This page intentionally left blank

Modern Compiler Implementation in Java

Second Edition

ANDREW W. APPEL
Princeton University

with JENS PALSBERG
Purdue University

Contents

Preface ix

5.Type Mixin a

13.4 G

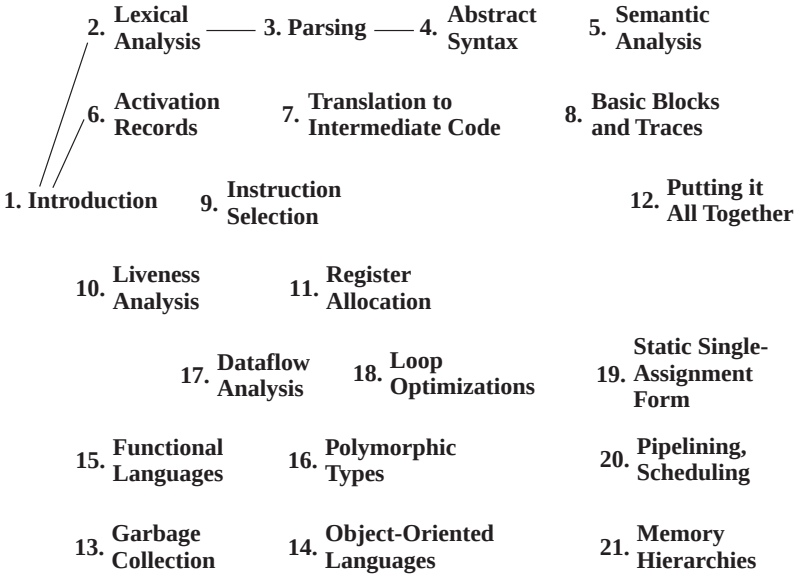
Preface

This book is intended as a textbook for a one- or two-semester course in compilers. Students will see the theory behind different components of a compiler, the programming techniques used to put the theory into practice, and the interfaces used to modularize the compiler. To make the interfaces and programming examples clear and concrete, we have written them in Java. Another edition of this book is available that uses the ML language.

Implementation project. The "student project coiler" that we have outlined is reasonably simple, but is organized to demonstrate some important techniques that are now in common use: abstract syntax trees to avoid tangling syntax and semantics, separation of instruction selection from register allocation, copy propagation to give flexibility to earlier phases of the com-

the occasional three-star exercises are not known to have a solution.

Course sequence. The figure shows how the chapters depend on each other.

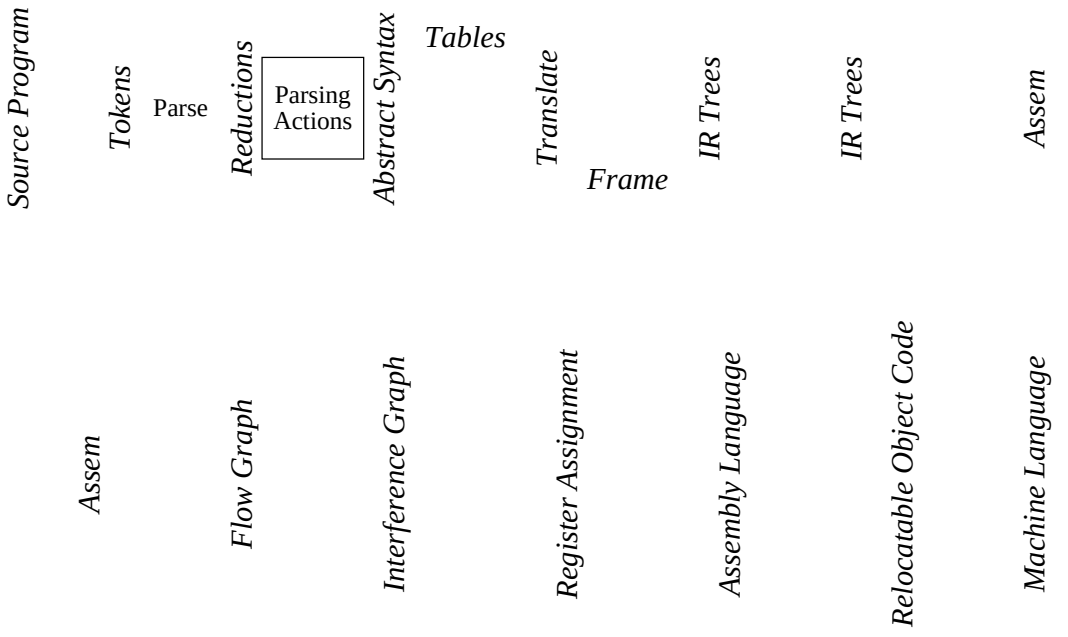


1 Introduction

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

Bauer and Eickel [1975]

This book describes techniques, data structures, and algorithms for translating programming languages into executable code. A modern compiler is often or-


Source Program
Tokens
Parse
Reductions
Parsing Actions
Abstract Syntax
Tables
Translate
Frame
IR Trees
IR Trees
Assem
Assem
Flow Graph
Interference Graph
Register Assignment
Assembly Language
Relocatable Object Code
Machine Language

```
public abstract class Stm {}

public class CompoundStm extends Stm {
   public Stm stm1, stm2;
   public CompoundStm(Stm s1, Stm s2) {stm1=s1; stm2=s2;}}

public class AssignStm extends Stm {
   public String id; public Exp exp;
   public AssignStm(String i, Exp e) {id=i; exp=e;}}

public class PrintStm extends Stm {
   public ExpList exps;
   public PrintStm(ExpList e) {exps=e;}}

public abstract class Exp {}

public class IdExp extends Exp {
   public String id;
   public IdExp(String i) {id=i;}}

public class NumExp extends Exp {
   public int num;
   public 41(2587(ins)-5n3(i))-6000(=np=e;}})]TJ-1.7867 -2.44 Td[(public)-587(clas
   publicId€Exp)-5l,lic inci, Expi)

public classExp extends Exp {
   public Stm (id;)-587(public)-587(Ext)-587(num;)]TJ0 -1.2133 TD[(public)-5Eseq00
   publicExpL€Exp)-5h,ass ExpListe

public classExpList extends ExpList {
   public Expip;
   publicExpL€Exp)-5h7(e)-50hea7=h;=e;}}
```

4. For each nontrivial symbol in the right-hand side of a rule, there will be one field in the corresponding class. (A trivial symbol is a punctuation symbol such as the semicolon in CompoundStm.)
5. Every class will have a constructor function that initializes all the fields.
6.

```
Stm prog =
new CompoundStm(new AssignStm("a",
                     new OpExp(new NumExp(5),
                               OpExp.Plus, new NumExp(3))),
 new CompoundStm(new AssignStm("b",
    new EseqExp(new PrintStm(new PairExpList(new IdExp("a"),
              new LastExpList(new OpExp(new IdExp("a"),
                            OpExp.Minus,new NumExp(1))))),

          new OpExp(new NumExp(10), OpExp.Times,
                    new IdExp("a")))),
  new PrintStm(new LastExpList(new IdExp("b"))));
```

