

Undergraduate Topics in Computer Science

Series editor

Ian Mackie

Advisory board

Samson Abramsky, University of Oxford, Oxford, UK

Karin Breitman, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil

Chris Hankin, Imperial College London, London, UK

Dexter C. Kozen, Cornell University, Ithaca, USA

Andrew Pitts, University of Cambridge, Cambridge, UK

Hanne Riis Nielson, Technical University of Denmark, Kongens Lyngby, Denmark

Steven S Skiena, Stony Brook University, Stony Brook, USA

Iain Stewart, University of Durham, Durham, UK

Undergraduate Topics in Computer Science (UTiCS) delivers high-quality instructional content for undergraduates studying in all areas of computing and information science. From core foundational and theoretical material to final-year topics and applications, UTiCS books take a fresh, concise, and modern approach and are ideal for self-study or for a one- or two-semester course. The texts are all authored by established experts in their fields, reviewed by an international advisory board, and contain numerous examples and problems. Many include fully worked solutions.

More information about this series at <http://www.springer.com/series/7592>

Torben Ægidius Mogensen

Introduction to Compiler Design

Second Edition



Springer

Torben Ægidius Mogensen
Datalogisk Institut
Københavns Universitet
Copenhagen
Denmark

ISSN 1863-7310 ISSN 2197-1781 (electronic)
Undergraduate Topics in Computer Science
ISBN 978-3-319-66965-6 ISBN 978-3-319-66966-3 (eBook)
<https://doi.org/10.1007/978-3-319-66966-3>

Library of Congress Control Number: 2017954288

1st edition: © Springer-Verlag London Limited 2011
2nd edition: © Springer International Publishing AG 2017

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

Language is a process of free creation; its laws and principles are fixed, but the manner in which the principles of generation are used is free and infinitely varied. Even the interpretation and use of words involves a process of free creation.

Noam Chomsky (1928–)

In order to reduce the complexity of designing and building computers, nearly all of these are made to execute relatively simple commands (but do so very quickly). A program for a computer must be built by combining these very simple commands into a program in what is called *machine language*. Since this is a tedious and error-prone process most programming is, instead, done using a high-level *programming language*. This language can be very different from the machine language that the computer can execute, so some means of bridging the gap is required. This is where the *compiler* comes in.

A compiler translates (or *compiles*) a program written in a high-level programming language, that is suitable for human programmers, into the low-level machine language that is required by computers. During this process, the compiler will also attempt to detect and report obvious programmer mistakes.

Using a high-level language for programming has a large impact on how fast programs can be developed. The main reasons for this are

- Compared to machine language, the notation used by programming languages is closer to the way humans think about problems.
- The compiler can detect some types of programming mistakes.
- Programs written in a high-level language tend to be shorter than equivalent programs written in machine language.

Another advantage of using a high-level language is that the same program can be compiled to many different machine languages and, hence, be brought to run on many different machines.

On the other hand, programs that are written in a high-level language and automatically translated to machine language may run somewhat slower than programs that are hand-coded in machine language. Hence, some time-critical programs are still written partly in machine language. A good compiler will, however, be able to get very close to the speed of hand-written machine code when translating well-structured programs.

The Phases of a Compiler

Since writing a compiler is a nontrivial task, it is a good idea to structure the work. A typical way of doing this is to split the compilation into several phases with well-defined interfaces between them. Conceptually, these phases operate in sequence (though in practice, they are often interleaved), each phase (except the first) taking the output from the previous phase as its input. It is common to let each phase be handled by a separate program module. Some of these modules are written by hand, while others may be generated from specifications. Often, some of the modules can be shared between several compilers.

A common division into phases is described below. In some compilers, the ordering of phases may differ slightly, some phases may be combined or split into several phases or some extra phases may be inserted between those mentioned below.

Lexical analysis	This is the initial part of reading and analyzing the program text: The text is read and divided into <i>tokens</i> , each of which corresponds to a symbol in the programming language, e.g., a variable name, keyword, or number. Lexical analysis is often abbreviated to <i>lexing</i> .
Syntax analysis	This phase takes the list of tokens produced by the lexical analysis and arranges these in a tree structure (called the <i>syntax tree</i>) that reflects the structure of the program. This phase is often called <i>parsing</i> .
Type checking	This phase analyses the syntax tree to determine if the program violates certain consistency requirements, e.g., if a variable is used but not declared, or if it is used in a context that does not make sense given the type of the variable, such as trying to use a Boolean value as a function pointer.
Intermediate code generation	The program is translated to a simple machine-independent intermediate language.
Register allocation	The symbolic variable names used in the intermediate code are translated to numbers, each of which corresponds to a register in the target machine code.

