

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

21

F. L. Bauer · F. L. De Remer · M. Griffiths
U. Hill · J. J. Horning · C. H. A. Koster
W. M. McKeeman · P. C. Poole · W. M. Waite

Compiler Construction

An Advanced Course

Edited by F. L. Bauer and J. Eickel

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

21

F. L. Bauer · F. L. De Remer · M. Griffiths
U. Hill · J. J. Horning · C. H. A. Koster
W. M. McKeeman · P. C. Poole · W. M. Waite

Compiler Construction

An Advanced Course

Edited by F. L. Bauer and J. Eickel



Springer-Verlag
Berlin · Heidelberg · New York 1974

Editorial Board: P. Brinch Hansen · D. Gries
C. Moler · G. Seegmüller · N. Wirth

Prof. Dr. F. L. Bauer
Prof. Dr. J. Eickel
Mathematisches Institut
der TU München
8 München 2
Arcisstraße 21
BRD

AMS Subject Classifications (1970): 00A10, 68-02, 69-02
CR Subject Classifications (1974): 4.12

ISBN 3-540-06958-5 Springer-Verlag Berlin · Heidelberg · New York
ISBN 0-387-06958-5 Springer-Verlag New York · Heidelberg · Berlin

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically those of translation, reprinting, re-use of illustrations, broadcasting, reproduction by photocopying machine or similar means, and storage in data banks.

Under § 54 of the German Copyright Law where copies are made for other than private use, a fee is payable to the publisher, the amount of the fee to be determined by agreement with the publisher.

© by Springer-Verlag Berlin · Heidelberg 1974 Printed in Germany.

Offsetdruck: Julius Beltz, Hemsbach/Bergstr.

Contents

CHAPTER 1.: INTRODUCTION

W. M. McKeeman	COMPILER CONSTRUCTION	1
	1. Definitions	1
	1.1. Source and Target Languages	1
	1.2. Implementation Languages	2
	1.3. Language Defining Languages	3
	2. Recursive Descent Compilation	4
	2.1. Introduction	4
	2.2. Writing a Recursive Descent Compiler	7
	2.3. Executing the Compiler	12
	2.4. Extending the Technique	14
	3. Modularization	15
	3.1. Modular Documentation	15
	3.2. Modular Programmer Assignment	15
	3.3. Modular Source Text	16
	3.4. Modular Target Text	16
	4. Intermodular Communication	16
	4.1. Specification	16
	4.2. "Need to Know"	18
	4.3. Test Environment	19
	4.4. Feedback-Free	19
	5. Vertical Fragmentation	20
	5.1. The Transformation $PT \rightarrow AST$	22
	5.2. The Transformation $AST \rightarrow ST$	24
	5.3. The Transformation $ST \rightarrow ACT$	24
	5.4. The Transformation $ACT \rightarrow ADT$	26
	5.5. The Transformation $ADT \rightarrow SET$	27
	5.6. The Transformation $SET \rightarrow SCT$	27
	6. Horizontal Fragmentation	28
	7. Equivalence Transformation	29
	8. Evaluation	34
	9. References	36

CHAPTER 2.: ANALYSIS

F. L. DeRemer	REVIEW OF FORMALISMS AND NOTATION	37
	1. Terminology and Definitions of Grammars	37
	1.1. Unrestricted Rewriting Systems	37
	1.2. The Chomsky Hierarchy	38
	1.3. Phrase Structure Implied by Context-Free Grammars	39
	1.4. Regular Grammars and Regular Expressions	41
	2. Parsing	42
	2.1. Syntactic Dominoes	42
	2.2. Parsing Strategies	44
	2.3. Ambiguity	45
	2.4. Debugging a Grammar	45
	3. Machines, Computational Complexity, Relation to Grammars	46

VI

4.	<i>Transduction Grammars</i>	46
4.1.	<i>String-To-String Grammars</i>	47
4.2.	<i>String-To-Tree Grammars</i>	48
5.	<i>Meta-Grammars</i>	50
5.1.	<i>Self-Describing Grammars</i>	50
5.2.	<i>Practical Applications</i>	53
	<i>References</i>	56

