

---

# Modern Compiler Implementation in ML

---

## Basic Techniques

ANDREW W. APPEL

*Princeton University*

Preliminary edition of *Modern Compiler Implementation in ML*



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, 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 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 Lexical Analysis</b>	13
2.1 Lexical tokens	14
2.2 Regular expressions	15
2.3 Finite automata	18
2.4 Nondeterministic finite automata	21
2.5 ML-Lex: a lexical analyzer generator	28
<b>3 Parsing</b>	37
3.1 Context-free grammars	39
3.2 Predictive parsing	44
3.3 LR parsing	53
3.4 Using parser generators	65
<b>4 Abstract Syntax</b>	78
4.1 Semantic actions	78
4.2 Abstract parse trees	83
<b>5 Semantic Analysis</b>	94
5.1 Symbol tables	94
5.2 Bindings for the Tiger compiler	102
5.3 Type-checking expressions	105

---

## CONTENTS

---

5.4	Type-checking declarations	108
<b>6</b>	<b>Activation Records</b>	<b>115</b>
6.1	Stack frames	117
6.2	Frames in the Tiger compiler	125
<b>7</b>	<b>Translation to Intermediate Code</b>	<b>138</b>
7.1	Intermediate representation trees	139
7.2	Translation into trees	142
7.3	Declarations	157
<b>8</b>	<b>Basic Blocks and Traces</b>	<b>162</b>
8.1	Canonical trees	163
8.2	Taming conditional branches	168
<b>9</b>	<b>Instruction Selection</b>	<b>174</b>
9.1	Algorithms for instruction selection	177
9.2	CISC machines	185
9.3	Instruction selection for the Tiger compiler	188
<b>10</b>	<b>Liveness Analysis</b>	<b>198</b>
10.1	Solution of dataflow equations	200
10.2	Liveness in the Tiger compiler	208
<b>11</b>	<b>Register Allocation</b>	<b>214</b>
11.1	Coloring by simplification	215
11.2	Coalescing	218
11.3	Graph coloring implementation	223
11.4	Register allocation for trees	232
<b>12</b>	<b>Putting It All Together</b>	<b>240</b>
 <b>Part II Advanced Topics</b>		
<b>13</b>	<b>Garbage Collection</b>	<b>249</b>
13.1	Mark-and-sweep collection	249
13.2	Reference counts	254
13.3	Copying collection	256
13.4	Generational collection	261

---

## CONTENTS

---

13.5	Incremental collection	263
13.6	Baker's algorithm	266
13.7	Interface to the compiler	267
<b>14</b>	<b>Object-oriented Languages</b>	<b>275</b>
14.1	Classes	275
14.2	Single inheritance of data fields	278
14.3	Multiple inheritance	280
14.4	Testing class membership	282
14.5	Private fields and methods	285
14.6	Classless languages	286
14.7	Optimizing object-oriented programs	287
<b>15</b>	<b>Functional Programming Languages</b>	<b>291</b>
15.1	A simple functional language	292
15.2	Closures	294
15.3	Immutable variables	295
15.4	Inline expansion	301
15.5	Closure conversion	307
15.6	Efficient tail recursion	310
15.7	Lazy evaluation	312
<b>16</b>	<b>Dataflow Analysis</b>	<b>325</b>
16.1	Intermediate representation for flow analysis	326
16.2	Various dataflow analyses	329
16.3	Transformations using dataflow analysis	333
16.4	Speeding up dataflow analysis	335
16.5	Alias analysis	343
<b>17</b>	<b>Loop Optimizations</b>	<b>351</b>
17.1	Dominators	354
17.2	Loop-invariant computations	357
17.3	Induction variables	361
17.4	Array bounds checks	366
17.5	Loop unrolling	370
	<b>Appendix: Tiger Language Reference Manual</b>	<b>373</b>
A.1	Lexical issues	373
A.2	Declarations	373

---

## CONTENTS

---

A.3	Variables and expressions	376
A.4	Standard library	380
	<i>Bibliography</i>	381
	<i>Index</i>	385

---

# 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 ML programming language. Other editions of this book are available that use the C and Java 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 ML: Basic Techniques*, is the preliminary edition of a more complete book to be published in 1998, entitled *Modern Compiler Implementation in ML*. 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, Maia Ginsburg, 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

