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.

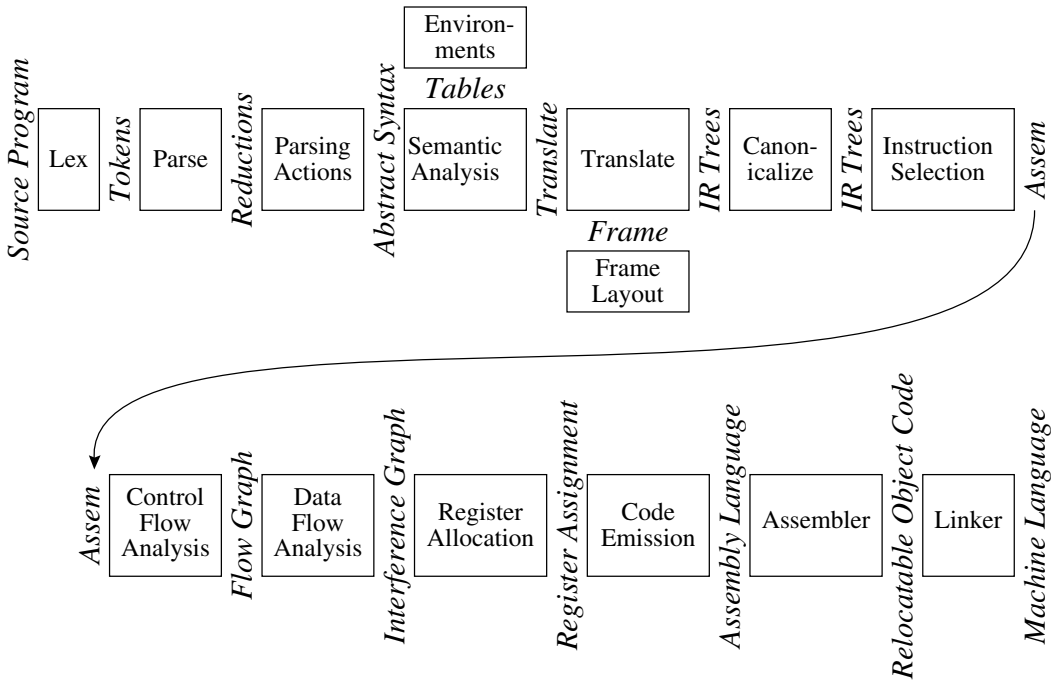


FIGURE 1.1. Phases of a compiler, and interfaces between them.

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 World Wide Web page

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

Chapter	Phase	Description
2	Lex	Break the source file into individual words, or *tokens*.
3	Parse	Analyze the phrase structure of the program.
4	Semantic Actions	Build a piece of *abstract syntax tree* 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 *intermediate representation trees* (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 *control flow graph* 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, *liveness analysis* calculates the places where each program variable holds a still-needed value (is *live*).
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, skeleton source code and support code for some of the programming exercises, example Tiger programs, and other useful files are also available from the same Web address. The programming exercises in this book refer to this directory as `$TIGER/` when referring to specific subdirectories and files contained therein.

Stm → *Stm* ; *Stm*	(CompoundStm)	*ExpList* → *Exp* , *ExpList*	(PairExpList)
Stm → `id :=` *Exp*	(AssignStm)	*ExpList* → *Exp*	(LastExpList)
Stm → `print (` *ExpList* `)`	(PrintStm)	*Binop* → +	(Plus)
Exp → `id`	(IdExp)	*Binop* → −	(Minus)
Exp → `num`	(NumExp)	*Binop* → ×	(Times)
Exp → *Exp Binop Exp*	(OpExp)	*Binop* → /	(Div)
Exp → `(` *Stm* , *Exp* `)`	(EseqExp)		

GRAMMAR 1.3. A straight-line programming language.

1.3 DATA STRUCTURES FOR TREE LANGUAGES

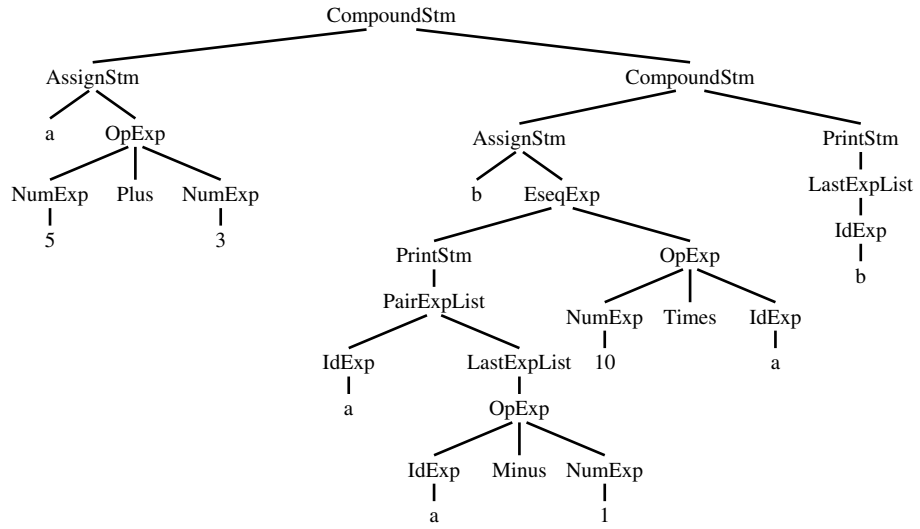
Many of the important data structures used in a compiler are *intermediate representations* of the program being compiled. Often these representations take the form of trees, with several node types, each of which has different attributes. Such trees can occur at many of the phase-interfaces shown in 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. `print`$(e_1, e_2, \ldots, e_n)$ 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 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.




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

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

For example, executing this program

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

prints

```
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 Program 1.5. Each grammar symbol corresponds to a typedef in the data structures:

Grammar	typedef
Stm	A_stm
Exp	A_exp
ExpList	A_expList
id	string
num	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 Program 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 = checked_malloc(sizeof(*s));
  s->kind = A_compoundStm;
  s->u.compound.stm1=stm1; s->u.compound.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` and the definition of the `struct` itself.
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;
typedef enum {A_plus,A_minus,A_times,A_div} A_binop;

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);

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 Program 1.5).
9. Typedef names (after the prefix) shall start with lowercase 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`: