

# Language Definition and Implementation

Michael Oudshoorn & Chris Marlin

*Department of Computer Science  
The University of Adelaide  
Adelaide, South Australia*

## **ABSTRACT**

Many languages have been designed to date, of which a large number have never been implemented and the majority are specified in a very imprecise manner. For a language to receive serious consideration among the computer science community, it generally must have been implemented after its design. However, it is the matter of a precise definition which is often forgotten or ignored and yet it is one of the most important aspects of a language when considered in the context of the evolution of programming languages. The language definition is the vehicle by which the rules regarding syntax and semantics are clearly stated and conveyed. It allows for the comparison of languages independent of any implementation or machine architecture. Furthermore, by employing techniques with a formal basis, it is also possible to use the language definition as the source for the automatic generation of a language implementation. This strongly suggests that language designers should precisely define their programming languages as they are designed. Designers would be encouraged to follow this paradigm if a language implementation could be generated from the definition with minimal effort. This paper describes a language definition technique suited to this approach and outlines its benefits.

**Keywords and Phrases:** Interpreter, Language Definition, Language Design

**CR Category:** D.3, Programming Languages; F.3, Logics and Meaning of Programs.

## **1. Introduction**

The designer of a new programming language has much to strive for and attain in order to gain any chance of widespread acceptance of the language. The language must be designed and implemented in order to produce a product that can be distributed, but it should also be defined clearly so that the problems and ambiguities encountered in the early definition of languages such as Pascal [Jensen78, Wirth71] and Modula-2 [Wirth85] can be avoided. The best means by which to avoid these problems is to employ a formal technique for the definition. Many such techniques exist, including attribute grammars [Uhl82], denotational semantics [Ganzinger82, Tennent76] and the Vienna Definition Method [Bjørner78]. This paper uses an operational semantics approach to the definition of programming languages, based on the model presented in [Marlin85, Oudshoorn88].

By expanding this model, a language definition can be developed which is easy to read and understand by programmers and compiler writers alike, whilst also being the basis for the generation of an interpreter for the language. This advantage is often ignored, or only partly used, as many of the systems available to automatically generate compilers from such formal specifications only produce code to handle one aspect of the language, such as static semantics.

By designing, implementing and defining a programming language as a single activity, the language designer is then able to provide the entire programming community with a formally defined language for which a compiler prototype is available. The principal benefit of adhering to this paradigm is that the language is defined formally, as it is designed, and an interpreter can be generated at any time, rather than the current practice of designing a language, then implementing it without adhering to a formal definition. Such a language definition would also serve as the basis for the comparison of programming languages, whilst the automatically generated implementation allows the designer to experiment with the language features and evaluate their usefulness cheaply and quickly. Using the approach described in this paper, a language designer can be confident of consistency between intended semantics of the language and that represented by the implementation.

This paper outlines a system which follows the above approach, and is called ATLANTIS (A Tool for LANguage definiTiOn and Interpreter Synthesis). Section 2 provides a brief summary of the model which underlies the system and which is described more fully in [Marlin85, Oudshoorn88]. Section 3 considers the language definition, and discusses the manner in which ATLANTIS defines the syntax and semantics of a language and the way in which this definition is transformed to produce an interpreter or compiler prototype. The final section puts forward some conclusions and gives an indication of planned future work.

## **2. Overview of ATLANTIS**

The model upon which ATLANTIS is based is a multi-layer, multi-pass information structure model based on Abstract Data Types (ADT's). By using a model consisting of multiple layers, it is possible to cater for the needs of various groups, such as compiler writers and programmers. The aim is to allow the groups to choose the level of detail to which they want to expose themselves; that is, a language definition has a formal basis capable of specifying the semantics to the level of detail required by a compiler writer and language designer, while still being sufficiently abstract to be useful to programmers. The multi-pass nature of the model provides a mechanism for the clearer specification of programming language semantics than a single pass technique. The model used is an information structure model, meaning that the programming language semantics is given in terms of manipulations of some data structures known as information structures.

Basing the model on ADT's provides the necessary formality [Gougen77, Gougen78, Guttag80] to be able to derive an implementation prototype automatically, and provides the key to the layering of the model. The innermost layer defines the information structure used in the model in terms of ADT's; these are specified algebraically, as illustrated in Figure□1. The middle layer combines ADT manipulations into useful operations known as High-Level Operations (HLO's) and the outermost layer describes the language semantics. Readers of the language definition are then able to read to the depth they require; as each layer is accompanied by a natural language commentary which is not considered to be a part of the language definition proper, the reader can depart from the formal definition to fill in any desired background information from the informal description.

