

The Anatomy of Programming Languages

Alice E. Fischer
University of New Haven

Frances S. Grodzinsky
Sacred Heart University

September 1992

Copyright ©1990, 1991, 1992 by Alice E. Fischer and Frances S. Grodzinsky

All rights reserved. No part of this manuscript may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author.

Contents

I	About Language	1
1	The Nature of Language	3
1.1	Communication	4
1.2	Syntax and Semantics	5
1.3	Natural Languages and Programming Languages	6
1.3.1	Structure	6
1.3.2	Redundancy	7
1.3.3	Using Partial Information: Ambiguity and Abstraction	8
1.3.4	Implicit Communication	9
1.3.5	Flexibility and Nuance	10
1.3.6	Ability to Change and Evolve	10
1.4	The Standardization Process	11
1.4.1	Language Growth and Divergence	12
1.5	Nonstandard Compilers	12
2	Representation and Abstraction	17
2.1	What Is a Program?	17
2.2	Representation	20
2.2.1	Semantic Intent	21
2.2.2	Explicit versus Implicit Representation	22
2.2.3	Coherent versus Diffuse Representation	22
2.3	Language Design	25
2.3.1	Competing Design Goals	25
2.3.2	The Power of Restrictions	27
2.3.3	Principles for Evaluating a Design	30
2.4	Classifying Languages	41
2.4.1	Language Families	41
2.4.2	Languages Are More Alike than Different	49

3	Elements of Language	51
3.1	The Parts of Speech	51
3.1.1	Nouns	51
3.1.2	Pronouns: Pointers	52
3.1.3	Adjectives: Data Types	53
3.1.4	Verbs	55
3.1.5	Prepositions and Conjunctions	58
3.2	The Metalanguage	59
3.2.1	Words: Lexical Tokens	59
3.2.2	Sentences: Statements	62
3.2.3	Larger Program Units: Scope	64
3.2.4	Comments	67
3.2.5	Naming Parts of a Program	70
3.2.6	Metawords That Let the Programmer Extend the Language	70
4	Formal Description of Language	77
4.1	Foundations of Programming Languages	78
4.2	Syntax	78
4.2.1	Extended BNF	82
4.2.2	Syntax Diagrams	87
4.3	Semantics	90
4.3.1	The Meaning of a Program	90
4.3.2	Definition of Language Semantics	90
4.3.3	The Abstract Machine	92
4.3.4	Lambda Calculus: A Minimal Semantic Basis	96
4.4	Extending the Semantics of a Language	107
4.4.1	Semantic Extension in FORTH	110
II	Describing Computation	115
5	Primitive Types	117
5.1	Primitive Hardware Types	118
5.1.1	Bytes, Words, and Long Words	118
5.1.2	Character Codes	118
5.1.3	Numbers	120
5.2	Types in Programming Languages	126
5.2.1	Type Is an Abstraction	126
5.2.2	A Type Provides a Physical Description	127
5.2.3	What Primitive Types Should a Language Support?	130
5.2.4	Emulation	133

5.3	A Brief History of Type Declarations	133
5.3.1	Origins of Type Ideas	133
5.3.2	Type Becomes a Definable Abstraction	137
6	Modeling Objects	143
6.1	Kinds of Objects	144
6.2	Placing a Value in a Storage Object	146
6.2.1	Static Initialization	146
6.2.2	Dynamically Changing the Contents of a Storage Object	148
6.2.3	Dereferencing	152
6.2.4	Pointer Assignment	154
6.2.5	The Semantics of Pointer Assignment	156
6.3	The Storage Model: Managing Storage Objects	158
6.3.1	The Birth and Death of Storage Objects	158
6.3.2	Dangling References	169
7	Names and Binding	175
7.1	The Problem with Names	175
7.1.1	The Role of Names	176
7.1.2	Definition Mechanisms: Declarations and Defaults	178
7.1.3	Binding	180
7.1.4	Names and Objects: Not a One-to-One Correspondence	185
7.2	Binding a Name to a Constant	186
7.2.1	Implementations of Constants	190
7.2.2	How Constant Is a Constant?	191
7.3	Survey of Allocation and Binding	191
7.4	The Scope of a Name	193
7.4.1	Naming Conflicts	193
7.4.2	Block Structure	195
7.4.3	Recursive Bindings	198
7.4.4	Visibility versus Lifetime.	200
7.5	Implications for the Compiler / Interpreter	205
8	Expressions and Evaluation	211
8.1	The Programming Environment	212
8.2	Sequence Control and Communication	213
8.2.1	Nesting	214
8.2.2	Sequences of Statements	215
8.2.3	Interprocess Sequence Control	216
8.3	Expression Syntax	216
8.3.1	Functional Expression Syntax	217

