Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

94

Semantics-Directed Compiler Generation

Proceedings of a Workshop
Aarhus, Denmark, January, 1980

Edited by Neil D. Jones

Springer-Verlag
Berlin Heidelberg New York

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

94

Semantics-Directed Compiler Generation

Proceedings of a Workshop
Aarhus, Denmark, January 14–18, 1980

Edited by Neil D. Jones

Springer-Verlag
Berlin Heidelberg New York 1980

Editor

Neil D. Jones
Datalogisk Afdeling
Matematisk Institut
Aarhus Universitet
8000 Aarhus C
Denmark

AMS Subject Classifications (1970): 68B10
CR Subject Classifications (1974): 4.12, 5.24

ISBN 3-540-10250-7 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-10250-7 Springer-Verlag New York Heidelberg Berlin

Printing and binding: Beltz Offsetdruck, Hemsbach/Bergstr.
2145/3140-543210

Lecture Notes in Computer Science

Vol. 1: GI-Gesellschaft für Informatik e.V. 3. Jahrestagung, Hamburg, 8.–10. Oktober 1973. Herausgegeben im Auftrag der Gesellschaft für Informatik von W. Brauer. XI, 508 Seiten. 1973.

Vol. 2: GI-Gesellschaft für Informatik e.V. 1. Fachtagung über Automatentheorie und Formale Sprachen, Bonn, 9.–12. Juli 1973. Herausgegeben im Auftrag der Gesellschaft für Informatik von K.-H. Böhling und K. Indermark. VII, 322 Seiten. 1973.

Vol. 3: 5th Conference on Optimization Techniques, Part I. (Series: I.F.I.P. TC7 Optimization Conferences.) Edited by R. Conti and A. Ruberti. XIII, 565 pages. 1973.

Vol. 4: 5th Conference on Optimization Techniques, Part II. (Series: I.F.I.P. TC7 Optimization Conferences.) Edited by R. Conti and A. Ruberti. XIII, 389 pages. 1973.

Vol. 5: International Symposium on Theoretical Programming. Edited by A. Ershov and V. A. Nepomniaschy. VI, 407 pages. 1974.

Vol. 6: B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, Matrix Eigensystem Routines – EISPACK Guide. XI, 551 pages. 2nd Edition 1974. 1976.

Vol. 7: 3. Fachtagung über Programmiersprachen, Kiel, 5.–7. März 1974. Herausgegeben von B. Schlender und W. Friellinghaus. VI, 225 Seiten. 1974.

Vol. 8: GI-NTG Fachtagung über Struktur und Betrieb von Rechensystemen, Braunschweig, 20.–22. März 1974. Herausgegeben im Auftrag der GI und der NTG von H.-O. Leilich. VI, 340 Seiten. 1974.

Vol. 9: GI-BIFOA Internationale Fachtagung: Informationszentren in Wirtschaft und Verwaltung. Köln, 17./18. Sept. 1973. Herausgegeben im Auftrag der GI und dem BIFOA von P. Schmitz. VI, 259 Seiten. 1974.

Vol. 10: Computing Methods in Applied Sciences and Engineering, Part 1. International Symposium, Versailles, December 17–21, 1973. Edited by R. Glowinski and J. L. Lions. X, 497 pages. 1974.

Vol. 11: Computing Methods in Applied Sciences and Engineering, Part 2. International Symposium, Versailles, December 17–21, 1973. Edited by R. Glowinski and J. L. Lions. X, 434 pages. 1974.

Vol. 12: GFK-GI-GMR Fachtagung Prozessrechner 1974. Karlsruhe, 10.–11. Juni 1974. Herausgegeben von G. Krüger und R. Friehmelt. XI, 620 Seiten. 1974.

Vol. 13: Rechnerstrukturen und Betriebsprogrammierung, Erlangen, 1970. (GI-Gesellschaft für Informatik e.V.) Herausgegeben von W. Händler und P. P. Spies. VII, 333 Seiten. 1974.

Vol. 14: Automata, Languages and Programming – 2nd Colloquium, University of Saarbrücken, July 29–August 2, 1974. Edited by J. Loeckx. VIII, 611 pages. 1974.

Vol. 15: L Systems. Edited by A. Salomaa and G. Rozenberg. VI, 338 pages. 1974.

Vol. 16: Operating Systems, International Symposium, Rocquencourt 1974. Edited by E. Gelenbe and C. Kaiser. VIII, 310 pages. 1974.