The model is multi-pass to cater for languages such as Modula-2, which are designed to be compiled by multi-pass compilers and which consequently require a multiple pass definition. However, such languages are not the only ones to require more than a single pass. Seemingly single pass languages, such as Pascal, frequently require more than a single pass to correctly define some aspect of their semantics. Pascal, for example, requires that the defining point of an identifier (the point at which it is defined and given semantic meaning) occur before an applied occurrence of the identifier (any point at which the identifier is used) within that block [BSI82]. Each of the Pascal programs in Figure□2 is erroneous, as the applied occurrence occurs before the defining point of the identifier within a block.



is procedure "P" and hence the call to "P2" within "P1" refers to the "P2" of the inner scope and not the outer scope; in other words, the rule of "declaration before use" is violated.

To model the semantics of Pascal correctly, the first pass over the source code records all locally defined identifiers in the information structure and immediately prior to the statement-part of the block, all identifiers defined local to the nearest textually enclosing block are inherited into the current block of interest provided that they were not redefined there. In the second pass, all subsequent identifiers which were inherited into the nearest textually enclosing block via the first pass, which have not been redefined in the current block, are inherited. This approach ensures that all errors in the programs in Figure 2 are detected and it enables the error to be flagged at the correct location within the program.

Although the Pascal definition [BSI82] goes to great lengths to explain what constitutes an error with respect to the scope rules of the language, it is clear that compiler writers had difficulty interpreting the natural language definition used precisely and consequently most compilers do not detect errors in all of the programs in Figure 2. In many cases, if the order of the declaration of parameters in the program shown in Figure 2(b) is reversed, the compiler detects only one or other of the programs as being incorrect. Clearly, a reversal of the parameters is not going to affect the semantics of the program to the extent that one of the programs is incorrect and the other is correct. A formal definition which allows multiple passes over the source code in order to build its information structures rectifies this situation.

### **3. The language definition**

#### **3.1 Outline**

Programming language syntax is generally specified in BNF [Naur60] or some variant of EBNF. This notation is well understood by all members of the programming community and is widely used. Programming language semantics, on the other hand, may be specified in a variety of ways ranging from very informal, to highly mathematical. ATLANTIS was developed to be of use to programmers wanting to find answers to simple questions about a language, and to compiler writers and language designers wanting answers to more complex questions.

Three aspects of a language definition are recognized by ATLANTIS: the lexical, syntactic and semantic aspects. The lexical symbols are defined first and this definition takes the form shown in Figure 3. In order to facilitate ease of change of the language design, the designer must specify a symbol to match each of the keywords in the language. Hence, if the designer wishes to change a keyword, only one change needs to be made to the language definition. Operators are also defined in this lexical section and it is here that their precedence level and associativity is specified. Finally, special lexical tokens are defined in terms of an EBNF definition, as shown in Figure 3, where an identifier is defined to be one or more arbitrary alphabetic letters.

## Language Example is Case Independent

### Keywords

```
begin_sym    begin ; -- comments are as in Ada
plus         \+ ;   -- \+ represents the plus sign
...
```

### Operators

```
plus        2    left ; -- precedence of 2, and left associative
...
```

### Special

```
identifier   {\l}+ ; -- \l denotes the set of all uppercase and lowercase letters.
              -- {...}+ denotes one or more occurrences of ...
...
```

**Figure 3.** The lexical portion of an ATLANTIS definition.

ATLANTIS uses EBNF to specify the language syntax and intersperses the semantic definition throughout the EBNF to highlight the places where the semantic actions have an effect. The result is a syntactic definition of the language which has a similar appearance to yacc [Johnson75]. Figure 4 shows an example of the mixed syntax and semantics for the definition of a Pascal block, according to that described in [BSI82]. Uppercase identifiers represent syntactic units which may be defined elsewhere. Text enclosed between "[...]" denote an optional object. The semantic calls are delimited by "%%" and correspond to calls to semantic routines in the outermost layer of the model described earlier. If the language requires multiple passes, then the pass on which the semantic action is to have an effect is specified, using "Pass *n*".

```
BLOCK:  %% Pass 2: Scope_Rules_Pass_2 %%
        [ LABEL_DECLARATION_PART ]
        [ CONSTANT_DEFINITION_PART ]
        [ TYPE_DEFINITION_PART ]
        [ VARIABLE_DECLARATION_PART ]
        [ PROCEDURE_AND_FUNCTION_DECLARATION_PART ]
        %% Pass 1: Scope_Rules_Pass_1 %%
        STATEMENT_PART
```