M. Griffiths

LL(1) GRAMMARS AND ANALYSERS 57

1.	<i>Introduction</i>	57
1.1.	<i>Predictive Analysis</i>	58
1.2.	<i>Efficiency</i>	60
1.3.	<i>Semantics</i>	61
2.	<i>LL(1) Conditions</i>	62
3.	<i>Decision Algorithm</i>	64
4.	<i>Production of Analyser</i>	67
5.	<i>Grammar Transformation</i>	70
5.1.	<i>Elimination of Left Recursion</i>	70
5.2.	<i>Factorisation and Substitution</i>	73
5.2.1.	<i>Ordering</i>	74
6.	<i>Semantic Insertion</i>	75
6.1.	<i>Generation of Postfixed Notation</i>	76
6.2.	<i>Symbol Table Insertion</i>	78
6.3.	<i>Interpretation of Postfixed Expressions</i>	79
7.	<i>Generalisation and Conclusion</i>	80
7.1.	<i>Vienna Notation</i>	80
7.2.	<i>LL(k) Grammars</i>	81
8.	<i>Practical Results</i>	82
	<i>References</i>	83

J. J. Horning

LR GRAMMARS AND ANALYSERS 85

1.	<i>Intuitive Description</i>	85
1.1.	<i>Definition of LR(k)</i>	86
1.2.	<i>Items</i>	86
1.3.	<i>States</i>	87
2.	<i>Interpreting LR Tables</i>	88
2.1.	<i>Form of Entries</i>	88
2.2.	<i>Example</i>	90
3.	<i>Constructing LR Tables</i>	93
3.1.	<i>The LR(0) Constructor Algorithm</i>	93
3.1.1.	<i>Initialization</i>	94
3.1.2.	<i>Closure</i>	94
3.1.3.	<i>Successor States</i>	94
3.1.4.	<i>Accessible States</i>	94
3.1.5.	<i>Deriving the Parsing Action Table</i>	95
3.2.	<i>Adding Lookahead</i>	96
3.2.1.	<i>Using the Follower Matrix</i>	96
3.2.2.	<i>Using the Shift Entries</i>	97
3.2.3.	<i>Adding Context to Items</i>	97
4.	<i>Representing LR Tables</i>	98
4.1.	<i>Matrix Forms</i>	98
4.2.	<i>List Form</i>	98
4.3.	<i>Efficiency Transformations</i>	99
4.3.1.	<i>LR(0) Reduce States</i>	99
4.3.2.	<i>Column Regularities</i>	100
4.3.3.	<i>Row Regularities</i>	100
4.3.4.	<i>Don't Cares</i>	100
4.3.5.	<i>Column Reduction</i>	101

VII

4.3.6.	<i>Row Reduction</i>	101
4.3.7.	<i>List Overlapping</i>	101
4.3.8.	<i>Matrix Factoring</i>	101
4.3.9.	<i>Eliminating Single Productions</i>	101
5.	<i>Properties of LR Grammars and Analysers</i>	102
6.	<i>Modification to Obtain LR Grammars</i>	103
6.1.	<i>Multiple-Use Separators</i>	103
6.2.	<i>Compound Terminal Symbols</i>	103
7.	<i>Comparison with other Techniques</i>	104
7.1.	<i>Grammar Inclusions</i>	104
7.2.	<i>Language Inclusions</i>	105
7.3.	<i>Error Detection</i>	106
7.4.	<i>Efficiency</i>	106
8.	<i>Choice of a Syntactic Analysis Technique</i>	106
	<i>References</i>	107

F. L. DeRemer

LEXICAL ANALYSIS 109

1.	<i>Scanning, Then Screening</i>	109
2.	<i>Screening</i>	110
3.	<i>Scanning</i>	111
3.1.	<i>Lexical Grammars</i>	111
3.1.1.	<i>Tokens</i>	112
3.1.2.	<i>A Regularity Condition</i>	112
3.1.3.	<i>Converting Regular Grammars to Regular Expressions</i>	113
3.2.	<i>Generating Scanners Via LR Tech- niques</i>	113
3.2.1.	<i>Using a Simplified LR Parser as a Scanner</i>	113
3.3.	<i>Hand-Written Scanners</i>	117
3.4.	<i>Error Recovery</i>	119
4.	<i>On Not Including "Conversion Rou- tines" in Lexical Analysers</i>	119
	<i>References</i>	120

F. L. DeRemer

TRANSFORMATIONAL GRAMMARS 121

1.	<i>Language Processing as Tree Manipulation</i>	121
1.1.	<i>Lexical and Syntactical Processing</i>	123
1.2.	<i>Standardization</i>	123
1.3.	<i>Flatting</i>	124
2.	<i>Description of Subtree Transfor- mational Grammars</i>	125
3.	<i>Compiler Structure</i>	128
4.	<i>Further Examples of Transformations</i>	129
4.1.	<i>Local Effects</i>	129
4.2.	<i>Global Effects</i>	129
4.3.	<i>Iterative Transformations</i>	131
4.4.	<i>Extension to Regular Expressions</i>	133
5.	<i>Summary and Conclusions</i>	136
6.	<i>Appendix - Meta-Grammars and PAL Grammars</i>	137
	<i>References</i>	145

VIII

C. H. A. Koster	TWO-LEVEL GRAMMARS	146
	1. Context Sensitivity	146
	1.1. On the Borderline between Syntax and Semantics	146
	2. Van Wijngaarden Grammars	148
	2.1. One-Level Van Wijngaarden Grammars	148
	2.1.1. Definition: 1VWG	148
	2.1.2. Notation	148
	2.1.3. Terminology	148
	2.1.4. Properties	149
	2.1.5. Example	149
	2.2. Two-Level Van Wijngaarden Grammars	150
	2.2.1. Definition: 2VWG	150
	2.2.2. Notation	150
	2.2.3. Terminology	150
	2.2.4. Properties	151
	2.2.5. Example: Assignment Statement	152
	2.2.6. Example: Defining and Applied Occurrences	152
	3. Conclusion	156
	References	156

W. M. Waite	SEMANTIC ANALYSIS	157
	1. Tree Traversal	158
	2. Attribute Propagation	162
	3. Operator Identification and Coercion	165
	References	168

CHAPTER 3.: SYNTHESIS

W. M. Waite	RELATIONSHIP OF LANGUAGES TO MACHINES	170
	1. Data Objects	172
	1.1. Encodings	172
	1.2. Primitive Modes	176
	1.3. Mode Conversions	181
	2. Formation Rules	182
	2.1. Expressions	182
	2.2. Names	185
	2.3. Aggregates	187
	2.4. Procedures	189
	References	193

M. Griffiths	RUN-TIME STORAGE MANAGEMENT	195
	1. Introduction	197
	2. Static Allocation	197
	3. Dynamic Allocation	198
	3.1. Block Linkage	200
	3.2. Displays	203
	3.3. Compaction of the Stack	206
	3.4. Parameter Linkage	208
	3.5. Labels	209
	4. Aggregates	210
	4.1. Arrays	210
	4.1.1. Reference by Multiplication	211

IX

4.1.2.	<i>Reference by Code Words</i>	213
4.1.3.	<i>Range Testing</i>	214
4.2.	<i>Structures</i>	215
5.	<i>Lists</i>	215
5.1.	<i>Free Lists and Garbage</i>	216
5.2.	<i>Storage Collapse</i>	217
6.	<i>Parallel Processes</i>	218
7.	<i>Conclusion</i>	220
	<i>References</i>	221