Files with the data type declarations for the trees, and this sample program,

2

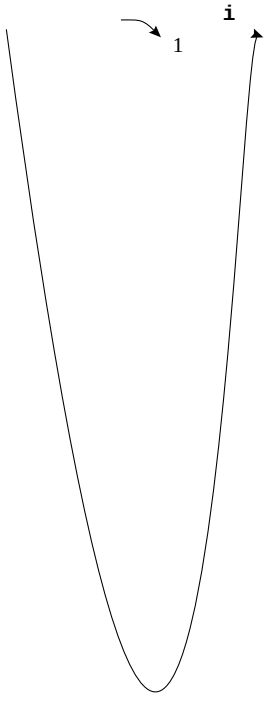
2.1 LEXICAL TOKENS

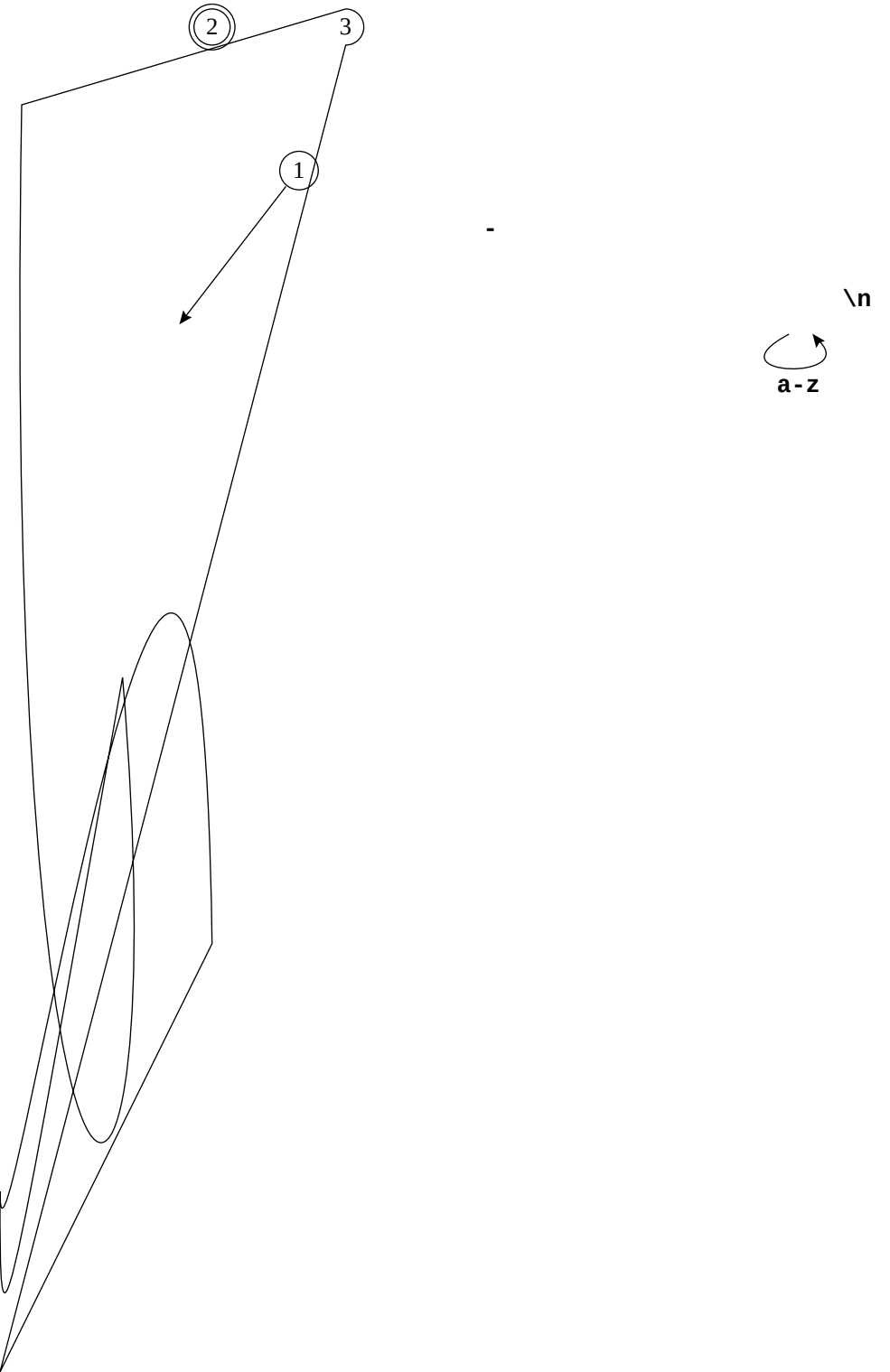
A lexical token is a sequence of characters that can be treated as a unit in the grammar of a programming language. A programming language classifies lexical tokens into a finite set of token types. For example, some of the token types of a typical programming language are

Type	Examples
ID	`foo  n14  last`
NUM	`73  0 00  515  082`
REAL	`66.1  .5  10.  1e67  5.5e-10`
COMMA	`,`

COMMA STRING(

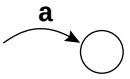
a	An ordinary character stands for itself.
	The empty string.
	Another way to write the empty string.
$M \mid N$	Alternation, choosing from M or N.
$M \cdot N$	Concatenation, an M followed by an N.
MN	Another way to write concatenation.
M	


1
i

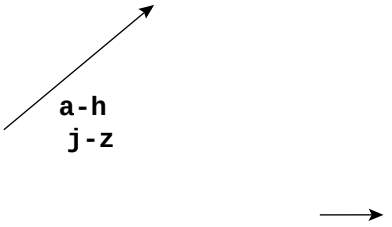

2
3
1
-
\n
a-z

and state 2 of the ID machine; since the latter is final, then the combined state must be final. State 3 t 5e.53ik3(A)-328(3)ate

In the start state, on input character a, the automaton can move either right or left. If left is chosen, then strings of aR0.9(v)14976 Tm(R0.9U277 0 0 -24.9U277 7

a 

M^+ constructed as $M \cdot M$


a-h
j-z

```
PARSER_BEGIN(MyParser)
   class MyParser {}
PARSER_END(MyParser)
```

```
Helpers
        digit  =  ['0'..'9'];
Tokens
        if  =  'if';
        id  =  ['a'..'z']( ['a'..'z']  |  (digit))*;
        number  =  digit+;
```

3. State declarations: support the state feature of, for example, GNU FLEX;

batch of characters is obtained at once; then the lexer can process one charac-

S

GRAMMAR 3.10.

S

GRAMMAR 3.11.

mustRreadRnotR onlyE terminal. symbolsRsuchE asn

Algorithm to compute FIRST, FOLLOW, *and* nullable.
Initialize FIRST and FOLLOW to all empty sets, and nullable to all false.
for each terminal symbol Z
    FIRST$[Z] \leftarrow \{Z\}$
repeat
    for each production $X \rightarrow$

$$S \rightarrow E\ \$$$

$$E \rightarrow T\ E'$$

$$E' \rightarrow +\ T\ E'$$
$$E' \rightarrow -\ T\ E'$$
$$E' \rightarrow$$

```
void T(k {switch (tokk {
        case ID:
        case NUM:
        case LPAREN:  F(k Tprime(k break;
        default:   error!
      }}

void Tprime(k {switch (tokk {
        case PLUS:    break;
        case TIMES:   eat(TIMESk F(k Tprime(k break;
        case EOF:     break;
        case RPAREN:  break;
        default:   error!
      }}
```

A blank entry in row T, column x of the LL(1k parsing table indicates that the parsing function does not expect to see token x – this will be a syntax error. How should *error* be handled? It is safe just to raise an exception and

Stack	*Input*	*Action*
1	a := 7 m 1 ; b a :7 m 1 c + 7 m 1 \ (d a :) 6 0 0 (7 m 1 + 7	
1		
		s h S a := 7 m 1
1		
1		
1		*s h S*
1		*s h S*

0 $S' \rightarrow S\$$

1 $S \rightarrow (L)$

2 $S \rightarrow x$

3 $L \rightarrow S$

4 $L \rightarrow L , S$

GRAMMAR 3.20.

0	$S \rightarrow E\ \$$	2	$E \rightarrow T$
1	$E \rightarrow T + E$	3	

	x	+	$	*E*	*T*
1	s5			g2	g3
2			a		
3		s4	r2		

0	$S' \rightarrow S\ \$$	3	$E \rightarrow V$
1	$S \rightarrow V = E$	4	$V \rightarrow x$
2	$S \rightarrow E$	5	$V \rightarrow * E$

GRAMMAR 3.26. A grammar capturing the essence of expressions, variables, and pointer-dereference (by the *) operator in the C language.

2

S′	→	S . $	?