**Figure 4.** Definition of a Pascal block.

The technique used in ATLANTIS not only ties the semantic actions to the syntax of the language, breaking it into multiple passes to highlight the relationship between aspects of the language, but also encourages the separation of the static and dynamic semantic. This separation makes the language definition easier to use, both when developing the language initially and when reading and understanding the definition. This is unlike most natural language descriptions which mix the two semantic aspects in an almost inseparable manner, as in [BSI82] and [DoD83]. The result of the ATLANTIS approach is a syntactic definition of the language which is reasonably easy to read and which has the semantic description firmly embedded.

Linking the semantics to the syntax in ATLANTIS does not force the language designer to define either the language syntax before the semantics, or vice versa. Such language design issues lie outside the scope of this paper; however, ATLANTIS will support either approach to language design with equal ease.

The semantic definitions are based on the information structure model outlined in Section 2. The interested reader is referred to [Marlin85, Oudshoorn88] for a more detailed discussion. The semantics are defined as for the outermost layer of the model and make heavy use of the layer below, namely the HLO's. Certain constructs are allowed throughout the semantic definitions as part of the manipulation of the information structures. These constructs include sequencing, as in

$$S_1; S_2;$$

which indicates that statement  $S_1$  is to be executed and completed before statement  $S_2$  commences. A conditional statement of the form

```
if <condition>
then S1;
else S2;
end if;
```

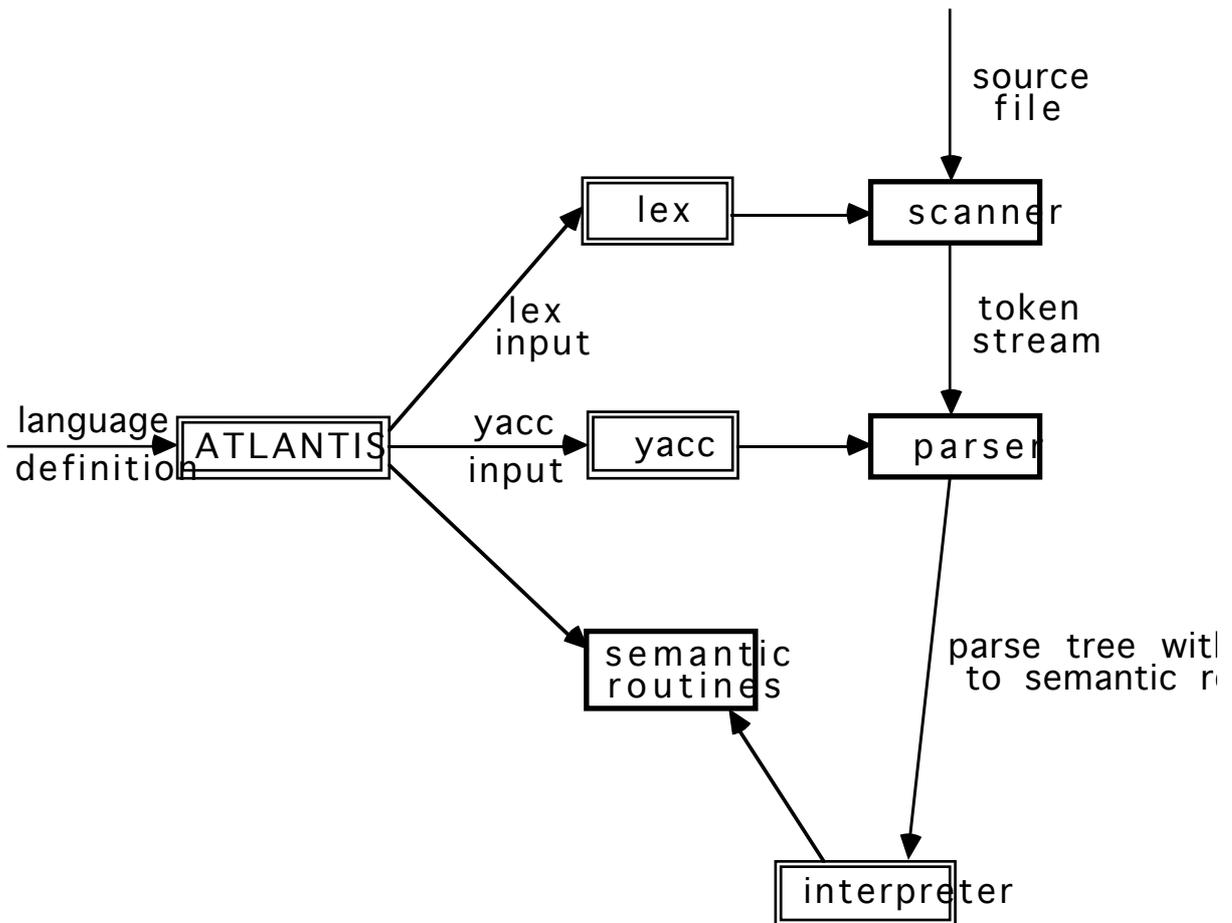
is permitted and has the conventional semantics. This definition can be applied recursively to build **repeat** and **while** loops, which in turn can be used to define a fixed iteration construct such as a **for** loop. Both sequencing and if construct can be defined formally [Guessarian84], hence the formal nature of the model is maintained.

The innermost layer of the semantic definition of the programming language is the definition of the information structures used in the description of the language. This is given in terms of an ADT definition in the style of the ADJ group [Gougen77] or Guttag [Guttag80]. The ADT's can be shown to be consistent and sufficiently-complete (i.e., showing that a unique result exists for every conceivable situation) [Gougen77, Guttag80, Oudshoorn85], thereby providing a formal framework for the model.

Annotating each layer of the semantic description with a natural language narrative provides a mechanism whereby the definition can be of use to programmers and compiler writers alike. Since the natural language description does not constitute part of the formal language definition, and because it is derived directly from the formal definition (by the language designer), the opportunity to introduce ambiguities is reduced. This abolishes the need to produce multiple description of the language to cater for the varying needs across the programming community. However, it is unlikely that any definitional technique will have an effect on the volume of programming language text book sales, but it is hoped that such a detailed language definition and narrative will prove useful in guiding authors through the programming language semantics and hence enable them to write better texts.

### 3.2 Generation of an interpreter

Figure 5 provides an outline of the ATLANTIS system; note that the components which have double outlines are supplied and are used unaltered, whereas those with a heavy outline are the parts of the resulting language implementation which are generated (directly or indirectly) from the language definition. The lexical and syntactic parts of an ATLANTIS definition map to input suitable for lex [Lesk75] and yacc, respectively. The generated scanner produces a token stream, which is passed to the generated parser; this parser, in turn, produces a parse tree of the source program provided there are no syntax errors. The parse tree is annotated with calls to semantic routines produced from the description of the semantics of the language in the ATLANTIS definition; these semantic routines are to be called in various passes over the parse tree.



**Figure 5.** The ATLANTIS system.

The semantic routines are a pool of Ada routines generated from the semantic actions and HLO's of the language definition. As the underlying model of ATLANTIS is based on ADT's, it is necessary to translate the ADT definitions into Ada packages. This aspect of the language definition is performed by hand at present. Although initially labour intensive, this aspect of the interpreter has a high degree of reusability, as the ADT's used are likely to be fundamental to many language definitions; hence an investment of time in this area will be rewarded later when another language is defined. Such a high level of reusability also results in a firm basis over which different languages may be compared.

The mapping of the information structure model used to define the language semantics to concrete code ensures that the interpreter correctly interprets the source program. If the language designer finds he is not satisfied with the results obtained when using the interpreter, the problem lies in the semantic description designed for the language, not the interpreter. By using the information structure model approach to language design, the language designer is forced to consider all aspects of the programming language semantics carefully; by using the generated interpreter, the language designer can check that the semantic definition corresponds to his or her intentions. Omissions and errors in the language definition may also be detected by use of the interpreter.

The generated parser checks the source for syntactic errors and flags them accordingly. If there are no syntax errors then an annotated parse tree is produced. This parse tree is used to perform semantic analysis and effectively interpret the source code by way of performing a post-order traversal of the parse tree. Each semantic action is executed during the tree traversal and hence the information structure described in the language definition is built and manipulated. The result of the source program will be reflected in the state of the information structure when all the necessary passes over the parse tree are complete. The language designer is now in a position to design and test his ideas as well as ensure that his language definition accurately reflects the semantics he intended. Any errors or omissions in the language definition will soon become apparent through usage of the interpreter.