U. Hill

	SPECIAL RUN-TIME ORGANIZATION TECH- NIQUES FOR ALGOL 68	222
1.	<i>Introduction</i>	222
1.1.	<i>Short Outline of Data Storage Prin- ciples</i>	224
1.1.1.	<i>Fixed and Variable Parts of Objects</i>	224
1.1.2.	<i>Modes and Objects in ALGOL 68</i>	224
1.1.3.	<i>Static and Dynamic Data Storage Areas</i>	228
1.1.4.	<i>The Heap</i>	231
1.2.	<i>Generative and Interpretative Hand- ling</i>	232
2.	<i>Special Objects in ALGOL 68</i>	233
2.1.	<i>Flexible Arrays</i>	233
2.2.	<i>Objects Generated by Slicing</i>	233
2.3.	<i>Objects Generated by Rowing</i>	234
2.4.	<i>Objects of Union Modes</i>	235
3.	<i>Scopes of Values (Life-Time)</i>	235
3.1.	<i>Definition</i>	235
3.2.	<i>Checking the Scope Conditions</i>	236
4.	<i>Scopes and Data Storage Allocation</i>	237
4.1.	<i>Scope of Routines and Allocation of Base Registers</i>	237
4.2.	<i>Storage for Data</i>	239
4.2.1.	<i>Local Objects Generated in the Block which is their Scope</i>	239
4.2.2.	<i>Local Objects Generated in an "Inner" Block</i>	240
4.2.3.	<i>Local Objects with Flexible Length</i>	241
4.2.4.	<i>Global Objects</i>	242
5.	<i>Special Topics</i>	243
5.1.	<i>Results of Blocks and Procedure Calls</i>	243
5.2.	<i>General Actual Parameters</i>	244
5.3.	<i>Local Generators</i>	245
5.4.	<i>General Mode Declarations</i>	247
5.5.	<i>"Empty" Flexible Arrays</i>	247
6.	<i>Garbage Collection</i>	248
	<i>References</i>	252

W. M. McKeemar

	SYMBOL TABLE ACCESS	253
1.	<i>Introduction</i>	253
2.	<i>Operations</i>	254
3.	<i>Method of Presentation</i>	258
4.	<i>Linear Symbol Table Access</i>	259
5.	<i>Sorted Symbol Table Access</i>	265
6.	<i>Tree Symbol Table Access</i>	273
7.	<i>Hash Symbol Table Access</i>	282

X

7.1.	Hash Functions	291
7.2.	Secondary Store	295
8.	Evaluation of Access Methods	296

W. M. Waite	CODE GENERATION	302
-------------	-----------------	-----

1.	A Model for Code Generation	304
1.1.	The Transducer	305
1.2.	The Simulator	306
1.3.	Common Subexpressions	307
2.	Sequencing and Control	309
3.	Generator Data Structures	316
3.1.	Value Descriptors	319
3.2.	Register Descriptors	321
4.	Instruction Generation	326
4.1.	Primitive Operations	327
4.2.	Interpretive Coding Language	329
	References	332

W. M. Waite	ASSEMBLY AND LINKAGE	333
-------------	----------------------	-----

1.	A Model for Assembly	335
1.1.	Object and Statement Procedures	335
1.2.	Cross Referencing	339
1.3.	Assembly under a Storage Constraint	342
1.4.	Operand Expressions	343
2.	Two-Pass Assembly	347
2.1.	Relative Symbol Definition	347
2.2.	Multiple Location Counters	352
3.	Partial Assembly and Linkage	353
	References	355

CHAPTER 4.: COMPILER-COMPILER

M. Griffiths	INTRODUCTION TO COMPILER-COMPILERS	356
--------------	------------------------------------	-----

1.	Motivation	356
2.	Compiler-Compilers based on Context-Free Syntax Methods	357
2.1.	Syntax	358
2.2.	Languages for Compiler Writing	358
2.2.1.	Production Language	359
2.2.2.	Machine Oriented Languages	360
3.	Current Research	361
3.1.	Extensible Languages	362
3.2.	Formal Semantics	363
4.	Conclusion	363
	References	364

C. H. A. Koster	USING THE CDL COMPILER-COMPILER	366
-----------------	---------------------------------	-----

0.	Introduction	366
1.	Affix Grammars and CDL	367
1.1.	CF Grammar as a Programming Language	367
1.2.	Extending CF Grammar	369
1.2.1.	Affixes	369