Machine code generation	The intermediate language is translated to assembly language (a textual representation of machine code) for a specific machine architecture.
Assembly and linking	The assembly language code is translated into binary representation and addresses of variables, functions, etc., are determined

The first three phases are collectively called *the front-end* of the compiler and the last three phases are collectively called *the back-end*. The middle part of the compiler is in this context only the intermediate code generation, but this often includes various optimisations and transformations on the intermediate code.

Each phase, through checking and transformation, establishes invariants on the data it passes on to the next phase. For example, the type checker can assume the absence of syntax errors, and the code generation can assume the absence of type errors. These invariants can reduce the burden of writing the later phases.

Assembly and linking are typically done by programs supplied by the machine or operating system vendor, and are hence not part of the compiler itself. We will not further discuss these phases in this book, but assume that a compiler produces its result as symbolic assembly code.

Interpreters

An *interpreter* is another way of implementing a programming language. Interpretation shares many aspects with compiling. Lexing, parsing, and type checking are in an interpreter done just as in a compiler. But instead of generating code from the syntax tree, the syntax tree is processed directly to evaluate expressions, execute statements, and so on. An interpreter may need to process the same piece of the syntax tree (for example, the body of a loop) many times and, hence, interpretation is typically slower than executing a compiled program. But writing an interpreter is often simpler than writing a compiler, and an interpreter is easier to move to a different machine, so for applications where speed is not of essence, or where each part of the program is executed only once, interpreters are often used.

Compilation and interpretation may be combined to implement a programming language. For example, the compiler may produce intermediate-level code which is then interpreted rather than compiled to machine code. In some systems, there may even be parts of a program that are compiled to machine code, some parts that are compiled to intermediate code that is interpreted at runtime, while other parts may be interpreted directly from the syntax tree. Each choice is a compromise between speed and space: Compiled code tends to be bigger than intermediate code, which tend to be bigger than syntax, but each step of translation improves running speed.

Using an interpreter is also useful during program development, where it is more important to be able to test a program modification quickly rather than run the

program efficiently. And since interpreters do less work on the program before execution starts, they are able to start running the program more quickly. Furthermore, since an interpreter works on a program representation that is closer to the source code than is compiled code, error messages can be more precise and informative.

We will discuss interpreters briefly in Chap. 4, but they are not the main focus of this book.

Why Learn About Compilers?

Few people will ever be required to write a compiler for a general-purpose language like C, Java, or SML. So why do most computer science institutions offer compiler courses and often make these mandatory?

Some typical reasons are

- (a) It is considered a topic that you should know in order to be “well-cultured” in computer science.
- (b) A good craftsman should know his tools, and compilers are important tools for programmers and computer scientists.
- (c) The techniques used for constructing a compiler are useful for other purposes as well.
- (d) There is a good chance that a programmer or computer scientist will need to write a compiler or interpreter for a domain-specific language.

The first of these reasons is somewhat dubious, though something can be said for “knowing your roots”, even in such a hastily changing field as computer science.

Reason “b” is more convincing: Understanding how a compiler is built will allow programmers to get an intuition about what their high-level programs will look like when compiled, and use this intuition to tune programs for better efficiency. Furthermore, the error reports that compilers provide are often easier to understand when one knows about and understands the different phases of compilation, such as knowing the difference between lexical errors, syntax errors, type errors, and so on.

The third reason is also quite valid. In particular, the techniques used for reading (*lexing* and *parsing*) the text of a program and converting this into a form (*abstract syntax*) that is easily manipulated by a computer, can be used to read and manipulate any kind of structured text such as XML documents, address lists, etc.

