

---

# Modern Compiler Implementation in C

---

## Basic Techniques

ANDREW W. APPEL

*Princeton University*

with MAIA GINSBURG

Preliminary edition of *Modern Compiler Implementation in C*



PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE  
The Pitt Building, Trumpington Street, Cambridge CB2 1RP, United Kingdom

CAMBRIDGE UNIVERSITY PRESS  
The Edinburgh Building, Cambridge CB2 2RU, United Kingdom  
40 West 20th Street, New York, NY 10011-4211, USA  
10 Stamford Road, Oakleigh, Melbourne 3166, Australia

© Andrew W. Appel and Maia Ginsburg, 1997

This book is in copyright. Subject to statutory exception  
and to the provisions of relevant collective licensing agreements,  
no reproduction of any part may take place without  
the written permission of Cambridge University Press.

First published 1997

Printed in the United States of America

Typeset in Times, Courier, and Optima

*Library of Congress Cataloguing-in-Publication data applied for*

*A catalog record for this book is available from the British Library*

0-521-58275-X	Modern Compiler Implementation in ML: Basic Techniques (hardback)
0-521-58775-1	Modern Compiler Implementation in ML: Basic Techniques (paperback)
0-521-58387-X	Modern Compiler Implementation in Java: Basic Techniques (hardback)
0-521-58654-2	Modern Compiler Implementation in Java: Basic Techniques (paperback)
0-521-58389-6	Modern Compiler Implementation in C: Basic Techniques (hardback)
0-521-58653-4	Modern Compiler Implementation in C: Basic Techniques (paperback)

---

# Contents

---

*Preface*

ix

## **Part I Fundamentals of Compilation**

<b>1</b>	<b>Introduction</b>	3
1.1	Modules and interfaces	4
1.2	Tools and software	5
1.3	Data structures for tree languages	7
<b>2</b>	<b>Lexical Analysis</b>	16
2.1	Lexical tokens	17
2.2	Regular expressions	18
2.3	Finite automata	21
2.4	Nondeterministic finite automata	24
2.5	Lex: a lexical analyzer generator	31
<b>3</b>	<b>Parsing</b>	39
3.1	Context-free grammars	41
3.2	Predictive parsing	46
3.3	LR parsing	56
3.4	Using parser generators	67
<b>4</b>	<b>Abstract Syntax</b>	80
4.1	Semantic actions	80
4.2	Abstract parse trees	84
<b>5</b>	<b>Semantic Analysis</b>	94
5.1	Symbol tables	94
5.2	Bindings for the Tiger compiler	103
5.3	Type-checking expressions	106

---

## CONTENTS

---

5.4	Type-checking declarations	109
<b>6</b>	<b>Activation Records</b>	<b>116</b>
6.1	Stack frames	118
6.2	Frames in the Tiger compiler	126
<b>7</b>	<b>Translation to Intermediate Code</b>	<b>140</b>
7.1	Intermediate representation trees	141
7.2	Translation into trees	144
7.3	Declarations	160
<b>8</b>	<b>Basic Blocks and Traces</b>	<b>166</b>
8.1	Canonical trees	167
8.2	Taming conditional branches	175
<b>9</b>	<b>Instruction Selection</b>	<b>180</b>
9.1	Algorithms for instruction selection	183
9.2	CISC machines	192
9.3	Instruction selection for the Tiger compiler	194
<b>10</b>	<b>Liveness Analysis</b>	<b>206</b>
10.1	Solution of dataflow equations	208
10.2	Liveness in the Tiger compiler	216
<b>11</b>	<b>Register Allocation</b>	<b>222</b>
11.1	Coloring by simplification	223
11.2	Coalescing	226
11.3	Graph coloring implementation	231
11.4	Register allocation for trees	240
<b>12</b>	<b>Putting It All Together</b>	<b>248</b>
 <b>Part II Advanced Topics</b>		
<b>13</b>	<b>Garbage Collection</b>	<b>257</b>
13.1	Mark-and-sweep collection	257
13.2	Reference counts	262
13.3	Copying collection	264
13.4	Generational collection	269