1.2.2.	<i>Primitive Actions and Predicates</i>	370
1.2.3.	<i>Actions</i>	371
1.2.4.	<i>Repetition</i>	372
1.2.5.	<i>Grouping</i>	373
1.2.6.	<i>Data Types</i>	374
1.3.	<i>Affix Grammars</i>	375
1.4.	<i>From Language Definition to Compiler Description</i>	375
2.	<i>The CDL Compiler-Compiler</i>	376
2.1.	<i>Extensional Mechanisms</i>	376
2.2.	<i>Realizing the CDL Machine</i>	378
2.2.1.	<i>Some Instructions</i>	378
2.2.2.	<i>Parametrization</i>	380
2.2.2.1.	<i>Parametrization of Rules</i>	380
2.2.2.2.	<i>Parametrization of System Macros</i>	381
2.2.2.3.	<i>Parametrization of User Macros</i>	381
2.2.2.4.	<i>Suggested Ameliorations</i>	382
2.3.	<i>Data Structures and Data Access</i>	383
2.4.	<i>The CDL Compiler-Compiler as a Tool</i>	384
2.4.1.	<i>High-Level and Low-Level Version</i>	385
2.4.2.	<i>Syntactic Debugging Aids</i>	385
2.4.3.	<i>Efficiency of the Resulting Compiler</i>	387
2.4.4.	<i>Conclusion</i>	388
3.	<i>Example: A Simple Editor</i>	389
3.1.	<i>Specification of a Simple Editor</i>	390
3.2.	<i>Environment, and Arithmetic Macros</i>	391
3.3.	<i>Table Administration Strategy</i>	392
3.4.	<i>Input and Output Primitives</i>	394
3.5.	<i>The Works</i>	395
3.6.	<i>Packing more Characters into a Word</i>	398
3.7.	<i>Various Target Languages</i>	400
4.	<i>Example 2: Symbol Table Administration</i>	402
4.1.	<i>The Environment</i>	403
4.1.1.	<i>Interface with the Machine</i>	403
4.1.2.	<i>Character Encoding</i>	404
4.1.3.	<i>The Basic Macros</i>	404
4.2.	<i>Storing Representation</i>	405
4.2.1.	<i>The repr Table</i>	405
4.2.2.	<i>Building a repr Table Element</i>	406
4.2.3.	<i>Entering an Element into the repr Table</i>	407
4.3.	<i>Output</i>	408
4.4.	<i>Input</i>	409
4.4.1.	<i>A One Character Stock</i>	409
4.4.2.	<i>Reading a Single Character</i>	410
4.4.3.	<i>Recognizing Characters</i>	410
4.4.4.	<i>Recognizing a Symbol</i>	411
4.4.5.	<i>Reading Items</i>	412
4.5.	<i>The Work Table</i>	412
4.5.1.	<i>The Block Administration</i>	413
4.5.2.	<i>Symbol Table</i>	414
4.6.	<i>Conclusion</i>	415
5.	<i>Example 3: Treatment of Declarations</i>	416
5.1.	<i>The Language to be Recognized</i>	416
5.1.1.	<i>Closed Clauses</i>	416
5.1.2.	<i>Modes</i>	417
5.1.3.	<i>Declarers</i>	418
5.2.	<i>Recognizing Closed Clauses</i>	419
5.3.	<i>Storing Modes</i>	421
5.4.	<i>Recognizing Declarers</i>	422
5.5.	<i>Conclusion</i>	425
	<i>References</i>	426