Reason “d” is becoming more and more important as domain-specific languages (DSLs) are gaining in popularity. A DSL is a (typically small) language designed for a narrow class of problems. Examples are database query languages, text-formatting languages, scene description languages for ray-tracers, and languages for setting up economic simulations. The target language for a compiler for a DSL may be traditional machine code, but it can also be another high-level language for which compilers already exist, a sequence of control signals for a machine, or formatted

text and graphics in some printer-control language (e.g., PostScript), and DSLs are often interpreted instead of compiled. Even so, all DSL compilers and interpreters will have front-ends for reading and analyzing the program text that are similar to those used in compilers and interpreters for general-purpose languages.

In brief, the methods needed to make a compiler front-end are more widely applicable than the methods needed to make a compiler back-end, but the latter is more important for understanding how a program is executed on a machine.

About the Second Edition of the Book

The second edition has been extended with material about optimisations for function calls and loops, and about dataflow analysis, which can be used for various optimisations. This extra material is aimed at advanced BSc-level courses or MSc-level courses.

To the Lecturer

This book was written for use in the introductory compiler course at DIKU, the Department of Computer Science at the University of Copenhagen, Denmark.

At times, standard techniques from compiler construction have been simplified for presentation in this book. In such cases, references are made to books or articles where the full version of the techniques can be found.

The book aims at being “language neutral”. This means two things

- Little detail is given about how the methods in the book can be implemented in any specific language. Rather, the description of the methods is given in the form of algorithm sketches and textual suggestions of how these can be implemented in various types of languages, in particular imperative and functional languages.
- There is no single through-going example of a language to be compiled. Instead, different small (sub-)languages are used in various places to cover exactly the points that the text needs. This is done to avoid drowning in detail, hopefully allowing the readers to “see the wood for the trees”.

Each chapter has a section on further reading, which suggests additional reading material for interested students. Each chapter has a set of exercises. Few of these require access to a computer, but can be solved on paper or blackboard. After some of the sections in the book, a few easy exercises are listed as suggested exercises. It

is recommended that the student attempts to solve these exercises before continuing reading, as the exercises support understanding of the previous sections.

Teaching with this book can be supplemented with project work, where students write simple compilers. Since the book is language neutral, no specific project is given. Instead, the teacher must choose relevant tools and select a project that fits the level of the students and the time available. Depending on the amount of project work and on how much of the advanced material added in the second edition is used, the book can support course sizes ranging from 5–10 ECTS points.

The following link contains extra material for the book, including solutions to selected exercises—<http://www.diku.dk/~torbenm/ICD/>.

Copenhagen, Denmark

Torben Ægidius Mogensen

Acknowledgements

*“Most people return small favors, acknowledge medium ones
and repay greater ones—with ingratitude.”*

Benjamin Franklin (1705–1790)

The author wishes to thank all people who have been helpful in making this book a reality. This includes the students who have been exposed to earlier versions of the book at the compiler courses “Dat1E”, “Oversættelse”, “Implementering af programmeringssprog” and “Advanced Language Processing” at DIKU, and who have found numerous typos and other errors in the earlier versions. I would also like to thank co-teachers and instructors at these courses, who have pointed out places where things were not as clear as they could be.

Copenhagen, Denmark
August 2017

Torben Ægidius Mogensen

Contents

1	Lexical Analysis	1
1.1	Regular Expressions	2
1.1.1	Shorthands	4
1.1.2	Examples	5
1.2	Nondeterministic Finite Automata	7
1.3	Converting a Regular Expression to an NFA	9
1.3.1	Optimisations	10
1.4	Deterministic Finite Automata	12
1.5	Converting an NFA to a DFA	13
1.5.1	Solving Set Equations	14
1.5.2	The Subset Construction	16
1.6	Size Versus Speed	19
1.7	Minimisation of DFAs	20
1.7.1	Example	21
1.7.2	Dead States	23
1.8	Lexers and Lexer Generators	25
1.8.1	Lexer Generators	29
1.9	Properties of Regular Languages	30
1.9.1	Relative Expressive Power	30
1.9.2	Limits to Expressive Power	32
1.9.3	Closure Properties	32
1.10	Further Reading	33
1.11	Exercises	34
	References	38
2	Syntax Analysis	39
2.1	Context-Free Grammars	40
2.1.1	How to Write Context Free Grammars	42
2.2	Derivation	44
2.2.1	Syntax Trees and Ambiguity	45