---

## CONTENTS

---

13.5	Incremental collection	271
13.6	Baker's algorithm	274
13.7	Interface to the compiler	275
<b>14</b>	<b>Object-oriented Languages</b>	<b>283</b>
14.1	Classes	283
14.2	Single inheritance of data fields	286
14.3	Multiple inheritance	288
14.4	Testing class membership	290
14.5	Private fields and methods	293
14.6	Classless languages	294
14.7	Optimizing object-oriented programs	295
<b>15</b>	<b>Functional Programming Languages</b>	<b>299</b>
15.1	A simple functional language	300
15.2	Closures	302
15.3	Immutable variables	303
15.4	Inline expansion	309
15.5	Closure conversion	315
15.6	Efficient tail recursion	318
15.7	Lazy evaluation	320
<b>16</b>	<b>Dataflow Analysis</b>	<b>333</b>
16.1	Intermediate representation for flow analysis	334
16.2	Various dataflow analyses	337
16.3	Transformations using dataflow analysis	341
16.4	Speeding up dataflow analysis	343
16.5	Alias analysis	351
<b>17</b>	<b>Loop Optimizations</b>	<b>359</b>
17.1	Dominators	362
17.2	Loop-invariant computations	365
17.3	Induction variables	369
17.4	Array bounds checks	374
17.5	Loop unrolling	378
	<b>Appendix: Tiger Language Reference Manual</b>	<b>381</b>
A.1	Lexical issues	381
A.2	Declarations	381

---

## CONTENTS

---

A.3	Variables and expressions	384
A.4	Standard library	388
	<i>Bibliography</i>	389
	<i>Index</i>	393

---

# Preface

---

Over the past decade, there have been several shifts in the way compilers are built. New kinds of programming languages are being used: object-oriented languages with dynamic methods, functional languages with nested scope and first-class function closures; and many of these languages require garbage collection. New machines have large register sets and a high penalty for memory access, and can often run much faster with compiler assistance in scheduling instructions and managing instructions and data for cache locality.

This book is intended as a textbook for a one-semester or two-quarter 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, I have written them in the C programming language. Other editions of this book are available that use the Java and ML languages.

The “student project compiler” that I 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, sophisticated copy propagation to allow greater flexibility to earlier phases of the compiler, and careful containment of target-machine dependencies to one module.

This book, *Modern Compiler Implementation in C: Basic Techniques*, is the preliminary edition of a more complete book to be published in 1998, entitled *Modern Compiler Implementation in C*. That book will have a more comprehensive set of exercises in each chapter, a “further reading” discussion at the end of every chapter, and another dozen chapters on advanced material not in this edition, such as parser error recovery, code-generator generators, byte-code interpreters, static single-assignment form, instruction scheduling

and software pipelining, parallelization techniques, and cache-locality optimizations such as prefetching, blocking, instruction-cache layout, and branch prediction.

**Exercises.** Each of the chapters in Part I has a programming exercise corresponding to one module of a compiler. Unlike many “student project compilers” found in textbooks, this one has a simple but sophisticated back end, allowing good register allocation to be done after instruction selection. Software useful for the programming exercises can be found at

`http://www.cs.princeton.edu/~appel/modern/`

There are also pencil and paper exercises in each chapter; those marked with a star \* are a bit more challenging, two-star problems are difficult but solvable, and the occasional three-star exercises are not known to have a solution.

**Acknowledgments.** Several people have provided constructive criticism, course-tested the manuscript, or helped in other ways in the production of this book. I would like to thank Stephen Bailey, David Hanson, Elma Lee Noah, Todd Proebsting, Barbara Ryder, Amr Sabry, Zhong Shao, Mary Lou Soffa, Andrew Tolmach, and Kwangkeun Yi.