Interpretation of the submitted program occurs via a post-order traversal of the parse tree. Each semantic action is executed during the tree traversal and hence the information structure described in the language definition is built and manipulated. This interpretation allows a language designer to verify that a language definition accurately reflects the intended semantics. Errors or omissions in the language definition will soon become apparent through usage of the interpreter.

As an example, given the rules in Figure 6(a), and "ACC" as input, then a parse tree as illustrated in Figure 6(b) will be produced. Performing a post-order traversal over the annotated parse tree results in the execution of the semantic actions at locations corresponding to their place in the rules of the language definition, that is Figure 6(a). If the language definition requires multiple passes to correctly specify its semantics, then multiple passes over the parse tree will be required. Figure 6(b) shows semantic actions,  $s_i$ , attached to leaf nodes of the parse tree. These semantic actions are executed as the node is left during the postorder traversal of the tree. This results in the need for empty leaf nodes to which semantic actions are attached to be introduced into the parse tree.

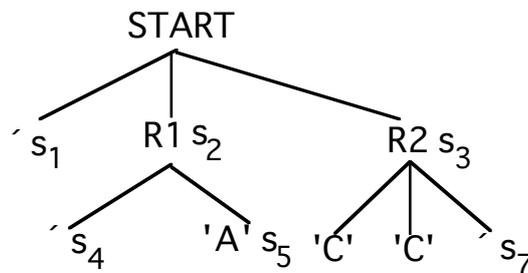
By using the information structure model of the programming language in order to execute a source program it is clear that although the interpreter may not be efficient in its execution, it is accurate in its interpretation of the language semantics.

```

START: %% s1 %%
R1
%% s2 %%
R2
%% s3 %% ;
R1:
%% s4 %%
[ 'A' %% s5 %% ]
| 'B' %% s6 %% ;
R2:
{ 'C' } %% s7 %% ;

```

(a)



(b)

**Figure 6.** A simple language definition and associated annotated parse tree.

The reason for generating an interpreter rather than a compiler at this stage is to ensure that the language definition remains free of any machine dependencies which may occur if a compiler producing native machine code were generated. The interpreter provides a system which allows language design experiments. The aim of ATLANTIS is not the generation of a production quality compiler, but rather a tool for the formal definition of programming languages and a mechanism whereby a language designer can thoroughly test and develop a design by using an implementation of the language even though it is still undergoing evolutionary changes.

#### **4. Conclusions and Future Work**

The approach described in this paper encourages formality in language design by linking together hitherto distinct processes: design, definition and implementation. The generation of the implementation from the language definition allows the designer to experiment and fully explore his ideas in language design comfortable in the knowledge that the implementation matches the formal definition precisely. It is also the generation of the interpreter which provides the incentive to the language designer to produce a formal, unambiguous language definition.

ATLANTIS provides a tool which supports this approach by automatically generating an interpreter for the language being defined. ATLANTIS also provides an additional benefit in that the language definition has a formal basis and yet, due to the layering and multi-pass nature of the information structure model employed, it provides a definition that is readable and usable to a wide class of users. Programmers, normally preferring an informal natural language approach because such descriptions are easier to read, will be able to find answers to questions they may have on the language by reading to a depth most appropriate to them; compiler writers, who require a more precise definition, are provided with a detailed, clearly defined specification. Finally, the language designer benefits by using ATLANTIS, since an implementation is obtained with little additional effort as a result of designing and defining the language. At this point in time the scanner and the parser is automatically generated from a language definition, and work is progressing on the translation of the semantic routines and HLO's to Ada routines. This is occurring in conjunction with the development of the driver to perform multiple post-order traversals over the parse tree and invoke the appropriate semantic routine.

Although the principal aim of ATLANTIS is not to produce a production quality compiler generating native code, future development will proceed in this area. Attempts will be made to employ a code generator generator [Ganapathi82, Ganapathi85, Graham80, Graham82] to produce a more efficient compiler, rather than the somewhat slower interpreter. Definition of the instructions which will need to be generated in order to handle language features for a specific machine architecture will have to be kept separate to the language definition. Those features which do not fit either category highlight the grey area between language definition and language implementation where designers believe it to be the implementors' job and vice versa.