8.3.2	Operator Expressions	218
8.3.3	Combinations of Parsing Rules	222
8.4	Function Evaluation	224
8.4.1	Order of Evaluation	224
8.4.2	Lazy or Strict Evaluation	227
8.4.3	Order of Evaluation of Arguments	230
9	Functions and Parameters	233
9.1	Function Syntax	234
9.1.1	Fixed versus Variable Argument Functions	234
9.1.2	Parameter Correspondence	235
9.1.3	Indefinite-Length Parameter Lists	237
9.2	What Does an Argument Mean?	239
9.2.1	Call-by-Value	240
9.2.2	Call-by-Name	242
9.2.3	Call-by-Reference	244
9.2.4	Call-by-Return	247
9.2.5	Call-by-Value-and-Return	248
9.2.6	Call-by-Pointer	249
9.3	Higher-Order Functions	254
9.3.1	Functional Arguments	255
9.3.2	Currying	257
9.3.3	Returning Functions from Functions	259
10	Control Structures	267
10.1	Basic Control Structures	268
10.1.1	Normal Instruction Sequencing	269
10.1.2	Assemblers	270
10.1.3	Sequence, Subroutine Call, IF, and WHILE Suffice	270
10.1.4	Subroutine Call	272
10.1.5	Jump and Conditional Jump	273
10.1.6	Control Diagrams	274
10.2	Conditional Control Structures	275
10.2.1	Conditional Expressions versus Conditional Statements	275
10.2.2	Conditional Branches: Simple Spaghetti	277
10.2.3	Structured Conditionals	278
10.2.4	The Case Statement	284
10.3	Iteration	289
10.3.1	The Infinite Loop	290
10.3.2	Conditional Loops	290
10.3.3	The General Loop	292

10.3.4	Counted Loops	293
10.3.5	The Iteration Element	298
10.4	Implicit Iteration	301
10.4.1	Iteration on Coherent Objects	301
10.4.2	Backtracking	303
11	Global Control	309
11.1	The GOTO Problem	310
11.1.1	Faults Inherent in GOTO	310
11.1.2	To GOTO or Not to GOTO	313
11.1.3	Statement Labels	315
11.2	Breaking Out	317
11.2.1	Generalizing the BREAK	320
11.3	Continuations	321
11.4	Exception Processing	327
11.4.1	What Is an Exception?	327
11.4.2	The Steps in Exception Handling	328
11.4.3	Exception Handling in Ada	331
III	Application Modeling	335
12	Functional Languages	337
12.1	Denotation versus Computation	338
12.1.1	Denotation	339
12.2	The Functional Approach	341
12.2.1	Eliminating Assignment	342
12.2.2	Recursion Can Replace WHILE	344
12.2.3	Sequences	347
12.3	Miranda: A Functional Language	350
12.3.1	Data Structures	351
12.3.2	Operations and Expressions	351
12.3.3	Function Definitions	352
12.3.4	List Comprehensions	355
12.3.5	Infinite Lists	358
13	Logic Programming	361
13.1	Predicate Calculus	362
13.1.1	Formulas	362
13.2	Proof Systems	365
13.3	Models	367

