Contents

I Introduction . . . **3**

1. Modules and Interfaces . . . 3
   - 1.1 Modules . . . 3
   - 1.2 Interfaces . . . 3
   - 1.3 Phases . . . 3
   - 1.4 Modularization(模块化) . . . 3
2. Tools and Software . . . 3
3. Data structures for tree languages . . . 4
   - 3.1 Intermediate Representations (IR) . . . 4
   - 3.2 An example of a program . . . 4
   - 3.3 Programming style . . . 4
   - 3.4 Modularity principle for C programs . . . 4

II Lexical Analysis . . . **4**

1. Lexical Token . . . 4
   - 1.1 A lexical token . . . 4
   - 1.2 Token types . . . 4
   - 1.3 Non-Tokens . . . 4
   - 1.4 An ad hoc lexer . . . 5
   - 1.5 A simpler and more readable lexical analyzers . . . 5
2. Regular Expression . . . 5
   - 2.1 Some Concepts . . . 5
   - 2.2 The notation of regular expressions . . . 5
   - 2.3 Two important disambiguation rules . . . 5
3. Finite Automata . . . 5
   - 3.1 A finite automaton . . . 5
   - 3.2 Deterministic Finite Automaton (DFA) . . . 5
   - 3.3 Combined finite automaton . . . 5
   - 3.4 A transition matrix . . . 5
4. Nondeterministic Finite Automata . . . 6
   - 4.1 NFA . . . 6
   - 4.2 From a RE to an NFA . . . 6
   - 4.3 From an NFA to a DFA . . . 6
   - 4.4 The equivalent states . . . 6
5. Lex: A Lexical Analyzer Generator . . . 6

III Parsing . . . **6**

1. Context-free Grammars . . . 6
   - 1.1 Definition for CFG . . . 6
   - 1.2 Derivations . . . 6
   - 1.3 Parse Trees . . . 6
   - 1.4 Ambiguous Grammars . . . 6
   - 1.5 End-Of-File Marker . . . 6
2. Predictive Parsing . . . 6
   - 2.1 Recursive-Descent Parser . . . 6
   - 2.2 Building a Predictive Parser . . . 7
   - 2.3 Building parsing table . . . 7
   - 2.4 Predictive Parsing: LL(1) . . . 7
   - 2.5 Eliminate left-recursion . . . 8
   - 2.6 Left Factoring . . . 8
   - 2.7 Error Recovery . . . 8
3. LR Parsing . . . 8
   - 3.1 Bottom-up Parsing . . . 8
   - 3.2 LR Parsing Engine . . . 8
   - 3.3 LR(0) Parsing . . . 9
   - 3.4 SLR Parsing . . . 10
   - 3.5 LR(1) Parsing . . . 10
   - 3.6 LALR(1) Parsing . . . 11
   - 3.7 Hierarchy of Grammar Classes . . . 11
4. Using Parser Generators . . . 11
   - 4.1 Yacc . . . 11
   - 4.2 Conflicts . . . 12
   - 4.3 Precedence Directives . . . 12
5. Error Recovery . . . 12
   - 5.1 Recovery Using the Error Symbol . . . 12
   - 5.2 Global Error Repair . . . 12

IV Overview of Semantic Analysis . . . **13**

1. Abstract Parse Trees . . . 13
   - 1.1 Positions . . . 13
2. Symbol Table . . . 13
   - 2.1 Imperative(命令式) . . . 13
   - 2.2 Functional(函数式) . . . 13

V Activation Record . . . **13**

1. Stack Frame(栈帧) . . . 13
2. Frame Pointer . . . 13

3. Parameter Passing . . . 14
4. Frame-Resident Variables . . . 14
5. Static Link . . . 14

VI Translation to Intermediate Code . . . 14
1. Intermediate Representation Trees . . . 14
1.1 Tree Operator . . . 14
2. Translation into Trees . . . 14
2.1 Kinds of Expressions . . . 14
2.2 Variables . . . 15
2.3 Conditionals . . . 15
2.4 Loops . . . 15
2.5 Function Call . . . 15
3. Declarations . . . 15

VII Basic Blocks and Traces . . . 16
1. Canonical Trees(规范树) . . . 16
1.1 Definition . . . 16
1.2 Transformations on ESEQ . . . 16
1.3 Moving CALLs to top level . . . 16
2. Taming Conditional Branches . . . 16
2.1 Basic Blocks(基本块) . . . 16
2.2 Trace(轨迹) . . . 16
2.3 Finishing Up . . . 16

VIII Instruction Selection . . . 17
1. Tree Patterns . . . 17
1.1 Optimal and Optimum Tilings . . . 17
2. Algorithm for Instruction Selection . . . 18
2.1 Maximal Munch . . . 18
2.2 Dynamic Programming . . . 18
2.3 Tree Grammar(树文法) . . . 18
2.4 Fast Match . . . 18
2.5 Efficiency of Tiling Algorithms . . . 18
3. CISC Machines . . . 18

IX Liveness(活跃性) Analysis . . . 19
1. Definition . . . 19
2. Solution of Dataflow Equations . . . 19
2.1 Calculation of Liveness . . . 20
2.2 Representation of Sets . . . 20
2.3 Least Fixed Points . . . 20
2.4 Static v.s. Dynamic Liveness . . . 20
2.5 Interference Graphs . . . 21

X Register Allocation . . . 21
1. Coloring by Simplification . . . 21
2. Coalescing . . . 22
3. Precolored Nodes . . . 22

XI Garbage Collection . . . 23
1. Mark-and-sweep Collection . . . 23
1.1 Cost of garbage collection . . . 23
1.2 Optimization . . . 23
1.3 An Array of Freelists . . . 23
1.4 Fragment . . . 23
2. Reference Counts . . . 23
3. Copying Collection . . . 24
3.1 Cheney's algorithm . . . 24
3.2 Cost of garbage collection . . . 24
4. Interface to the Compiler . . . 24
4.1 Fast Allocation . . . 24
4.2 Describing Data Layouts . . . 24
4.3 Derived Pointers . . . 25

XII Object-Oriented Languages . . . 26
1. Classes . . . 26
2. Single Inheritance of Data Fields . . . 26
2.1 Field layout . . . 26
2.2 Method dispatch . . . 26
3. Multiple Inheritance of Data Fields . . . 27
3.1 Field layout . . . 27
3.2 Method dispatch . . . 27
4. Testing Class Membership . . . 28
5. Private Fields and Methods . . . 28

XIII Loop Optimizations . . . 29
1. Loop in CFG . . . 29
1.1 Dominator tree . . . 29
1.2 Natural Loop . . . 29
1.3 Loop-nest Tree . . . 30

2. Loop Invariant Hoisting 30
2.1 Code hoisting 31

I Introduction

A compiler is a program to translates one language to another.

A Real program language Tiger: Simple and Nontrivial
Two Important Concepts:

- Phases(阶段): one or more modules
  Operating on the different abstract "languages" during compiling process
- Interfaces(接口)
  Describe the information exchanged between modules of the compiler

1. Modules and Interfaces

1.1 Modules

Role: implementing each phase
Advantage: allowing for reuse of the components

1.2 Interfaces

The data structures: Abstract Syntax, IR Trees and Assem.

A set of functions: The translate interface.

1.3 Phases

1) Lex
2) Parse
3) Parsing Actions
4) Semantic Analysis
5) Frame Layout
6) Translate
7) Canonicalize
8) Instruction Selection
9) Control Flow Analysis
10) Dataflow Analysis
11) Register Allocation
12) Code Emission

1.4 Modularization(模块化)

2. Tools and Software

Two of the most useful abstractions:

1) Context-Free Grammars for parsing
2) Regular Expressions for lexical analysis

Two tools for compiling:

1) Yacc converts a grammar into a parsing program
2) Lex converts a declarative specification(声明性规范) into a lexical analysis program

3. Data structures for tree languages

3.1 Intermediate Representations (IR)

The form of a compiling program: Trees Representation(TR):

- The main representation forms
- Several node types with different attributes

3.2 An example of a program

3.3 Programming style

Several conventions for representing tree data structures in C:

1) Trees are described by a grammar

2) A tree is described by one or more typedef, each corresponding to a symbol in the grammar.

3) Each typedef defines a pointer to a corresponding struct.

The struct name, which ends in an underscore, is never used anywhere except in the declaration of the typedef and the definition of the struct itself.

4) Each struct contains a kind fields

An enum showing different variants, one of each grammar rule; and a u field, which is a union.

5) There is more than one nontrivial(value-carraying) symbol in the right-hand side of a rule. The union has a component that is itself a struct comprising these values

6)

3.4 Modularity principle for C programs

1)

II Lexical Analysis

To translate a program from one language into another, a compiler first pull it apart and understand its structure and meaning, then put it together in a different way.

- The front end(前端): performs analysis
- The back end(后端): performs synthesis

The analysis is usually broken up into:

1) Lexical analysis
2) Syntax analysis
3) Semantic analysis

Task of the lexical analyzer:

1) Taking a stream of characters
2) Produces a stream of tokens
3) Discarding white space and comments

1. Lexical Token

1.1 A lexical token

- A sequence of characters
- A unit in the grammar of a programming language

1.2 Token types

Classification of lexical tokens: A finite set of token types.

Type	Examples
ID	`foo  n14  last`
NUM	`73  0 00  515  082`
REAL	`66.1  .5  10.  1e67  5.5e-10`
IF	`if`
COMMA	`,`
NOTEQ	`!=`
LPAREN	`(`
RPAREN	`)`

Figure II.1: Token types

1.3 Non-Tokens

comment	`/* try again */`
preprocessor directive	`#include<stdio.h>`
preprocessor directive	`#define NUMS 5 , 6`
macro	`NUMS`
blanks, tabs, and newlines	

Figure II.2: Non-Tokens

The preprocessor deletes the non-tokens.

Example II.1.

```
float match0(char *s){ /* find a zero */
    if (!strncmp(s, "0.0", 3))
        return 0.;
}
```

```
FLOAT ID(match0) LPAREN CHAR STAR ID(s) RPAREN LBRACE
IF LPAREN BANG ID(strncmp) LPAREN ID(s) COMMA
  ↪  STRING(0.0) COMMA NUM(3) RPAREN RPAREN
RETURN REAL(0.0) SEMI
RBRACE EOF
```

1.4 An ad hoc lexer

Any reasonable programming language can be used to implement it. (人和代码有一个能跑就行)

1.5 A simpler and more readable lexical analyzers

- Regular expressions: Specify lexical tokens
- Deterministic finite automata: Implementing lexers
- Mathematics: Connecting the above two

2. Regular Expression

2.1 Some Concepts

- A language is a set of strings
- A string is a finite sequence of symbols (string 没有被赋予意义)
- A symbol is taken from a finite alphabet

2.2 The notation of regular expressions

a	An ordinary character stands for itself.
ϵ	The empty string.
	Another way to write the empty string.
$M \mid N$	Alternation, choosing from M or N.
$M \cdot N$	Concatenation, an M followed by an N.
MN	Another way to write concatenation.
M^*	Repetition (zero or more times).
M^+	Repetition, one or more times.
$M?$	Optional, zero or one occurrence of M.
$[\mathbf{a}-\mathbf{zA}-\mathbf{Z}]$	Character set alternation.
.	A period stands for any single character except newline.
"a.+*"	Quotation, a string in quotes stands for itself literally.

Figure II.3: Regular expression notation

优先级: $* > \cdot > |$.

Example II.2.

1) $(0|1)*\cdot 0$

   Binary numbers that are multiples of two

```
if                                      {return IF;}
[a-z][a-z0-9]*                          {return ID;}
[0-9]+                                  {return NUM;}
([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)     {return REAL;}
("--"[a-z]*"\n")|(" "|"\n"|"\t")+       { /* do nothing */ }
.                                       {error();}
```

Figure II.4: Regular expressions for some tokens

2.3 Two important disambiguation rules

These rules are a bit ambiguous.

1) Longest match

   The longest initial substring of the input that can match any regular expression is taken as the next token.

2) Rule priority

   The first regular expression that can match determines its token-type

3. Finite Automata

3.1 A finite automaton

Definition II.1 (A finite automaton)**.**

- *A finite set of states;*
- *Edges lead from one state to another, and each edge is labeled with a symbol* ;
- *One state is the start state, and certain of the states are distinguished as final states.*

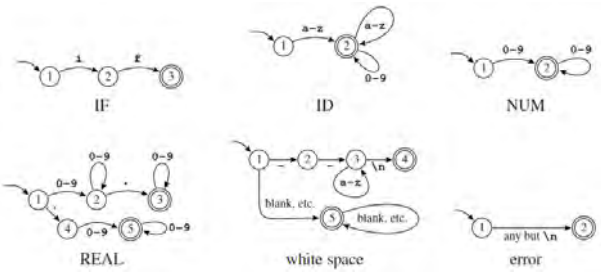


Figure II.5: Finite automata for lexical tokens.

3.2 Deterministic Finite Automaton (DFA)

A DFA accepts or rejects a string as follows: TLDR. 看计算理论 II.1-6.

3.3 Combined finite automaton

Using ad hoc method.

3.4 A transition matrix

Encoding **Figure** II.6

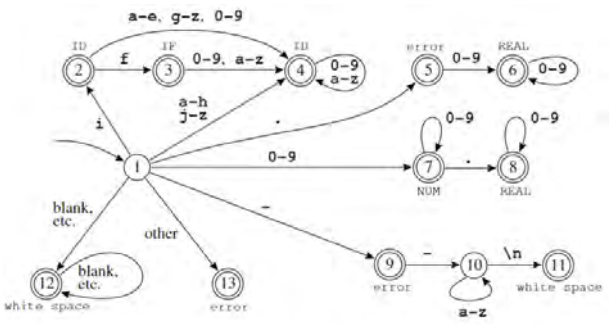


Figure II.6: Combined finite automaton

4. Nondeterministic Finite Automata

4.1 NFA

A NFA:

- Have to choose one from the edges to follow out of a state
- Have special edges labeled with ϵ

4.2 From a RE to an NFA

The conversion algorithm: (计算理论 III.2 上说这个转换是 naive 的)

1) Turning each regular expression into an NFA with a tail (start edge) and a head (ending state).
2) The rules for translating

4.3 From an NFA to a DFA

To avoid guesses by trying every possibility at once.
计算理论 II.13.