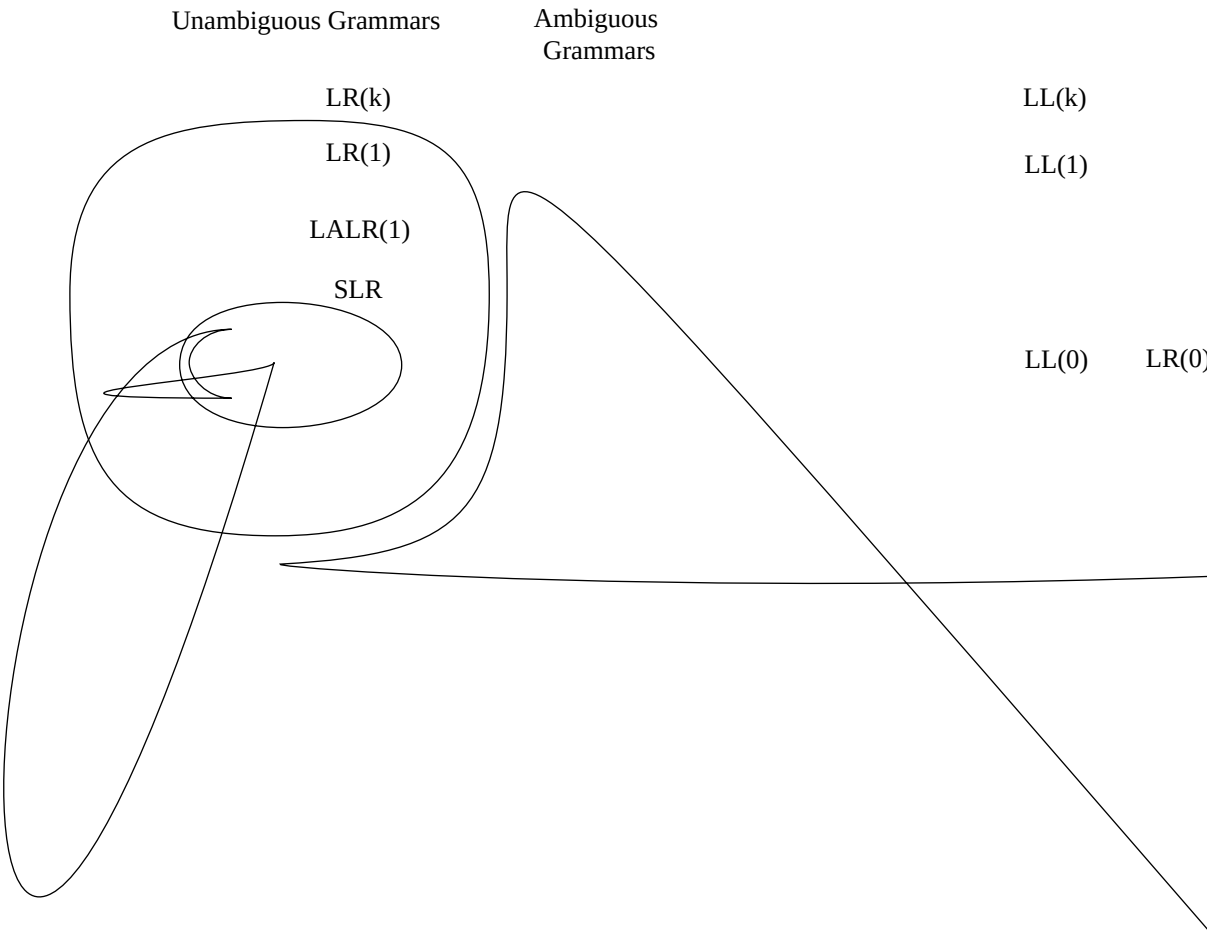
S	→	V . = E	$
E		V .	$

1

S′	. S $	?	
S	. V = E	$	
S	. E	$	
E	.V	$	
V	. x	$,=	
V	.* E	$,=	

6

V	*. E	$,=	
E	. V	$,=	
V	. x	$,=	
V	. *E	$,=	


Unambiguous Grammars
Ambiguous Grammars
LR(k)
LR(1)
LALR(1)
SLR
LL(k)
LL(1)
LL(0)
LR(0)

1 $P \rightarrow L$

2 $S \rightarrow$ id := id
3 $S \rightarrow$ while id do S
4 $S \rightarrow$ begin L end
5 $S \rightarrow$ if id then S
6 $S \rightarrow$ if id then S else S
7 $L \rightarrow$ [illegible] S

GRAMMAR 3.30.

It is often possible to use ambiguous grammars by resolving shift-reduce conflicts in favor of shifting or reducing, as appropriate. But it is best to use this technique sparingly, and only in cases (such as the

mar 3.31. Notice that if we had written the production for `StmList()` in the

	id	num	+	-	*	/	(	)	$	*E*
1	s2	s3					s4			g7
2			r1	r1	r1	r1		r1	r1	
3			r2	r2	r2	r2		r2	r2	
4	s2	s3					s4			g5
5								s6		

		+	-	*	/	
				⋮		
9		r5	r5	s12	s14	
11	...			s12	s14	...
13		r3	r3	r3	r3	
15		r4	r4			
				⋮		

TABLE 3.36.

RECOVERY USING THE ERROR

such [illegible] theseRcalledYhronizingYtoaddingYerrorR20(-recov [illegible] -329(aR)-

rules that contain *error*

e. All subsets and permutations (without repetition) of the keywords `public final static synchronized transient`. (Then comment on how

3.14 Show that this grammar is LL(1) but not LALR(1):

1

4

Abstract Syntax

```
class Token {int kind; Object val;
             Token(int k, Object v) {kind=k; val=v;}
            }
final int EOF=0, ID=1, NUM=2, PLUS=3, MINUS=4,  ···

int lookup(String id) {  ···  }

int F_follow[] [(i"0.73(k,)-600(Obj14TIME(Obj14RPAREN,low[] -583(k,)-}7(val;0.6924
```

```
public abstract class Exp {
   public abstract int accept(Visitor v);
}
public class PlusExp extends Exp {
   public Exp e1,e2;
   public PlusExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
   public int accept(Visitor v) {
       return v.visit(this);
   }
}
public class MinusExp extends Exp {
   public Exp e1,e2;
   public MinusExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
   public int accept(Visitor v) {
       return v.visit(this);
   }
}
public class Timeass Timeass Timeass Timeass Timeass Timeass Timeass Timeass Tim5.
public int accept(Visitor v) {
   pubDividea2;7(v.visit(this);)]TJ-2.4 -1.2133 TD0 Tc(})Tj-1.7873(}133 TD0 Tc(})
      public Exp e1,e2;
      public PlusExp(Exp a1, Exp a2) { IntegerLiteralusExp
```

ing Sparc code but we might also want to remove (in this configuration) the Pentium code-generation functions. This would be very inconvenient in the object-oriented style, requiring each class to be edited. In the *syntax separate from interpretations* style, such a change is modular: We remove a Pentium-related module and add a Sparc module.

We prefer a syntax-separate-from-interpretations style. Fortunately, we can use this style without employing `instanceof` expressions for accessing syntax trees. Instead, we can use a technique known as the Visitor pattern. A visitor implements an interpretation; it is an object which contains a `visit` method for each syntax-tree class. Each syntax-tree class should contain an `accept` method. An

4.3. VISITORS

PROGRAM

5

Semantic Analysis

se-man-tic: of or relating to meeting in language

Webster's Dictionary

The *semantic analysis* phase of a compiler connects variable definitions to their uses, checks that each expression has a correct type, and translates the abstract syntax into a simpler representation suitable for generating machine code.

5.1 SYMBOL TABLES

This phase is characterized by the maintenance of *symbol tables* (also called *environments*

```
class C {
  int a; int b; int c;
  public void m(){
    System.out.println(a+c);
    int j = a+b;
    String a = "hello";
    System.out.println(a);
    System.out.println(j);
    System.out.println(b);
   }
}
```

Suppose we compile this class in the environment σ_0

```
class Bucket {String key; Object binding; Bucket next;
        Bucket(String k, Object b, Bucket n) {key=k; binding=b; next=n;}
}

class Hashc {
   final int SIZE = 256;
   Bucket table[] = new Bucket[SIZE];

   private int hash(String s) {
        int h=0;
```

```
package Symbol;

public class Symbol {
  public String toString();
  public static Symbol symbol(String s);
}

public class Table {
  public Table();
  public void put(Symbol key, Object value);
  public Object get(Symbol key);
  public void beginScope();
  public void endScope();
  public java.util.Enumeration keys();
}
```

PROGRAM 5.5. The interface of package `Symbol`.