2.3	Operator Precedence	48
2.3.1	Rewriting Ambiguous Expression Grammars	50
2.4	Other Sources of Ambiguity	53
2.5	Syntax Analysis	54
2.6	Predictive Parsing	54
2.7	<i>Nullable</i> and <i>FIRST</i>	55
2.8	Predictive Parsing Revisited	59
2.9	FOLLOW	61
2.10	A Larger Example	63
2.11	LL(1) Parsing	65
2.11.1	Recursive Descent	66
2.11.2	Table-Driven LL(1) Parsing	68
2.11.3	Conflicts	70
2.12	Rewriting a Grammar for LL(1) Parsing	70
2.12.1	Eliminating Left-Recursion	70
2.12.2	Left-Factorisation	72
2.12.3	Construction of LL(1) Parsers Summarised	73
2.13	SLR Parsing	74
2.14	Constructing SLR Parse Tables	77
2.14.1	Conflicts in SLR Parse-Tables	81
2.15	Using Precedence Rules in LR Parse Tables	82
2.16	Using LR-Parser Generators	84
2.16.1	Conflict Handling in Parser Generators	84
2.16.2	Declarations and Actions	86
2.16.3	Abstract Syntax	86
2.17	Properties of Context-Free Languages	89
2.18	Further Reading	90
2.19	Exercises	91
	References	95
3	Scopes and Symbol Tables	97
3.1	Symbol Tables	98
3.1.1	Implementation of Symbol Tables	98
3.1.2	Simple Persistent Symbol Tables	99
3.1.3	A Simple Imperative Symbol Table	100
3.1.4	Efficiency Issues	101
3.1.5	Shared or Separate Name Spaces	101
3.2	Further Reading	102
3.3	Exercises	102
	Reference	102
4	Interpretation	103
4.1	The Structure of an Interpreter	104
4.2	A Small Example Language	104

4.3	An Interpreter for the Example Language	105
4.3.1	Evaluating Expressions	106
4.3.2	Interpreting Function Calls	108
4.3.3	Interpreting a Program	108
4.4	Advantages and Disadvantages of Interpretation	110
4.5	Further Reading	111
4.6	Exercises	111
	References	113
5	Type Checking	115
5.1	The Design Space of Type Systems	115
5.2	Attributes	117
5.3	Environments for Type Checking	117
5.4	Type Checking of Expressions	118
5.5	Type Checking of Function Declarations	120
5.6	Type Checking a Program	121
5.7	Advanced Type Checking	122
5.8	Further Reading	124
5.9	Exercises	125
	References	126
6	Intermediate-Code Generation	127
6.1	Designing an Intermediate Language	128
6.2	The Intermediate Language	129
6.3	Syntax-Directed Translation	131
6.4	Generating Code from Expressions	131
6.4.1	Examples of Translation	135
6.5	Translating Statements	136
6.6	Logical Operators	138
6.6.1	Sequential Logical Operators	140
6.7	Advanced Control Statements	142
6.8	Translating Structured Data	143
6.8.1	Floating-Point Values	144
6.8.2	Arrays	144
6.8.3	Strings	149
6.8.4	Records/Structs and Unions	149
6.9	Translation of Declarations	150
6.9.1	Simple Local Declarations	151
6.9.2	Translation of Function Declarations	151
6.10	Further Reading	152
6.11	Exercises	152
	References	155

7	Machine-Code Generation	157
7.1	Conditional Jumps	158
7.2	Constants	159
7.3	Exploiting Complex Instructions	159
7.3.1	Two-Address Instructions	163
7.4	Optimisations	164
7.5	Further Reading	166
7.6	Exercises	166
	References	167
8	Register Allocation	169
8.1	Liveness	170
8.2	Liveness Analysis	171
8.3	Interference	174
8.4	Register Allocation by Graph Colouring	176
8.5	Spilling	177
8.6	Heuristics	179
8.6.1	Removing Redundant Moves	180
8.6.2	Using Explicit Register Numbers	181
8.7	Further Reading	182
8.8	Exercises	182
	References	184
9	Functions	185
9.1	The Call Stack	185
9.2	Activation Records	186
9.3	Prologues, Epilogues and Call-Sequences	187
9.4	Letting the Callee Save Registers	190
9.5	Caller-Saves Versus Callee-Saves	191
9.6	Using Registers to Pass Parameters	192
9.7	Interaction with the Register Allocator	194
9.8	Local Variables	196
9.9	Accessing Non-local Variables	196
9.9.1	Global Variables	197
9.9.2	Call-by-Reference Parameters	198
9.10	Functions as Parameters	199
9.11	Variants	199
9.11.1	Variable-Sized Frames	199
9.11.2	Variable Number of Parameters	200
9.11.3	Direction of Stack-Growth and Position of <i>FP</i>	200
9.11.4	Register Stacks	200
9.12	Optimisations for Function Calls	201
9.12.1	Inlining	201
9.12.2	Tail-Call Optimisation	203