4.4 The equivalent states

略

5. Lex: A Lexical Analyzer Generator

看书

III Parsing

- Lexical Analysis: Create sequence of tokens from characters
- Parsing: Create abstract syntax tree from sequence of tokens

Syntax: the way in which words are put together to form phrases, clauses, or sentences

- Input: sequence of tokens from lexer;
- Output: parse tree of the program (But some parsers never produce a parse tree ...)

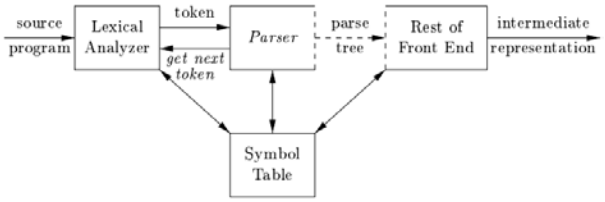


Figure III.1: compiler

1. Context-free Grammars

1.1 Definition for CFG

计算理论 IV.1.

1.2 Derivations

计算理论 IV.2-3
用 CFG 生成 string

1.3 Parse Trees

计算理论 IV.5
Representing derivations as a tree. Parse trees have meaning.

1.4 Ambiguous Grammars

A grammar is ambiguous if the same sequence of tokens can give rise to two or more parse trees.

解决二义性 (Ambiguity) 的方式就是重写文法. 解决二义性没有通用的方法, 一般都是通过声明 precedence(优先级) 与 associativity(结合性) 来消除文法中的二义性.

1.5 End-Of-File Marker

Use $ to represent end of file

2. Predictive Parsing

2.1 Recursive-Descent Parser

Each grammar production turns into one clause of a recursive function. 自顶向下分析.

Problem: 预测分析需要每个子表达式的 the first terminal symbol 提供足够的信息来决定生成的是什么.

Definition III.1 (Nullable). *Non-terminal X is Nullable if X can derive the empty string.*

Definition III.2 (First Sets). *$First(X)$ is the set of terminals that can begin strings derived from X.*

$$First(X) = \{t | X \to^* t\alpha\} \cup \{\epsilon | X \to^* \epsilon\}$$

即, 如果 X 可以经 0 步或多步推导出以 terminal t 开头的串, 那么 t 属于 $First(X)$; 如果 X 可以经 0 步或者多步推导出空串, 那么 ϵ 属于 $First(X)$.

Definition III.3 (Follow Sets). *$Follow(X)$ is the set of terminals that can immediately follow X. That is, $t \in Follow(X)$ if there is any derivation containing Xt. This can occur if the derivation contains $XYZt$ where Y and Z both derive ϵ.*

$$Follow(X) = \{t | S \to^* \alpha X t \beta\}$$

ϵ 不会出现在 Follow Sets 中. Start Symbol 的 Follow Set 包含文件结束符 \$.

Algorithm III.1 Compute *First*, *Follow*, and *nullable*

Initialize $First$ and $Follow$ to all empty sets, and nullable to all false.
for each terminal symbol t **do**
  $First(t) \leftarrow \{t\}$
end for
repeat
  for each production $X \to Y_1 Y_2 \dots Y_k$ **do**
    for each i from 1 to k, each j from $i+1$ to k **do**
      if all the Y_i are nullable **then**
        $nullable(X) \leftarrow true$
      end if
      if $Y_1 \dots Y_{i-1}$ are all nullable **then**
        $First(X) \leftarrow First(X) \cup First(Y_i)$
      end if
      if $Y_{i+1} \dots Y_k$ are all nullable **then**
        $Follow(Y_i) \leftarrow Follow(Y_i) \cup Follow(X)$
      end if
      if $Y_{i+1} \dots Y_{j-1}$ are all nullable **then**
        $Follow(Y_i) \leftarrow Follow(Y_i) \cup First(Y_j)$
      end if
    end for
  end for
until $First, Follow$ and nullable didn't change in this iteration.

2.2 Building a Predictive Parser

- $Z \to XYZ | d$
- $Y \to c | \epsilon$
- $X \to a | Y$

	Nullable	First	Follow
Z	no	d, a, c	
Y	yes	c	a, c, d
X	yes	a, c	a, c, d

2.3 Building parsing table

- if $T \in First(s)$ then enter $(X \to s)$ in row X, col T
- if s is Nullable and $T \in Follow(X)$, enter $(X \to s)$ in row X, col T

Build parsing table where row X, col T tells parser which clause to execute in function X with next-token T:

	a	c	d
Z	$Z \to XYZ$	$Z \to XYZ$	$Z \to d$ $Z \to XYZ$
Y	$Y \to$	$Y \to$ $Y \to c$	$Y \to$
X	$X \to a$ $X \to Y$	$X \to Y$	$X \to Y$

2.4 Predictive Parsing: LL(1)

依据 grammar 构造的 parsing table 没有冲突, 此 grammar 才能被称为 LL(1) grammar.

LL(1): Left-to-right parse, Left-most derivation, 1 symbol lookahead.

In LL(k) parsing table, columns include every k-length sequence of terminals.

用栈来存储正在生成的 parse tree, 栈顶为 leftmost nonterminal 或即将匹配的 leftmost terminal.

Example III.1.

- $E \to TX$
- $T \to \text{int } Y | (E)$
- $X \to +E | \epsilon$
- $Y \to *T | \epsilon$

	Nullable	First	Follow
E	no	$(, \text{int}$	$), \$$
X	yes	$+, \epsilon$	$), \$$
T	no	$(, \text{int}$	$+,), \$$
Y	yes	$*, \epsilon$	$+,), \$$

	int	$*$	$+$	$($	$)$	$\$$
E	TX			TX		
X			$+E$		ϵ	ϵ
T	int Y			(E)		
Y		$*T$	ϵ		ϵ	ϵ

Stack	Input	Action
$E\$$	int $*$ int$\$$	TX
$TX\$$	int $*$ int$\$$	int Y
int $YX\$$	int $*$ int$\$$	terminal
$YX\$$	$*$int$\$$	$*T$
$*TX\$$	$*$int$\$$	terminal
$TX\$$	int$\$$	int Y
int $YX\$$	int$\$$	terminal
$YX\$$	$\$$	ϵ
$X\$$	$\$$	ϵ
$\$$	$\$$	Accept

2.5 *Eliminate left-recursion*

消除左递归 Rewrite the grammar so it parses the same language but the rules are different.

- $E \to E + T | T \Rightarrow E \to TE', E' \to +TE' | \epsilon$
- $A \to A\alpha | \beta \Rightarrow A \to \beta A', A' \to \alpha A' | \epsilon$

2.6 *Left Factoring*

提取左因子

- $E \to T + E | T \Rightarrow E \to TX, X \to +E | \epsilon$
- $P \to \alpha\beta | \alpha\gamma \Rightarrow P \to \alpha Q, Q \to \beta | \gamma$

2.7 *Error Recovery*

How should error be handled?

- Raise an exception and quit parsing
- Print an error message and recover from the error

This can proceed by deleting, replacing, or inserting tokens

3. LR Parsing

3.1 *Bottom-up Parsing*

自底向上分析.

LL(k) 只看前面 k 个 token.

LR(k): Left-to-right parse, Rightmost derivation, k-token lookahead 可以看到代表输入的全部右侧的生成.

Shift-reduce parsing:

- Reduce(规约): token 到 non-terminal
- Shift(移进): 右移一位, 考虑下一个 terminal

LALR variant: The basis for parsers for most modern programming languages, Implemented in tools such as Yacc.

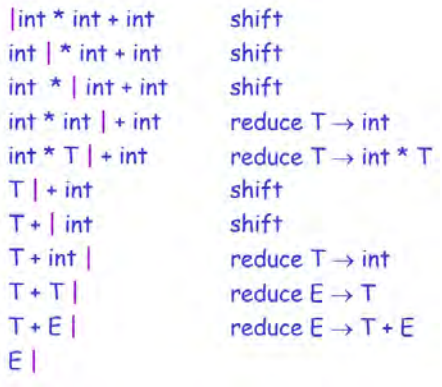


Figure III.2: Bottom-up Parsing Example

3.2 *LR Parsing Engine*

LR parser 使用 DFA 来决定何时 shift/reduce. 具体的:

1) 通过 LR items 构造 NFA
2) NFA 转换为 DFA
3) DFA 转换为 LR parser table
4) 依据 LR parser table 决定何时 shift/reduce

一般来说, LR parser table 有以下 elements:

- s_n: Shift into state n
- g_n: Goto state n
- r_k: Reduce by rule k
- a: Accept
- : Error

LR parser table 具体的使用方式:

- $Shift(n)$: Advance input one token; push n on stack.
- $Reduce(k)$:

  1) Pop stack as many times as the number of symbols on the right-hand side of rule k;

2) Let X be the left-hand-side symbol of rule k;

3) Push X into stack, and look up X to get "goto n";

4) Push n on top of stack.

- *Accept*: Stop parsing, report success
- *Error*: Stop parsing, report failure

如 **Figure** III.4 所示, stack 需要维护 token 与 state 两个量, 这里使用 (state, token) 对表示.

Algorithm III.2 LR parser table 使用

a 表示当前入读的 token.
a_{top}, s_{top} 分别为栈顶的 token 与 state.
$T_{(i,a)}$ 表示 parser table 中, state i 行, token a 列所对应的 element.
repeat
  if $T_{(s_{top},a)}$ is s_n **then**
    PUSH(n, a)
    a=GETCHAR()
  else if $T_{(s_{top},a)}$ is r_k **then**
    Assume rule k is $A \to \beta$
    POP() $|\beta|$ times
    $T_{(s_{top},A)}$ is g_n
    PUSH(n, A)
  else
    Error
  end if
until $T_{(s_{top},a)}$ is accept
Accept

Example III.2.

1) $S \to S;S$
2) $S \to id := E$
3) $S \to print(L)$
4) $E \to id$
5) $E \to num$
6) $E \to E + E$
7) $E \to (S, E)$
8) $L \to E$
9) $L \to L, E$

	id	num	print	;	,	+	:=	(	)	$	S	E	L
1	s4		s7								g2		
2				s3						a			
3	s4		s7								g5		
4							s6						
5				r1	r1					r1			
6	s20	s10						s8				g11	
7								s9					
8	s4		s7								g12		
9												g15	g14
10				r5	r5	r5			r5	r5			
11				r2	r2	s16				r2			
12				s3	s18								
13				r3	r3					r3			
14					s19				s13				
15					r8				r8				
16	s20	s10						s8				g17	
17				r6	r6	s16			r6	r6			
18	s20	s10						s8				g21	
19	s20	s10						s8				g23	
20				r4	r4	r4			r4	r4			
21									s22				
22				r7	r7	r7			r7	r7			
23					r9	s16			r9				

Figure III.3: LR parsing table

Stack	Input	Action
	a := 7 ; b := c + (d := 5 + 6 , d) $	*shift*
id$_4$	:= 7 ; b := c + (d := 5 + 6 , d) $	*shift*
id$_4$:=$_6$	7 ; b := c + (d := 5 + 6 , d) $	*shift*
id$_4$:=$_6$ num$_{10}$	; b := c + (d := 5 + 6 , d) $	*reduce* $E \to$ num
id$_4$:=$_6$ E_{11}	; b := c + (d := 5 + 6 , d) $	*reduce* $S \to$ id:=E
S_2	; b := c + (d := 5 + 6 , d) $	*shift*
S_2 ;$_3$	b := c + (d := 5 + 6 , d) $	*shift*
S_2 ;$_3$ id$_4$	:= c + (d := 5 + 6 , d) $	*shift*
S_2 ;$_3$ id$_4$:=$_6$	c + (d := 5 + 6 , d) $	*shift*
S_2 ;$_3$ id$_4$:=$_6$ id$_{20}$	+ (d := 5 + 6 , d) $	*reduce* $E \to$ id
S_2 ;$_3$ id$_4$:=$_6$ E_{11}	+ (d := 5 + 6 , d) $	*shift*
S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$	(d := 5 + 6 , d) $	*shift*
S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$	d := 5 + 6 , d) $	*shift*
S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ id$_4$	:= 5 + 6 , d) $	*shift*
S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ id$_4$:=$_6$	5 + 6 , d) $	*shift*
S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ id$_4$:=$_6$ num$_{10}$	+ 6 , d) $	*reduce* $E \to$ num
S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ id$_4$:=$_6$ E_{11}	+ 6 , d) $	*shift*
S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ id$_4$:=$_6$ E_{11} +$_{16}$	6 , d) $	*shift*
S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ id$_4$:=$_6$ E_{11} +$_{16}$ num$_{10}$	, d) $	*reduce* $E \to$ num
S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ id$_4$:=$_6$ E_{11} +$_{16}$ E_{17}	, d) $	*reduce* $E \to E + E$
S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ id$_4$:=$_6$ E_{11}	, d) $	*reduce* $S \to$ id:=E
S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ S_{12}	, d) $	*shift*
S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ S_{12} ,$_{18}$	d) $	*shift*
S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ S_{12} ,$_{18}$ id$_{20}$	) $	*reduce* $E \to$ id
S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ S_{12} ,$_{18}$ E_{21}	) $	*shift*
S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ S_{12} ,$_{18}$ E_{21})$_{22}$	$	*reduce* $E \to (S, E)$
S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ E_{17}	$	*reduce* $E \to E + E$
S_2 ;$_3$ id$_4$:=$_6$ E_{11}	$	*reduce* $S \to$ id:=E
S_2 ;$_3$ S_5	$	*reduce* $S \to S; S$
S_2	$	*accept*

Figure III.4: Shift-reduce parse of a sentence

3.3 LR(0) Parsing

Making shift/reduce decisions without any lookahead.