- Comparing two symbols for Ògreater-thanÓ (in some arbitrary ordering) is fast (in case we want to make binary search trees).

C H A P T E R

e v E
u b
e r

E R R O R

e h W
h s
i l
f o
o c

```
{ i n t i =
 i n t j = i
 . . .
}
```

n v s e e o n d e w C
s i e t h t n i i
o s
t f I

```
fun f(x) =
  let fun g(y) = x+y
   in g
  end

val h = f(3)
val j = f(4)

val z = h(5)
val w = j(7)
```

tion. Some *local variables* are in this frame; other local variables are kept in machine registers. Sometimes a local variable kept in a register needs to be *saved* into the frame to make room for other uses of the register; there is an area in the frame for this purpose. Finally, when the current function calls other functions, it can use the *outgoing argument* space to pass parameters.

THE FRAME POINTER

registers! How has the use of registers saved any time?

consecutive with the memory locations in which arguments $k + 1$, $k + 2$, etc., are written. Therefore, C programs can't have some of the arguments saved in one place and some saved in another – they must all be saved contiguously.

So in the standard calling convention of many modern machines the *calling*

```
type tree = {key: string, left: tree, right: tree}

function prettyprint(tree: tree) : string =
 let
     var output := ""

     function write(s: string) =
         output := concat(output,s)

     function show(n:int, t: tree) =
        let function indent(s: string) =
                (for i := 1 to n
                  do write(" ");
                 output := concat(output,s); write("\n"))
         in if t=nil then indent(".")
            else (indent(t.key);
                  show(n+1,t.left);
                  show(n+1,t.right))
        end

  in show(0,tree); output
 end
```

PROGRAM 6.3. Nested functions.

There are seselse (indent(t.key);

10 `show` stores its static link (the address of `prettyprint`'s frame) into its own frame.

15 `show` calls `indent`, passing its own frame pointer as `indent`'s static link.

17 `show` calls `show`, passing its own static link (not its own frame pointer) as the static link.

12 `indent` uses the value *n* from 's frame. To do so, it fetches at an appropriate offset from

```
public abstract class Frame {
  abstract public Frame newFrame(Label name,
                                 Util.BoolList formals);
           public Label name;
           public AccessList formals;
  abstract public Access allocLocal(boolean escape);
   /* . . . other stuff, eventually . . . */
}
```

`InFrame(`X`)`

static links. Instead, this is the responsibility of the translation phase. The translation phase would know that each frame contains a static link. The static link `would be` passed to a function in a register and stored into the frame. Since the static link behaves so much like a formal parameter, we can treat it as one (as much as possible).

Hopkins [1986] designed calling conventions optimized for the common case: The first four arguments are passed in registers, with the (rare) extra argu-

Copy i to *save location* in stack frame
Copy frame pointer to i
$\cdots$ body of
Copy *save location* back to i

In Program 6.3, function `prettyprint` is at depth 1, `write` and

```
package Tree;

abstract class Exp
 CONST(int value)
```

NT-18(ET)]TJ/T1_1 1 Tf0.00011 Tc 304399 0 Td[(Labelt)-60(labele))]TJ/T1_2 1 Tf0.0011 Tc- 304399 -1.2670 Td(TEM

```
 Stmp--573stm,s Exp-600(expe))]TJTf913919 -2.44 dD[(abstract)-573(class)-587Stmp
```

CONSMOVJ/T1_1 1 Tf0.00011 Tc 304399 532Td[((Exp--573exmp)ds587(Exp--587rightsrcTJ/T1_2 1 Tf-0.0091 Tc

7.2 TRANSLATION INTO TREES

Translation of abstract syntax expressions into intermediate trees is reasonably straightforward; but there are many cases to handle. We will cover the translation of various language constructs, including many from MiniJava.

KINDS OF EXPRESSIONS

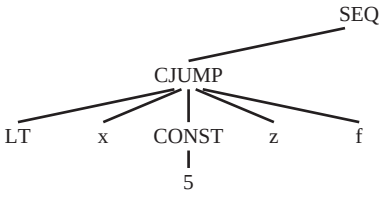
The MiniJava grammar has clearly distinguished statements and expressions. However, in languages such as C, the distinction is blurred; for example, an assignment in C can be used as (in)-295>eva3E2(asa320(for)-I)-372(Cranslatiog)-2

```
if (a<b && c<d) {// true block}else {// false block}
```

might translate to a

7.2. TRANSLATION INa69649b26072RANE.1(A)E.1(A)S

Some bugs are very subtle. But it should not take a genius to detect an out-of-bounds array subscript: If the array bounds are $L..$


SEQ
CJUMP
LT
x
CONST
z
f
5

a series of MOVE trees can initialize offsets $0, 1W, 2W, \ldots, (n-1)W$ from r with the translations of expressions e_i

while

To implement 7, the `Translate` phase should generate a move instruction

MOVE(RV, body)

that puts the result of evaluating the body in the return value (RV) location specified by the machine-specific `frame` structure:

```
package Frame;
public abstract class Frame {
   ⋮
   abstract public Temp RV();
}
```

Item 4 (moving incoming formal parameters), and 5 and 8 (the saving and restoring of callee-save registers), are part of the *view shift* described on page 128. They should be done by a function in lement 7, the

Methodsand the *this* pointer. Method calls in MiniJava are similar to func-

Then identify all the components of the calling sequence (items 1–11), and explain what each line of assembly language does (especially the pseudo-

8

Basic Blocks and Traces

ca-non-i-cal: reduced to the simplest or clearest schema possible

Webster's Dictionary

The trees generated by the semantic analysis phase must be translated into assembly or machine language. The operators of the `Tree` language are chosen carefully to match the capabilities of most machines. However, there are certain aspects of the tree language that do not correspond exactly with machine languages, and some aspects of the

TRANSFORMATIONS ON ESEQ

How can the ESEQ nodes be eliminated? The idea is to lift them higher and higher in the tree, until they can become

(1)

```
abstract public class Stm {
        abstract public ExpList kids();
        abstract public Stm build(ExpList kids);
```

```
public class MOVE extends Stm {
  public Exp dst, src;
  public MOVE(Exp d, Exp s) {dst=d; src=s;}
  public ExpList kids() {
    if (dst instanceof MEM)
       return new ExpList(((MEM)dst).exp,
                          new ExpList(src,null));
    else return new ExpList(src,null);
  }
  public Stm build(ExpList kids) {
    if (dst instanceof MEM)
     return new MOVE(new MEM(kids.head), kids.tail.head);
    else return new MOVE(dst, kids.head);
  }
}
```

Now, given a list of "kids," we pull the ESEQs out, from right to left.

MOVING CALLS TO TOP LEVEL

The `Tree` language permits CALL nodes to be used as subexpressions. However, the actual implementation of CALL will be that each function returns its result in the same dedicated return-value register TEMP(RV)

and does not call `reorder` on the CALL node in that case, but treats the f

```
Put all the blocks of the program into a list Q.
while Q is not empty
    Start a new (empty) trace, call it T.
    Remove the head element b from Q.
    while b is not marked
        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
          b ← c
    End the current trace T.
```

EXERCISES

***8.1**

Write a statement matching the left side of this rewrite rule that produces a different result when rewritten.

