# Contents

**Preface**  

1. **Introduction**  

1.1 Why study compiler construction?  

1.1.1 Compiler construction is very successful  

1.1.2 Compiler construction has a wide applicability  

1.1.3 Compilers contain generally useful algorithms  

1.2 A simple traditional modular compiler/interpreter  

1.2.1 The abstract syntax tree  

1.2.2 Structure of the demo compiler  

1.2.3 The language for the demo compiler  

1.2.4 Lexical analysis for the demo compiler  

1.2.5 Syntax analysis for the demo compiler  

1.2.6 Context handling for the demo compiler  

1.2.7 Code generation for the demo compiler  

1.2.8 Interpretation for the demo compiler  

1.3 The structure of a more realistic compiler  

1.3.1 The structure  

1.3.2 Run-time systems  

1.3.3 Short-cuts  

1.4 Compiler architectures  

1.4.1 The width of the compiler  

1.4.2 Who’s the boss?  

1.5 Properties of a good compiler
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

2 From program text to abstract syntax tree 52
  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
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.5</td>
<td>Creating a bottom-up parser automatically</td>
<td>150</td>
</tr>
<tr>
<td>2.3</td>
<td>Conclusion</td>
<td>179</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>181</td>
</tr>
<tr>
<td></td>
<td>Further reading</td>
<td>184</td>
</tr>
<tr>
<td></td>
<td>Exercises</td>
<td>185</td>
</tr>
<tr>
<td>3</td>
<td>Annotating the abstract syntax tree – the context</td>
<td>194</td>
</tr>
<tr>
<td>3.1</td>
<td>Attribute grammars</td>
<td>195</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Dependency graphs</td>
<td>200</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Attribute evaluation</td>
<td>202</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Cycle handling</td>
<td>210</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Attribute allocation</td>
<td>217</td>
</tr>
<tr>
<td>3.1.5</td>
<td>Multi-visit attribute grammars</td>
<td>218</td>
</tr>
<tr>
<td>3.1.6</td>
<td>Summary of the types of attribute grammars</td>
<td>229</td>
</tr>
<tr>
<td>3.1.7</td>
<td>L-attributed grammars</td>
<td>230</td>
</tr>
<tr>
<td>3.1.8</td>
<td>S-attributed grammars</td>
<td>235</td>
</tr>
<tr>
<td>3.1.9</td>
<td>Equivalence of L-attributed and S-attributed grammars</td>
<td>235</td>
</tr>
<tr>
<td>3.1.10</td>
<td>Extended grammar notations and attribute grammars</td>
<td>237</td>
</tr>
<tr>
<td>3.1.11</td>
<td>Conclusion</td>
<td>238</td>
</tr>
<tr>
<td>3.2</td>
<td>Manual methods</td>
<td>238</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Threading the AST</td>
<td>239</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Symbolic interpretation</td>
<td>245</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Data-flow equations</td>
<td>253</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Interprocedural data-flow analysis</td>
<td>260</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Carrying the information upstream – live analysis</td>
<td>262</td>
</tr>
<tr>
<td>3.2.6</td>
<td>Comparing symbolic interpretation and data-flow equations</td>
<td>267</td>
</tr>
<tr>
<td>3.3</td>
<td>Conclusion</td>
<td>269</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>269</td>
</tr>
<tr>
<td></td>
<td>Further reading</td>
<td>273</td>
</tr>
<tr>
<td></td>
<td>Exercises</td>
<td>274</td>
</tr>
<tr>
<td>4</td>
<td>Processing the intermediate code</td>
<td>279</td>
</tr>
<tr>
<td>4.1</td>
<td>Interpretation</td>
<td>281</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Recursive interpretation</td>
<td>281</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Iterative interpretation</td>
<td>285</td>
</tr>
<tr>
<td>4.2</td>
<td>Code generation</td>
<td>290</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Avoiding code generation altogether</td>
<td>295</td>
</tr>
<tr>
<td>4.2.2</td>
<td>The starting point</td>
<td>296</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Trivial code generation</td>
<td>297</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Simple code generation</td>
<td>302</td>
</tr>
</tbody>
</table>
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

7 Functional programs 538
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
## Contents

### 7.2 Compiling functional languages
- 7.2.1 The functional core

### 7.3 Polymorphic type checking
- 7.3.1 Polymorphic function application

### 7.4 Desugaring
- 7.4.1 The translation of lists
- 7.4.2 The translation of pattern matching
- 7.4.3 The translation of list comprehension
- 7.4.4 The translation of nested functions