Definition III.4 (LR(0) item)**.** *A grammar rule, combined with the dot that indicates a position in its right-hand side, is called an item (specifically, an LR(0) item).*

A state is just a set of items.

Example III.3. For grammar

1) $S' \to S$
2) $S \to (S)S$
3) $S \to \epsilon$

The LR(0) items include:

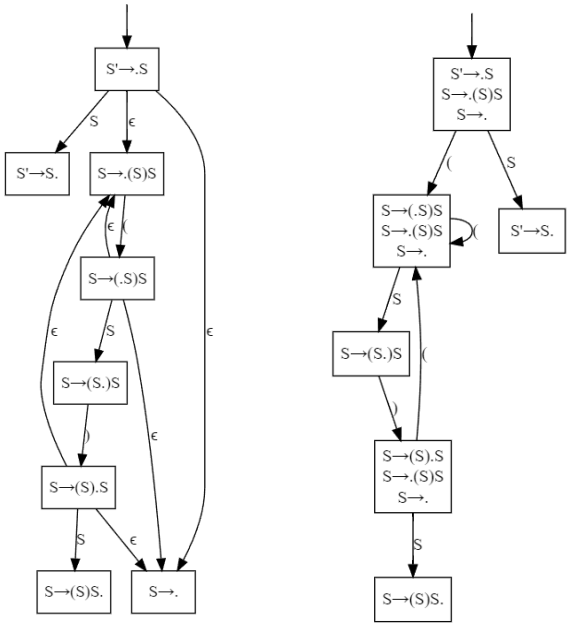
- $S' \to .S$, $S' \to S.$

- $S \to .(S)S$, $S \to (.S)S$, $S \to (S.)S$, $S \to (S).S$, $S \to (S)S.$
- $S \to .\epsilon$, $S \to \epsilon.$

LR(0) Item 之间存在一些转换关系:

- $X \to .\alpha\beta$, 接受 α 变为 $X \to \alpha.\beta$
- 若有 $X \to \gamma T\omega, Y \to \alpha\beta$, 则 $X \to \gamma.Y\omega$ 可以转换为 $Y \to .\alpha\beta$ (因为凑 X 必须先凑 Y)

通过这些转换关系将 LR(0) items 写为一个 NFA, 然后将 NFA 转换为 DFA.



(a) NFA (b) DFA

Figure III.5: LR(0) NFA and DFA for **Example** III.3

然后依据 DFA 与如下规则:

- 对每条 t 是 terminal 的边 $S_i \xrightarrow{t} S_j$, 令 $T_{(i,t)}$ 为 s_j. 其中, S_i 表示状态 i,
- 对每条 X 是 non-terminal 的边 $S_i \xrightarrow{X} S_j$, 令 $T_{(i,X)}$ 为 g_j.
- 对每个包含 $S' \to S.\$$ 的状态 i, 令 $T_{(i,\$)}$ 为 accept.
- 对每个包含 $X \to \gamma.$ (结尾带点的规则 k) 的状态 i, 对每个 terminal t, 令 $T_{(i,t)}$ 为 r_k.

这样就可以构造出 LR(0) Parsing Table, 如 **Table** III.1 所示.

注意到 $T_{(1,()}, T_{(3,()}, T_{(5,()}$ 出现了冲突, 这三个都属于 shift-reduce conflict. 只有构造出的 LR(0) Parsing Table 没有冲突时, grammar 才能被称为 LR(0) grammar.

Table III.1: LR(0) Parsing Table for **Example** III.3

	(	)	$	S
1	s_3, r_3	r_3	r_3	g_2
2	r_1	r_1	r_1, accept	
3	s_3, r_3	r_3	r_3	g_4
4		s_5		
5	s_3, r_3	r_3	r_3	g_6
6	r_2	r_2	r_2	

3.4 SLR Parsing

SLR 中的 S 表示 Simple. SLR Parsing 在 LR(0) 的基础上通过简单的判断尝试解决冲突.

SLR 在构造形如 LR(0) 的 DFA 之上, 还需要计算每个 non-terminal 的 Follow Set. SLR 只对那些下一个符号在对应 non-terminal 的 Follow Set 的情况进行 reduc. 具体更改如下:

- 对每个包含 $X \to \gamma.$ (结尾带点的规则 k) 的状态 i, 对每个 terminal $t \in Follow(X)$, 令 $T_{(i,t)}$ 为 r_k.

构造出的 SLR Parsing Table, 如 **Table** III.2 所示.

Table III.2: SLR Parsing Table for **Example** III.3

	(	)	$	S
1	s_3	r_3	r_3	g_2
2			r_1, accept	
3	s_3	r_3	r_3	g_4
4		s_5		
5	s_3	r_3	r_3	g_6
6		r_2	r_2	

3.5 LR(1) Parsing

Definition III.5. *An LR(1) item consists of a grammar production, a right-hand-side position (represented by the dot), and a lookahead symbol. The idea is that an item $(A \to \alpha.\beta, x)$ indicates that the sequence α is on top of the stack, and at the head of the input is a string derivable from βx.*

LR(1) Item 存在如下两种转化:

- $X \to .\alpha\beta, t$ 接受 α 变为 $X \to \alpha.\beta, t$

- 若有 $X \to \gamma T \omega, Y \to \alpha\beta$, 则对于每个 $t_i \in First(\omega t)$ (ω 可以是 ϵ), $X \to \gamma.Y\omega, t$ 可以转换为 $Y \to .\alpha\beta, t_i$

Example III.4. 对于如下 grammar:

- $S' \to S$
- $S \to aAd$
- $S \to bBd$
- $S \to aBe$
- $S \to bAe$
- $A \to c$
- $B \to c$

Start Symbol 有 LR(1) Item $S' \to .S, \$$

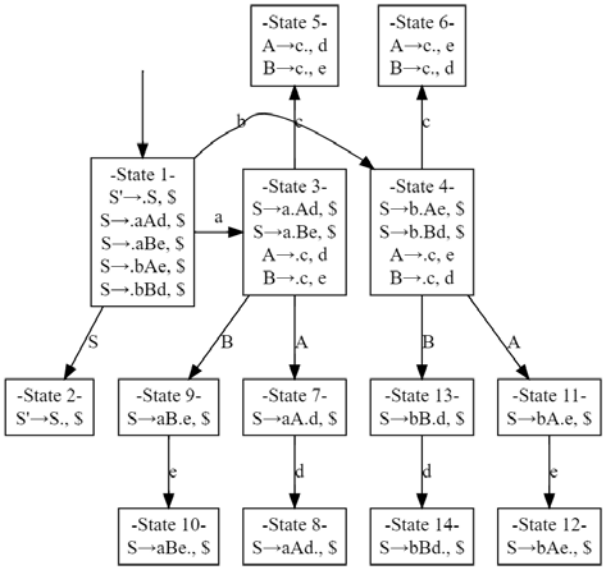


Figure III.6: LR(1) DFA for **Example** III.4

构造 LR(1) Parsing Table 的方式, 也是只基于 LR(0) 更改了 reduce:

- 对每个包含 $X \to \gamma.$ (结尾带点的规则 k) 的状态 i, 对每个 lookahead symbol t, 令 $T_{(i,t)}$ 为 r_k.

3.6 LALR(1) Parsing

LALR(1)(Look-Ahead LR(1)) 是 LR(1) 的简化版本.

对于每一个状态, 将其包含的所有 LR(1) items 的第一个分量的集合称为这个状态的核心 (core).

例如, 对于 **Figure** III.6, 其状态 5 和 6 的核心均为 $\{A \to c., B \to c.\}$. 将这样的具有相同核心的状态进行合并, 通常能够减少许多状态.

但是, 这有时 (虽然很少) 也可能引入 reduce-reduce conflict. 不过问题不大.

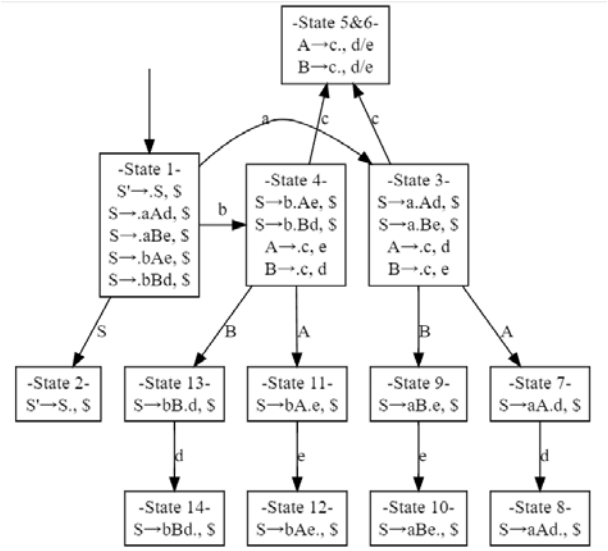


Figure III.7: LALR(1) DFA for **Example** III.4

3.7 Hierarchy of Grammar Classes

The relationship between several classes of grammars.

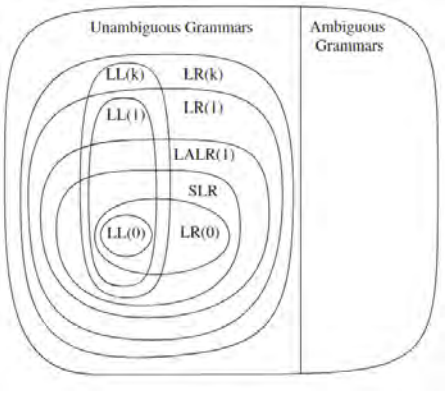


Figure III.8: A hierarchy of grammar classes.

4. Using Parser Generators

4.1 Yacc

Yacc ("Yet another compiler-compiler"): A classic and widely used parser generator

A Yacc specification is divided into three sections, separated by %% marks:

```
parser declarations
%%
grammar rules
%%
programs
```

- Parser declaration: a list of N, T and so on
- Programs: ordinary C code usable from the semantic action
- Grammar rules: production of the form
  `exp: exp PLUS exp {semantic action}`
  where `exp` is a nonterminal producing a right-hand side of `exp + exp`, and `PLUS` is a terminal symbol (token). The semantic action is written in ordinary C and will be executed whenever the parser reduces using this rule.

Example III.5. For grammar

1) $P \to L$
2) $S \to id := id$
3) $S \to$ while id do S
4) $S \to$ begin L end
5) $S \to$ if id then S
6) $S \to$ if id then S else S
7) $L \to S$
8) $L \to L; S$

```
%{
int yylex(void);
void yyerror(char *s) { EM_error(EM_tokPos, "%s", s); }
%}
%token ID WHILE BEGIN END DO IF THEN ELSE SEMI ASSIGN
%start prog
%%
prog: stmlist
stm : ID ASSIGN ID
    | WHILE ID DO stm
    | BEGIN stmlist END
    | IF ID THEN stm
    | IF ID THEN stm ELSE stm
stmlist : stm
        | stmlist SEMI stm
```

4.2 Conflicts

- shift-reduce conflict: Resolved using shift by default in Yacc
- reduce-reduce conflict: Resolved using the rule appears early in the grammar

4.3 Precedence Directives

Ambiguous grammars are still be useful if finding ways to resolve the conflict.

Yacc uses precedence directives to resolve this class of shift-reduce conflicts

5. Error Recovery

5.1 Recovery Using the Error Symbol

Local error recovery mechanisms.

If a syntax error is encountered in the middle of an expression, the parser should skip to the next semicolon or right parenthesis(called synchronizing tokens) and resume parsing.

error is considered a terminal symbol. When the LR parser reaches an error state, it takes the following actions:

1) Pop the stack (if necessary) until a state is reached in which the action for the error token is shift.
2) Shift the error token.
3) Discard input symbols (if necessary) until a lookahead is reached that has a nonerror action in the current state.
4) Resume normal parsing.

5.2 Global Error Repair

Global error repair: finds the smallest set of insertions and deletions that would turn the source string into a syntactically correct string, even if the insertions and deletions are not at a point where an LL or LR parser would first report an error.

Burke-Fisher error repair: 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.

IV Overview of Semantic Analysis

1. Abstract Parse Trees

这种方法限制编译器按照解析程序的顺序来分析程序.

parse tree(解析树) 从 semantic 中分离 issues of syntax(parsing), 并让编译器后期可以遍历它.

- Concrete parse tree: 代表了源码的 concrete syntax. 从技术上讲, 对于输入的每个 token 只有一个 leaf, 对于 parse 中减少的每个语法规则只有一个内部节点. 但其中的标点符号是冗余信息.
- Abstract parse tree/Abstract syntax tree: abstract syntax 表达了源码的 phrase structure, 在没有任何语义解释下解决所有 parsing 问题.

一般是语义分析 concrete syntax 然后构建 abstract syntax tree.

1.1 Positions

2. Symbol Table

Definition IV.1. *A symbol table is a data structure that tracks the current bindings of identifiers*

Code IV.1: A Fancier Symbol Table

```
enter_scope()   // start a new nested scope
find_symbol(x)  // finds current x (or null)
add_symbol(x)   // add a symbol x to the table
check_scope(x)  // true if x defined in current scope
exit_scope()    // exit current scope
```

局部变量都有一个作用域 (scope), 变量仅在自己的作用域中可见.

环境是由绑定 (binding) 组成的集合, 指标识符和含义之间的一种映射关系, 用箭头表示. e.g. $\{g \to string\}$, $\{a \to int\}$

2.1 Imperative(命令式)

- 实现: bucket list(hash table)

插入 identifiers 时插入到头部, 在退出 scope 时方便 pop.