Hint: It is very reasonable to say that the statement MOVE TEMP_a TEMP_b commutes with expression TEMP_b (if a and b are not the same), since TEMP_b

Name	*EffectTrees*	
—	r_i	TEMP
ADD	$d_i \leftarrow d_j + d_k$	d +
MUL	$d_i \leftarrow d_j \times d_k$	d * 0

B16ET231.64301 534.33099 m216.98599 523.94701 lSBT/T1_2 1 Tf7.994

141 rules and 35 nonterminal symbols. However, instructions that produce more than one result – such as autoincrement instructions on the VAX – are difficult to express using tree patterns.

Code-generator generators are probably overkill for RISC machines. The

Suppose there are N nodes in the input tree. Then maximal munch will

$t_2 + t_3$ we produceg

t

$t + t$

ABSTRACT ASSEMBLY LANGUAGE INSTRUCTIONS

We will invent a data type for "assembly language instruction without register assignments," called

```
package TempQ
```

```
package Jouette;
public class Codegen {
  Frame frame;
  public Codegen(Frame f) {frame=f;}

  private Assem.InstrList ilist=null, last=null;

  private void emit(Assem.Instr inst) {}}}
  munchStm(s);
  l=ilist;ilist=last=null;
  return l;}}

  package Frame;
  public class Frame {}
```

PROGRAM 9.7. Codegen class.

both listed as destinations of this instruction. (The high-order bits can be very useful for programs written in assembly language to do multiprecision arithmetic, but most programming languages do not support any way to access them.)

IF THERE'S NO FRAME POINTER

implemented by James Larus), described on the Web page for this book.

First write the class μ.`Codegen` implementing the "maximal munch" translation algorithm from IR trees to the `Assem` data structure.

Use the `Canon` module (described in Chapter 8) to simplify the trees before applying your `Codegen` module to them. Use the `format`

(on some machines) the zero register ZEROr Some machines may have other special registers;

`argregs`

munch algorithm to use multiple nonterminal symbols to represent differ-

10

Liveness Analysis

live: of continuing or current interest

Webster's Dictionary

Basic blocks. Flow-graph nodes that have only one predecessor and one successor are not very interesting. Such nodes can be merged with their predecessors and successors; what results is a graph with many fewer nodes, where each node represents a basic block. The algorithms that operate on flow graphs, such as liveness analysis, go much faster on the smaller graphs. Chapter 17 explains how to adjust the dataflow equations to use basic blocks. In this chapter we keep things simple.

One variable at a time. Instead of doing dataflow "in parallel" using set equations, it can be just as practical to compute dataflow for one variable at a time as information about that variable is needed. For liveness, thisw4s mean repeating the dataflow traversal once for each temporary. Starting from each *use* site of a temporary t, a *predecessor* edges of the flow graph) using depth-first search, we note the liveness of t at each flow-graph node. The search stops at any definition of the temporary. Although thiswmight seem expensive, many temporaries have very short live pre-

X Y Z

The answer is that any solution to the dataflow equations is a *conservative approximation*. If the value of variable a will truly be needed in some execution of the program when execution reaches node n of the flow graph, then

The constructor `AssemFlowGraph` takes a list of instructions and returns a flow graph. In making the flow graph, the

tions would be executed if *n*

11.1 COLORING BY SIMPLIFICATION

Register allocation is an NP-complete problem (except in special cases, such

```
live-in: k j
         g := mem[j+12]
         h := k - 1
         f := g * h
         e := mem[j+8]
         m := mem[j+16]
         b := mem[f]
         c := e + 8
         d := c
         k := m + 4
         j := b
live-out: d k j
```

11.2 COALESCING

It is easy to eliminate redundant move instructions with an interference graph. If there is no edge in the interference graph between the sourceand destina-

j

enter: $c \leftarrow r_{3\}}$

$1\}$

$2\}$

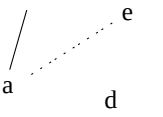
```
int f(int a, int b) {
   int d=0;
   int e=a;
   do {d = d+b;
       e = e-1;
   } while (e>0);
   return d;
}
```

e

r_1 a

d

8. Now we build a new interference graph:



the neighbor must be move-related, and is not added to the `simplifyWorklsst`. When the degree of a neighbor transitions from K

```
procedure AssignColors()
    while
```

```
function SimpleAlloc(t)
    for each nontrivial tile u that is a child of t
        SimpleAlloc(u)
    for each nontrivial tile u that is a child of t
        n ← n − 1
    n ← n + 1
    assign rn
```

```
function Label(t)
    for each tile u that is a child of t
        Label(u)
    if t is trivial
        then need[t] ← 0
    else if t has two children, u_left and u_right
        then if need[u_left] = need[u_right]
                then need[t] ← 1 + need[u_left]
                else need[t] ← max(1, need[u_left], need[u_right])
    else if t has one child, u
        then need[t] ← max(1, need[u])
    else if t has no children
        then need[t] ← 1
```

ALGORITHM 11.10. Sethi-UII.005hn l005hbeling 005hlg05horithm.

$r_1 \leftarrow M[$

function SethiUllman(t, n)

spillCost

simplifyWorklist

freezeWorklist spillWorklist

simplifyWorklist

simplifyWorklist

EXERCISES

a. 4-color the graph without coalescing. SC1(r)8w the *select*-stack, indicating the order inCich youm1(r4(ve)-8(d)-260(node)-8(s.)-276(Is)-279(the)-88re)-271(a) a aua spill?

12

Putting It All Together

de-bug: to eliminate errors in or malfunctions of

Webster's Dictionary

Chapters 2–11 have described the fundamental components of a good compiler: a *frontend* , which does lexicalanalysis, parsing, construction of abstract syntax, type-checking, and translation to intermediate code; and a *back end*, which does instruction selection, dataflow analysis, and register allocation.

What lessons have we learned? We hope that the reader has learned about the algorithms used in different components of a compiler and the interfaces used to connect the components. But the authors have also learned quite a bit from the exercise.

Our goal was to describe a good compiler that is, to use Einstein's phrase, "as simple as possible – but no simpler." we will now discuss the thorny issues that arose in designing the MiniJava compiler.

Sttctu7.3(tr21(3(t)-2.76()-310.76(l)43(t-v9.9(a)lu7.3(tes.]TJ/F3 1 Tf10.9418 0 0 10.94

```
package Frame;
import Temp.Temp;

public abstract class Frame implements Temp.TempMap {
abstract public Temp RV();     (see p. 157)
abstract public Temp FP();     (p. 143)
abstract public Temp.TempList registers();
abstract public String tempMap(Temp temp);
abstract public int wordSize();   (p. 143)
abstract public Tree.Exp externalCall(String func,Tree.ExpList args);  (p. 153)
abstract public Frame newFrame(Temp.Label name,
                               Util.BoolList formals);   (p. 127)
public AccessList formals;    (p. 128)
public Temp.Label name;       (p. 127)
abstract public Access allocLocal(boolean escape);    (p. 129)
abstract public Tree.Stm procEntryExit1(Tree.Stm body);  (p. 251)
abstract public Assem.InstrList procEntryExit2(Assem.InstrList body);  (p. 199)
abstract public Proc procEntryExit3(Assem.InstrList body);
abstract public Assem.InstrList codegen(Tree.Stm stm);   (p. 196)
}
```

PROGRAM 12.1. Package `Frame`.

13

Garbage Collection

CHAPTER THIRTEEN. G7(H)-2(A)RBAGE7(H80(C)-7(0gs0001k23.982100-5ELE8210)-9(BA-2(A)

-
-
-
-
-

function DFS(x)**if** x is a pointer and record x is not marked **mark**(x)**for** each record r in x **search**(r)

point to each record: This is the *reference count* of the record, and it is stored with each record.

next collection. The amortized cost of collection is thus

$$\frac{3R}{\frac{H}{2} - R}$$

instructions per word allocated. As H grows, this cost approaches zero. That is, *there is no inherent lower bound to the cost of garbage collection*. In a more realistic setting, wh

Algorithm 13.13 generalizes all of the mark-sweep and copying algorithms shown so far: Algorithms 13.2, 13.3, 13.5, 13.6, and 13.9.

All these algorithms preserve two natural invariants:

1. No black object points to a white object.
2. Every grey object is on the collector's (stack or queue) data structure (which we will call the *grey-set*).