---

## **PART ONE**

# **Fundamentals of Compilation**

---



# 1

---

## Introduction

---

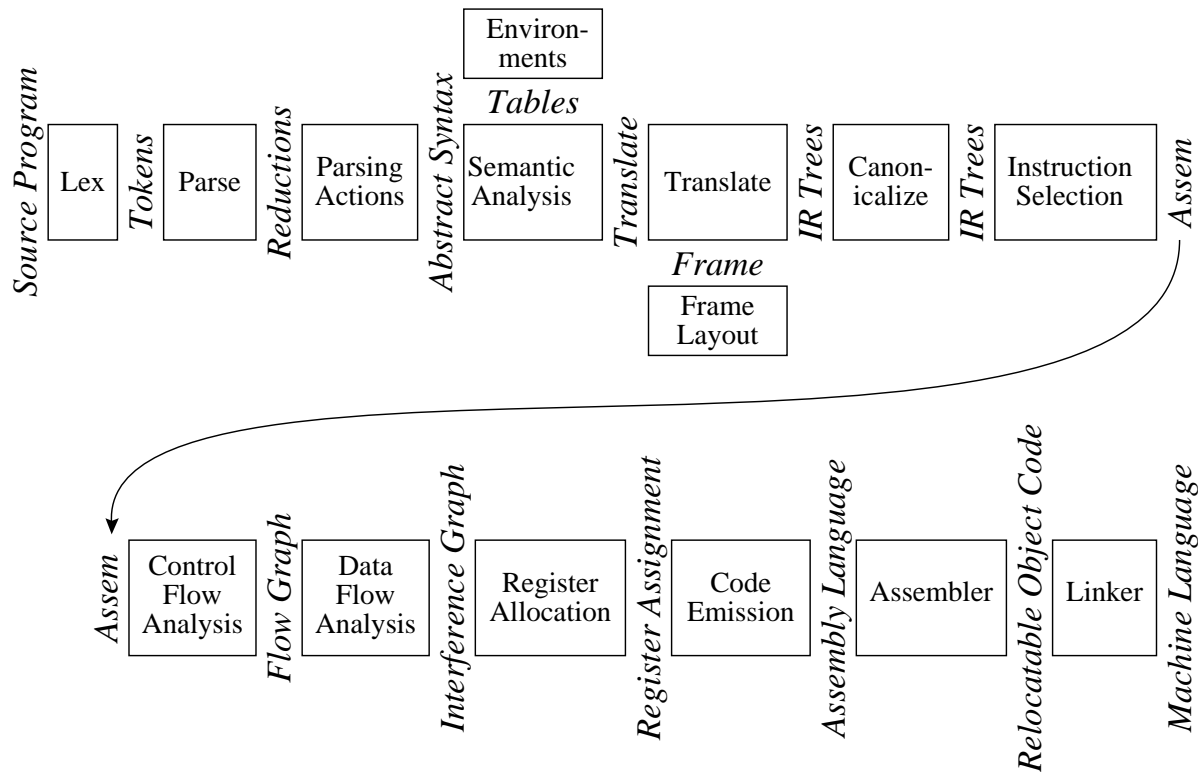
A **compiler** was originally a program that “compiled” subroutines [a link-loader]. When in 1954 the combination “algebraic compiler” came into use, or rather into misuse, the meaning of the term had already shifted into the present one.

Bauer and Eickel [1975]

This book describes techniques, data structures, and algorithms for translating programming languages into executable code. A modern compiler is often organized into many phases, each operating on a different abstract “language.” The chapters of this book follow the organization of a compiler, each covering a successive phase.

To illustrate the issues in compiling real programming languages, I show how to compile Tiger, a simple but nontrivial language of the Algol family, with nested scope and heap-allocated records. Programming exercises in each chapter call for the implementation of the corresponding phase; a student who implements all the phases described in Part I of the book will have a working compiler. Tiger is easily modified to be *functional* or *object-oriented* (or both), and exercises in Part II show how to do this. Other chapters in Part II cover advanced techniques in program optimization. Appendix A describes the Tiger language.

The interfaces between modules of the compiler are almost as important as the algorithms inside the modules. To describe the interfaces concretely, it is useful to write them down in a real programming language. This book uses the C programming language.




---

FIGURE 1.1. Phases of a compiler, and interfaces between them.

---

## 1.1

## MODULES AND INTERFACES

Any large software system is much easier to understand and implement if the designer takes care with the fundamental abstractions and interfaces. Figure 1.1 shows the phases in a typical compiler. Each phase is implemented as one or more software modules.

Breaking the compiler into this many pieces allows for reuse of the components. For example, to change the target-machine for which the compiler produces machine language, it suffices to replace just the Frame Layout and Instruction Selection modules. To change the source language being compiled, only the modules up through Translate need to be changed. The compiler can be attached to a language-oriented syntax editor at the *Abstract Syntax* interface.