2.2 Functional(函数式)

- 实现: 可持久化二叉搜索树 (persistent BST)

使用可持久化来控制 scope

V Activation Record

1. Stack Frame(栈帧)

Definition V.1. *栈中存放函数的局部变量/参数/返回地址/临时变量的这片区域为该函数的活动记录 (activation record) 或栈帧 (stack frame).*

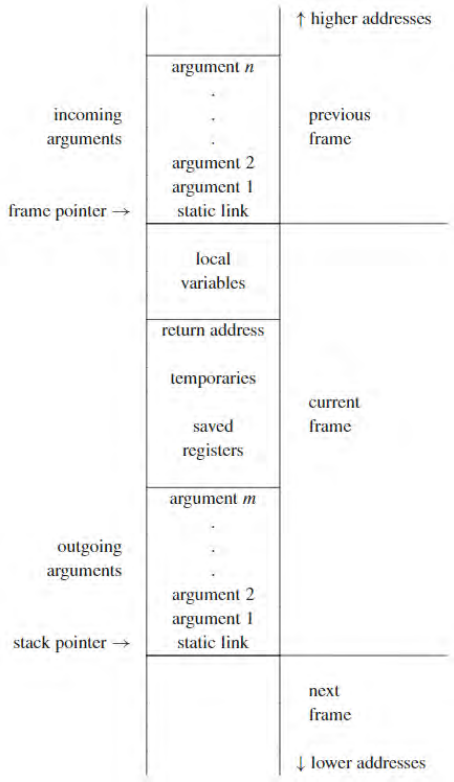


Figure V.1: a typical stack frame layout

- incoming arguments: 是前一个帧的一部分, 但其与 frame pointer 的偏移量已知.
- return address: 由 CALL 指令创建, 告知完成此函数后需要返回何处.
- local variables: 一些在帧中, 一些在寄存器上. 在寄存器与帧中临时空间之间移动.

2. Frame Pointer

指向当前帧的指针, 一般是上一个 sp; 有些栈帧会分配一个寄存器存 fp; 虚寄存器: fp=sp+size(frame).

帧指针的变化: 一个函数 g 调用 f 时,

1) sp 指向 g 传给 f 的第一个参数
2) f 分配栈帧 (sp-栈帧大小)
3) 进入 f 时旧的 sp 变成当前帧指针;fp 旧值被保存到栈帧内, 新的帧指针变成旧 sp
4) f 退出时把 fp 拷贝给 sp, 再取回原先保存的 fp 即可.

3. Parameter Passing

现代计算机传参约定: 前 $k(k = 4$ or $k = 6$, typically) 个参数放在寄存器里传递, 剩余在储存器传递.

寄存器传参的方法 (4 种):

1) 不给叶过程 (leaf procedure) 分配栈帧. 绝大多数过程都是叶过程, 不分配栈帧能节省很多开销.

叶过程: 不调用其他过程的过程

2) 过程间寄存器分配 (interprocedural register allocation): 先分析代码中全部的函数, 然后再根据分析分配寄存器

3) 若变量 x 不再被使用, 可以直接写其寄存器, 不需要再保存 x 到栈帧中.

4) 寄存器窗口技术 (register windows): 每个函数调用分配一组新的寄存器, 无需内存传输.

4. Frame-Resident Variables

一般来说局部变量和中间结果会放到寄存器中, 以下情况需要将变量储存到栈帧内 (memory):

1) 变量传地址 / 引用 (passed by reference)
2) 被嵌套在当前过程的函数调用 (nested accessed)
3) 太大了放不下 (too big to fit)
4) 变量是数组
5) 有特殊用途的变量 (传参等)
6) 存在过多的临时变量和局部变量 (溢出 spill)

在以下情况, 称变量为逃逸 (escape):

1) 传地址
2) 被取地址
3) 被内层嵌套函数访问

5. Static Link

静态链本质是指向上一层嵌套层级的栈帧的指针. 内层嵌套函数调用外层定义的变量需要用到静态链, 否则无法寻址.

其他访问外层变量的方法:

1) 嵌套层次显示表 (display): 一个全局数组, 位置 i 存放最近一次的, 静态嵌套深度为 i 的过程的栈帧. 是管理静态链的全局数组 (不是栈指针)

2) λ 提升 (lambda shifting): 内层函数访问的外层声明变量, 会作为函数参数传给内层嵌套函数.

注意: 静态链层级是函数的嵌套深度 (函数之间), 不是递归调用的深度 (函数自己), 两者不同概念

VI Translation to Intermediate Code

intermediate represent(中间表示): 抽象的机器语言, 链接前端和后端, 解决了高级语言和目标机器汇编语言之间的转化.

- 前端 (front end): 词法分析, 语法分析, 语义分析, 翻译成中间代码
- 后端 (back end):IR 优化, 翻译成机器语言.

1. Intermediate Representation Trees

1.1 Tree Operator

Expressions(T_exps):

- CONST(i): 整型常数
- NAME(n): 符号常数
- TEMP(t): 临时变量
- BINOP(o, e_1, e_2): 对操作数 e_1, e_2 的二元操作
- MEM(e): 作为 MOVE 操作的左子式时表示对储存器 e 地址的存入; 其他位置表示读取该地址的内容
- CALL(f, l): 过程调用
- ESEQ(s, e): 先计算语句 s 形成副作用, 然后计算 e 违该表达式的值

Statements(T_stm):

- MOVE(TEMP t, e): 计算 e 的值然后存到临时变量 t 中
- MOVE(MEM(e_1),e_2): 计算 e_2 的值然后存入到 e_1 作为地址的内存中
- JUMP(e,$labs$): 跳转到 e 地址或者 $labs$ 为 label 的地址
- CJUMP(o, e_1, e_2, t, f): 依次计算 e_1 和 e_2, 生成值 a, b; 然后用比较运算符操作 aob, 如果结果为 true 跳到 t, 反之跳转到 f;
- SEQ(s_1, s_2): 语句 s_1 后面跟 s_2
- LABEL(n): 定会一名字后的常数值为当前机器代码的地址.

2. Translation into Trees

2.1 Kinds of Expressions

- Ex 代表 expression
- Nx 代表无结果的 statement
- Cx 代表条件分支, 可能跳转到 true label 或 false label.

对于 CJUMP 和 JUMP 语句, 还不知道 label 的具体值, 需要使用两张表:

- 真值标号回填表 (true patch list)

- 假值标号回填表 (false patch list)

2.2 *Variables*

- Simple variables:
  MEM(BINOP(PLUS, TEMP fp, CONST k))
- Following static links:
  MEM(+(CONST k_n, MEM(+(CONST k_{n-1}, ... MEM(+(CONST k_1, TEMP fp))...))))
- Array variables:
  MEM(+(MEM(e), BINOP(MUL, I, CONST W)))

2.3 *Conditionals*

e.g. if $x < 5$ then $a > b$ else 0,
$x < 5$ translates into Cx(s_1), $a > b$ translates into Cx(s_2)
SEQ(s_1(z,f), SEQ(LABEL z, s_2(t,f)))

2.4 *Loops*

- while

```
test:
    if not(condition) goto done
    body
    goto test
done:
```

- for

```
for i:=lo to hi          let var i:=lo
    do body                  var limit:=hi
                         in while i<=limit
                             do(body; i:=i+1)
                         end
```

2.5 *Function Call*

CALL(NAME l_f, $[sl, e_1, e_2, \ldots, e_n]$)
l_f is the label for f, sl is the static link

3. Declarations

变量声明将会在 frame 中额外保留部分空间；函数声明会在 Tree code 中保留一个新的 fragment.

变量的初值会被转换成一个 Tree 表达式，transDec 返回一个 Tr_Exp，这个 Tr_Exp 应当包含完成赋初值的赋值表达式；如果对函数和类型声明施加 transDec，结果将会得到 Ex(CONST(0)) 这样的空操作

函数被翻译为入口处理代码 (prologue), 函数体 (body) 和出口处理函数 (epilogue) 组成的汇编语言代码.

入口包含:

1) 声明一个函数开始的伪指令 (pseudo-instructions)
2) 函数名 label 的定义
3) 调整栈指针的一条指令, 用于分配新的栈帧
4) 将逃逸 (escaping) 参数保存至栈帧的指令, 以及将非逃逸参数传送的新临时寄存器指令
5) 保存此函数用到的 callee-save 寄存器 (包括返回地址寄存器)

本体:

6) 函数体

出口包含:

7) 将返回值传送至专用于返回结果的寄存器
8) 用于恢复 callee-save 的寄存器取数指令
9) 恢复栈指针, 释放栈帧
10) return 指令
11) 声明函数结束的伪指令

VII Basic Blocks and Traces

1. Canonical Trees(规范树)

1.1 Definition

Definition VII.1. *A canonical trees have these properties:*

1) No SEQ or ESEQ

2) The parent of each CALL is either EXP(...) or MOVE(TEMP t, ...).

Why?

1) CJUMP 能够跳转到两个标号的任意一个, 但实际的是条件为假时跳转到下一条

2) ESEQ 会使得子树的不同计算顺序产生不同结果

3) 表达式使用 CALL 会有计算顺序不同的问题

4) CALL 的嵌套调用 (作为另一个 CALL 的参数) 会出问题, 覆盖存放返回值的寄存器的值

重写流程:

1) 一棵树重写成规范树

2) 将树分组合成不含转移和标号的基本块 (basicblock) 集合

3) 对基本块进行排序形成一组轨迹 (trace); 每一个 CJUMP 后就是其 false 标号

1.2 Transformations on ESEQ

Table VII.1: Transformations on ESEQ

Expression	Transforms to
ESEQ(s_1, ESEQ(s_2, e))	ESEQ(SEQ(s_1, s_2), e)
BINOP(op, ESEQ(s, e_1), e_2)	ESEQ(BINOP(op,e_1, e_2))
MEM(ESEQ(s, e_1))	ESEQ(s, MEM(e_1))
JUMP(ESEQ(s, e_1))	SEQ(s, JUMP(e_1))
CJUMP(op, ESEQ(s, e_1), e_2, l_1, l_2)	SEQ(s, CJUMP(op, e_1, e_2, l_1, l_2))
BINOP(op, e_1, ESEQ(s, e_2))	ESEQ(MOVE (TEMP t, e_1), ESEQ(s, BINOP(op, TEMP t, e_2)))
CJUMP(op, e_1, ESEQ(s, e_2), l_1, l_2)	SEQ(MOVE(TEMP t, e_1), SEQ(s, CJUMP(op, TEMP t, e_2, l_1, l_2))).

如果 ESEQ 中 s 和 e_1 是可交换的 (commute), 那么可以直接把 s 和 e_1 交换,ESEQ 提出来.

1.3 Moving CALLs to top level

以 BINOP(op,CALL(),CALL() ...) 为例, 第二个 CALL 会在 BINOP 执行前覆盖第一个 CALL 返回在 RV 寄存器里的值. 解决办法是使用 ESEQ 将返回值保存到一个新的临时变量里: CALL(fun,args) → ESEQ(MOVE(TEMP t, CALL(fun, args)), TEMP t)

2. Taming Conditional Branches

2.1 Basic Blocks(基本块)

取一列规范树, 块的开始是 label, 以跳转指令为结尾. 即:

1) 第一个语句是 LABEL

2) 最后一个语句是 JUMP 或 CJUMP

3) 没有其他的 LABEL,JUMP 或 CJUMP

划分基本块方法: 从头到尾扫描语句序列, 每次发现一个 LABEL 就开始一个新的基本块并结束上一个基本块; 每发现一个 JUMP 或 CJUMP 就结束一个基本块 (并开始下一个基本块). 如果过程还遗留任何基本块不是 JUMP 或 CJUMP 结尾的, 则在基本块块末尾增加一条转移到下一个基本快标号处的 JUMP; 如果有任何基本块不是以 LABEL 开始的, 则生成一个新的标号插入到基本块开始; 在全部末尾添加 done LABEL, 将 JUMP(NAME done) 放到最后一个基本快末尾.

2.2 Trace(轨迹)

程序执行期间可能连贯执行的语句序列. 要寻找一组能够覆盖整个程序的轨迹集合, 且每一个基本块仅出现在一条轨迹中.

Algorithm VII.1 Generation of traces

```
Put all the blocks of the program into a list Q
while Q is not empty do
    Start a new (empty) trace, call it T.
    Remove the head element b from Q.
    while b is not marked do
        Mark b; append b to the end of the current trace T.
        Examine the successors of b (the blocks to which b branches);
        if there is any unmarked successor c then
            b ← c
        end if
    end while
    (All the successors of b are marked.)
    End the current trace T.
end while
```

2.3 Finishing Up

1) 所有后面跟 false 标记的 CJUMP 不变

2) 对任何后面跟 true 标号的 CJUMP, 交换其 true 标号和 false 标号以及判断条件取反

3) 对其后跟随的既不是 true 也不是 false 标号的 CJUMP, 生成新的标号 f' 并重写 CJUMP, 使得其 false 标号紧跟其后.

VIII Instruction Selection

Definition VIII.1 (in-struc-tion)**.** *A code that tells a computer to perform a particular operation*

1. Tree Patterns

在 intermediate representation (Tree) language 中, 一个操作用一个树节点表示. 一条机械指令需要执行一些基本操作, 可以用 IR tree 的一个片段表示, 称为 tree pattern(树模式).

指令选择的任务就是使用树模式的最小集合来覆盖 (tile).

使用 Jouette 体系, 将树模式映射为指令, 如 VIII.1 所示.