13.4	Automatic Theorem Proving	368
13.4.1	Resolution Theorem Provers	370
13.5	Prolog	375
13.5.1	The Prolog Environment	375
13.5.2	Data Objects and Terms	375
13.5.3	Horn Clauses in Prolog	376
13.5.4	The Prolog Deduction Process	379
13.5.5	Functions and Computation	380
13.5.6	Cuts and the “not” Predicate	385
13.5.7	Evaluation of Prolog	389
14	The Representation of Types	393
14.1	Programmer-Defined Types	394
14.1.1	Representing Types within a Translator	394
14.1.2	Finite Types	396
14.1.3	Constrained Types	397
14.1.4	Pointer Types	398
14.2	Compound Types	400
14.2.1	Arrays	400
14.2.2	Strings	406
14.2.3	Sets	408
14.2.4	Records	412
14.2.5	Union Types	419
14.3	Operations on Compound Objects	422
14.3.1	Creating Program Objects: Value Constructors	422
14.3.2	The Interaction of Dereferencing, Constructors, and Selectors	423
14.4	Operations on Types	430
15	The Semantics of Types	435
15.1	Semantic Description	436
15.1.1	Domains in Early Languages	436
15.1.2	Domains in “Typeless” Languages	437
15.1.3	Domains in the 1970s	441
15.1.4	Domains in the 1980s	444
15.2	Type Checking	444
15.2.1	Strong Typing	445
15.2.2	Strong Typing and Data Abstraction	446
15.3	Domain Identity: Different Domain/ Same Domain?	448
15.3.1	Internal and External Domains	448
15.3.2	Internally Merged Domains	449
15.3.3	Domain Mapping	450

15.4	Programmer-Defined Domains	452
15.4.1	Type Description versus Type Name	452
15.4.2	Type Constructors	453
15.4.3	Types Defined by Mapping	454
15.5	Type Casts, Conversions, and Coercions	459
15.5.1	Type Casts.	460
15.5.2	Type Conversions	465
15.5.3	Type Coercion	466
15.6	Conversions and Casts in Common Languages	470
15.6.1	COBOL	470
15.6.2	FORTRAN	470
15.6.3	Pascal	471
15.6.4	PL/1	472
15.6.5	C	472
15.6.6	Ada Types and Treatment of Coercion	475
15.7	Evading the Type Matching Rules	479
16	Modules and Object Classes	489
16.1	The Purpose of Modules	490
16.2	Modularity Through Files and Linking	492
16.3	Packages in Ada	497
16.4	Object Classes.	500
16.4.1	Classes in C++	501
16.4.2	Represented Domains	505
16.4.3	Friends of Classes	506
17	Generics	511
17.1	Generics	512
17.1.1	What Is a Generic?	512
17.1.2	Implementations of Generics	513
17.1.3	Generics, Virtual Functions, and ADTs	515
17.1.4	Generic Functions	516
17.2	Limited Generic Behavior	521
17.2.1	Union Data Types	521
17.2.2	Overloaded Names	521
17.2.3	Fixed Set of Generic Definitions, with Coercion	523
17.2.4	Extending Predefined Operators	524
17.2.5	Flexible Arrays	525
17.3	Parameterized Generic Domains	527
17.3.1	Domains with Type Parameters	530
17.3.2	Preprocessor Generics in C	531