The learning experience of coming to the right abstraction by several iterations of *think–implement–redesign* is one that should not be missed. However, the student trying to finish a compiler project in one semester does not have

this luxury. Therefore, I present in this book the outline of a project where the abstractions and interfaces are carefully thought out, and are as elegant and general as I am able to make them.

Some of the interfaces, such as *Abstract Syntax*, *IR Trees*, and *Assem*, take the form of data structures: for example, the Parsing Actions phase builds an *Abstract Syntax* data structure and passes it to the Semantic Analysis phase. Other interfaces are abstract data types; the *Translate* interface is a set of functions that the Semantic Analysis phase can call, and the *Tokens* interface takes the form of a function that the Parser calls to get the next token of the input program.

### DESCRIPTION OF THE PHASES

Each chapter of Part I of this book describes one compiler phase, as shown in Table 1.2

This modularization is typical of many real compilers. But some compilers combine Parse, Semantic Analysis, Translate, and Canonicalize into one phase; others put Instruction Selection much later than I have done, and combine it with Code Emission. Simple compilers omit the Control Flow Analysis, Data Flow Analysis, and Register Allocation phases.

I have designed the compiler in this book to be as simple as possible, but no simpler. In particular, in those places where corners are cut to simplify the implementation, the structure of the compiler allows for the addition of more optimization or fancier semantics without violence to the existing interfaces.

---

## 1.2

---

## TOOLS AND SOFTWARE

Two of the most useful abstractions used in modern compilers are *context-free grammars*, for parsing, and *regular expressions*, for lexical analysis. To make best use of these abstractions it is helpful to have special tools, such as *Yacc* (which converts a grammar into a parsing program) and *Lex* (which converts a declarative specification into a lexical analysis program).

The programming projects in this book can be compiled using any ANSI-standard C compiler, along with *Lex* (or the more modern *Flex*) and *Yacc* (or the more modern *Bison*). Some of these tools are freely available on the Internet; for information see the Wide-World Web page

<http://www.cs.princeton.edu/~appel/modern/>

Chapter	Phase	Description
2	Lex	Break the source file into individual words, or <i>tokens</i> .
3	Parse	Analyze the phrase structure of the program.
4	Semantic Actions	Build a piece of <i>abstract syntax tree</i> corresponding to each phrase.
5	Semantic Analysis	Determine what each phrase means, relate uses of variables to their definitions, check types of expressions, request translation of each phrase.
6	Frame Layout	Place variables, function-parameters, etc. into activation records (stack frames) in a machine-dependent way.
7	Translate	Produce <i>intermediate representation trees</i> (IR trees), a notation that is not tied to any particular source language or target-machine architecture.
8	Canonicalize	Hoist side effects out of expressions, and clean up conditional branches, for the convenience of the next phases.
9	Instruction Selection	Group the IR-tree nodes into clumps that correspond to the actions of target-machine instructions.
10	Control Flow Analysis	Analyze the sequence of instructions into a <i>control flow graph</i> that shows all the possible flows of control the program might follow when it executes.
10	Dataflow Analysis	Gather information about the flow of information through variables of the program; for example, <i>liveness analysis</i> calculates the places where each program variable holds a still-needed value (is <i>live</i> ).
11	Register Allocation	Choose a register to hold each of the variables and temporary values used by the program; variables not live at the same time can share the same register.
12	Code Emission	Replace the temporary names in each machine instruction with machine registers.

---

**TABLE 1.2.** Description of compiler phases.

---

Source code for some modules of the Tiger compiler, support code for some of the programming exercises, example Tiger programs, and other useful files are also available from the same Web address.

Skeleton source code for the programming assignments is available from this Web page; the programming exercises in this book refer to this directory as `$TIGER/` when referring to specific subdirectories and files contained therein.