Name	*Effect*	*Trees*
—	r_i	TEMP
ADD	$r_i \leftarrow r_j + r_k$	+
MUL	$r_i \leftarrow r_j \times r_k$	*
SUB	$r_i \leftarrow r_j - r_k$	-
DIV	$r_i \leftarrow r_j / r_k$	/
ADDI	$r_i \leftarrow r_j + c$	+ CONST; + CONST; CONST
SUBI	$r_i \leftarrow r_j - c$	- CONST
LOAD	$r_i \leftarrow M[r_j + c]$	MEM + CONST; MEM + CONST; MEM CONST; MEM
STORE	$M[r_j + c] \leftarrow r_i$	MOVE MEM + CONST CONST; MOVE MEM + CONST CONST; MOVE MEM CONST; MOVE MEM
MOVEM	$M[r_j] \leftarrow M[r_i]$	MOVE MEM MEM

Figure VIII.1: Jouette architecture

一棵树可以有多种覆盖方式.

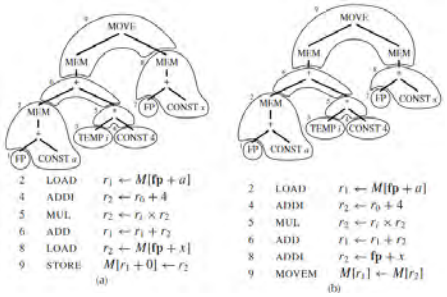


Figure VIII.2: A tree tiled in two ways

1.1 *Optimal and Optimum Tilings*

- optimum tiling (最优覆盖): 覆盖的代价和是最小的覆盖, 全局最优.
- optimal tiling (最佳覆盖): 不存在两个相邻的覆盖能连接成一个代价更小的覆盖, 局部最优.

全局最优属于局部最优.

2. Algorithm for Instruction Selection

2.1 Maximal Munch

最佳覆盖的算法.

从树的根节点开始寻找适合他的最大覆盖 (覆盖的节点数最多, 如果相等的可以任意选择其一), 按照 Jouette 体系结构可能会覆盖其他几个节点; 对遗留的其他子树也进行相同操作.

Maximal Munch 算法的 tiling 是从顶向下的, 但是指令的生成是逆序的 (很好理解, 因为上层的覆盖指令需要下层的指令提供操作数, 所以是逆序).

2.2 Dynamic Programming

可以找到最优覆盖, 子问题是子树的覆盖, 自下而上工作.

会给每个节点计算一个代价, 表示可以覆盖该节点为根的子树的指令序列最优的代价之和.

对于一个节点 n, 先找到其所有子树的代价 c_i, 然后枚举节点 n 所有可能的覆盖, 计算每种覆盖的代价 $c + \sum c_i$, 最后选择代价最小的覆盖.

不同的覆盖代价不同, 具体的代价书中没有给出, 看着默认都是 1, 即目标是覆盖的数量最少.

2.3 Tree Grammar(树文法)

DP 的推广. 使用 brain-damaged Jouette 体系.

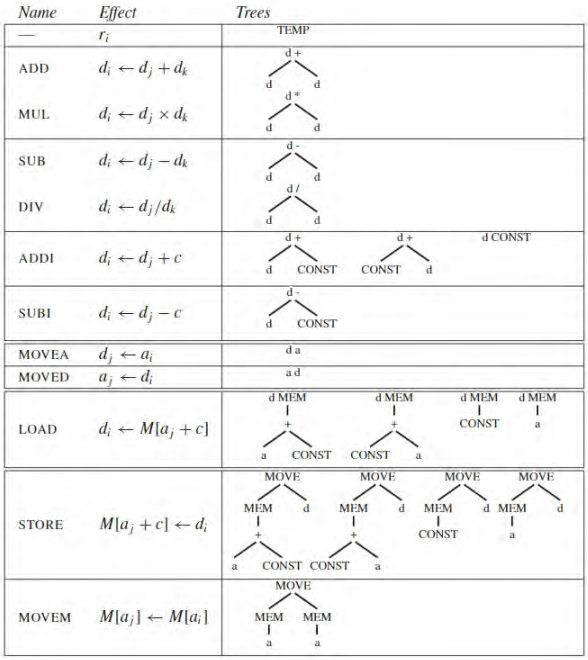


Figure VIII.3: The Schizo-Jouette architecture

使用 CFG 来描述覆盖, 文法具有高度歧义性, 但是 DP 可以很好处理给出最优的覆盖.

有两类寄存器:

- a 寄存器: 存地址;
- d 寄存器: 存数据

$$
\begin{aligned}
d &\rightarrow \text{MEM}(+(a, \text{CONST})) \\
d &\rightarrow \text{MEM}(+(\text{CONST}, a)) \\
d &\rightarrow \text{MEM}(\text{CONST}) \\
d &\rightarrow \text{MEM}(a) \\
d &\rightarrow a \\
a &\rightarrow d
\end{aligned}
$$

Figure VIII.4: The grammar rules for the LOAD , MOVEA and MOVED instructions

2.4 Fast Match

使用 switch-case 来匹配非叶子节点的 label.

2.5 Efficiency of Tiling Algorithms

T 个覆盖, 平均每个匹配的覆盖有 K 个非叶子节点. K' 是在给定子树中为确定匹配那个覆盖需要检查的最大节点个数 (近似于最大覆盖的大小). 假定平均每个树节点可以与 T' 个覆盖匹配. 输入树的节点为 N.

- Maximal Munch: 平均只需要遍历 N/K 个节点, 就可以覆盖整棵树, 所以复杂度为 $O((K' + T')N/K)$
- DP: 需要遍历所有节点的所有覆盖可能, 复杂度为 $O((K' + T')N)$

DP 的其他常数也比 Maximal 大, 因为要遍历两遍. K', K, T 是常数, 线性复杂度.

3. CISC Machines

RISC 机器特征:

1) 32 个寄存器
2) 仅有一类整数/指针寄存器
3) 算数运算仅对寄存器进行操作
4) 采用 “三地址” 指令 ($r_1 \leftarrow r_1 + r_2$)
5) 取指令和村指令只有 M[reg+const] 模式
6) 每条指令长度固定为 32 位
7) 每一条指令产生一个结果或作用, 无副作用

CISC 机器特征:

1) 不多的几个寄存器 (16,8,6)
2) 寄存器分不同类型, 某些操作只能在特定种类的寄存器上进行
3) 算术运算可以通过不同的寻址模式访问寄存器和储存器

4) 指令是 “两地址” 指令
5) 有不同的寻址模式
6) 有由变长操作码加变长寻址模式形成的变长指令
7) 指令具有副作用 (自增寻址方式)

CISC 机器的特点解决难题:

1) 寄存器较少: 不限制生成 TEMP 节点, 假设寄存器分配能完成分配工作
2) 寄存器分类: 将操作数显示地传送到相应的寄存器中
3) 两地址指令: 增加一条额外的传送指令
4) 算数运算可以访问存储器: 指令选择阶段将每一个 TEMP 节点转化成一个寄存器引用.
5) 若干种寻址模式: 优点 (破坏寄存器少; 指令代码短)
6) 变长指令: 不管;
7) 副作用指令

三种解决办法:

1) 忽略地址自增指令, 希望其自动消失
2) 在采取树型匹配的代码生成器的上下文中使用特别方式匹配特殊的习惯用法
3) 使用完全不同的指令算法, 基于 DAG 样式.

IX Liveness(活跃性) Analysis

1. Definition

编译器需要分析程序的中间表示, 以确定那些临时变量在同时被使用. 如果一个变量的值在将来还需要使用, 则变量是活跃的 (live), 这种分析叫做活跃分析.

控制流图 (control flow graph): 程序的每条语句都是流图的节点.

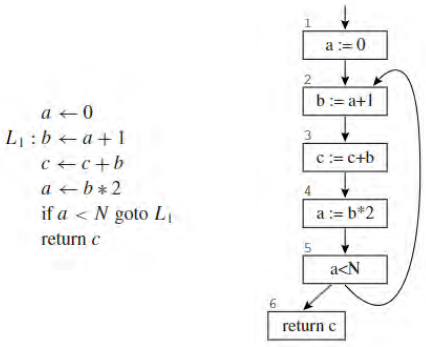


Figure IX.1: Control-flow graph of a program

活跃范围 (Liveness): 变量位索引的集合, 变量在那几条边上活跃的边集合.

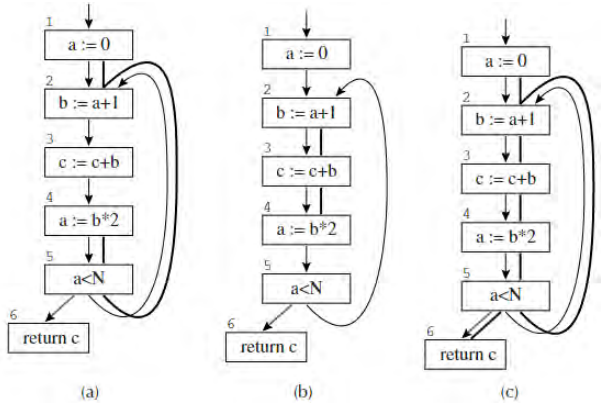


Figure IX.2: Liveness of variables a, b, c.

2. Solution of Dataflow Equations

- 出边 (out-edge): 节点引向后继节点的边;
- 入边 (in-edge): 由前继节点指向的边.

$succ[n]$ 是节点 n 的后继节点; $pred[n]$ 是节点 n 的前驱节点

- 定义 (define): 对变量和临时变量的赋值称为变量的定义;
- 使用 (use): 使用出现在赋值号右边的变量.

活跃性 (Liveness): 变量在边上活跃是指存在一条边通向一个使用该变量的有向路径, 且不经过该变量的任何定义.

- 如果变量在一个节点的所有入边上全是活跃的, 那么该变量是入口活跃的 (live-in);
- 若一个变量在一个节点的所有出边上都是活跃的, 那么该变量在该节点是出口活跃的 (live-out).

2.1 Calculation of Liveness

就是计算流图每一个节点的 in 和 out 集合

$$\begin{aligned} in[n] &= use[n] \cup (out[n] - def[n]) \\ out[n] &= \bigcup_{s \in succ[n]} in[s] \end{aligned} \tag{IX.1}$$

活跃性计算的迭代方法

Algorithm IX.1 Computation of liveness by iteration

for each n **do**
  $in[n] \leftarrow \{\}$; $out[n] \leftarrow \{\}$
end for
repeat
  for each n **do**
    $in'[n] \leftarrow in[n]$; $out'[n] \leftarrow out[n]$
    $in[n] \leftarrow use[n] \cup (out[n] - def[n])$
    $out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$
  end for
until $in'[n] = in[n]$ and $out'[n] = out[n]$ for all n

			1st		2nd		3rd		4th		5th		6th		7th	
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1		a				a		a		ac	c	ac	c	ac	c	ac
2	a	b	a		a	bc	ac	bc	ac	bc	ac	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	c	bc	b	bc	bc	bc	bc
4	b	a	b		b	a	b	a	b	ac	bc	ac	bc	ac	bc	ac
5	a		a	a	a	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac
6	c		c		c		c		c		c		c		c	

Figure IX.3: Liveness calculation following forward control-flow edges.

			1st		2nd		3rd	
	use	*def*	*out*	*in*	*out*	*in*	*out*	*in*
6	c			c		c		c
5	a		c	ac	ac	ac	ac	ac
4	b	a	ac	bc	ac	bc	ac	bc
3	bc	c	bc	bc	bc	bc	bc	bc
2	a	b	bc	ac	bc	ac	bc	ac
1		a	ac	c	ac	c	ac	c

Figure IX.4: Liveness calculation following reverse control-flow edges.

适当排序可以显著加快算法的收敛过程, 一般要从程序末尾往前算, 先算 out 再算 in, 可以显著提高速度和正确率. 信息活跃性是沿控制流箭头的反方向流动的, 计算顺序同理.

时间复杂度:

- for 循环初始化节点 in, out 需要 $O(N^2)$;
- repeat 循环的时间复杂度是 $O(N^4)$.

由于活跃信息大部分稀疏, 实际运行时间在 $O(N)$ 和 $O(N^2)$ 之间.

2.2 Representation of Sets

- 位数组 (bit array): 程序中有 N 个变量, 用 N 位二进制数表示集合
  求并集对位数组求按位或
  时间效率: 对每个字有 K 位的计算机, 并运算需要 N/K 次操作
- 有序变量表: 链表的成员是组成集合的元素
  并集通过合并链表实现
  时间开销和求并集的集合大小成线性.

集合稀疏 (平均少于 N/K) 用有序链表表示速度会更快 (越稀疏越快); 集合密集位数组表示更好.

2.3 Least Fixed Points

Equation IX.1 的解只是保守的近似解, 只能保证生成的代码是正确的, 但是所使用的寄存器可能比实际需要的多.

Theorem IX.1. ***Equation*** *IX.1* *有一个以上的解.*

Proof. 增加更多的变量后, 结果仍是 **Equation** IX.1 的解. Q.E.D.

Theorem IX.2. ***Equation*** *IX.1* *的所有解都包含最小解 (least solution).*

Algorithm IX.1 所计算的就是最小解.

2.4 Static v.s. Dynamic Liveness

Theorem IX.3 (Halting Problem)**.** *不存在程序 H, 其以任意程序 P 和输入 X 作为自己的输入. 当 $P(X)$ 停止时返回真, 否则返回假.*

Corollary IX.4. *不存在程序 $H'(X, L)$, 对任何程序 X 和其中标记 L, 可以判断 X 在执行中是否到达过 L.*

意思是不存在一个通用程序能算出每一个标记是否达到. 动态判断及其静态的近似:

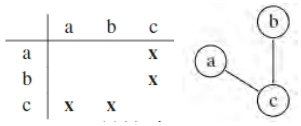
- 动态活跃 (Dynamic liveness): 如果程序从节点 n 执行到使用 a 之间没有经过 a 的任何定义, 那么变量 a 在节点 n 是动态活跃的.
- 静态活跃 (Static liveness): 如果存在着一条从 n 到使用 a 的控制流路径, 且此路径上没有 a 的任何定义, 那么变量 a 在节点 n 静态活跃的.

2.5 Interference Graphs

Definition IX.5. 两个临时变量不能分配到同一个寄存器的情况称为冲突 *(interference)*.

冲突原因:

1) 临时变量在程序的同一点同时活跃
2) 某些寄存器必须被使用时, 临时变量不能占用这些寄存器.



(a) Matrix (b) Graph

Figure IX.5: Representations of interference

绘制冲突图的办法: 为新定义 (def) 添加冲突边, 即

1) 对变量 a 非 MOVE 指令的定义, 以及在该指令节点 n 处, 任意 $b_i \in out[n]$, 添加冲突边 (a, b_i)
2) 对于节点标号为 n 的 MOVE 指令 $a \leftarrow c$, 对 $b_i \in out[n]$ 且 $b_i \neq c$, 添加边 (a, b_i).

注: 可以给 (a, c) 画上虚线, 便于寄存器分配的 coalesce.

X Register Allocation

通过染色算法对冲突图求解.

1. Coloring by Simplification

寄存器分配与图染色都是 NP-complete 问题. 这里使用一种线性近似算法, 分为构建, 简化, 溢出, 选择四个阶段.

1) 构建: 构建冲突图.
2) 简化: 启发式图染色. 将图中度数 $< K$ (寄存器个数) 的节点删除, 并压入一个栈中. 直到剩下节点度数都 $\geq K$, 无法继续简化.
3) 溢出: 称目前剩下为高度数 (significant degree, 度 $\geq K$) 点. 选择与剩下其他节点无冲突的, 代表临时变量的点作为溢出, 将其删除并标记为潜在溢出 (potential spill) 压入栈中. 然后继续简化.
4) 选择: 从空图开始, 将点从栈中弹出, 加入到图中并染色. 每次加入节点, 其必须被染色.

若为潜在溢出的点, 其可能无法被染色, 则将其标记为实际溢出 (actual spill), 然后不管其继续弹出节点.

但潜在溢出的点也可能被成功染色 (其周围有节点颜色相同), 此时称为乐观染色 (optimistic coloring).

5) 重开: 对实际溢出的节点, 重写程序通过读写内存对其进行操作. 但就算如此, 读写也需要额外几个临时寄存器, 此时对于重写的程序, 重新构建冲突图, 跑染色算法. 多次迭代直到简化阶段不会溢出, 一般而言, 一两次迭代即可.

假设 a 发生了实际溢出, 则 a 必须通过读写内存对其进行操作. 对其 def 与 use 的修改如下:

1) 对于 a 的每个 use, 新建 a_i, 从 $M[a_{loc}]$ 读取 a.
   $c \leftarrow a \Rightarrow a_1 \leftarrow M[a_{loc}], c \leftarrow a_1$
2) 对于 a 的每个 def, 新建 a_i, 通过其写入 $M[a_{loc}]$.
   $a \leftarrow c \Rightarrow a_2 \leftarrow c, M[a_{loc}] \leftarrow a_2$

Example X.1. 假设有四个寄存器.

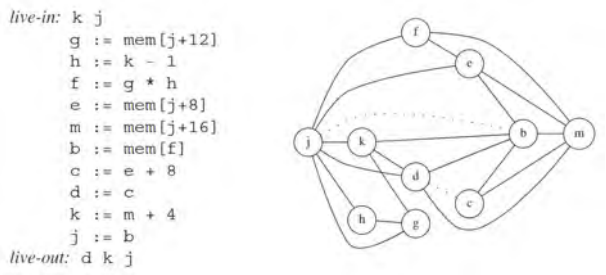


Figure X.1: Interference graph for a program

Figure X.1 简化就可以删除全部节点了. 然后依次弹出并染色.

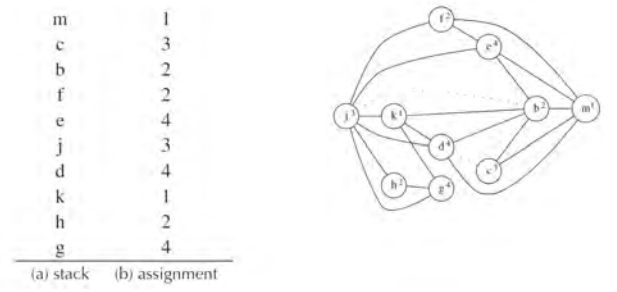


Figure X.2: Simplification stack, and a possible coloring

2. Coalescing

删除冗余的 MOVE 指令. 若 MOVE 的 src 与 dst 之间不冲突, 则将其合并为新节点, 与新节点冲突的点是二者的并集. 理论上, 任意一对不冲突的点都可以被合并. 但合并可能导致原本可被染色图不能被染色.

所以引出安全的合并策略:

1) Briggs: 如果 a, b 合并后的节点 ab 的高度数邻居个数 $< K$, 则 a, b 可以合并.

2) George: 对于 a 的每个邻居 t, 若 t 与 b 冲突, 或者 t 是低度数点 (度 $< K$), 则 a, b 可以合并.

这两种策略都是保守的 (conservative), 因为其合并不会改变图的着色性. 策略执行后可能仍有多余的 MOVE 指令, 但这至少比溢出好.

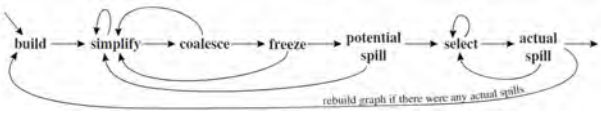


Figure X.3: Graph coloring with coalescing

带合并的图染色算法:

1) 构建: 构建冲突图, 将节点分类为 move-related 与 non-move-related. move-related 节点指此点是 MOVE 的 src 或 dst.

2) 简化: 删除低度数 ($< K$) 的 non-move-related 节点, 并压入栈中. 直到无法简化.

3) 合并: 运行保守的合并策略, 每次合并两个节点 (删除相关 MOVE 指令), 若结果是 non-move-related, 则继续简化. 重复简化合并直到剩下的都是高度数或 move-related 节点.

4) 冻结: 寻找一个度数较低的 move-related 节点, 放弃对其相关 MOVE 指令的合并, 将此 MOVE 指令相关的点标记为 non-move-related. 然后继续简化合并.

5) 溢出: 若没有低度数节点, 选择高度数节点作为潜在溢出并删除压入栈中.

6) 选择: 将点从栈中弹出, 加入到图中并染色.

Example X.2. 仍是四个寄存器, 以 **Figure** X.1 为例.

只有 b, c, d, j 四个点是 move-related 的. 简化 g, h, k. 然后合并 cd, bj. 最后完成所有的简化. 不需要溢出.

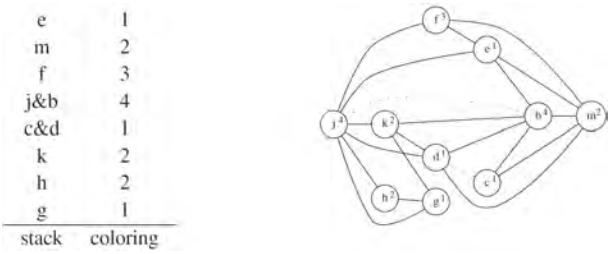


Figure X.4: A coloring, with coalescing

若冲突图中两个节点既有虚线又有实现, 则称之为受约束的 (constrained). 对此不将其考虑为 move-related.

3. Precolored Nodes

有一些临时变量是预着色的, 代表的是机器寄存器, 参数寄存器, 帧指针, 返回值寄存器等. 预着色的点必须相互冲突, 一般来说, 一个颜色只会对一个节点预着色.

选择与合并操作可以给一个普通点被预着色的颜色, 只要没有冲突. 特别的, 当一个预着色节点与普通节点使用 George 合并时, 需要将普通节点作为 a, 检查普通节点的邻居是否满足合并条件.

预着色节点特性:

1) 无法简化
2) 无法溢出 (认为此点的度无限大)
3) 可以参与合并.

染色算法通过简化合并溢出来工作, 直到只剩下预着色节点, 然后才能开始向冲突图中加入其他节点. 预着色节点不能溢出所以前端必须使他们的活跃范围保持较小, 可以通过 MOVE 指令来实现.

对于节点 a, 其溢出优先级计算公式:

$$Priority = \frac{Out_{use+def} + 10 \times In_{use+def}}{D}$$

其中,

- $Out_{use+def}$ 为 a 在循环外的 use 与 def 总数.
- $In_{use+def}$ 为 a 在循环内的 use 与 def 总数.
- D 为 a 的度, 只包含实线.

优先级值越小, 说明优先级越高. 表示优先溢出不被经常使用的高度数节点.

XI Garbage Collection

Definition XI.1 (Garbage)**.** *Heap-allocated records that are not reachable by any chain of pointers from program variables are garbage.*

不存在一个通用程序能算出每一个变量是否活跃, 所以使用一个保守的近似: 要求编译器保证所有活跃记录是可达的 (reachable), 并最小化可达的不活跃记录.

1. Mark-and-sweep Collection

程序变量与堆分配记录可以构成一张有向图. 变量就是图的一系列根.

Definition XI.2 (reachable)**.** *A node n is reachable if there is a path of directed edges $r \to \cdots \to n$ starting at some root r.*

算法原理:

1) 标记: 使用 DFS 标记所有可达节点
2) 清扫: 未被标记的节点都是垃圾, 从头到尾扫描堆内存, 将未标记的节点加到空闲表 (freelist) 中, 同时清除已标记节点的标记.

1.1 Cost of garbage collection

假设大小为 H 个字的堆中, 有 R 个字可达, 则一次垃圾回收的代价是 $c_1R + c_2H$, 其中 c_1, c_2 为常数.

最终每个垃圾的均摊回收代价是

$$\frac{c_1R + c_2H}{H - R}$$

若 $\frac{R}{H} > 0.5$, 回收器应该向 OS 申请更大的内存, 以增大 H.

1.2 Optimization

对算法的空间与时间常数进行优化.

- Using an explicit stack: 将 DFS 的递归改写为循环. 因为递归的 DFS 其运行时的栈可能超过总的堆大小, 所以用显式的栈代替使用堆.
- Pointer reversal: 压栈时使用 $x.f_i$ 指向其父节点, 弹栈时再恢复 $x.f_i$ 原本的值, 这样连显示的栈都不需要. 只需要额外的 *done* 数组作为 x 子域的下标.

DFS 中, x 是当前节点, t 是父节点, y 是子节点, $done[x]$ 用于索引 $x.f_{done[x]}$ 作为下一个子域.

1.3 An Array of Freelists

$freelists$ 使用空闲域的大小作为索引, 存储邻接链表, 方便之后使用. 比如想要大小为 i 的域, 把 $freelists[i]$ 的头取出即可. 也可以取出 $freelists[j](j > i)$, 然后把剩下的空闲域插入 $freelists[j - i]$.

```
function DFS(x)
 if x is a pointer and record x is not marked
  t ← nil
  mark x;  done[x] ← 0
  while true
       i ← done[x]
       if i  <  # of fields in record x
         y ← x.f_i
         if y is a pointer and record y is not marked
           x.f_i ← t;  t ← x;  x ← y
           mark x;  done[x] ← 0
         else
           done[x] ← i + 1
       else
         y ← x;  x ← t
         if x = nil then return
         i ← done[x]
         t ← x.f_i;  x.f_i ← y
         done[x] ← i + 1
```

Figure XI.1: DFS

1.4 Fragment

- 外部碎片 (external fragmentation): 想分配一个 n 大小的空间, 但是空闲空间均小于 n.
- 内部碎片 (internal fragmentation): 实际使用大小为 n , 却分配了大小为 $K(K > n)$ 的空间, 未使用的空间在记录内而不是在空闲表中.

2. Reference Counts

算法原理: 记住每个记录有多少指针指向其, 计数和每个记录储存在一起.

- 当 p 写入 $x.f_i$ 时, p 的引用数要增加, $x.f_i$ 之前内容的引用数要减少.
- 若 r 的引用数为 0, 将 r 加入到 $freelist$ 中, 并减少 r 指向的其他记录的引用数.

有一个改进是在将 r 从 $freelist$ 中删除时, 再递归地减少 $r.f_i$ 的计数:

1) 能将递归减少的动作分解为较短的操作, 使程序的运行更加平滑 (对对交互式程序或实时程序好)
2) 递归减少动作只需要在分配 r 时进行.

但有两个大问题:

1) 无法回收成环的垃圾.
2) 增加引用计数所需的操作代价很大.

解决 “环” 的办法:

1) 让程序员手动断开所有循环引用, 这比手动管理内存要简单, 但也并不优雅.
2) 将标记清理 (快速, 无中断地回收) 和引用计数 (回收环) 相结合.

引用计数的缺点大于优点, 所以极少使用.

3. Copying Collection

1) 将堆分为两块区域 from-space 和 to-space; 当前的内存放到 from-space 中

2) 在 to-space 构造一个同构的副本, 副本空间利用上更紧凑: 占据连续的不含碎片的储存单元

3) from-space 和 to-space 互换, 然后抛弃 from-space.

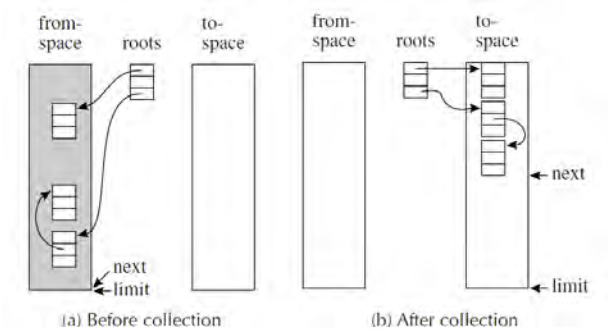


Figure XI.2: Copying collection

算法流程:

1) 回收初始化: 初始化指针 next 指向 to-space 的开始. 每当 from-space 发现一个可达记录, 便把它复制到 to-space 的 next 所指位置, 同时使 next 增加该记录的大小.