---

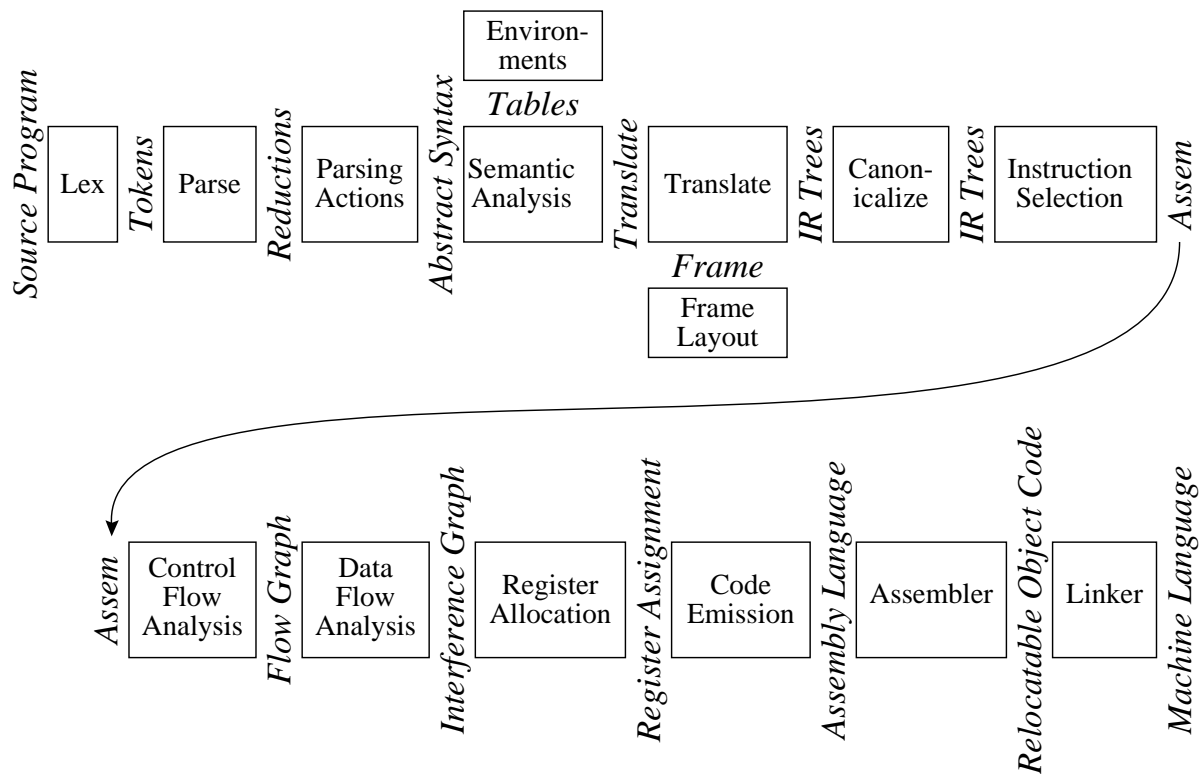
A **compiler** was originally a program that “compiled” subroutines [a link-loader]. When in 1954 the combination “algebraic compiler” came into use, or rather into misuse, the meaning of the term had already shifted into the present one.

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 ML – a strict, statically typed functional programming language with modular structure. ML is well suited to many applications, but compiler implementation in particular seems to hit all of its strong points and few




---

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

---

of its weaknesses. Implementing a compiler in ML is quite a pleasant task. Furthermore, a well rounded introduction to compilers should include some acquaintance with modern programming language design.

This is not a textbook on ML programming. Students using this book who do not know ML already should be able to pick it up as they go along, using an ML programming book such as Paulson [1996] or Ullman [1994] as a reference.

---

## 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.

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*

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.

---

(which converts a grammar into a parsing program) and *Lex* (which converts a declarative specification into a lexical analysis program). Fortunately, good versions of these tools are available for ML, and the project described in this book makes use of them.

The programming projects in this book can be compiled using the *Standard*

---

### 1.3. DATA STRUCTURES FOR TREE LANGUAGES

---

$Stm \rightarrow Stm ; Stm$	(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.

---

*ML of New Jersey* system, including associated tools such as its ML-Yacc, ML-Lex, and the *Standard ML of New Jersey Software Library*. All of this software is available free of charge on the Internet; for information see the Wide-World Web page

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

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.