### 7.5 Graph reduction
- 7.5.1 Reduction order
- 7.5.2 The reduction engine

### 7.6 Code generation for functional core programs
- 7.6.1 Avoiding the construction of some application spines

### 7.7 Optimizing the functional core
- 7.7.1 Strictness analysis
- 7.7.2 Boxing analysis
- 7.7.3 Tail calls
- 7.7.4 Accumulator transformation
- 7.7.5 Limitations

### 7.8 Advanced graph manipulation
- 7.8.1 Variable-length nodes
- 7.8.2 Pointer tagging
- 7.8.3 Aggregate node allocation
- 7.8.4 Vector apply nodes

### 7.9 Conclusion
- Summary
- Further reading
- Exercises

### 8 Logic programs
- 8.1 The logic programming model
- 8.1.1 The building blocks
- 8.1.2 The inference mechanism
- 8.2 The general implementation model, interpreted
- 8.2.1 The interpreter instructions
- 8.2.2 Avoiding redundant goal lists
- 8.2.3 Avoiding copying goal list tails
- 8.3 Unification
- 8.3.1 Unification of structures, lists, and sets
8.3.2 The implementation of unification
8.3.3 Unification of two unbound variables
8.3.4 Conclusion

8.4 The general implementation model, compiled
8.4.1 List procedures
8.4.2 Compiled clause search and unification
8.4.3 Optimized clause selection in the WAM
8.4.4 Implementing the ‘cut’ mechanism
8.4.5 Implementing the predicates assert and retract

8.5 Compiled code for unification
8.5.1 Unification instructions in the WAM
8.5.2 Deriving a unification instruction by manual partial evaluation
8.5.3 Unification of structures in the WAM
8.5.4 An optimization: read/write mode
8.5.5 Further unification optimizations in the WAM
8.5.6 Conclusion

Summary
Further reading
Exercises

9 Parallel and distributed programs

9.1 Parallel programming models
9.1.1 Shared variables and monitors
9.1.2 Message passing models
9.1.3 Object-oriented languages
9.1.4 The Linda Tuple space
9.1.5 Data-parallel languages

9.2 Processes and threads

9.3 Shared variables
9.3.1 Locks
9.3.2 Monitors

9.4 Message passing
9.4.1 Locating the receiver
9.4.2 Marshaling
9.4.3 Type checking of messages
9.4.4 Message selection

9.5 Parallel object-oriented languages
9.5.1 Object location
9.5.2 Object migration
9.5.3 Object replication
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.6</td>
<td>Tuple space</td>
<td>678</td>
</tr>
<tr>
<td>9.6.1</td>
<td>Avoiding the overhead of associative addressing</td>
<td>679</td>
</tr>
<tr>
<td>9.6.2</td>
<td>Distributed implementations of the tuple space</td>
<td>682</td>
</tr>
<tr>
<td>9.7</td>
<td>Automatic parallelization</td>
<td>684</td>
</tr>
<tr>
<td>9.7.1</td>
<td>Exploiting parallelism automatically</td>
<td>685</td>
</tr>
<tr>
<td>9.7.2</td>
<td>Data dependencies</td>
<td>686</td>
</tr>
<tr>
<td>9.7.3</td>
<td>Loop transformations</td>
<td>688</td>
</tr>
<tr>
<td>9.7.4</td>
<td>Automatic parallelization for distributed-memory machines</td>
<td>690</td>
</tr>
<tr>
<td>9.8</td>
<td>Conclusion</td>
<td>693</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>693</td>
</tr>
<tr>
<td></td>
<td>Further reading</td>
<td>695</td>
</tr>
<tr>
<td></td>
<td>Exercises</td>
<td>695</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>Syntax-determined classes and semantics-determining methods</td>
<td>699</td>
</tr>
<tr>
<td>A.2</td>
<td>The simple object-oriented compiler</td>
<td>701</td>
</tr>
<tr>
<td>A.3</td>
<td>Object-oriented parsing</td>
<td>702</td>
</tr>
<tr>
<td>A.4</td>
<td>Evaluation</td>
<td>708</td>
</tr>
<tr>
<td></td>
<td>Exercise</td>
<td>708</td>
</tr>
</tbody>
</table>

**Answers to exercises**

**References**

**Index**