CHAPTER 5.: ENGINEERING A COMPILER

P. C. Poole	PORTABLE AND ADAPTABLE COMPILERS	427
1.	<i>Basic Concepts</i>	427
1.1.	<i>Portability and Adaptability</i>	427
1.2.	<i>Problems with Current Compilers</i>	430
1.3.	<i>Portable and Adaptable Standards</i>	437
2.	<i>Survey of Techniques</i>	439
2.1.	<i>Portability through High Level Language Coding</i>	439
2.1.1.	<i>Bootstrapping</i>	440
2.1.2.	<i>Language-Machine Interface</i>	442
2.2.	<i>Portability through Abstract Machine Modelling</i>	444
2.2.1.	<i>A Standard Abstract Machine</i>	446
2.2.2.	<i>Janus Assembly Language Formats</i>	453
2.2.3.	<i>Some Janus Examples</i>	458
2.2.4.	<i>Realizing the Abstract Machine by Interpretation</i>	464
2.3.	<i>Portability through Generation</i>	466
2.4.	<i>Adaptability</i>	467
3.	<i>Case Studies</i>	470
3.1.	<i>AED</i>	470
3.2.	<i>LSD</i>	471
3.3.	<i>BCPL</i>	473
3.4.	<i>Pascal</i>	478
3.4.1.	<i>Structure of Pascal</i>	478
3.4.2.	<i>The Abstract Machine for Pascal</i>	481
3.4.3.	<i>Incorporating the Abstract Machine into the LDT</i>	488
3.4.4.	<i>Measurements</i>	491
3.5.	<i>IBM S/360 FORTRAN (G) Compiler</i>	493
	<i>References</i>	497
J. J. Horning	STRUCTURING COMPILER DEVELOPMENT	498
1.	<i>Goals of Compiler Development</i>	498
1.1.	<i>Typical Compiler Goals</i>	498
1.1.1.	<i>Correctness</i>	498
1.1.2.	<i>Availability</i>	500
1.1.3.	<i>Generality and Adaptability</i>	501
1.1.4.	<i>Helpfulness</i>	501
1.1.5.	<i>Efficiency</i>	502
1.2.	<i>The Effects of Trade-Offs</i>	502
1.2.1.	<i>Compilation Efficiency VS. Execution Efficiency</i>	503
1.2.2.	<i>Compilation Efficiency VS. Helpfulness</i>	503
1.2.3.	<i>Generality VS. Efficiency</i>	503
1.2.4.	<i>Reliability VS. Complexity</i>	504
1.2.5.	<i>Development Speed VS. Everything Else</i>	504
2.	<i>Processes in Development</i>	504
2.1.	<i>Specification</i>	504
2.2.	<i>Design</i>	505
2.3.	<i>Implementation</i>	505
2.4.	<i>Validation</i>	506
2.5.	<i>Evaluation</i>	506
2.6.	<i>Maintenance</i>	506
3.	<i>Management Tools</i>	507

XIII

3.1.	<i>Project Organisation</i>	507
3.2.	<i>Information Distribution and Validation</i>	508
3.3.	<i>Programmer Motivation</i>	509
4.	<i>Technical Tools</i>	509
4.1.	<i>Compiler Compilers</i>	509
4.2.	<i>Standard Designs</i>	510
4.3.	<i>Design Methodologies</i>	510
4.4.	<i>Off-The-Shelf Components and Techniques</i>	510
4.5.	<i>Structured Programming</i>	511
4.6.	<i>Structured Programs</i>	511
4.7.	<i>Appropriate Languages</i>	511
	<i>References</i>	512

W. M. McKeeman

PROGRAMMING LANGUAGE DESIGN 514

1.	<i>Who Should (Not?) Do It?</i>	514
2.	<i>Design Principles</i>	517
3.	<i>Models for Languages</i>	519
3.1.	<i>The ALGOL Family as Models</i>	520
3.2.	<i>Mathematics as a Model</i>	523
3.3.	<i>Street Language as a Model</i>	524
	<i>References</i>	524