---

### 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

```
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 `type` in the data structures:

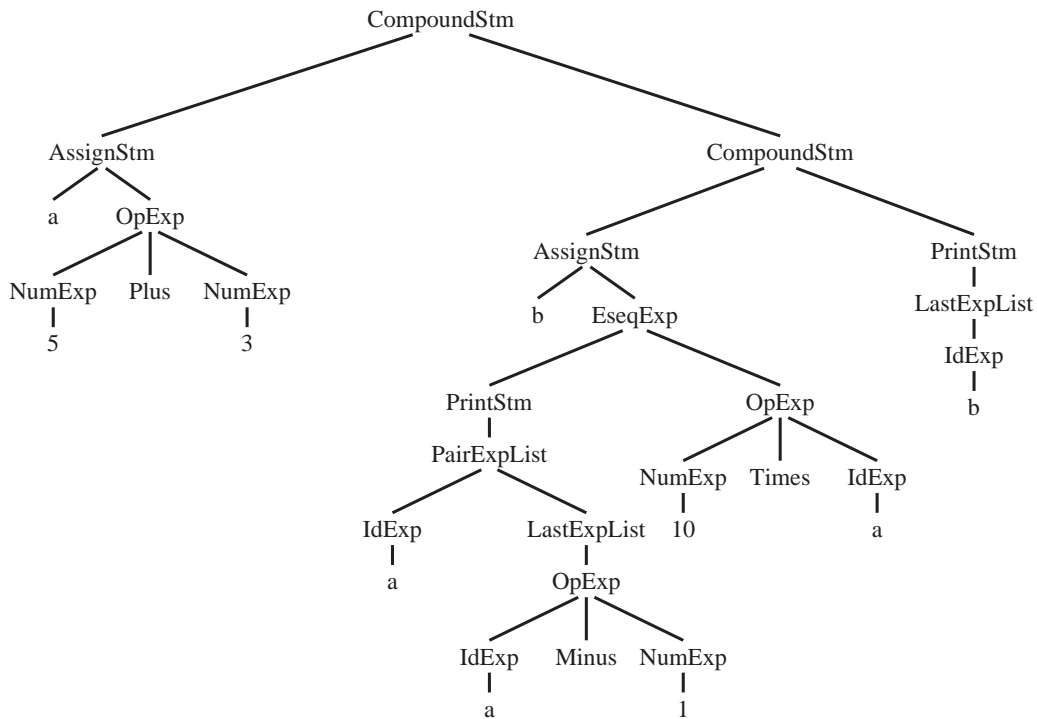
Grammar	type
<i>Stm</i>	stm
<i>Exp</i>	exp
<i>ExpList</i>	exp list
<i>id</i>	id
<i>num</i>	int

For each grammar rule, there is one *constructor* that belongs to the type for its left-hand-side symbol. The ML `datatype` declaration works beautifully to describe these trees. The constructor names are indicated on the right-hand side of Grammar 1.3.

---

### 1.3. DATA STRUCTURES FOR TREE LANGUAGES

---



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

---

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

---

```
type id = string

datatype binop = Plus | Minus | Times | Div

datatype stm = CompoundStm of stm * stm
             | AssignStm of id * exp
             | PrintStm of exp list

and exp = IdExp of id
        | NumExp of int
        | OpExp of exp * binop * exp
        | EseqExp of stm * exp
```

---

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

---

**Modularity principles for ML 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 ML:

1. Each phase or module of the compiler belongs in its own structure.

2. open declarations will not be used. If an ML file begins with

```
open A.F; open A.G; open B; open C;
```

then you (the human reader) *will have to look outside this file* to tell which structure defines the  $X$  that is used in the expression  $X.put()$ .

Structure abbreviations are a better solution. If the module begins,

```
structure W=A.F.W and X=A.G.X and Y=B.Y and Z=C.Z
```

then you can tell *without looking outside this file* that  $X$  comes from  $A.G$ .

---

## 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 ML programming. Programmers experienced in other languages but new to ML should be able to do this exercise, but will need supplementary material (such as textbooks) on ML.

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:

```
val prog =
  CompoundStm(AssignStm("a", OpExp(NumExp 5, Plus, NumExp 3)),
    CompoundStm(AssignStm("b",
      EseqExp(PrintStm[IdExp "a", OpExp(IdExp "a", Minus,
        NumExp 1)],
        OpExp(NumExp 10, Times, IdExp "a"))),
      PrintStm[IdExp "b"])))
```

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

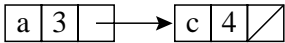
Writing interpreters without side effects (that is, assignment statements that update variables and data structures) is a good introduction to *denotational semantics* and *attribute grammars*, which are methods for describing what programming languages do. It’s often a useful technique in writing compilers,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

Therefore, do not use reference variables, arrays, or assignment expressions in implementing these programs:

1. Write an ML function (`maxargs : stm → int`) 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 an ML function `interp : stm → unit` that “interprets” a program in this language. To write in a “functional” style – without assignment (`:=`) or arrays – maintain a list of (variable, integer) pairs, and produce new versions of this list at each `AssignStm`.

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

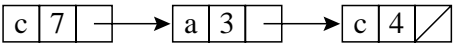
For part 2, make two mutually recursive functions `interpStm` and `interpExp`. Represent a “table,” mapping identifiers to the integer values assigned to them, as a list of `id × int` pairs. Then `interpStm` should have the type `stm × table → table`, taking a table  $t_1$  as argument and producing 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  , written in ML as `("a", 3) :: ("c", 4) :: nil`.

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

```
val lookup: table * id -> int
```

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 make `interpExp` have type `exp × table → int × table`. 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.

```

type key = string
datatype tree = LEAF | TREE of tree * key * tree

val empty = LEAF

fun insert(key,LEAF) = TREE(LEAF,key,LEAF)
  | insert(key,TREE(l,k,r)) =
    if key<k
    then TREE(insert(key,l),k,r)
    else if key>k
    then TREE(l,k,insert(key,r))
    else TREE(l,key,r)

```

- 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:

```

datatype 'a tree = ...
insert: 'a tree * key * 'a -> 'a tree
lookup: 'a tree * key -> 'a

```

- 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.)