Vol. 17: Rechner-Gestützter Unterricht RGU '74, Fachtagung, Hamburg, 12.–14. August 1974, ACU-Arbeitskreis Computer-Unterstützter Unterricht. Herausgegeben im Auftrag der GI von K. Brunnstein, K. Haefner und W. Händler. X, 417 Seiten. 1974.

Vol. 18: K. Jensen and N. E. Wirth, PASCAL – User Manual and Report. VII, 170 pages. Corrected Reprint of the 2nd Edition 1976.

Vol. 19: Programming Symposium. Proceedings 1974. V, 425 pages. 1974.

Vol. 20: J. Engelfriet, Simple Program Schemes and Formal Languages. VII, 254 pages. 1974.

Vol. 21: Compiler Construction, An Advanced Course. Edited by F. L. Bauer and J. Eickel. XIV. 621 pages. 1974.

Vol. 22: Formal Aspects of Cognitive Processes. Proceedings 1972. Edited by T. Storer and D. Winter. V, 214 pages. 1975.

Vol. 23: Programming Methodology. 4th Informatik Symposium, IBM Germany Wildbad, September 25–27, 1974. Edited by C. E. Hackl. VI, 501 pages. 1975.

Vol. 24: Parallel Processing. Proceedings 1974. Edited by T. Feng. VI, 433 pages. 1975.

Vol. 25: Category Theory Applied to Computation and Control. Proceedings 1974. Edited by E. G. Manes. X, 245 pages. 1975.

Vol. 26: GI-4. Jahrestagung, Berlin, 9.–12. Oktober 1974. Herausgegeben im Auftrag der GI von D. Siefkes. IX, 748 Seiten. 1975.

Vol. 27: Optimization Techniques. IFIP Technical Conference. Novosibirsk, July 1–7, 1974. (Series: I.F.I.P. TC7 Optimization Conferences.) Edited by G. I. Marchuk. VIII, 507 pages. 1975.

Vol. 28: Mathematical Foundations of Computer Science. 3rd Symposium at Jadwisin near Warsaw, June 17–22, 1974. Edited by A. Blikle. VII, 484 pages. 1975.

Vol. 29: Interval Mathematics. Procedings 1975. Edited by K. Nickel. VI, 331 pages. 1975.

Vol. 30: Software Engineering. An Advanced Course. Edited by F. L. Bauer. (Formerly published 1973 as Lecture Notes in Economics and Mathematical Systems, Vol. 81) XII, 545 pages. 1975.

Vol. 31: S. H. Fuller, Analysis of Drum and Disk Storage Units. IX, 283 pages. 1975.

Vol. 32: Mathematical Foundations of Computer Science 1975. Proceedings 1975. Edited by J. Bečvář. X, 476 pages. 1975.

Vol. 33: Automata Theory and Formal Languages, Kaiserslautern, May 20–23, 1975. Edited by H. Brakhage on behalf of GI. VIII, 292 Seiten. 1975.

Vol. 34: GI – 5. Jahrestagung, Dortmund 8.–10. Oktober 1975. Herausgegeben im Auftrag der GI von J. Mühlbacher. X, 755 Seiten. 1975.

Vol. 35: W. Everling, Exercises in Computer Systems Analysis. (Formerly published 1972 as Lecture Notes in Economics and Mathematical Systems, Vol. 65) VIII, 184 pages. 1975.

Vol. 36: S. A. Greibach, Theory of Program Structures: Schemes, Semantics, Verification. XV, 364 pages. 1975.

Vol. 37: C. Böhm, λ-Calculus and Computer Science Theory. Proceedings 1975. XII, 370 pages. 1975.

Vol. 38: P. Branquart, J.-P. Cardinael, J. Lewi, J.-P. Delescaille, M. Vanbegin. An Optimized Translation Process and Its Application to ALGOL 68. IX, 334 pages. 1976.

Vol. 39: Data Base Systems. Proceedings 1975. Edited by H. Hasselmeier and W. Spruth. VI, 386 pages. 1976.

Vol. 40: Optimization Techniques. Modeling and Optimization in the Service of Man. Part 1. Proceedings 1975. Edited by J. Cea. XIV, 854 pages. 1976.

Vol. 41: Optimization Techniques. Modeling and Optimization in the Service of Man. Part 2. Proceedings 1975. Edited by J. Cea. XIII, 852 pages. 1976.

Vol. 42: James E. Donahue, Complementary Definitions of Programming Language Semantics. VII, 172 pages. 1976.

Vol. 43: E. Specker und V. Strassen, Komplexität von Entscheidungsproblemen. Ein Seminar. V, 217 Seiten. 1976.

Vol. 44: ECI Conference 1976. Proceedings 1976. Edited by K. Samelson. VIII, 322 pages. 1976.

Vol. 45: Mathematical Foundations of Computer Science 1976. Proceedings 1976. Edited by A. Mazurkiewicz. XI, 601 pages. 1976.

Vol. 46: Language Hierarchies and Interfaces. Edited by F. L. Bauer and K. Samelson. X, 428 pages. 1976.

Vol. 47: Methods of Algorithmic Language Implementation. Edited by A. Ershov and C. H. A. Koster. VIII, 351 pages. 1977.

Vol. 48: Theoretical Computer Science, Darmstadt, March 1977. Edited by H. Tzschach, H. Waldschmidt and H. K.-G. Walter on behalf of GI. VIII, 418 pages. 1977.

INTRODUCTION

For several reasons it is becoming more and more common to provide formal definitions of the semantics of new programming languages, using techniques such as denotational semantics, attribute and affix grammars, algebraic semantics, operational definitions and axiomatic definitions. The construction of such a definition tends to expose ambiguities and unexpected implications of proposed language features, and thus can be a significant aid when designing a new language. A precise definition of the "meaning" of a program is of course essential when implementing a compiler or other language processor, and such definitions have in some cases guided the development of implementations. Further, the existence of formal definitions of source and object languages makes it possible to formally prove the correctness of compilers.

A number of translator-writing systems have been devised to systematize and simplify the task of compiler construction. These are usually syntax-directed, and provide in addition to parsing some means of manipulating symbol tables, parse tree attributes, etc. Still, compiler writing is at present largely handcraft - construction of such a large and complex piece of software requires considerable creativity, is quite prone to errors, and involves an enormous amount of work.

Correctness proofs for real compilers seem to be more of an ideal than a reality at this time, since construction of a correctness proof seems to require even more creativity and labor than construction of the compiler itself.

Clearly both problems would be alleviated if there were a closer connection between the semantic definition of the language and the structure of its compiler, just as parsing problems were much simplified after a firm connection was made between syntax definition and parser structure.

An ideal solution would be a true compiler generator, which if given definitions of the syntax and semantics of a programming language would automatically produce a compiler of acceptable compile-time and run-time efficiency. The purpose of the workshop was to bring together researchers whose work brings us closer to this goal.

The papers presented at the workshop naturally fall into four categories. The first group contains three papers with a common goal: to produce a compiler from the denotational semantics of a programming language. The second group is concerned with the use of abstract algebra to define semantics, to specify compilers and to prove them correct. The third group has to do with several aspects of attribute or affix grammars. These are a powerful and natural medium for expressing compilers, and thus provide a promising output language for compiler generators. The last group contains three papers which are related to compiler generation but not in the earlier categories, including one on the formal semantic definition of the ADA programming language. The definition is intended to serve

"for the validation of implementations and as a guideline for implementors ... (and) as an input for a compiler generator when the technology becomes available." As such it is likely to stimulate further research in automating the compiler generation process.

The "Workshop on Semantics-Directed Compiler Generation" was held January 14-18 at the University of Aarhus, Denmark. The meeting was made possible by grants from the Danish Research Council (Forskningsråd) and the Aarhus University Computer Science Department (Datalogisk Afdeling). Local arrangements were handled by an organizing committee consisting of Neil Jones, Peter Mosses and Mogens Nielsen and by the workshop secretary, Karen Møller. The department deserves a round of thanks for providing (in addition to funds) the use of its reproduction, secretarial and library facilities, and for providing an excellent milieu for work and discussion. The contributions of a number of individuals are warmly acknowledged, including Karen Møller, Mogens Nielsen, Lene Rold and of course the participants in the workshop, without whose professional expertise the workshop would not have been possible.

Vol. 49: Interactive Systems. Proceedings 1976. Edited by A. Blaser and C. Hackl. VI, 380 pages. 1976.

Vol. 50: A. C. Hartmann, A Concurrent Pascal Compiler for Minicomputers. VI, 119 pages. 1977.

Vol. 51: B. S. Garbow, Matrix Eigensystem Routines – Eispack Guide Extension. VIII, 343 pages. 1977.

Vol. 52: Automata, Languages and Programming. Fourth Colloquium, University of Turku, July 1977. Edited by A. Salomaa and M. Steinby. X, 569 pages. 1977.

Vol. 53: Mathematical Foundations of Computer Science. Proceedings 1977. Edited by J. Gruska. XII, 608 pages. 1977.

Vol. 54: Design and Implementation of Programming Languages. Proceedings 1976. Edited by J. H. Williams and D. A. Fisher. X, 496 pages. 1977.

Vol. 55: A. Gerbier, Mes premières constructions de programmes. XII, 256 pages. 1977.

Vol. 56: Fundamentals of Computation Theory. Proceedings 1977. Edited by M. Karpiński. XII, 542 pages. 1977.

Vol. 57: Portability of Numerical Software. Proceedings 1976. Edited by W. Cowell. VIII, 539 pages. 1977.

Vol. 58: M. J. O'Donnell, Computing in Systems Described by Equations. XIV, 111 pages. 1977.

Vol. 59: E. Hill, Jr., A Comparative Study of Very Large Data Bases. X, 140 pages. 1978.

Vol. 60: Operating Systems, An Advanced Course. Edited by R. Bayer, R. M. Graham, and G. Seegmüller. X, 593 pages. 1978.

Vol. 61: The Vienna Development Method: The Meta-Language. Edited by D. Bjørner and C. B. Jones. XVIII, 382 pages. 1978.

Vol. 62: Automata, Languages and Programming. Proceedings 1978. Edited by G. Ausiello and C. Böhm. VIII, 508 pages. 1978.

Vol. 63: Natural Language Communication with Computers. Edited by Leonard Bolc. VI, 292 pages. 1978.

Vol. 64: Mathematical Foundations of Computer Science. Proceedings 1978. Edited by J. Winkowski. X, 551 pages. 1978.

Vol. 65: Information Systems Methodology, Proceedings, 1978. Edited by G. Bracchi and P. C. Lockemann. XII, 696 pages. 1978.

Vol. 66: N. D. Jones and S. S. Muchnick, TEMPO: A Unified Treatment of Binding Time and Parameter Passing Concepts in Programming Languages. IX, 118 pages. 1978.

Vol. 67: Theoretical Computer Science, 4th GI Conference, Aachen, March 1979. Edited by K. Weihrauch. VII, 324 pages. 1979.

Vol. 68: D. Harel, First-Order Dynamic Logic. X, 133 pages. 1979.

Vol. 69: Program Construction. International Summer School. Edited by F. L. Bauer and M. Broy. VII, 651 pages. 1979.

Vol. 70: Semantics of Concurrent Computation. Proceedings 1979. Edited by G. Kahn. VI, 368 pages. 1979.

Vol. 71: Automata, Languages and Programming. Proceedings 1979. Edited by H. A. Maurer. IX, 684 pages. 1979.

Vol. 72: Symbolic and Algebraic Computation. Proceedings 1979. Edited by E. W. Ng. XV, 557 pages. 1979.

Vol. 73: Graph-Grammars and Their Application to Computer Science and Biology. Proceedings 1978. Edited by V. Claus, H. Ehrig and G. Rozenberg. VII, 477 pages. 1979.

Vol. 74: Mathematical Foundations of Computer Science. Proceedings 1979. Edited by J. Bečvář. IX, 580 pages. 1979.

Vol. 75: Mathematical Studies of Information Processing. Proceedings 1978. Edited by E. K. Blum, M. Paul and S. Takasu. VIII, 629 pages. 1979.

Vol. 76: Codes for Boundary-Value Problems in Ordinary Differential Equations. Proceedings 1978. Edited by B. Childs et al. VIII, 388 pages. 1979.

Vol. 77: G. V. Bochmann, Architecture of Distributed Computer Systems. VIII, 238 pages. 1979.

Vol. 78: M. Gordon, R. Milner and C. Wadsworth, Edinburgh LCF. VIII, 159 pages. 1979.

Vol. 79: Language Design and Programming Methodology. Proceedings, 1979. Edited by J. Tobias. IX, 255 pages. 1980.

Vol. 80: Pictorial Information Systems. Edited by S. K. Chang and K. S. Fu. IX, 445 pages. 1980.

Vol. 81: Data Base Techniques for Pictorial Applications. Proceedings, 1979. Edited by A. Blaser. XI, 599 pages. 1980.

Vol. 82: J. G. Sanderson, A Relational Theory of Computing. VI, 147 pages. 1980.

Vol. 83: International Symposium Programming. Proceedings, 1980. Edited by B. Robinet. VII, 341 pages. 1980.

Vol. 84: Net Theory and Applications. Proceedings, 1979. Edited by W. Brauer. XIII, 537 Seiten. 1980.

Vol. 85: Automata, Languages and Programming. Proceedings, 1980. Edited by J. de Bakker and J. van Leeuwen. VIII, 671 pages. 1980.

Vol. 86: Abstract Software Specifications. Proceedings, 1979. Edited by D. Bjørner. XIII, 567 pages. 1980

Vol. 87: 5th Conference on Automated Deduction. Proceedings, 1980. Edited by W. Bibel and R. Kowalski. VII, 385 pages. 1980.

Vol. 88: Mathematical Foundations of Computer Science 1980. Proceedings, 1980. Edited by P. Dembiński. VIII, 723 pages. 1980.

Vol. 89: Computer Aided Design - Modelling, Systems Engineering, CAD-Systems. Proceedings, 1980. Edited by J. Encarnacao. XIV, 461 pages. 1980.

Vol. 90: D. M. Sandford, Using Sophisticated Models in Resolution Theorem Proving.
XI, 239 pages. 1980

Vol. 91: D. Wood, Grammar and L Forms: An Introduction. IX, 314 pages. 1980.

Vol. 92: R. Milner, A Calculus of Communication Systems. VI, 171 pages. 1980.

Vol. 93: A. Nijholt, Context-Free Grammars: Covers, Normal Forms, and Parsing. VII, 253 pages. 1980.

Vol. 94: Semantics-Directed Compiler Generation. Proceedings, 1980. Edited by N. D. Jones. V, 489 pages. 1980.

CONTENTS

COMPILERS BASED ON DENOTATIONAL SEMANTICS

Transforming denotational semantics into practical attribute grammars 1
Harald Ganzinger

Compiler generation from denotational semantics 70
Neil D. Jones, David A. Schmidt

From standard to implementation denotational semantics 94
Martin Raskovsky, Phil Collier

COMPILING AND ALGEBRAIC SEMANTICS

Specification of compilers as abstract data type representations 140
Marie-Claude Gaudel

More on advice on structuring compilers and proving them correct 165
James W. Thatcher, Eric G. Wagner, Jesse B. Wright

A constructive approach to compiler correctness 189
Peter Mosses

Using category theory to design implicit conversions and generic operators .. 211
John Reynolds

ATTRIBUTE AND AFFIX GRAMMARS

On defining semantics by means of extended attribute grammars 259
Ole Lehrmann Madsen

Tree-affix dendrogrammars for languages and compilers 300
Frank DeRemer, Richard Jullig

An implementation of affix grammars 320
Hans Meijer

Experiences with the compiler writing system HLP 350
Kari-Jouko Räihä

Rule splitting and attribute-directed parsing 363
David A. Watt

Attribute-influenced LR parsing 393
Neil D. Jones, C. Michael Madsen

On the definition of attribute grammar 408
Martti Tienari

RELATED TOPICS

State transition machines for lambda calculus expressions 415
David A. Schmidt

Semantic definitions in REFAL and the automatic production of compilers 441
Valentin F. Turchin

On the formal definition of ADA 475
Veronique Donzeau-Gouge, Gilles Kahn, Bernard Lang

TRANSFORMING DENOTATIONAL SEMANTICS INTO PRACTICAL ATTRIBUTE GRAMMARS*

Harald Ganzinger

Institut für Informatik
Technische Universität München
Postfach 202420
D-8000 München 2
Fed. Rep. of Germany

1. INTRODUCTION

It is generally accepted that, in order to reason about programs, a formal definition of the semantics of the programming language is a prerequisite. Additionally, such a definition must be the basis for the systematic composition and verification of specific implementations of the language, especially compilers. The mathematical method of Scott and Strachey [ScS 71] has been developed to provide a universal concept for describing formally languages and implementation issues [MiS 76].

Independently, much effort has been devoted to the development of practical compiler generators (see [Räi 77] for an overview). These systems are mostly based on the concept of attribute grammars (AGs), as introduced in [Knu 68]. Since attribute grammar definitions describe the compilation of a language in many of its technical details, they cannot be viewed to describe the abstract semantics. Thus, the problem arises, to prove a compiler description correct with respect to an abstract semantics definition, or, even better, to develop a compiler description correctly. This paper is a contribution to a solution of this problem.

*The work reported in this paper was sponsored by the Sonderforschungsbereich 49 -Programmiertechnik- at the Technical University of Munich.

Much work has already been done concerning the development of compilers from denotational definitions of languages and target machines [Bjø 77, Jon 76, MiS 76]. Thereby, the development process has been divided into two major phases. In the first phase, the abstract language definition is transformed into an implementation-oriented definition. The latter defines a language in terms of the data types (operations and objects) of an abstract machine, which is relatively close to the real target machine. In this step, well-known implementation techniques are used implicitly. The transformation is guided by very general software development principles, as that of (stepwise) refining data abstractions [Hoa 72]. The methods for formulating the correctness proofs are well-understood [MNV 73, Rey 74]. The resulting definition is denotational but already effective in the sense that it can be input to an experimental compiler generating system, as described in [Mos 75].

The second phase has to develop the concrete compiler from that implementation-oriented definition. Apart from some very informal arguments in [Bjø 77], we do not know of any satisfactory solution in the literature, illustrating the principles of this second phase. The work reported here is based on the observation that this phase is mostly language and implementation independent, but depends on the meta language to be used for compiler descriptions.

The development phase is divided into three major steps. The first modifies the given denotational definition such that it can considered to be an attribute grammar in some generalized sense: The semantic domains are re-structured into domains of n-ary functions, yielding m-ary results. The function parameters become the inherited, the results the synthesized attributs of that syntactic constructs with which the domain is associated. Equations over parameters and results serve to define the (functional) meaning. The equations are similar to attribute rules.

In the second step, the analyzation of the dependencies between the arguments and results of the semantic objects yields a separation between static and dynamic semantics. The denotational definition in attribute grammar form is splitted into two parts. An attribute grammar (in the classical sense) is obtained for the static semantics. The dynamic remainder is embodied in a description of the interpretation of the attributed program tree with re-

spect to given input data. The handling of semantic errors is added. This leads to rejecting erroneous programs from being interpreted. The principle of defunctionalization [Rey 72] is applied to handle semantic objects such as "procedure addresses" in symbol tables by references to the corresponding nodes in the program tree.

In a last step, the interpretation description has to be transformed into a description of code generation. This step is not dealt with in this paper. For a first approach which is based upon relating interpretation schemes with code templates on a flow graph level, we refer to [Gan 79c]. The primitive operations in these flow graphs constitute an interface to the description of concrete target machines.

The main result of the paper will be that most of the transformation steps are *mechanizable* or, at least, can be supported by *automatic methods*.

We assume that the reader is familiar with the denotational semantics method, at least at the level of [Ten 76]. Additionally, it is necessary that the reader has some acquaintance with the use if attribute grammars for the definition of semantic analysis and the preparation of code generation, at least at the level of [LRS 76].

2. BASIC NOTATIONS

Domains and basic functions

Domains are complete lattices. B denotes, as usually, the boolean domain $\{\bot,\underline{true},\underline{false},T\}$. If $D_1,\ldots,D_m$ are $m\geq 0$ domains and $s_1,\ldots,s_m$ are pairwise distinct symbols, then $D = [s_1 : D_1, s_2 : D_2,\ldots,s_m : D_m]$ is the cartesian product of the domains, where s_i is the name of the i-th selector, i.e. $\lambda x.\ x.s_i$ denotes the projection from D to its i-th component D_i. For $m=0$, D is the domain of empty records, $D=\{\bot,(),T\}$. (We write simply $D = D_1\times D_2\times\cdots\times D_m$, if the selector names are of no interest.) $\underline{make}_D(x_1,\ldots,x_m)$, für $x_i\in D_i$, denotes the tupel $(x_1,\ldots,x_m)\in D$ (We write simply $(x_1,\ldots,x_m)$, if D is determined by the context.)

$D = D_1+\cdots+D_m$, $1\leq m\leq\infty$, is the (separated) sum of the D_i, i.e. $D=\{(i,x)\mid 1\leq i\leq m,\ x\in D_i\}\cup\{\bot,T\}$. For $x\in D$, x $\underline{to}$ D_i denotes that $x_i\in D_i$ which corresponds to x in D, i.e. (i,x_i) $\underline{to}$ $D_i = x_i$ T $\underline{to}$ $D_i = T$, and $\bot$ otherwise. Conversely, for $x_i\in D_i$ it is x_i $\underline{in}$ $D=(i,x_i)$, if $x_i\notin\{\bot,T\}$. Otherwise it is x_i $\underline{in}$ $D = x_i$. (If D is uniquely determined by the context, we sometimes omit $\underline{in}$ D.) $\underline{is}$ D_i tests the type of a $x\in D$, i.e. (i,x) $\underline{is}$ $D_j = \underline{true}$, if $i=j$, and (i,x) $\underline{is}$ $D_j = \underline{false}$, if $i\neq j$. $\bot$ $\underline{is}$ $D_j = \bot$, T $\underline{is}$ $D_j = T$.

Let D be a domain and $D^i = [s_1 : D,\ldots,s_i : D]$, $i\geq 0$, with arbitrary selectors s_j. Then $D^* = D^0+D^1+D^2+\cdots$ is the domain of all lists with elements from D. $\underline{nil}$ (or $\underline{nil}_D$) denotes the empty list $()$ in D^0. $\underline{append}(d^*,d) = (d_1,\ldots,d_n,d)$, if $d^* = (d_1,\ldots,d_n)$, adds an element $d\in D-\{\bot,T\}$ to d^*. $\underline{hd}(d^*) = d$, if $d^* = (d,\ldots)$, yields the head of d^*, whereas $\underline{tl}(d^*) = (d',\ldots)$, if $d^* = (d,d',\ldots)$, eliminates the head in d^*. $\underline{append}$, $\underline{hd}$, and $\underline{tl}$ are *call-by-value* in their arguments, i.e. yield $\bot$, if one of the arguments is $\bot$, and yields T, if one of the arguments is T and if none is $\bot$.

If S is some countable set then $D = S^{\square}$ denotes the flat domain obtained by adjoining $\perp$ and T to S (and by defining $\perp \leq s$, $s \leq T$, for all $s \in S$, $\perp \leq T$, and $s_1 \leq s_2 \Rightarrow s_1 = s_2$, for $s_1, s_2 \in S$). If D' is also a domain and if f is a partial mapping from S into D', then the doubly strict extension of f to $S^{\square}$ (i.e. $\perp \mapsto \perp$, $T \mapsto T$, $x \in S \mapsto f(x)$, if $f(x)$ defined, $x \in S \mapsto \perp$, otherwise) is also denoted by f.

If D_1 and D_2 are domains, then $D = [D_1 \to D_2]$ is the domain of continuous functions from D_1 into D_2. For $f \in D$, $x \in D_1$, and $y \in D_2$, $f[x \leftarrow y]$ denotes that function which is identical to f, except that it maps x to y, i.e. $f[x \leftarrow y](z) = f(z)$, if $z \neq x$, and $f[x \leftarrow y](x) = y$. fix denotes the fixpoint operator, i.e. fix(f) yields the least fixpoint of $f \in D$. The conditional $c =$ *if* p *then* x_1 *else* x_2, for $p \in B$, $x_1, x_2 \in D, D$ any domain, is doubly strict in p, i.e. $c = \perp$, if $p = \perp$, $c = T$, if $p = T$, $c = x_1$, if $p =$ *true*, and $c = x_2$, if $p =$ *false*. A function f is said to be total in an argument x, if $f(\ldots,x,\ldots) \in \{\perp, T\}$ implies $x \in \{\perp, T\}$. Our notation for defining (continuous) functions used in the sequel is very close to the notation for untyped lambda expressions introduced in [Don 76]. It is defined in the appendix.

Grammars and syntactic domains

We assume, for simplicity, that in any context-free production no grammar symbol occurs twice. To achieve this, indexes can be attached to the symbols. Moreover, ε-productions are forbidden.

Syntactic trees t, which we call *program trees*, are defined as usual: t is a labelled, ordered, rooted, and finite tree. The label $X(t,u)$ of a node u in t is a grammar symbol. At each node u in t, which is not a leaf, some syntactic production rule $prod(t,u) = X_0 \to X_1 \ldots X_n$ is applied, i.e. it is $X(t,u) = X_0$ and u has n sons $u_1, \ldots, u_n$ in t, such that $X(t,u_i) = X_i$. The root of t is labelled with the start symbol. Subsequently, $son_i(t,u)$ denotes the i-th son of u in t, if $i > 0$. For $i = 0$ it denotes u itself. Instead of $son_i(t,u)$ we also write $u \circ i$ or $son_i(u)$, if t is determined by the context.

Additionally, lexical information lexinf(t,u) is attached to the terminal leafs u in t.

We denote by TREES the flat domain $\{(t,u) \mid t \text{ syntactic tree, } u \text{ node in } t\}^{\square}$ of tree configurations given by a context-free grammar. Members of TREES are denoted by τ, τ', τ_1, etc. (The doubly strict extensions of the above functions X, prod, son_i, and lexinf are the only standard functions over TREES which we allow to use in the following.)

3. DENOTATIONAL DEFINITIONS AND INTERPRETATIONS

Denotational definitions are given by semantic valuation functions which map constructs in the program to the abstract values (numbers, functions, etc.) which they denote. The valuation functions are defined recursively. The value denoted by a construct - the meaning of the construct - is specified in terms of the values denoted by its syntactic components.

For later purposes, we have to define this more formally. A denotational definition associates

- *semantic domains* $D[X]$ with each grammar symbol X,
- *semantic functions* Σ_p with each syntactic rule p, such that $\Sigma_p \in \left[\prod_{i \geq 0} D[X[p,i]] \right] \to D[X[p,0]]$, if $X[p,i]$ is the i-th symbol in p.

Thereby it specifies for any program tree t and any node u in t a *meaning* $\mu[u] \in D[t(u)]$, given by

- $\mu[u] = \text{lexinf}(t,u)$, if u is a (terminal) leaf
- $\mu[u] = \text{fix}(\lambda\mu.\Sigma_p(\mu,\mu[u \circ 1],\ldots,\mu[u \circ n_p]))$, if the rule p is applied at u, where the fixpoint operator is applied to solve the possible recursions.

When considering the syntactic domain TREES, we extend μ to $\mu \in [\text{TREES} \to D]$, D being the sum of all semantic domains, defined by

$$\mu(\tau) = \begin{cases} \bot, & \tau = \bot \\ \top, & \tau = \top \\ \mu[u] \text{ in } D, & \tau = (t,u). \end{cases}$$

Denotational definitions of a language can be given on different levels of abstraction. Following the classification in [MiS 76] and [Sto 77], the most abstract definition operates with domains of complicated functionality and rich structure, and is free of implementation details, i.e. avoids prejudicing an implementer towards any particular technique. Such an definition is called the *standard semantics* of the language. Our example which will be giv-

en in the next section is located somewhere between the *store* and the *stack semantics* levels (cf. [MiS 76]), since it already contains strategies for symbol table administration and storage allocation. In fact, we need such a level as our starting point in this paper: Since we want to study the both language and implementation *independent* aspects of compiler description development, we start from a level of abstraction where the standard semantics definition has already been refined by introducing various implementation-oriented concepts (as shown in [MiS 76, Bjø 77, Jon 76, Sto 77]). This description is neither an interpreter nor a compiler description yet, but it describes the meanings by combining primitive and implementation-oriented data types, using the standard constructors (including fix), into objects of almost arbitrary complexity.

Such definitions are the starting-point for applying the development principles to be investigated in this paper.

Interpreter definitions are used to define an interpretation η of program constructs. In contrast to denotational definitions, they operate explicitly on the tree configurations, i.e. on the syntactic domain TREES. They also attach semantic domains D[X] to the grammar symbols X and semantic functions ε_p, called *interpretation functions*, to the syntactic rules $p = X_0 \to X_1 \ldots X_n$. But this time it is $\varepsilon_p \in [\text{TREES} \to D[X_0]]$. Thus, the ε_p can refer to all operations on trees, e.g. son_i, lexinf, in particular, recursively, to the interpretation function η itself. η is specified by the ε_p as follows: $\eta \in [\text{TREES} \to D]$, where D is the sum of all D[X], and

$$
\begin{aligned}
\eta(\tau) = \ & \underline{\text{rec}}\ \underline{\text{if}}\quad \text{prod}(\tau) = p_1\ \underline{\text{then}}\ \varepsilon_{p_1}(\tau)\ \underline{\text{in}}\ D \\
& \underline{\text{elsf}}\ \text{prod}(\tau) = p_2\ \underline{\text{then}}\ \varepsilon_{p_2}(\tau)\ \underline{\text{in}}\ D \\
& \ldots\ . \\
& \underline{\text{elsf}}\ \text{prod}(\tau) = p_n\ \underline{\text{then}}\ \varepsilon_{p_n}(\tau)\ \underline{\text{in}}\ D \\
& \underline{\text{else}}\quad -\ -\quad \tau \text{ is terminal} \\
& \qquad \text{lexinf}(\tau)\ \underline{\text{in}}\ D,
\end{aligned}
$$

if $p_1, \ldots, p_n$ are the syntactic rules of the grammar.

Any denotational definition can be regarded as an interpreter definition by defining

$\varepsilon_p(\tau) = \Sigma_p(\eta(\tau)\ \underline{\text{to}}\ D[X_0], \eta(son_1(\tau))\ \underline{\text{to}}\ D[X_1], \ldots, \eta(son_n(\tau))\ \underline{\text{to}}\ D[X_n])$, if p as above.

One verifies easily that in that case in fact $\mu = \eta$ holds.

Note that due to the fact that our syntactic domain TREES is a primitive one, there is no possibility of defining semantic aspects by complex manipulations of the source programs as in [AdB 77].

The next section presents a denotational language definition from which we shall develop a compiler description.