

# Contents

<b>Preface</b>	xiii
<b>1 Introduction</b>	1
1.1 Why study compiler construction?	4
1.1.1 Compiler construction is very successful	6
1.1.2 Compiler construction has a wide applicability	8
1.1.3 Compilers contain generally useful algorithms	8
1.2 A simple traditional modular compiler/interpreter	9
1.2.1 The abstract syntax tree	9
1.2.2 Structure of the demo compiler	11
1.2.3 The language for the demo compiler	12
1.2.4 Lexical analysis for the demo compiler	13
1.2.5 Syntax analysis for the demo compiler	14
1.2.6 Context handling for the demo compiler	19
1.2.7 Code generation for the demo compiler	19
1.2.8 Interpretation for the demo compiler	21
1.3 The structure of a more realistic compiler	22
1.3.1 The structure	22
1.3.2 Run-time systems	24
1.3.3 Short-cuts	25
1.4 Compiler architectures	25
1.4.1 The width of the compiler	26
1.4.2 Who's the boss?	27
1.5 Properties of a good compiler	29

1.6	Portability and retargetability	32
1.7	Place and usefulness of optimizations	32
1.8	A short history of compiler construction	33
1.8.1	1945–1960: code generation	33
1.8.2	1960–1975: parsing	33
1.8.3	1975–present: code generation and code optimization; paradigms	34
1.9	Grammars	34
1.9.1	The form of a grammar	35
1.9.2	The production process	35
1.9.3	Extended forms of grammars	37
1.9.4	Properties of grammars	38
1.9.5	The grammar formalism	38
1.10	Closure algorithms	40
1.10.1	An iterative implementation of the closure algorithm	44
1.11	The outline code used in this book	45
1.12	Conclusion	47
	Summary	47
	Further reading	48
	Exercises	49
<b>2</b>	<b>From program text to abstract syntax tree</b>	<b>52</b>
2.1	From program text to tokens – the lexical structure	56
2.1.1	Reading the program text	56
2.1.2	Lexical versus syntactic analysis	57
2.1.3	Regular expressions and regular descriptions	58
2.1.4	Lexical analysis	60
2.1.5	Creating a lexical analyzer by hand	61
2.1.6	Creating a lexical analyzer automatically	68
2.1.7	Transition table compression	86
2.1.8	Error handling in lexical analyzers	93
2.1.9	A traditional lexical analyzer generator – lex	94
2.1.10	Lexical identification of tokens	96
2.1.11	Symbol tables	98
2.1.12	Macro processing and file inclusion	103
2.1.13	Conclusion	109
2.2	From tokens to syntax tree – the syntax	110
2.2.1	Two classes of parsing methods	111
2.2.2	Error detection and error recovery	115
2.2.3	Creating a top-down parser manually	117
2.2.4	Creating a top-down parser automatically	120

2.2.5	Creating a bottom-up parser automatically	150
2.3	Conclusion	179
	Summary	181
	Further reading	184
	Exercises	185
<b>3</b>	<b>Annotating the abstract syntax tree – the context</b>	<b>194</b>
3.1	Attribute grammars	195
3.1.1	Dependency graphs	200
3.1.2	Attribute evaluation	202
3.1.3	Cycle handling	210
3.1.4	Attribute allocation	217
3.1.5	Multi-visit attribute grammars	218
3.1.6	Summary of the types of attribute grammars	229
3.1.7	L-attributed grammars	230
3.1.8	S-attributed grammars	235
3.1.9	Equivalence of L-attributed and S-attributed grammars	235
3.1.10	Extended grammar notations and attribute grammars	237
3.1.11	Conclusion	238
3.2	Manual methods	238
3.2.1	Threading the AST	239
3.2.2	Symbolic interpretation	245
3.2.3	Data-flow equations	253
3.2.4	Interprocedural data-flow analysis	260
3.2.5	Carrying the information upstream – live analysis	262
3.2.6	Comparing symbolic interpretation and data-flow equations	267
3.3	Conclusion	269
	Summary	269
	Further reading	273
	Exercises	274
<b>4</b>	<b>Processing the intermediate code</b>	<b>279</b>
4.1	Interpretation	281
4.1.1	Recursive interpretation	281
4.1.2	Iterative interpretation	285
4.2	Code generation	290
4.2.1	Avoiding code generation altogether	295
4.2.2	The starting point	296
4.2.3	Trivial code generation	297
4.2.4	Simple code generation	302

4.2.5	Code generation for basic blocks	320
4.2.6	BURS code generation and dynamic programming	337
4.2.7	Register allocation by graph coloring	357
4.2.8	Supercompilation	363
4.2.9	Evaluation of code generation techniques	364
4.2.10	Debugging of code optimizers	365
4.2.11	Preprocessing the intermediate code	366
4.2.12	Postprocessing the target code	371
4.2.13	Machine code generation	374
4.3	Assemblers, linkers, and loaders	375
4.3.1	Assembler design issues	378
4.3.2	Linker design issues	381
4.4	Conclusion	383
	Summary	383
	Further reading	389
	Exercises	389
<b>5</b>	<b>Memory management</b>	<b>396</b>
5.1	Data allocation with explicit deallocation	398
5.1.1	Basic memory allocation	399
5.1.2	Linked lists	403
5.1.3	Extensible arrays	404
5.2	Data allocation with implicit deallocation	407
5.2.1	Basic garbage collection algorithms	407
5.2.2	Preparing the ground	409
5.2.3	Reference counting	415
5.2.4	Mark and scan	420
5.2.5	Two-space copying	425
5.2.6	Compaction	428
5.2.7	Generational garbage collection	429
5.3	Conclusion	431
	Summary	431
	Further reading	434
	Exercises	435
<b>6</b>	<b>Imperative and object-oriented programs</b>	<b>438</b>
6.1	Context handling	440
6.1.1	Identification	441
6.1.2	Type checking	449
6.1.3	Conclusion	460