While the collector operates, the mutator creates new objects (of what color?) and updates pointer fields of existing objects. If the mutator breaks one of the

Any implementation of a write or read barrier must synchronize with the collector. For example, a Dijkstra-style collector might try to change a white node to grey (and put it into the grey-set) at the same time the mutator is also greying the node (and putting it into the grey-set). Thus, software implementations of the read or write barrier will need to use explicit synchronization instructions, which can be expensive.

But implementations using virtual-memory hardware can take advantage of the synchronizationimplicit ina page fault: If the mutator faults ona page, the operating systemitheprocessset)itinheo5(p4(antage)]TJ0 -1.30409 befooftwang)pr

The type- or class-descriptor must be generated by the compiler from the

PROGRAMMING EXERCISE

Design a descriptor format for arrays, and implement it in the compiler and runtime system.

Implement a temp-map with a boolean for each temporary: Is it a pointer or not? Also make a similar map for the offsets in each stack frame, for frame-resident pointer variables. You will not need to handle derived pointers, as your MiniJava compiler probably does not keep derived pointers live across function calls.

For each procedure call, put a new return-address label L_{ret} immediately after the

concurrent algorithm that they could prove correct, trying to keep the synchronization requirements as weak as possible. Baker [1978] invented the incremental copying algorithm in which the mutator sees onlAngthe41395intsyn-

EXERCISES

14

Object-Oriented Languages

ob-ject

```
class A {           int a = 0;}
class B extends A {int b = 0;
                    int c = 0;}
class C extends A {int d = 0;}
class D extends B {int e = 0;}
```

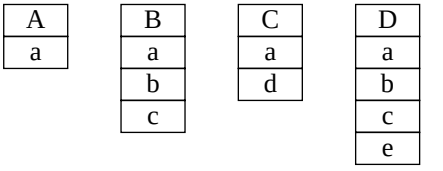


FIGURE 14.2.

goto L_1

where t_1.super is the superclass (parent class) of class t_1.

D

`b` is in fact an instance of `C`, so that the type-check in the narrowing operation can be eliminated.

C++ is an unsafe object-oriented language. It has a *static cast* mechanism

ML:

```
case i
 of 0 => s0
  | 1 => s1
  | 2 => s2
  | 3 => s3
  | 4 => s4
  | _ => sd
```

C, Java:

```
switch (i) {
 case 0: s0; break;
 case 1: s1; break;
 case 2: s2; break;
 case 3: s3; break;
 case 4: s4; break;
 default: sd;
}
```

is compiled as follows: First a range-check comparison is made to ensure that i is within the range of case labels (0–4, in this case); then the address of the ith statement is fetched from the

There are many varieties of privacy and protection. Different languages allow

•

PROGRAM

MiniJava WITH CLASS EXTENSION

Implement class extension in your MiniJava compiler.

FURTHER READING

Dahl and Nygaard's Simula-67 language [Birtwistle et al. 1973] introduced

```
class B extends A {    int f() { return 1; } }
class D extends C {    int f() { this.g(); return 3; } }
```

Similarly, add new instances `E_f`, `F_f`, and `C_g`

15

Functional Programming Languages

func-tion: a mathematical correspondence that assigns exactly one element of one set to each element of the same or another set

Webster's Dictionary

The mathematical notion of function is that if $f(x) = a$

```
type intfun = int -> int;

class C {
 public intfun add(n: int) {
   public int h(int m) { return n+m;}
   return h;
 }
 public intfun twice(f: intfun) {
   public int g(int x) {return f(f(x));}
   return g;
 }
 public int test() {
   intfun addFive = add(5);
   intfun addSeven = add(7);
   int twenty = addFive(15);
   int twentyTwo = addSeven(15);
   intfun addTen = twice(addFive);
   int seventeen = twice(add(5))(7);
   intfun addTwentyFour = twice(twice(add(6)));
   return addTwentyFour(seventeen);
 }
}
```

Program 15.1 illustrates the use of function types. The function `add` takes an integer argument `n` and returns a function

HEAP-ALLOCATED ACTIVATION RECORDS

The local variables for must not be destroyed when returns, because is still needed for the execution of . To solve this problem, we can create a heap-allocated object to hold each function's local variables; then we rely on the garbage collector to reclaim the object when all references (including inner-nested function values) have disappeared.

A refinement of this technique is to save on the heap only those variables that *escape* (that are used by inner-nested functions). The stack frame will hold spilled registers, return address, and so on, and also a pointer to the *escaping-variable record*. The escaping-variable record holds (1) any local variables that an inner-nested procedure might need and (2) a pointer to the environment (escaping-variable record) provid-1.315092escas enclosing function. This pointer from one closure to the closure of the statically enclosing function is called the *static link*; see Figure 15.2.

15.3 IMMUTABLE VARIABLES

The FunJava language has higher-order functions with nested scope, but it is still not really possible to use *equational reasoning* about FunJava programs. ThatT1_t1f1lnalfape0.ape3ape

This seems rather Draconian: How is the program to get any work done? To program without assignments, in a functional style, you produce new values instead of updating old ones. For example, Program 15.3 shows the im-

```
type answer
```

```
class Factorial {
  boolean isDigit (int c) {
      return c >= 48 && c <= 48+9;  // 48 == (int)'0'
  }

  public answer getInt(intConsumer done) {
      public answer nextDigit(int accum) {
          public answer eatChar(int dig) {
              return if (isDigit(dig))
                       nextDigit(accum*10+dig-48)
                     else done(accum);
          }
          return ContIO.readBy-1.2133 TD0 (Digit(dig)))]TJ5.3733nswer g-600(donesn
              retuic aontIO.readByigit'
```

fetches fields from it:

```
class recrd {int a; int b;
          public recrd(int a, int b) {this.a=a; this.b=b;}
}

int a1 = 5;
int b1 = 7;
```

(a) When the actual parameters are simple variables $i_1, \ldots, i_n$.

```
int f(a_1, ..., a_n)
    B
```

```
public answer printTable(list l, cont c) {
   public answer doListX(list l) {
     return if (l==null)
         c();
     else {
         public answer doRest() {return doListX(l.tail);}
         return printDouble(l.head, doRest);
     };
   }
   return doListX(l);
}
```

Cascading inlining. In this version of `printTable`, we have

local) variables. This is done by turning each free-variable access into an instance-variable access.

Some local variables declared in a function f are also accessed by functions nested within f; we say these variables *escape*. For example, in Program 15.5, in the function `putInt`, the variables `dig` and `c` escape (because they are used in the inner-nested function `doDigit`), but the variable `rest` does not escape.

Given a function $f(a_1, \ldots, a_n)\ B$ at nesting depth d with escaping local variables (and formal parameters) $x_1, x_2, \ldots, x_n$ and nonescaping variables $y_1, \ldots, y_n$, we can rewrite into

x

f`(this,` $a_1, \ldots, a_n$`) {`

`c272` r `= newc272(this,` $x_1 x_2$,

15.6 EFFICIENT TAIL RECURSION

Clearly, if we want to claim that two programs are equivalent, then they must behave the same. In PureFunJava, if we obtain program A by doing substition on program B, then A and B they

point to a record containing the `addFive` function. The representation of `addFive(15)` is not a function call that will go and compute the answer *now*, but a thunk that will remember how to compute it on demand, *later*. We might translate this fragment of the LazyJava program into FunJava as follows:

```
/* this already points to a record containing addFive */
c_void_int twenty = new intThunk(this);
```

Invariant hoisting.

```
{int fact(int i) {
      return if (i==0) 1 else i * fact(i-1)
  }
  tree t0 = new tree("",0,null,null);
  tree t1 = t0.enter("-one", if i=0 then 1 else i * fact(i-1));
  tree t2 = t1.enter("three", fact(3));
  return putInt(t2.look("three", exit));
}
```

A curious thing about this program is that `fact(-1)` is undefined. Thus, if this program is compiled by a (strict) PureFunJava compiler, it will infinite-loop. (or will eventually overflow the memory)

Figure 15.7: (a) [illegible] (b) [illegible] (c) [illegible] (d) [illegible]

```
class intList  {int head; intList tail; intList(head,tail){
```

```
bindingThunk look(tree t, key k) {
  return if (k < t.key.eval()) look(t.left.eval(), k)
         else if (k > t.key.eval()) look(t.right.eval(), k)
         else t.binding;
}
```

PROGRAM 15.16. Partial call-by-name using the results of strictness analysis; compare with Program 15.14.

