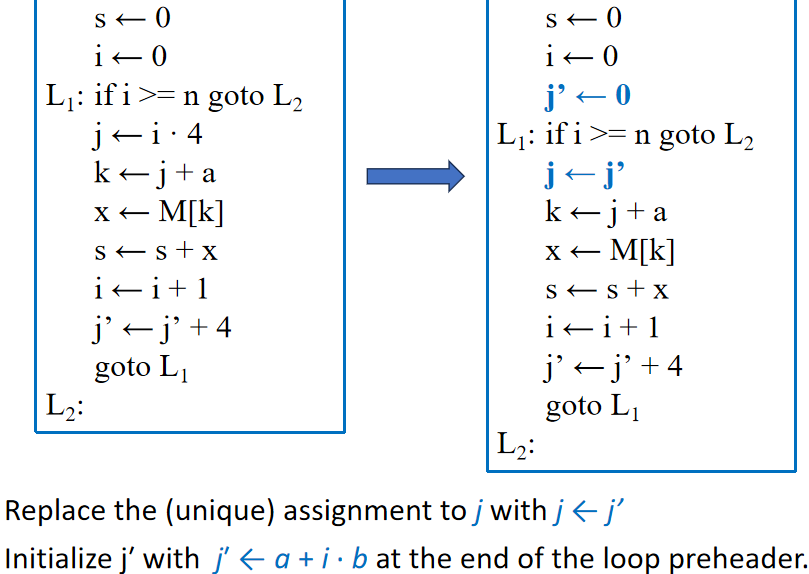
Derivation induction variable（有family的概念，由哪个basic variable引导获得）

Linear induction variable：loop中每次迭代时，增加量均相同

Strength reduction（强度下降）

首先，乘法远较加法消耗大；可以将derived变量的基于basic变量的乘法赋值更换为加法赋值





在上图实例中，可以同理修改变量K

删除useless变量代码

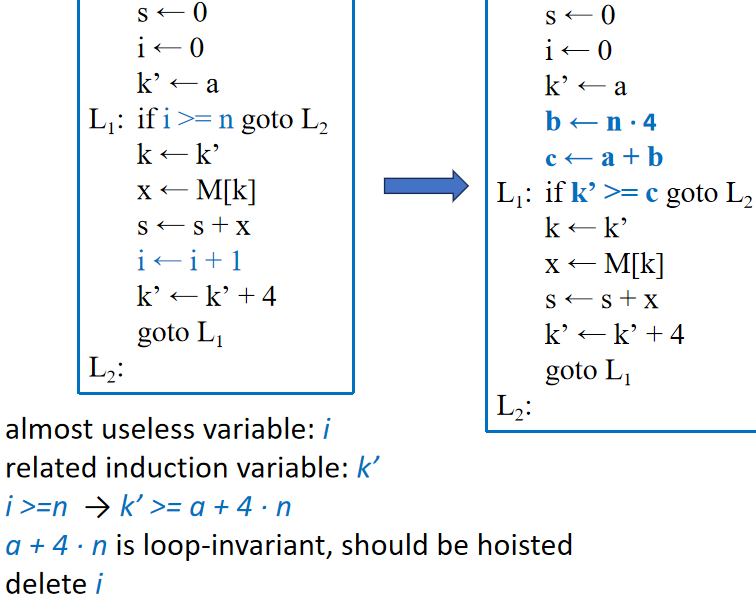
仅定义而未使用的变量

在loop L所有出口均为使用的变量

Almost useless variable

同族induction variable中有非useless变量

only in comparisons against loop-invariant values and in deﬁnitions of itself



继续简化一些其他类型的冗余代码，则可获得相对简洁的结果

Array-bound check

Compiler想移除一些不必要的array-bound check代码（实现复杂）

Loop Unrolling（循环展开）

Unrolling: putting two or more copies of the loop body in a row（loop time）

Loop的判断条件可能会随着循环的展开而发生一定的调整，且因为循坏的展开，可能会在loop的边界增加一些新的条件判断