$Stm \rightarrow Stm ; Stm$	(CompoundStm)	$ExpList \rightarrow Exp , ExpList$	(PairExpList)
$Stm \rightarrow id := Exp$	(AssignStm)	$ExpList \rightarrow Exp$	(LastExpList)
$Stm \rightarrow \text{print} ( ExpList )$	(PrintStm)	$Binop \rightarrow +$	(Plus)
$Exp \rightarrow id$	(IdExp)	$Binop \rightarrow -$	(Minus)
$Exp \rightarrow \text{num}$	(NumExp)	$Binop \rightarrow \times$	(Times)
$Exp \rightarrow Exp Binop Exp$	(OpExp)	$Binop \rightarrow /$	(Div)
$Exp \rightarrow ( Stm , Exp )$	(EseqExp)		

---

**GRAMMAR 1.3.** A straight-line programming language.

---



---

## 1.3

---

## DATA STRUCTURES FOR TREE LANGUAGES

Many of the important data structures used in a compiler are *intermediate representations* of the program being compiled. Often these representations take the form of trees, with several node types, each of which has different attributes. Such trees can occur at many of the phase-interfaces shown in Figure 1.1.

Tree representations can be described with grammars, just like programming languages. To introduce the concepts, I will show a simple programming language with statements and expressions, but no loops or if-statements (this is called a language of *straight-line programs*).

The syntax for this language is given in Grammar 1.3.

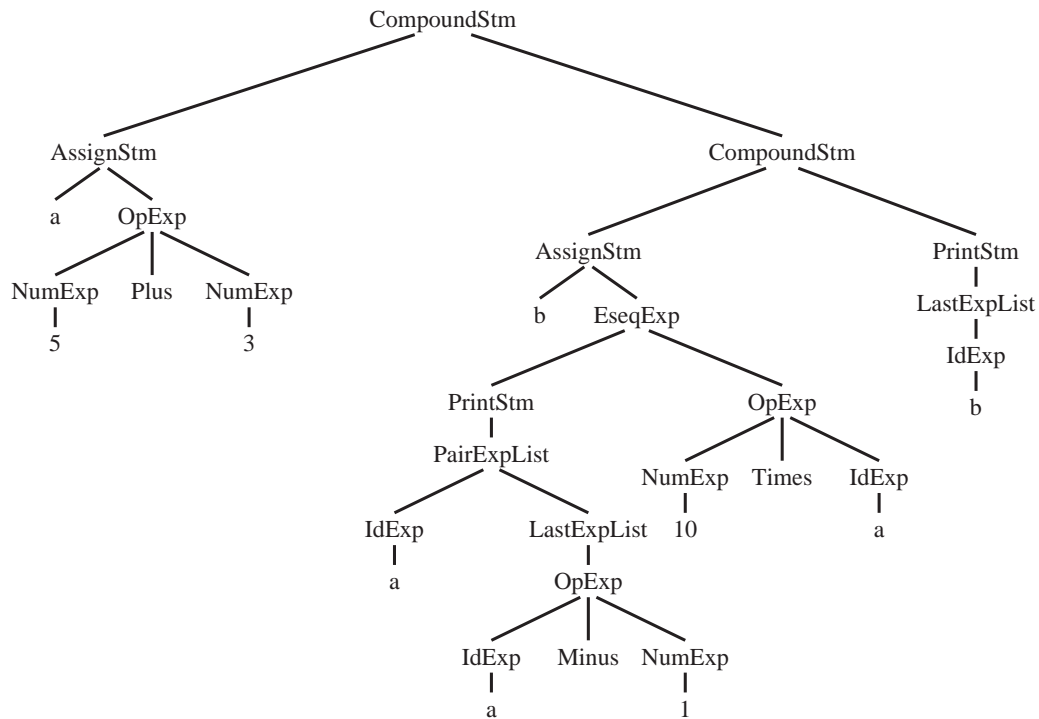
The informal semantics of the language is as follows. Each  $Stm$  is a statement, each  $Exp$  is an expression.  $s_1; s_2$  executes statement  $s_1$ , then statement  $s_2$ .  $i := e$  evaluates the expression  $e$ , then “stores” the result in variable  $i$ .  $\text{print}(e_1, e_2, \dots, e_n)$  displays the values of all the expressions, evaluated left to right, separated by spaces, terminated by a newline.

An *identifier expression*, such as  $i$ , yields the current contents of the variable  $i$ . A *number* evaluates to the named integer. An *operator expression*  $e_1 \text{ op } e_2$  evaluates  $e_1$ , then  $e_2$ , then applies the given binary operator. And an *expression sequence*  $s, e$  behaves like the C-language “comma” operator, evaluating the statement  $s$  for side effects before evaluating (and returning the result of) the expression  $e$ .