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

int g(int x, int y) { return if (x>0) y else x; }

tree h(String x, int y) { return new tree(x,y,null,null); }

int j(int x) { return j(0); }
```

The function `f` is *strict* in its argument `x`, the `f(x,y)` is demanded then

`f` will certainly touch (demand the value of) `x`. Similarly, `f` is

Function M:

$$M(7, \sigma) = 1$$

$$M(\mathsf{x}, \sigma) = \mathsf{x} \square \sigma$$

$$M(E_1 + E_2, \sigma) = M(E_1, \sigma) \square M(E_2, \sigma)$$

$M($`newobject`$(E$

FURTHER READING

16

list. We could, of course, use the word `Object` instead of `Integer` to declare `head`, but then if we unintentionally mixed Integers and Strings as elements of the same List, the compiler's type-checker wouldn't be able to give us useful feedback: we'drutensame10o(nte)1xcepentied do(b)27(w)-ncastul2227(w

ClassDecl

checkType(T) =
  if T is primitive (`int`, `boolean`, etc.) then OK

Method types. It's convenient to separate the type of a method from the name of the method. For example, consider the declaration

For any type X that's a subclass of `Printable`, tM . L o e o f

checkExp*(e*

6	1

s p i e e PrintableInt (
n w – t d l e x fi
n o
u PointableInt w
n a
e b

the previous sections.

EXERCISES

17

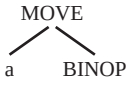
Dataflow Analysis

Therefore, if we had a fully optimizing compiler, we could use it to solve the halting problem; to see if there exists an input on which P halts, just see if **Opt**(P) is the one-line infinite loop. But we know that no computable algorithm can always tell whether programs halt, so a fully optimizing compiler

Each of the optimizing transformations listed at the beginning of the chapter can be applied using the following generic recipe:

Dataflow analysis: Traverse the flow graph, gathering information about what may happen at run time (this will necessarily be a conservative approximation).

Transformation: Modify the program to make it faster in some way; the in-


MOVE
a
BINOP

Statement s	$gen[s]$

executing all the assignments until none of the sets changes any more.

There are several ways to speed up the evaluation of dataflow equations.

BIT VECTORS

A set S over a finite domain (that is, where the elements are integers in the range $1 - N$ or can be put in an array indexed by $1 - N$) can be represented by a *bit vector*. The ith bit in the vector is a 1 if the element i is in the set S

In the bit-vector representation, unioning two sets S and T is done by a bitwise-*or* W, and the

p□pred[n]

more optimizations can be found. At best this iterates two or three times, so that on the third round there are no more transformations to perform.

But the worst case is very bad indeed. Consider a program in which the statement $z; \quad a_1 + a_2 + a_3 + \cdots + a_n$

```
function f( ref i: int,  ref j: int) =
   (i := 5; j := 7; return i)
```

a naive computation of reaching definitions would miss the fact that `i` might be the same variable as `j`, if `f` is called with `f(x,x)`.

The may-alias relation We use *alias analysis*, a kind of dataflow analysis, to learn about different names that may point to the same memory locations. The result of alias analysis *may-alias* relation: *p* may-alias *q* if, in some run of the program, *p* and *q* might point to the same data. As with most dataflow analyses, static (compile-time) information cannot be completely accurate, so the may-alias relation 37.1630(conserv)24(ati)21(v)13(e:)-264(W)73(e)-219(say)-228

The semantic analysis phase of the compiler must compute these classes, as they inolve the concept of *type*

c l a s s c a n b e r e p r e s e

The `Translate` functions must label every fetch and store (that is, every MEM language) with its class. We will need to modify the

```
type list = {head: int,
             tail: list}
var p : list := nil
var q : list := nil
q := list{head=0, tail=nil};
p := list{head=0, tail=q};
q.head := 5;
a := p.head
```

(a) MiniJava program