J. J. Horning

WHAT THE COMPILER SHOULD TELL THE USER 525

0.	<i>Introduction</i>	525
1.	<i>Normal Output</i>	527
1.1.	<i>Headings</i>	527
1.2.	<i>Listings</i>	528
1.3.	<i>Summaries</i>	530
2.	<i>Reaction to Errors</i>	532
2.1.	<i>Styles of Reaction</i>	532
2.1.1.	<i>Crash or Loop</i>	532
2.1.2.	<i>Produce Invalid Output</i>	532
2.1.3.	<i>Quit</i>	532
2.1.4.	<i>Recover and Continue Checking</i>	532
2.1.5.	<i>Repair and Continue Compilation</i>	533
2.1.6.	<i>"Correct"</i>	533
2.2.	<i>Ensuring Chosen Reactions</i>	533
2.3.	<i>Error Sources, Detectors and Sinks</i>	534
3.	<i>Syntactic Errors</i>	535
3.1.	<i>Point of Detection</i>	535
3.2.	<i>Recovery Techniques</i>	535
3.3.	<i>Systematic Recovery Strategies</i>	536
3.4.	<i>Repair Techniques</i>	537
4.	<i>Other Errors</i>	539
4.1.	<i>Lexical Errors</i>	539
4.2.	<i>Static Semantic Errors</i>	539
4.3.	<i>Dynamically Detected Errors</i>	540
4.4.	<i>Limit Failures</i>	542
4.5.	<i>Compiler Self-Checking</i>	543
5.	<i>Error Diagnosis</i>	543
5.1.	<i>Locating the Problem</i>	544
5.2.	<i>Describing the Symptom</i>	544
5.3.	<i>Suggesting Corrections</i>	544
5.4.	<i>Localisation of Error Processing</i>	545
5.5.	<i>Synthesis and Placement of Messages</i>	545
5.6.	<i>Error Logging</i>	546
5.7.	<i>Run-Time Diagnosis</i>	547
	<i>References</i>	548

W.M.WAITE

OPTIMIZATION

549

1. Classification of Techniques	551
1.1. Transformations	552
1.2. Regions	555
1.3. Efficacy	561
2. Local Optimization	564
2.1. Rearrangement of an expression	564
2.2. Redundant Code Elimination	570
2.3. Basic Blocks	579
3. Global Optimization	585
3.1. Redundancy and Rearrangement	587
3.2. Frequency and Strength Reduction	590
3.3. Global Analysis	596
References	600

CHAPTER 6.: APPENDIX

F.L.BAUER

HISTORICAL REMARKS ON COMPILER
CONSTRUCTION

603

1. Prehistorical 'Gedanken-Compilers'	605
2. The First Compilers	606
3. Sequentially Working Compilers	609
4. "Syntax Controlled Compilers"	612
5. Compilers Based on Precedence	612
6. Syntax Directed Compilers	613
7. Concluding Remarks	614
References	615

CHAPTER 1.A
COMPILER CONSTRUCTION

W. M. McKeeman
The University of California at
Santa Cruz
U. S. A.

"If PL/I is the Fatal Disease,
then perhaps Algol-68 is
Capital Punishment".

An Anonymous Compiler Writer

1. DEFINITIONS

1.1. SOURCE AND TARGET LANGUAGES

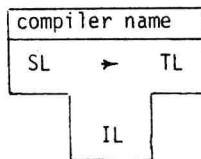
A compiler is a program, written in an implementation language, accepting text in a source language and producing text in a target language. Language description languages are used to define all of these languages and themselves, as well. The source language is an algorithmic language to be used by programmers. The target language is suitable for execution by some particular computer.

If the source and target languages are reasonably simple, and well matched to each other, the compiler can be short and easy to implement. (See Section 1.A.2 of these notes). The more complex the requirements become, the more elaborate the compiler must be and, the more elaborate the compiler, the higher the payoff in applying the techniques of structured programming.

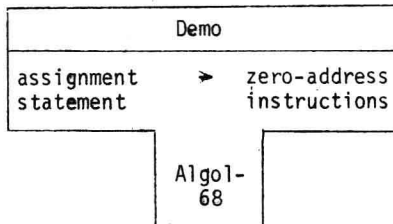
1.2. IMPLEMENTATION LANGUAGES

Compilers can, and have, been written in almost every programming language, but the use of structured programming techniques is dependent upon the implementation language being able to express structure. There are some existing languages which were explicitly designed for the task of compiler writing (FSL [Feldman 66], XPL [McKeeman 70], CDL [Koster 71b], and some for structuring (Pascal [Wirth 71], Algol 68 [van Wijngaarden 68]). The criterion for choosing an implementation language is quite straight forward: it should minimize the implementation effort and maximize the quality of the compiler. Lacking explicit knowledge of this kind, the compiler writer is advised to seek a language as close as possible to the ones mentioned above. The number, quality and availability of such languages is generally on the increase. It may be advantageous to write a compiler to run on a different machine than the target text will run on if better tools can thus be used (especially common for very small target machines). In any case, we shall simply assume an appropriate implementation language is available.