For example, executing this program

```
a := 5+3; b := (print(a, a-1), 10*a); print(b)
```

prints



`a := 5 + 3 ; b := ( print ( a , a - 1 ) , 10 * a ) ; print ( b )`

---

**FIGURE 1.4.** Tree representation of a straight-line program.

---

8 7  
80

How should this program be represented inside a compiler? One representation is *source code*, the characters that the programmer writes. But that is not so easy to manipulate. More convenient is a tree data structure, with one node for each statement (Stm) and expression (Exp). Figure 1.4 shows a tree representation of the program; the nodes are labeled by the production labels of Grammar 1.3, and each node has as many children as the corresponding grammar production has right-hand-side symbols.

We can translate the grammar directly into data structure definitions, as shown in Figure 1.5. Each grammar symbol corresponds to a typedef in the data structures:



Grammar	typedef
<i>Stm</i>	A_stm
<i>Exp</i>	A_exp
<i>ExpList</i>	A_expList
<i>id</i>	string
<i>num</i>	int

For each grammar rule, there is one *constructor* that belongs to the union for its left-hand-side symbol. The constructor names are indicated on the right-hand side of Grammar 1.3.

Each grammar rule has right-hand-side components that must be represented in the data structures. The CompoundStm has two Stm's on the right-hand side; the AssignStm has an identifier and an expression; and so on. Each grammar symbol's struct contains a union to carry these values, and a kind field to indicate which variant of the union is valid.

For each variant (CompoundStm, AssignStm, etc.) we make a *constructor function* to malloc and initialize the data structure. In Figure 1.5 only the prototypes of these functions are given; the definition of A\_CompoundStm would look like this:

```
A_stm A_CompoundStm(A_stm stm1, A_stm stm2) {  
    A_stm s = malloc(sizeof(*s));  
    s->stm1=stm1; s->stm2=stm2;  
    return s;  
}
```

For Binop we do something simpler. Although we could make a Binop struct – with union variants for Plus, Minus, Times, Div – this is overkill because none of the variants would carry any data. Instead we make an enum type A\_binop.

**Programming style.** We will follow 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 typedefs, 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.
4. Each struct contains a kind field, which is an enum showing different variants, one for each grammar rule; and a u field, which is a union.

```
typedef char *string;
typedef struct A_stm_ *A_stm;
typedef struct A_exp_ *A_exp;
typedef struct A_expList_ *A_expList;

struct A_stm_ {enum {A_compoundStm, A_assignStm, A_printStm} kind;
              union {struct {A_stm stm1, stm2;} compound;
                    struct {string id; A_exp exp;} assign;
                    struct {A_expList exps;} print;
              } u;
};

A_stm A_CompoundStm(A_stm stm1, A_stm stm2);
A_stm A_AssignStm(String id, A_exp exp);
A_stm A_PrintStm(A_expList exps);

struct A_exp_ {enum {A_idExp, A_numExp, A_opExp, A_eseqExp} kind;
              union {String id;
                    int num;
                    struct {A_exp left; A_binop oper; A_exp right;} op;
                    struct {A_stm stm; A_exp exp;} eseq;
              } u;
};

A_exp A_IdExp(String id);
A_exp A_NumExp(int num);
A_exp A_OpExp(A_exp left, A_binop oper, A_exp right);
A_exp A_EseqExp(A_stm stm, A_exp exp);

typedef enum {A_plus, A_minus, A_times, A_div} A_binop;

struct A_expList_ {enum {A_pairExpList, A_lastExpList} kind;
                  union {struct {A_exp head; A_expList tail;} pair;
                        A_exp last;
                  } u;
}
```

---

**PROGRAM 1.5.** Representation of straight-line programs.

---

5. If there is more than one nontrivial (value-carrying) symbol in the right-hand side of a rule (example: the rule CompoundStm), the union will have a component that is itself a struct comprising these values (example: the compound element of the A\_stm\_ union).
6. If there is only one nontrivial symbol in the right-hand side of a rule, the union will have a component that is the value (example: the num field of the A\_exp union).
7. Every class will have a constructor function that initializes all the fields. The malloc function shall never be called directly, except in these constructor

functions.