9.13	Further Reading	207
9.14	Exercises	207
	References	209
10	Data-Flow Analysis and Optimisation	211
10.1	Data-Flow Analysis	211
10.2	How to Design a Data-Flow Analysis	212
10.3	Liveness Analysis	212
	10.3.1 Improving Liveness Analysis	213
10.4	Generalising from Liveness Analysis	214
10.5	Common Subexpression Elimination	215
	10.5.1 Available Assignments	215
	10.5.2 Example of Available-Assignments Analysis	217
	10.5.3 Using Available Assignment Analysis for Common Subexpression Elimination	220
10.6	Index-Check Elimination	221
10.7	Jump-to-Jump Elimination	224
10.8	Resources Used by Data-Flow Analysis	225
10.9	Pointer Analysis	227
10.10	Limitations of Data-Flow Analyses	231
10.11	Further Reading	232
10.12	Exercises	232
	References	233
11	Optimisations for Loops	235
11.1	Loops	235
11.2	Code Hoisting	236
11.3	Memory Prefetching	238
11.4	Incrementalisation	240
	11.4.1 Rules for Incrementalisation	242
11.5	Further Reading	244
11.6	Exercises	244
	Reference	245
	Appendix A: Set Notation and Concepts	247
	Index	255

List of Figures

1.1	Regular expressions and their derivation.	3
1.2	Some algebraic properties of regular expressions	4
1.3	Example of an NFA	8
1.4	Constructing NFA fragments from regular expressions.	10
1.5	NFA for the regular expression $(a b)^*ac$	11
1.6	Optimised NFA construction for regular expression shorthands	11
1.7	Optimised NFA for $[0-9]^+$	12
1.8	Example of a DFA.	12
1.9	DFA constructed from the NFA in Fig. 1.5	19
1.10	Non-minimal DFA	21
1.11	Minimal DFA.	23
1.12	Combined NFA for several tokens	26
1.13	Combined DFA for several tokens	27
1.14	A 4-state NFA that gives 15 DFA states	31
2.1	From regular expressions to context free grammars	42
2.2	Simple expression grammar	43
2.3	Simple statement grammar	43
2.4	Example grammar.	45
2.5	Derivation of the string <code>aabbbcc</code> using Grammar 2.4	45
2.6	Leftmost derivation of the string <code>aabbbcc</code> using Grammar 2.4	45
2.7	Syntax tree for the string <code>aabbbcc</code> using Grammar 2.4	46
2.8	Alternative syntax tree for the string <code>aabbbcc</code> using Grammar 2.4.	47
2.9	Unambiguous version of Grammar 2.4	47
2.10	Fully reduced tree for the syntax tree in Fig. 2.7	48
2.11	Preferred syntax tree for $2+3*4$ using Grammar 2.2, and the corresponding fully reduced tree.	49
2.12	Unambiguous expression grammar.	52
2.13	Syntax tree for $2+3*4$ using Grammar 2.12 , and the corresponding fully reduced tree	52