18 Dispatching with Inheritance	541
18.1 Representing Domain Relationships	542
18.1.1 The Mode Graph and the Dispatcher	542
18.2 Subdomains and Class Hierarchies.	549
18.2.1 Subrange Types	549
18.2.2 Class Hierarchies	550
18.2.3 Virtual Functions in C++.	554
18.2.4 Function Inheritance	556
18.2.5 Programmer-Defined Conversions in C++	558
18.3 Polymorphic Domains and Functions	561
18.3.1 Polymorphic Functions	561
18.3.2 Manual Domain Representation and Dispatching	562
18.3.3 Automating Ad Hoc Polymorphism	563
18.3.4 Parameterized Domains	567
18.4 Can We Do More with Generics?	568
18.4.1 Dispatching Using the Mode Graph	571
18.4.2 Generics Create Some Hard Problems	575
A Exhibits Listed by Topic	585
A.1 Languages	585
A.1.1 Ada	585
A.1.2 APL	586
A.1.3 C++	587
A.1.4 C and ANSI C	588
A.1.5 FORTH	590
A.1.6 FORTRAN	591
A.1.7 LISP	591
A.1.8 Miranda	592
A.1.9 Pascal	593
A.1.10 Prolog	596
A.1.11 Scheme and T	597
A.1.12 Other Languages	597
A.2 Concepts	598
A.2.1 Application Modeling, Generics, and Polymorphic Domains	598
A.2.2 Control Structures	599
A.2.3 Data Representation	600
A.2.4 History	600
A.2.5 Lambda Calculus	600
A.2.6 Language Design and Specification	601
A.2.7 Logic	601
A.2.8 Translation, Interpretation, and Function Calls	602

A.2.9 Types 602

Preface

This text is intended for a course in advanced programming languages or the structure of programming language and should be appropriate for students at the junior, senior, or master's level. It should help the student understand the principles that underlie all languages and all language implementations.

This is a comprehensive text which attempts to dissect language and explain how a language is really built. The first eleven chapters cover the core material: language specification, objects, expressions, control, and types. The more concrete aspects of each topic are presented first, followed by a discussion of implementation strategies and the related semantic issues. Later chapters cover current topics, including modules, object-oriented programming, functional languages, and concurrency constructs.

The emphasis throughout the text is on semantics and abstraction; the syntax and historical development of languages are discussed in light of the underlying semantical concepts. Fundamental principles of computation, communication, and good design are stated and are used to evaluate various language constructs and to demonstrate that language designs are improving as these principles become widely understood.

Examples are cited from many languages including Pascal, C, C++, FORTH, BASIC, LISP, FORTRAN, Ada, COBOL, APL, Prolog, Turing, Miranda, and Haskell. All examples are annotated so that a student who is unfamiliar with the language used can understand the meaning of the code and see how it illustrates the principle.

It is the belief of the authors that the student who has a good grasp of the structure of computer languages will have the tools to master new languages easily.

The specific goals of this book are to help students learn:

- To reason clearly about programming languages.
- To develop principles of communication so that we can evaluate the wisdom and utility of the decisions made in the process of language design.
- To break down language into its major components, and each component into small pieces so that we can focus on competing alternatives.
- To define a consistent and general set of terms for the components out of which programming languages are built, and the concepts on which they are based.

- To use these terms to describe existing languages, and in so doing clarify the conflicting terminology used by the language designers, and untangle the complexities inherent in so many languages.
- To see below the surface appearance of a language to its actual structure and descriptive power.
- To understand that many language features that commonly occur together are, in fact, independent and separable. To appreciate the advantages and disadvantages of each feature. To suggest ways in which these basic building blocks can be recombined in new languages with more desirable properties and fewer faults.
- To see the similarities and differences that exist among languages students already know, and to learn new ones.
- To use the understanding so gained to suggest future trends in language design.

Acknowledgement

The authors are indebted to several people for their help and support during the years we have worked on this project. First, we wish to thank our families for their uncomplaining patience and understanding.

We thank Michael J. Fischer for his help in developing the sections on lambda calculus, functional languages and logic. and for working out several sophisticated code examples. In addition, his assistance as software and hardware systems expert and TeX guru made this work possible.

Several reviewers read this work in detail and offered invaluable suggestions and corrections. We thank these people for their help. Special thanks go to Robert Fischer and Roland Lieger for reading beyond the call of duty and to Gary Walters for his advice and for the material he has contributed.

Finally, we thank our students at the University of New Haven and at Sacred Heart University for their feedback on the many versions of this book.

Parts of this manuscript were developed under a grant from Sacred Heart University.