2) 转递 (forwarding): 使一个指向 from-space 的指针 p 转而指向 to-space. 有三种情况:

a. 若 p 指向的 from-space 已经被复制, 则 $p.f_1$ 是一个特殊的转递指针 (forwarding pointer) 指向了副本所在. 因为只有这种指针指向了 to-space, 可以很容易地分辨.

b. 若 p 指向的 from-space 未被复制, 则将其复制到 next 所在位置, 然后将 $p.f_1$ 变为转递指针. 因为其已经被复制了, 所以可以覆写 f_1.

c. 若 p 不是指针或指向了 from-space 之外, 不做任何事.

3.1 Cheney's algorithm

使用 BFS 遍历可达数据.

- 在 to-space 上, 位于 scan 与 next 之间包含了已经被复制到 to-sapce, 但是其子域还没有转递的记录, 他们的子域仍指向着 from-space.
- 在 scan 之前包含了被复制且被转递的记录, 其中所有指针都指向 to-space.

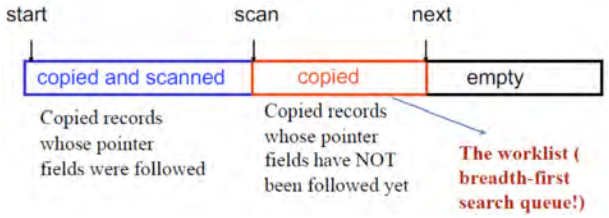


Figure XI.3: Cheney's algorithm

但其空间局部性不好, x 中的指针 $x.f_i$ 指向的对象可能地址离 x 很远, 这导致无法很好的利用虚拟内存, 也会造成许多 cache misses. DFS 的局部性很好, 但需要各种复杂的常数优化. 所以结合 BFS 与 DFS, 用 BFS 寻找需要复制的记录, 复制时使用 DFS 复制.

3.2 Cost of garbage collection

假设大小为 H 个字的堆中, 有 R 个字可达, 因为 H 被分为了两部分, 所以只需要回收 $\frac{H}{2} - R$ 的垃圾. 对每个垃圾的均摊回收代价为

$$\frac{c_3 R}{\frac{H}{2} - R}$$

其中 c_3 是常数.

4. Interface to the Compiler

4.1 Fast Allocation

使用复制回收 (copying collection) 的方式回收垃圾, 以便分配空间是一个连续的空闲区域. 区域的末端是 limit, next 指向下一个空闲单元.

分配大小为 N 的记录的步骤如下:

1) 调用分配函数
2) 若 $next + N < limit$ 则继续, 否则调用垃圾回收.
3) 将 $next$ 复制到 $result$
4) 清理 $M[next], M[next + 1], \ldots, M[next + N - 1]$
5) $next \leftarrow next + N$
6) 从分配函数返回.

a. 将 $result$ 赋值到计算上有用的地方.

b. 将要用到的值储存到该记录

使用内联展开 (inline expanding, 将函数调用替换为函数本体) 可以省略 1) 和 6); 与 a. 结合可以消除 3); b. 也可以消除 4); 2), 5) 不能被消除. 但若是多个分配可以被合并优化. 至此, 分配就减少到了大约 4 条指令.

4.2 Describing Data Layouts

垃圾回收器需要对任意数据类型进行操作, 所以需要确定每条记录的子域类型及数量. 简单的方法是用每个对象指针

的第一个字指向一个描述记录 (descripter record), 其包含对象总大小, 以及每个域指针的位置.

垃圾回收器需要知道哪些内存地址是指针, 才能正确地跟踪对象间的引用关系, 避免错误地回收正在使用的对象. 所以编译器需要生成指针映射 (pointer map) 来描述程序运行过程中每个时刻哪些位置存储着指针.

具体步骤:

1) 识别指针: 编译器需要识别出所有可能包含指针的变量, 包括堆记录, 临时变量和局部变量, 无论它们存储在寄存器还是活动记录 (activation record) 中.

2) 指针映射的时机: 由于活跃指针的集合在每条指令执行后都可能发生变化, 所以为每条指令都生成指针映射是不现实的. 因此, 编译器只在垃圾回收可能开始的点生成指针映射, 这些点包括:

- 调用 alloc 函数的地方, 因为分配新内存可能会触发垃圾回收.
- 每个函数调用的地方, 因为被调用的函数可能直接或间接地调用 alloc 函数.

3) 指针映射的组织: 指针映射使用函数返回地址作为键, 因为返回地址是垃圾回收器在下一个活动记录中看到的内容. 每个返回地址对应一个活跃指针集合, 描述了在该返回地址处哪些寄存器或栈帧位置存储着指针.

4) 垃圾回收过程: 垃圾回收器从栈顶开始向下扫描, 逐帧查找活跃指针.

- 每个返回地址都指向一个指针映射条目, 该条目描述了下一帧的信息.
- 在每一帧中, 垃圾回收器根据指针映射标记/转递 (取决于垃圾回收方式) 该帧中的所有活跃指针.

5) callee-save 寄存器: 对于 callee-save 寄存器, 需要特殊处理.

例如, 函数 f 调用 g, g 调用 h. 函数 h 知道它将一些 callee-save 寄存器保存在其栈帧中, 并在其指针映射中记录了这一事实, 但 h 不知道哪些寄存器是指针. 因此, g 的指针映射必须描述在调用 h 时, 哪些 callee-save 寄存器包含指针, 哪些是从 f "继承" 下来的.

4.3 Derived Pointers

派生指针是指那些不直接指向堆记录起始地址的指针, 它们可能指向记录的中间、之前或之后.

比如说, 考虑表达式 $a[i-2000]$, 它可以被编译器优化为:

$$
\begin{aligned}
t_1 &\leftarrow a - 2000 \\
t_2 &\leftarrow t_1 + i \\
t_3 &\leftarrow M[t_2]
\end{aligned}
$$

其中, a 是一个指向数组的指针, t_1 就是一个派生指针, 它指向 a 所指向数组的前 2000 个元素之前的位置.

如果在循环中使用 $a[i-2000]$, 编译器可能会将 $t_1 \leftarrow a - 2000$ 提升到循环外, 以避免重复计算. 但它不指向对象的起始地址, 甚至可能指向不相关的对象. 垃圾回收器可能会被迷惑.

为了解决这个问题, 编译器需要在指针映射中标识出每个派生指针, 并记录它所依赖的基指针 (base pointer). 当垃圾回收器将基指针 a 移动到新地址 a' 时, 它需要根据基指针的移动量来调整派生指针 t_1 的值, 即将 t_1 更新为 $t_1 + a' - a$.

由于派生指针依赖于基指针, 所以在编译器的活跃性分析中, 派生指针会隐式地保持其基指针的活跃状态. 也就是说, 只要派生指针还活跃, 它的基指针也必须被视为活跃的, 不能被回收.

XII Object-Oriented Languages

面向对象语言 = object-oriented language = OO language = class-based language

- (几乎) 所有值都是对象 (object)
- 对象属于某个类, 或者说对象是某个类的实例 (instance)
- 对象封装了状态 (也就是成员变量 fields) 与行为 (也就是成员方法 methods)

重要概念:

- 继承 (inheritance): 派生类继承基类的特性
- 封装 (encapsulation): 隐藏不该被外部接触到的接口
- 多态 (polymorphism): 对象可以以不同形态呈现

1. Classes

拓展 Tiger 以支持对象. 不考大题, 就不详细记了.
在 Tiger 中实现类需要解决如下问题:

- Field layout: 某个类的各个 fields 在内存中如何放置、访问? 换句话说, 如何确定某个实例的 field 内存地址相对这个实例初始地址的偏移量?
- Method dispatch: 调用某个实例的方法时, 如何找到正确的方法位于何地址? 是派生类的实现, 还是基类的实现, 还是基类的基类的实现?
- Membership test: 如何检查给定实例是否是给定类的实例?

2. Single Inheritance of Data Fields

单继承 single-inheritance(SI): 每个类最多只能继承一个基类, 因此继承关系图是一棵树.

2.1 Field layout

使用 *prefixing

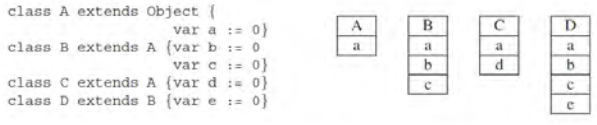


Figure XII.1: Single inheritance of data fields.

派生类新增的 fields 跟在基类的后面. 这使得我们可以正确处理多态: 例如把 Cat 当作 Animal 看待, 访问 Animal 的 fields 时相当于屏蔽了新增的 fields; 基类的各个 fields 偏移量不会因为这是个派生类实例发生变化, 仍然是已知的. 这避免了不安全的内存访问.

2.2 Method dispatch

每个 method 编译成一段代码 (称为 method instance), 编译方式和普通函数几乎无异. 比如 A 中的 $f()$ 就编译为一段代码, 可用 A_f 这样 label 标记函数地址. 机器码中, 函数起始地址使用一个 LABEL 标出.

每个类都对应一个 class decriptor, 里面包含了描述这个类的一些必要信息:

- 一个指向基类的指针
- 一个列表, 包含这个类所有的 method instances

对于 static method 的调用 $x.f()$, 编译器将会:

1) 找到对象 x 对应的类, 记为 C.
2) 如果 C 中有 f, 则直接得出 $x.f()$ 翻译结果为 C_f; 否则继续向上 (在基类中) 寻找.
3) 假设 C 的基类为 B, 在 B 中查找 f, 如找到则得出 $x.f()$ 翻译结果为 B_f; 否则继续向上 (在基类中) 寻找.
4) ...
5) 直到在某个祖先中找到为止 (或是一路找到 Object 还没有则报错), 调用它.

对于 dynamic method 的处理略复杂:

- 每个类维护一个 dispatch vector (例如 C++ 中的虚表 vtable = virtual table = VMT = virtual methods table) 储存每个 method 的地址.

  建立方式类似 prefixing: 派生类中新声明的方法跟在基类 dispatch vector 的后面; 不过如果有基类的 method 被重写了, 也要替换成自己重写后 method 的地址. 这样, 每个方法的偏移量是确定的.
- 每个对象都关联某个 vtable: 对象的开头储存一个指针指向对应 class descriptor , 里面就有 vtable.
- 需要动态查找 (lookup), 有额外的开销.

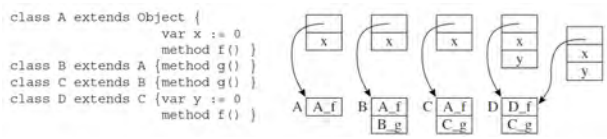


Figure XII.2: Class descriptors for dynamic method lookup

对于 dynamic method 的调用 $x.f()$, 编译器将会:

1) 在 x 的 0 偏移处 (开头) 找到 class descriptor d.
2) 由于方法 f 的偏移量是确定的 (记为 F), 从 d 中 F 偏移处获取 f 的函数地址.
3) 调用 f.

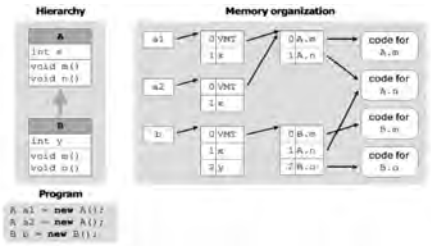


Figure XII.3: dynamic method

3. Multiple Inheritance of Data Fields

多继承 multiple-inheritance(MI): 每个类可以继承多个基类, 因此继承关系图是有向无环图 (DAG).

```
class A { int x; }
class B1 extends A { … }
class B2 extends A { … }
class C extends B1, B2 { … }
```

Figure XII.4: 菱形继承

引入了经典的菱形继承问题:

- 歧义: 如果 $B1$ 和 $B2$ 中都有 method m, 那么 C 的实例 c 上调用 $c.m()$ 时应该调用哪个 m? 无法确定.
- field replication: 对于 A 中的一个 field x, 由于 $B1$ 和 $B2$ 中都继承了 $A.x$, 最终 C 中会有重复的两个 x 都来自 A.

3.1 Field layout

通过图染色获取.

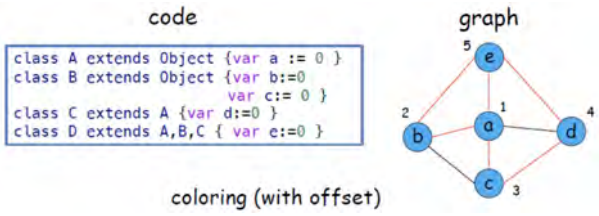


Figure XII.5: Multiple inheritance

目标: 静态分析所有类, 为每个 field 的找到一个固定偏移量. 如果不同 field 在同一个类中出现, 则不能共享同一个偏移量.

- 节点: 不同的 field 名
- 边: 同时在某个类中出现的 field 则连边
- 颜色: 最终偏移量 $(0, 1, 2, \cdots)$

例如从 **Figure** XII.5 将得到 **Figure** XII.6.

优化: 可以看出存在许多空的 slot 被浪费了. 我们可以把 fields 在内存上合并, 转而在每个类的 class descriptors 中记录各个 field 的真实偏移量. 如 **Figure** XII.7 所示.

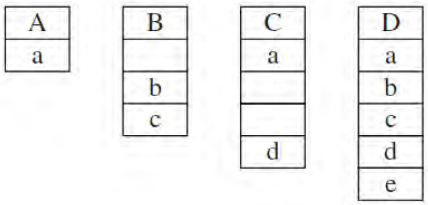


Figure XII.6: Multiple inheritance of data fields.