```
{int *p, *q;
 int h,i;
 p = &h;
 q = &i;
 *p = 0;
 *q = 5;
 a = *p;
}
```

(b) C program

PRORAM17.9. p and q are not aliases.

ALIAS ANALYSIS BASED ON FLOW

Instead of, or in addition to, alias classes based on types, we can also make alias classes based on *point of creation*.

In Program p and q are the same type, we know they point a must be 0; the to definition q.head:=5 cannot affect a.18 251Similarly, in Program 17.9b we know p and q cannot be aliases, so a must be 0.

To catch these distinctions automatically, we will make an alias class for each point of creation.18 60(T)-9(h)1(at)-243(is,)-2251for every different statement w allocated (that malloc in C or new in Pascal or Java) we make a new alias class. Also, each address is taken is an alias class.

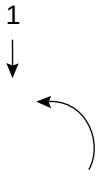
A pointer(or call-by-reference parameter)can point to variables of more than one alias class.18 81.1(In)-250.1(the)-262.1(program)]TJ8.9933 0 0 8.9933 128.

Producing may-alias information.

where the numbers 1–6 show the correspondence with the flow-graph nodes of Figure 18.1f.

Because the programmer can arrange these functions in arbitrary ways, flow graphs produced by the tail-call structure of functional programs are sometimes irreducible.

Advantages of reducible flow graphs. Many dataflow analyses (presented in


1

x i

2. If b_j or b_k

Let $k_1, \quad k$

quadruples for `A[i]` will include $n \leftarrow M[A]$, to fetch n for doing the bounds

$$
\begin{aligned}
L_1 : x &\leftarrow M[i] \\
s &\leftarrow s +
\end{aligned}
$$

EXERCISES

18.1 a. Calculate the dominators of each node of this flowgraph:

19.1. CONVERTING TO SSA FORM

$DF_{\text{local}}[n]$:

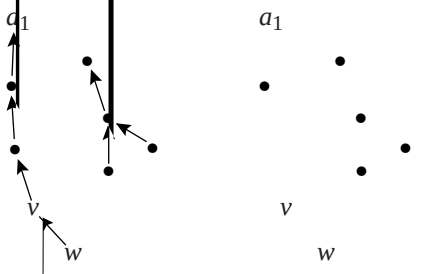
Place-φ-Functions =

edge in the control-flow graph, *(b)* for each statement in the program, *(c)* for each element of every dominance frontier, and *(d)* for each inserted φ-function. For a program of size N, the amounts a and b are proportional to N,

*Initialization:***for**each variable$aCount[a] \leftarrow 0Stack[a] \leftarrow$

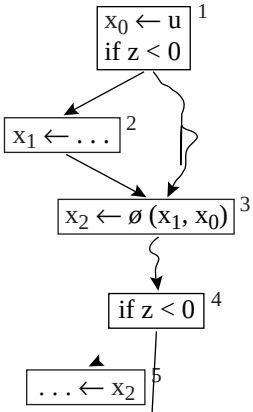
SEMIDOMINATORS

Paths that bypass ancestors of n are useful for proving that those ancestors do not dominate n. Let us consider, for now, only those bypassing paths that rejoin the spanning tree at n (not above n). Let's find the path that departs

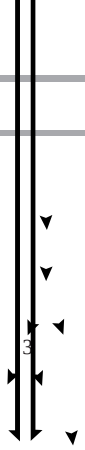

a_1
v
w

a_1
v
w

If we run this algorithm on the SSA program of FiINgure 19.4g, then the assignment $j_3 \leftarrow$



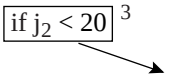
propeof



return 1 [4]

19.6. CONVERTING BACK FROM SSA FORM

19.7. A FU.Z02(TIO)87(20(ALA)-702.9I(.ZEF)9.1RM)11(EF)9.1D.ŁA F


if $j_2 < 20$
3

Translate(*node*) =
  let C be the children of *node* in the dominator tree
  let $p_1, \ldots, p_n$

obeys all the scope rules, but arguments are not atomic and variables are not unique. It is a simple matter to introduce well-scoped intermediate temporaries by a recursive walk of expression trees; dominator and SSA calculations are unnecessary.

u o n l

FURTHER READING

$\boxed{\text{if } v < n}$ [3]

return m

$\boxed{\text{if } r \geq n}$ [5]

issues two or more instructions in parallel *if they are not related by data dependence* (which it can check quickly in the instruction-decode hardware);

ADD	**I-Fetch**	

MultA	MultA	MultA	

a c

j d

b

f

h

g

constraint has been violated; this can be solved by also moving the calculations that use d to a later cycle.

Conversely, if a data-dependence edge grows many cycles long, then we must carry several "versions" of a value around the loop (as we carry f, f', f'' around the loop of Figure 20.8), and this means that we are using more temporaries, so that register allocation may fail. In fact, we should consider register allocation simultaneously

COMPARE	I-Fetch	Read	Arith ↓	Write				
BRANCH		I-Fetch	Read	↑ Arith ↓				
ADD			*wait*	*wait*	↑ I-Fetch	Read	Arith	Write

FIGURE 20.11. Dependence of ADD's instruction-fetch on result of BRANCH.

ter renaming can use hundreds of actual registers with a few static names (see the Further Reading section). Optimal static scheduling depends on knowing the precise pipeline state that will be reached by the hardware, which is sometimes difficult to determine in practice. Finally, dynamic scheduling does not require that the program be recompiled (i.e., rescheduled) for each different implementation of the same 1 Tf0 tion set.

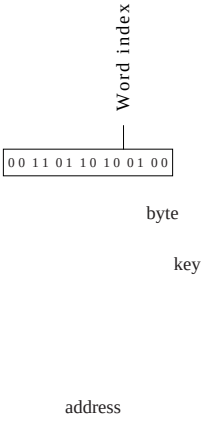
CHAPTER SEVENTEEN

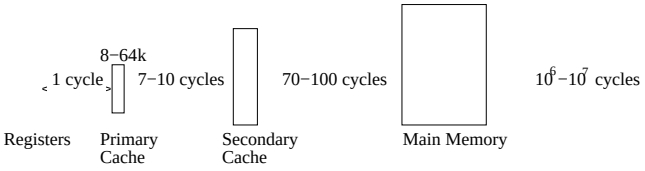
Prediction based on

I. Move the shallow-slope group(s) down to close the gap.

heuristics *(pointer, call, return, loop header, loop preheader, guard)* described on pages 457–458. Label each instance.

20.6 Use branch-prediction heuristics to predict the direction of each of the conditional branches in the programs of Exercise 8.6 (page 175) and Figure 18.7b

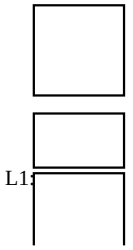

Word index
00 11 01 10 10 01 00
byte
key
address


8−64k
1 cycle
7−10 cycles
70−100 cycles
10^6−10^7 cycles
Registers
Primary Cache
Secondary Cache
Main Memory

Temporal locality: Accessing (fetching or storing) a word that has been recently accessed will usually be a cache hit.

Spatial locality: Accessing a word in the same cache block as one that has been accessed recently will usually be a cache hit.

Cache conflict: If address a and address $a + i \cdot 2$ are both frequently


L1

Cache Delay	Instruction issued	Live or reserved registers
	$x_1 \leftarrow M[i_0]$	$s_0 i_0 j_0 x_1$
|	$y_1 \leftarrow M[j_0]$	$s_0 i_0 j_0 x_1 y_1$
| |	$i_1 \leftarrow i_0 + 4$	$s_0 i_1 j_0 x_1 y_1$
|	$j_1 \leftarrow j_0 + 4$	$s_0 i_1$ *7961 Tm(.51151 52943 0 0 7.43 140.4032 532.2446 Tm(0)Tj/T1_2 1 Tf8.9933 0 0 8.9939933*

block, so x_1, x_2, x_3, x_4

In the example shown above, there is *no* dependence between any iterations of the nested loops, so interchange is legal.

See the Further Reading section for a discussion of the analysis of dependence relations for array accesses in nested loops.

21.5 BLOCKING

The technique of *blocking* reorders a computation so that all the computa-

21.5. BLOCKING

c. Modify your program to optimize on both levels of the cache; measure its run time.

APPENDIX

MiniJava Language Reference Manual

MiniJava is a subset of Java. The meaning of a MiniJava program is given by its meaning as a Java program. Overloading is not allowed in MiniJava. The MiniJava statement `System.out.println( ...);` f4m(262(only)-273(print(

Bibliography

CHAMBERS, C. AND LEAVENS, G. T. 1995. Typechecking and modules for multimethods. *ACM Trans. on Programming Languages and Systems* 17(6), 805–843.

CHAMBERS, C., UNGAR, D., AND LEE

statically typed object-oriented programs. In OOPSLA ’96: 11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications. *SIGPLAN Notices* 31, 292–305.

DIWAN, A., TARDITI, D., AND MOSS, E. 1994. Memory subsystem performance of programs using copying garbage collection. In *Proc. 21st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. ACM Press, New York, 1–14.

DIXON, R., MCKEE, T., SCHWEIZER, P., AND VAUGHAN, M. 1989. A fast method dispatcher for compiled languages with multiple inheritance. In OOPSLA ’89: Object-Oriented Programming: Systems, Languages, and Applications. *SIGPLAN Notices* 24(10), 211–214.

ERSHOV, A. P. 1958. On programming of arithmetic operations. *Commun. ACM* 1(8), 3–6.

FELDMAN, J. AND GRIES, D. 1968. Tr omm. ACM 11(3), 91–100.
C1 3-91 TC30133 5Td(F)Tj0.07277J0 T.22.9933 0 01.2267

GORDON, , A. J. R. G., MORRIS, L., NEWEY AND WADSWORTH, C. P. 1518. A metalanguage for interactive proof in LCF. In *Fifth ACM Symp. on Principles of Programming Languages*

AO GOVINDARAJAN, R., ALTMAN, E. R., AND G , G. R. 1596. A framework for resource-constrained rate-optimal software pipelining. *IEEE Transactions on Parallel and Distributed Systems* 7(11), 1133–1149.

GRAY, R. W. 1988. γ-GLA—a generator for lexical analyzers that programm5.9(yz81.9(can)-250.9(u York.

H

ALL

Addison Wesley, Reading, MA.
KOOPMAN, P. J., LEE, P., AND SIEWIOREK, D. P. 1992. Cache behavior of combinator

Index

abstract data type, 5
abstract syntax, *see* syntax, abstract
access link, *see* static link
activation record, 6, 116–125
Ada, 336, 348, 349
addressing mode, 183, 188
ADT, *see* abstract data type
Aiken-Nicolau algorithm, 444–448, 459
alias
    analysis, 357, 369–374, 392
    in coalescing register allocation, 234
alignment, *see* cache alignment
`alloca`, 197
allocation
    of activation records, 116, 118, 156
    of arrays and records, 151
    of heap data, 275
    register, *see* register allocation
alphabet, 18
ambiguous grammar, *see* grammar
analysis
    dataflow, *see* dataflow analysis
    liveness, *see* liveness
ab99 0 TD06pede

edge splitting, 408
edge-split SSA, *see* static single-assignment form

hash table, 106, 114
hazard, 441, *see also* constraint, functional-

IBM 360/91, 455
induction variable, 385–391
coordinated, 388, 390, 392
linear, 387
inheritance, 283, 284
multiple, 286
single, 285, 294, 295

invariant, 314, 326, 382, 384–389, 398

LR(1), 63
st l0 0 50 0 nk(,)-63(125(,)-63(134(,)-76(155(,)-63(156(,)-76(1993)]TJ03 -1.1533 T