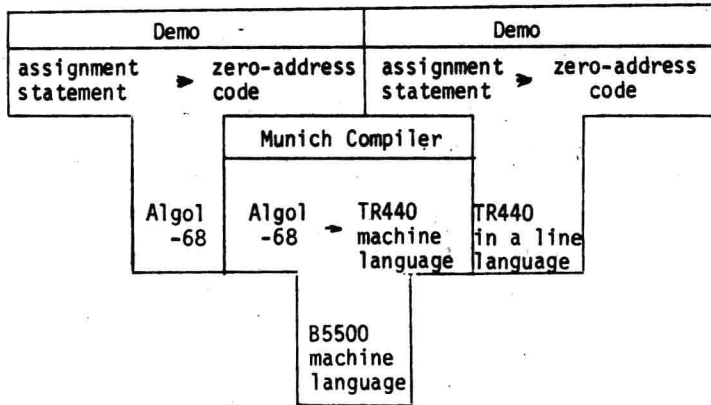
Since there are so many languages involved, and thus so many translations, we need a notation to keep the interactions straight. A given translator has three main languages (SL, TL, IL above) which are objects of the prepositions from, to and in respectively. A T diagram of the form



gives all three [Bratman 61]. If the compiler in Section 1.A.2 (below) is called Demo, then it can be described by the diagram



Now a compiler written in Algol-68 is of no use unless there is also a running compiler for Algol-68 available. Suppose it is on the Burroughs B5500. Then if we apply it to the T above, we will get a new T as follows:



where the arms of the middle T must match the tails of the Ts to the left and right. Complicated, multistage, multilanguage, multimachine translation processes can be described by appropriate cascades of such T diagrams [McKeeman 70 pp. 16-18].

1.3 Language Defining Languages

Language defining languages are almost always based on grammars (see Chapter 2 of these notes) but frequently have additional features designed to define the target text (i.e., translation defining languages). Thus the distinction between language definition and implementation language has not always been very clear. There was a tradition at one point of time to define a programming language as "what the compiler would translate" but this turned out to be of no value to the user who was not prepared to explore the idiosyncracies of a compiler to be able to write programs. The problem then has been to define languages without leaning on the compiler itself.

The ultimate solution is a definitional mechanism both clear enough for human reference and usable as input to a translator writing system which automatically creates the compiler.

2. Recursive Descent Compilation

2.1 Introduction

It is the intent of the following pages to give a concrete example of translation and also to review a particular, rather simple; rather successful, translation technique. The example, a translation of assignment statements to an assembly language for a stack machine, is trivial by contemporary standards but serves to elucidate the process. We can, in fact, present the entire translator as a whole without becoming mired in either detail or side issues. For the example, the source text

$$A = - A + 5 * B / (B-1)$$

will be translated to the following zero-address target text

LIT A LIT A LOAD NEG LIT 5 LIT B LOAD MUL LIT B LOAD LIT 1 NEG ADD DIV ADD STORE

which closely approximates the instructions of the Burroughs B5500 computer [Organik 71] (See also Chapters 3.A and 3.E of these notes). The target text is executed, step by step, in Figure 2.1. Although the meaning of the target text is probably obvious to the reader, we will take a few words to describe it. There is an evaluation stack into which values can be pushed. Furthermore, the top values (hence the last ones pushed into the stack) are available to be used in computations. The LIT instruction pushes one value onto the stack. That value is either an address (e.g., the address of the variable A in LIT A) or a constant (e.g. the value 5 in LIT 5).

The LOAD instruction assumes the top value on the stack is an address. The address is removed from the stack and the value found in the indicated cell in memory is pushed onto the stack in its place. The STORE instruction must be supplied with two items at the stack top. One is the address of a cell in memory. The other is a value to be stored into the indicated cell (the address is below the value in the stack). After the STORE instruction has completed its action, both address and value are removed from the stack. The remainder of the instructions are arithmetic operations. NEG changes the sign of the top value on the stack and leaves it where it found it. ADD, MUL and DIV operate on the two top elements on the stack, removing