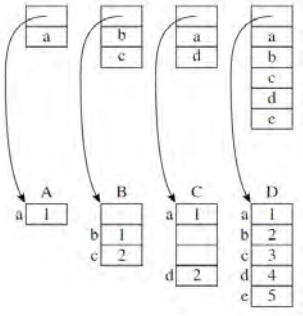


Figure XII.7: Field offsets in descriptors

由于类的数量远少于对象 (类的实例) 数量, 所以这样能节省空间. 不过这种优化导致每个 field 的具体偏移不固定了, 因此需要在运行时在 class descriptor 中动态查找 (lookup) field 的真实偏移量.

3.2 Method dispatch

仍然使用图染色. 直接把 method 名混合进上述的图中, 一起染色; 也就是不仅记录 field 的偏移量, 也记录 method 的地址. 也有动态查找的开销.

不过, 并非所有时候都可以静态知晓所有类的存在并统筹规划.

解决方案: Hashing. 其实就是又包了一层新表, 允许通过 field/method 的名字本身索引到 field offset 或 method address. 更简单地说, 原先是 `OffsetOrAddr[]` , 现在是 `hashmap<Name, OffsetOrAddr>` .

- Ftab(field table): field offset 或 method address (之前就有的)
- Ktab(key-table): 记录注册过的名字 (因为有哈希冲突的问题, 需要确认名字是否真的匹配)

若要在 object c 获取 field b, 编译器会:

1) 在 c 的 0 偏移处 (开头) 找到 class descriptor d.
2) 从偏移量 $d + Ktab + hash_b$ 获取函数名 f
3) 对比 f 是否与 b 相同
4) 从偏移量 $d + Ftab + hash_b$ 获取 field b

5) 获取 field b 的内容.

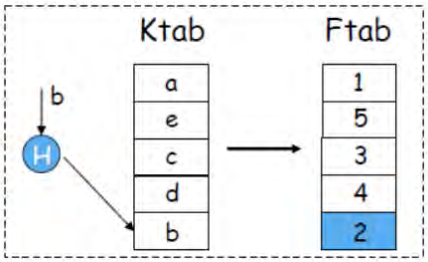


Figure XII.8: class descriptor

4. Testing Class Membership

每个类都是一种类型. 派生类可以看作一种 sub-type. 在进行类型转换时:

- upcast: 派生类转为基类. 永远是安全的. 在这种转换中相当于“丢失”了一些信息, upcast 后能调用/访问的 method/field 变少了.
- downcast: 反之, 基类转为派生类不一定安全. 冒然转换就可能通过错误的 offset 访问到错误的、越界的内存, 很危险.

Membership test: 为了安全的类型转换, 需要测试某个对象是否是某个类的实例.

如何判断 x 是不是类 C 的实例?

朴素 (慢) 的做法: 递归地检查 x 的 class descriptor (记作 $x.0$) 开始的继承链 $x.0$, $x.0.super$, $x.0.super.super$, $x.0.super.super.super$, ... 如果发现某个是 C 的 class descriptor 则 x 是 C 的实例; 否则如果到最上层 ($...super == NIL$) 仍未发现, 则说明不是 C 的实例.

更快的做法: display. 每个 class descriptor 储存一个 display, 也就是一个足够长 (比最长继承链长) 的定长列表, 记录对象的整条继承链. 就像:

```
0: Object
1: GrandparentClass
2: ParentClass
3: MeClass
4: (nil)
5: (nil)
...
```

我们可以给每个类一个专属的数字 ID (例如对于 SI, 可以按照继承关系树的 BFS 序为每个类编号), 然后 display 中储存这些 ID 代表类. 由于对每个类, 其继承关系的嵌套深度在编译期已知, 因此可以立刻找到需要比较 display 中的哪一项. 例如, 假设 MeClass 的继承深度是第 3 层, 要检查 x 是否为 MeClass 的实例, 只需检查 x 的继承深度是否大于等于 3, 且 $x.0.display[3]$ 是否指向 MeClass 的 class descriptor 即可.

5. Private Fields and Methods

private field/method: 私有的 field/method 只能被类的其他 method 访问/调用, 而不能被外部调用. 这是封装思想的体现: 调用者不该知道内部实现细节. 通过类型检查确保私密性 (privacy): 每个访问/调用处检查是否 private.

XIII Loop Optimizations

1. Loop in CFG

循环在控制流图 (CFG) 体现为一个节点集合 S, 包含 header node h, 并且对于 S 中的任何节点 x:

- 都有一条从 x 到 h 的路径
- 都有一条从 h 到 x 的路径
- 除了 h, 没有任何其他 S 以外的节点能到达 x

1.1 Dominator tree

重要概念:

- Loop entry: h 是唯一一个能从外部到达的节点, 是循环的唯一入口
- Loop exit: 可以有多个节点能跳出循环 (i.e. 后继节点不都属于 S)
- Predecessor: pred, 前驱节点
- Successor: succ, 后继节点
- Dominator: 如果 CFG 的入口节点 s_0 到节点 n 的所有路径都经过节点 d, 我们就称 d 是 n 的支配节点 (dominator), 记作 $d \operatorname{dom} n$.

  每个节点都支配 (dominate) 自己

  节点可以有多个 dominators

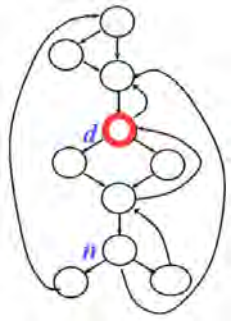


Figure XIII.1: dominator

求解每个节点的 dominators $D[n]$:

– 入口节点的唯一 dominator 就是自己: $D[s_0] = \{s_0\}$

– 对于其它节点:

$$D[n] = n \cup \left(\bigcap_{p \in pred[n]} D[p] \right) \text{ for } n \neq s_0$$

– 求解过程: 一开始时令 $D[s_0] = \{s_0\}$, 其余节点 $D[x] =$ 所有节点; 然后使用上面的式子不断迭代更新直到不动点.

- Immediate dominator: 直接支配节点, 支配 n 的节点中距离 n 最近 (但不是自己) 的节点.

  也就是从入口结点到达 n 的任何路径 (不含 n) 中, 路径中最后一个支配 n 的结点.

  n 的直接支配节点记作 $idom(n)$.

  除了初始节点 s_0 以外, 每个节点有且仅有一个直接支配节点. 可以证明, 如果 d 和 e 都支配 n, 那么要么 d 支配 e, 要么 e 支配 d. 因此对于某个节点 n 的支配节点集合 $D[n]$, $D[n]$ 中的 dominators 上有全序关系. 而 $idom(n)$ 就是 "最小" 的那个 dominator: 不支配 $D[n]$ 中任何其他的 dominator.
- Dominator tree: 支配节点树

  对于每个节点 n, 连接边: $idom(n) \to n$.

  每个节点支配以自己为根的子树中的所有节点

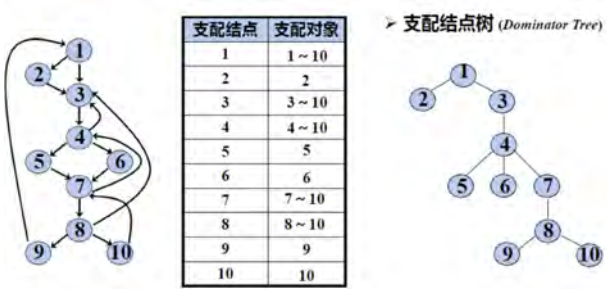


Figure XIII.2: Dominator tree

1.2 Natural Loop

自然循环. 并不关心循环具体代码形式, 而是关心能否提取出易于优化的循环结构.

如果边 $n \to h$ 满足 $h \operatorname{dom} n$, 则这是一个 back edge. Back edge 指向的节点 h 称为 loop header.

可以基于 back edge 严谨地定义 CFG 图中的一个自然循环:

Definition XIII.1. *每个 back edge $n \to h$ 对应一个 natural loop. 这个 natural loop 中的节点集合包含一些节点 x 满足:*

- $h \operatorname{dom} x$
- *存在一条从 x 到 n 的路径不经过 h.*

说人话就是 back edge 是从循环尾回到循环头 (loop header) 的边; 循环体就是中间那些被 loop header 支配的节点. 而 back edge 的意义是保证至少有一条路径能返回首节点 h.

可以有多个 back edge 对应一个 natural loop: 例如 for 语句中的每个 continue 都给这个 natural loop 新增了一个 back edge.

注:

- 循环可以嵌套, 可以共享首节点.
- 除了共享首节点的情况外, 两个循环要么完全不相交, 要么一个完全嵌入另一个 (或者说后者包含前者).
- 最内循环 (innermost loop): 最里的循环, 不包含其他循环 (不被其他循环嵌入).

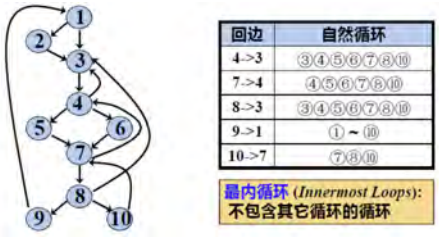


Figure XIII.3: innermost loop

1.3 Loop-nest Tree

表达循环的嵌套关系. 树中每个节点对应一个 loop header 及其对应的 natural loops 节点集合. 如果这个 loop heder 对应多个 natural loops (共享首节点), 也合并到同一个节点.

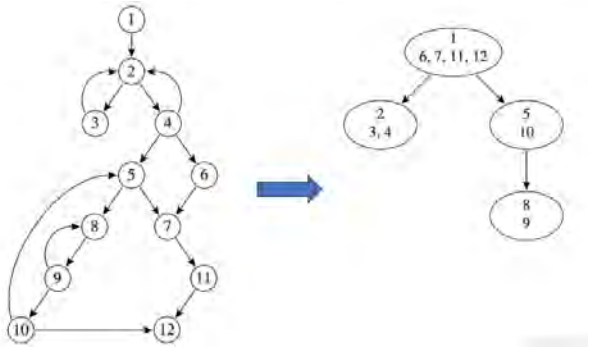


Figure XIII.4: Loop-nest Tree

节点的上半部分表示该节点对应的 loop header. 可以把整个 procedure 视为在一个假想的大循环中, 作为树的根节点. 叶节点即对应最内循环.

Loop preheader: 前置首节点. 许多优化操作会在进入循环前进行一些准备工作, 也就是在紧挨着循环头之前插入一些语句. 因此我们可以在 loop header 之前插入一个 loop preheader 用来安置这些语句.

- loop preheader 的唯一后继就是 loop header
- 循环 L 外到达首节点的边 $x \to h$ 改为进入前置首节点: $x \to p$
- 循环 L 里到达首节点的边不变

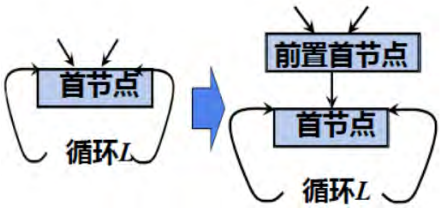


Figure XIII.5: Loop preheader

2. Loop Invariant Hoisting

循环不变代码外提.

Loop-invariant: 如果某个表达式的值在循环中不会改变, 对循环来讲是固定值, 则称表达式 loop-invariant. 显然, 所有常数都 loop-invariant.

赋值语句 $x := v_1$ OP v_2 是 invariant 的当且仅当其操作数 v_1 和 v_2 都满足:

- 操作数是常数, 或是
- 对于操作数中使用到的变量, 其 def 都在循环外, 或是
- 对于操作数中使用到的变量, 在执行赋值语句时其 def 唯一, 且 loop-invariant.

说人话就是操作数及其使用的变量也得 loop-invariant, 而且不能在循环中因为控制流跳转等原因导致变量 def 不唯一.

可以归纳地检查赋值语句 $x := v_1$ OP v_2 的操作数是否 invariant:

- Base cases:
  常数: 一定是 invariant 的.
  变量的 use: 该变量所有 defs 都在循环外则 invariant.
- Inductive cases:
  表达式: 多个 invariant 表达式进行运算仍然 invariant.
  变量的 use: 要求在执行该语句时只可能有唯一的 def 有效, 且这个 def 的右侧 (RHS) 是 loop-invariant 的.

优化: 这样的表达式在循环外就可以求值, 而非每轮循环都计算一次. 这就是 loop-invariant code motion.

- Code hoisting: 如果这样的表达式在循环中需要使用, 则可以将求值提升到循环开始前.
- Code sinking: 如果这样的表达式在循环结束后需要使用, 则可以将求值下沉到循环结束后.

注意表达式的值是 loop-invariant 的不代表 hoisting 就是合法的: 归根结底, 这相当于只关心了表达式的值, 但忽视了其副作用; 盲目上提会破坏代码的语义 (semantics) 信息.

2.1 Code hoisting

hoisting 的判据: 当且仅当满足如下所有条件时可以提升形如 $t := a$ OP b 的赋值语句 (赋值语句就是 def)d:

- d 必须支配所有 t 在其上 live-out 的循环出口 (loop exits).
- 在循环中 t 只能有唯一的 def.
- t 不属于 loop preheader 的 live-out 集合; 换句话说, t 在进入循环前不是活跃的.

如果这个赋值语句有一定的副作用, 那么上述规则就不足够, 需要添加更多判断、约束.

将 while 转写为 for 有助于优化: 拷贝一份条件判断块到循环体末尾即可. 这是由于 while 特殊的控制流结构可能会导致判据中最后一条要求无人能达到, 要把 while 转化为刚刚研究的 repeat-until 这种模式的 natural loops.

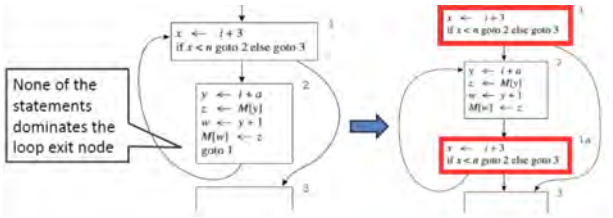


Figure XIII.6: while