2.14	Unambiguous grammar for statements	53
2.15	Fixed-point iteration for calculation of <i>Nullable</i>	57
2.16	Fixed-point iteration for calculation of <i>FIRST</i>	58
2.17	Recursive descent parser for Grammar 2.9	66
2.18	Tree-building recursive descent parser for Grammar 2.9.	67
2.19	LL(1) table for Grammar 2.9	68
2.20	Program for table-driven LL(1) parsing	68
2.21	Input and stack during table-driven LL(1) parsing	69
2.22	Tree-building program for table-driven LL(1) parsing	69
2.23	Removing left-recursion from Grammar 2.12	72
2.24	Left-factorised grammar for conditionals	73
2.25	Example shift-reduce parsing	75
2.26	SLR table for Grammar 2.9	76
2.27	Algorithm for SLR parsing	77
2.28	Example SLR parsing	77
2.29	Example grammar for SLR-table construction	78
2.30	NFAs for the productions in Grammar 2.29	78
2.31	Combined NFA for Grammar 2.29: Epsilon transitions are added, and A is the only start state.	79
2.32	DFA constructed from the NFA in Fig. 2.31	79
2.33	DFA table for Grammar 2.9, equivalent to the DFA in Fig. 2.32	80
2.34	Summary of SLR parse-table construction	80
2.35	Textual representation of NFA states	85
4.1	Example language for interpretation	105
4.2	Evaluating expressions	107
4.3	Evaluating a function call	109
4.4	Interpreting a program	109
5.1	The design space of type systems	116
5.2	Type checking of expressions	119
5.3	Type checking a function declaration	121
5.4	Type checking a program	122
6.1	The intermediate language	130
6.2	A simple expression language	132
6.3	Translating an expression	134
6.4	Statement language	136
6.5	Translation of statements	137
6.6	Translation of simple conditions	137
6.7	Example language with logical operators	140
6.8	Translation of sequential logical operators	141
6.9	Translation for one-dimensional arrays	145
6.10	A two-dimensional array	146
6.11	Translation of multi-dimensional arrays	148
6.12	Translation of simple declarations	151
7.1	Pattern/replacement pairs for a subset of the MIPS instruction set	162

8.1	Example program for liveness analysis and register allocation	171
8.2	Gen and kill sets.	172
8.3	<i>succ</i> , <i>gen</i> and <i>kill</i> for the program in Fig. 8.1.	173
8.4	Fixed-point iteration for liveness analysis.	174
8.5	Interference graph for the program in Fig. 8.1	175
8.6	Algorithm 8.3 applied to the graph in Fig. 8.5	178
8.7	Program from Fig. 8.1 after spilling variable <i>a</i>	179
8.8	Interference graph for the program in Fig. 8.7	179
8.9	Colouring of the graph in Fig. 8.8	179
9.1	Simple activation record layout.	187
9.2	Prologue for the header $f(p_1, \dots, p_m)$ using the frame layout shown in Fig. 9.1	188
9.3	Epilogue for the instruction RETURN <i>result</i> using the frame layout shown in Fig. 9.1.	189
9.4	Call sequence for $x := \text{CALL } g(a_1, \dots, a_n)$ using the frame layout shown in Fig. 9.1.	188
9.5	Activation record layout for callee-saves	190
9.6	Prologue for the header $f(p_1, \dots, p_m)$ using callee-saves	190
9.7	Epilogue for the instruction RETURN <i>result</i> using callee-saves	190
9.8	Call sequence for $x := \text{CALL } g(a_1, \dots, a_n)$ using callee-saves	190
9.9	Possible division of registers for a 16-register architecture	191
9.10	Activation record layout for the register division shown in Fig. 9.9.	192
9.11	Prologue for the header $f(p_1, \dots, p_m)$ using the register division shown in Fig. 9.9	193
9.12	Epilogue for the instruction RETURN <i>result</i> using the register division shown in Fig. 9.9	193
9.13	Call sequence for $x := \text{CALL } g(a_1, \dots, a_n)$ using the register division shown in Fig. 9.9	194
9.14	Variable capture when inlining.	202
9.15	Renaming variables when inlining	202
9.16	Recursive inlining.	202
10.1	Gen and kill sets for available assignments	216
10.2	Example code for available-assignments analysis	218
10.3	<i>pred</i> , <i>gen</i> and <i>kill</i> for the program in Fig. 10.2	218
10.4	Fixed-point iteration for available-assignment analysis	219
10.5	The program in Fig. 10.2 after common subexpression elimination	220
10.6	<i>gen</i> and <i>kill</i> sets for index-check elimination	223
10.7	Intermediate code for for-loop with index check	224
10.8	Equations for pointer analysis.	229
11.1	Incrementalisation of nested loop	241
11.2	Eliminating weakly dead variables	242