6.2	Source language data representation and handling	460
6.2.1	Basic types	460
6.2.2	Enumeration types	461
6.2.3	Pointer types	461
6.2.4	Record types	465
6.2.5	Union types	466
6.2.6	Array types	467
6.2.7	Set types	470
6.2.8	Routine types	471
6.2.9	Object types	471
6.2.10	Interface types	480
6.3	Routines and their activation	481
6.3.1	Activation records	482
6.3.2	Routines	485
6.3.3	Operations on routines	486
6.3.4	Non-nested routines	489
6.3.5	Nested routines	491
6.3.6	Lambda lifting	499
6.3.7	Iterators and coroutines	500
6.4	Code generation for control flow statements	501
6.4.1	Local flow of control	502
6.4.2	Routine invocation	512
6.4.3	Run-time error handling	519
6.5	Code generation for modules	523
6.5.1	Name generation	524
6.5.2	Module initialization	524
6.5.3	Code generation for generics	525
6.6	Conclusion	527
	Summary	528
	Further reading	531
	Exercises	532
<b>7</b>	<b>Functional programs</b>	<b>538</b>
7.1	A short tour of Haskell	540
7.1.1	Offside rule	540
7.1.2	Lists	541
7.1.3	List comprehension	542
7.1.4	Pattern matching	543
7.1.5	Polymorphic typing	543
7.1.6	Referential transparency	544
7.1.7	Higher-order functions	545
7.1.8	Lazy evaluation	547

7.2	Compiling functional languages	548
7.2.1	The functional core	549
7.3	Polymorphic type checking	551
7.3.1	Polymorphic function application	551
7.4	Desugaring	553
7.4.1	The translation of lists	553
7.4.2	The translation of pattern matching	553
7.4.3	The translation of list comprehension	557
7.4.4	The translation of nested functions	558
7.5	Graph reduction	560
7.5.1	Reduction order	564
7.5.2	The reduction engine	566
7.6	Code generation for functional core programs	568
7.6.1	Avoiding the construction of some application spines	573
7.7	Optimizing the functional core	575
7.7.1	Strictness analysis	575
7.7.2	Boxing analysis	582
7.7.3	Tail calls	582
7.7.4	Accumulator transformation	584
7.7.5	Limitations	587
7.8	Advanced graph manipulation	587
7.8.1	Variable-length nodes	587
7.8.2	Pointer tagging	588
7.8.3	Aggregate node allocation	588
7.8.4	Vector apply nodes	588
7.9	Conclusion	589
	Summary	589
	Further reading	592
	Exercises	593
<b>8</b>	<b>Logic programs</b>	<b>596</b>
8.1	The logic programming model	598
8.1.1	The building blocks	598
8.1.2	The inference mechanism	600
8.2	The general implementation model, interpreted	601
8.2.1	The interpreter instructions	603
8.2.2	Avoiding redundant goal lists	606
8.2.3	Avoiding copying goal list tails	606
8.3	Unification	607
8.3.1	Unification of structures, lists, and sets	607

8.3.2	The implementation of unification	609
8.3.3	Unification of two unbound variables	612
8.3.4	Conclusion	614
8.4	The general implementation model, compiled	615
8.4.1	List procedures	616
8.4.2	Compiled clause search and unification	619
8.4.3	Optimized clause selection in the WAM	623
8.4.4	Implementing the 'cut' mechanism	628
8.4.5	Implementing the predicates <code>assert</code> and <code>retract</code>	630
8.5	Compiled code for unification	634
8.5.1	Unification instructions in the WAM	636
8.5.2	Deriving a unification instruction by manual partial evaluation	638
8.5.3	Unification of structures in the WAM	639
8.5.4	An optimization: read/write mode	646
8.5.5	Further unification optimizations in the WAM	648
8.5.6	Conclusion	650
	Summary	650
	Further reading	653
	Exercises	653
<b>9</b>	<b>Parallel and distributed programs</b>	<b>656</b>
9.1	Parallel programming models	659
9.1.1	Shared variables and monitors	659
9.1.2	Message passing models	661
9.1.3	Object-oriented languages	663
9.1.4	The Linda Tuple space	664
9.1.5	Data-parallel languages	665
9.2	Processes and threads	667
9.3	Shared variables	668
9.3.1	Locks	669
9.3.2	Monitors	669
9.4	Message passing	671
9.4.1	Locating the receiver	672
9.4.2	Marshaling	672
9.4.3	Type checking of messages	673
9.4.4	Message selection	674
9.5	Parallel object-oriented languages	674
9.5.1	Object location	675
9.5.2	Object migration	676
9.5.3	Object replication	677

9.6	Tuple space	678
9.6.1	Avoiding the overhead of associative addressing	679
9.6.2	Distributed implementations of the tuple space	682
9.7	Automatic parallelization	684
9.7.1	Exploiting parallelism automatically	685
9.7.2	Data dependencies	686
9.7.3	Loop transformations	688
9.7.4	Automatic parallelization for distributed-memory machines	690
9.8	Conclusion	693
	Summary	693
	Further reading	695
	Exercises	695

**Appendix A – A simple object-oriented compiler/interpreter** 699

A.1	Syntax-determined classes and semantics-determining methods	699
A.2	The simple object-oriented compiler	701
A.3	Object-oriented parsing	702
A.4	Evaluation	708
	Exercise	708

**Answers to exercises** 709

**References** 720

**Index** 728