8. Each module (header file) shall have a prefix unique to that module (example, `A_` in Figure 1.5).
9. Typedef names (after the prefix) shall start with lower-case letters; constructor functions (after the prefix) with uppercase; enumeration atoms (after the prefix) with lowercase; and union variants (which have no prefix) with lowercase.

**Modularity principles for C programs.** A compiler can be a big program; careful attention to modules and interfaces prevents chaos. We will use these principles in writing a compiler in C:

1. Each phase or module of the compiler belongs in its own “.c” file, which will have a corresponding “.h” file.
2. Each module shall have a prefix unique to that module. All global names (structure and union fields are not global names) exported by the module shall start with the prefix. Then the human reader of a file will not have to look outside that file to determine where a name comes from.
3. All functions shall have prototypes, and the C compiler shall be told to warn about uses of functions without prototypes.
4. We will `#include "util.h"` in each file:

```
/* util.h */
#include <assert.h>

typedef char *string;
string String(char *);

typedef char bool;
#define TRUE 1
#define FALSE 0

void *checked_malloc(int);
```

The inclusion of `assert.h` encourages the liberal use of assertions by the C programmer.

5. The `string` type means a heap-allocated string that will not be modified after its initial creation. The `String` function builds a heap-allocated `string` from a C-style character pointer (just like the standard C library function `strdup`). Functions that take `strings` as arguments assume that the contents will never change.
6. C’s `malloc` function returns `NULL` if there is no memory left. The Tiger compiler will not have sophisticated memory management to deal with this problem. Instead, it will never call `malloc` directly, but call only our own function, `checked_malloc`, which guarantees never to return `NULL`:

```
void *checked_malloc(int len) {
    void *p = malloc(len);
    assert(p);
    return p;
}
```

7. We will never call `free`. Of course, a production-quality compiler must free its unused data in order to avoid wasting memory. The best way to do this is to use an automatic garbage collector, as described in Chapter 13 (see particularly *conservative collection* on page 280). Without a garbage collector, the programmer must carefully `free(p)` when the structure `p` is about to become inaccessible – not too late, or the pointer `p` will be lost, but not too soon, or else still-useful data may be freed (and then overwritten). In order to be able to concentrate more on compiling techniques than on memory deallocation techniques, we can simply neglect to do any freeing.

---

**PROGRAM**

---

**STRAIGHT-LINE PROGRAM INTERPRETER**

Implement a simple program analyzer and interpreter for the straight-line programming language. This exercise serves as an introduction to *environments* (symbol tables mapping variable-names to information about the variables); to *abstract syntax* (data structures representing the phrase structure of programs); to *recursion over tree data structures*, useful in many parts of a compiler; and to a *functional style* of programming without assignment statements.

It also serves as a “warm-up” exercise in C programming. Programmers experienced in other languages but new to C should be able to do this exercise, but will need supplementary material (such as textbooks) on C.

Programs to be interpreted are already parsed into abstract syntax, as described by the data types in Program 1.5.

However, we do not wish to worry about parsing the language, so we write this program by applying data constructors:

```
A_stm prog =
A_CompoundStm(A_AssignStm("a",
    A_OpExp(A_NumExp(5), A_plus, A_NumExp(3))),
A_CompoundStm(A_AssignStm("b",
    A_EseqExp(A_PrintStm(A_PairExpList(A_IdExp("a"),
    A_LastExpList(A_OpExp(A_IdExp("a"), A_minus,
    A_NumExp(1)))))),
    A_OpExp(A_NumExp(10), A_times, A_IdExp("a")))),
A_PrintStm(A_LastExpList(A_IdExp("b"))));
```

Files with the data type declarations for the trees, and this sample program, are available in the directory `$TIGER/chap1`.

Writing interpreters without side effects (that is, assignment statements that update variables and data structures) is a good introduction to *denotational semantics* and *attribute grammars*, which are methods for describing what programming languages do. It's often a useful technique in writing compilers, too; compilers are also in the business of saying what programming languages do.

Therefore, in implementing these programs, never assign a new value to any variable or structure-field except when it is initialized. For local variables, use the initializing form of declaration (for example, `int i=j+3;`) and for each kind of struct, make a “constructor” function that allocates it and initializes all the fields, similar to the `A_CompoundStm` example on page 9.

1. Write a function `int maxargs(A_stm)` that tells the maximum number of arguments of any `print` statement within any subexpression of a given statement. For example, `maxargs(prog)` is 2.
2. Write a function `void interp(A_stm)` that “interprets” a program in this language. To write in a “functional programming” style – in which you never use an assignment statement – initialize each local variable as you declare it.

For part 1, remember that `print` statements can contain expressions that contain other `print` statements.

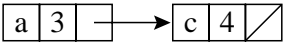
For part 2, make two mutually recursive functions `interpStm` and `interpExp`. Represent a “table,” mapping identifiers to the integer values assigned to them, as a list of `id × int` pairs.

```
typedef struct table *Table_;
Table_ {string id; int value; Table_ tail};
Table_ Table(string id, int value, struct table *tail) {
    Table_ t = malloc(sizeof(*t));
    t->id=id; t->value=value; t->tail=tail;
    return t;
}
```

Then `interpStm` is declared as

```
Table_ interpStm(A_stm s, Table_ t)
```

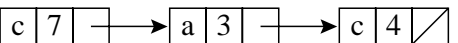
taking a table  $t_1$  as argument and producing the new table  $t_2$  that's just like  $t_1$  except that some identifiers map to different integers as a result of the statement.

For example, the table  $t_1$  that maps  $a$  to 3 and maps  $c$  to 4, which we write  $\{a \mapsto 3, c \mapsto 4\}$  in mathematical notation, could be represented as the linked list .

Now, let the table  $t_2$  be just like  $t_1$ , except that it maps  $c$  to 7 instead of 4. Mathematically, we could write,

$$t_2 = \text{update}(t_1, c, 7)$$

where the update function returns a new table  $\{a \mapsto 3, c \mapsto 7\}$ .

On the computer, we could implement  $t_2$  by putting a new cell at the head of the linked list:  as long as we assume that the *first* occurrence of  $c$  in the list takes precedence over any later occurrence.

Therefore, the update function is easy to implement; and the corresponding lookup function

```
int lookup(Table_ t, string key)
```

just searches down the linked list.

Interpreting expressions is more complicated than interpreting statements, because expressions return integer values *and* have side effects. We wish to simulate the straight-line programming language's assignment statements without doing any side effects in the interpreter itself. (The `print` statements will be accomplished by interpreter side effects, however.) The solution is to declare `interpExp` as

```
struct IntAndTable {int i; Table_ t;};
struct IntAndTable interpExp(A_exp e, Table_ t) ...
```

The result of interpreting an expression  $e_1$  with table  $t_1$  is an integer value  $i$  and a new table  $t_2$ . When interpreting an expression with two subexpressions (such as an `OpExp`), the table  $t_2$  resulting from the first subexpression can be used in processing the second subexpression.

---

## EXERCISES

---

- 1.1 This simple program implements *persistent* functional binary search trees, so that if `tree2=insert(x,tree1)`, then `tree1` is still available for lookups even while `tree2` can be used.

---

## EXERCISES

---

```
typedef struct tree *T_tree;
struct tree {T_tree left; String key; T_tree right;};
T_tree Tree(T_tree l, String k, T_tree r) {
    T_tree t = checked_malloc(sizeof(*t));
    t->left=l; t->key=k; T->right=r;
    return t;
}

T_tree insert(String key, T_tree t) {
    if (t==NULL) return Tree(NULL, key, NULL)
    else if (key < t->key)
        return Tree(insert(key,t->left),t->key,t->right);
    else if (key > t->key)
        return Tree(t->left,t->key,insert(key,t->right));
    else return Tree(t->left,key,t->right);
}
```

- a. Implement a member function that returns true if the item is found, else false.
- b. Extend the program to include not just membership, but the mapping of keys to bindings:

```
T_tree insert(String key, void *binding, T_tree t);
void * lookup(String key, T_tree t);
```

- c. These trees are not balanced; demonstrate the behavior on the following two sequences of insertions:
  - (a) t s p i p f b s t
  - (b) a b c d e f g h i
- \*d. Research balanced search trees in Sedgewick [1988] and recommend a balanced-tree data structure for functional symbol tables. (Hint: to preserve a functional style, the algorithm should be one that rebalances on insertion but not on lookup.)