At present, the ATLANTIS system only handles sequential languages such as Pascal. In order to provide the same benefits to the developers of parallel languages, the underlying information structure model needs to be expanded. This can be done by introducing a new layer, directly above the ADT layer, to handle several processes trying to access the information structures; this layer would employ Shared Data Abstractions (SDA's), described in [Freidel84, Freidel88, Mallgren83]. ATLANTIS would then provide a language definition technique able to describe both sequential and parallel languages, whilst providing a formal and readable language definition from which an interpreter or compiler prototype can be produced automatically.

#### **5. Acknowledgements**

Part of the work described here was undertaken at the University of St. Andrews during the study leave period of the second author, and was supported by SERC grant number GR/E□75395.

## References

- [Bjørner78] D. Bjørner and C.B. Jones (Editors). *The Vienna Development Method: The Meta-Language*. Volume 61 of *Lecture Notes in Computer Science*, Springer-Verlag, 1978.
- [BSI82] Specification for the computer programming language Pascal. Publication BS□6192:1982, British Standards Institution, London, 1982.
- [DoD83] U.S. Department of Defense. *Ada Programming Language, ANSI/MIL-STD-1815A*, Military Standard, U.S. Department of Defense, Washington D.C., January 1983.
- [Freidel84] D.H. Freidel. *Modelling Communication and Synchronization in Parallel Programming Languages*. Ph.D. Thesis, The University of Iowa, Iowa City, Iowa, May 1984 (Technical Report 84-01).
- [Freidel88] D.H. Freidel, C.D. Marlin and M.J. Oudshoorn. *Modelling communication in Ada with shared data abstractions*. Technical Report 88-06, Department of Computer Science, The University of Adelaide, Adelaide, South Australia, December, 1988.
- [Ganapathi82] M. Ganapathi and C.N. Fischer. Description-driven code generation using attribute grammars. In *Conference Record of the Ninth Annual A.C.M. Symposium on Principles of Programming Languages*. Albuquerque, New Mexico, pages 108–119, January 1982.
- [Ganapathi85] M. Ganapathi and C.N. Fischer. Affix grammar driven code generation. *A.C.M. Transactions on Programming Languages and Systems*. Volume 7, number 4, pages 560–599, October 1985.
- [Ganzinger82] H. Ganzinger. Denotational semantics for languages with modules. In *Formal Description of Programming Concepts*, D. Bjørner (Ed), North-Holland, pages 3–23, 1982.
- [Gougen77] J.A. Gougen, J.W. Thatcher, E.G. Wagner and J.B. Wright. Initial algebra semantics and continuous algebras. *Journal of the A.C.M.* Volume 24, number 1, pages 68–95, 1977
- [Gougen78] J.A. Gougen, J.W. Thatcher and E.G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R.T. Yeh, editor, *Current Trends in Programming Methodology*, chapter 5, pages 80–149, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.
- [Graham80] S.L. Graham. Table-driven code generation. *I.E.E.E. Computer*. Volume 13, number 8, pages 25–34, August 1980.
- [Graham82] S.L. Graham, R.R. Henry and R.A. Schulman. An experiment in table driven code generators. In *Proceedings of the SIGPLAN'82 Symposium on Compiler Construction*. Boston, Massachusetts, June 1982. Also in *A.C.M. SIGPLAN Notices*, Volume 17, number 6, pages 32–43, June 1982.
- [Guessarian84] I. Guessarian and J. Meseguer. *Axiomatisation of "IF...THEN...ELSE" revisited*. Technical Report 84-18, Laboratoire Informatique Theorique et Programmation, Université P. et M. Curie, Paris, April 1984.

- [Gutttag80] J.V. Gutttag. Notes on type abstraction (version 2). *I.E.E.E Transactions on Software Engineering*. Volume SE-6, pages 13–23, January 1980.
- [Jensen78] K. Jensen and N. Wirth. *Pascal. User Manual and Report*. Springer-Verlag, second edition, 1978.
- [Johnson75] S.C. Johnson. *Yet Another Compiler Compiler*. Computer Science Technical Report Number 32, Bell Laboratories, Murray Hill, New Jersey, 1975.
- [Lesk75] M.E. Lesk and E. Schmidt. *Lex –□A Lexical Analyzer Generator*. Computer Science Technical Report Number 39, Bell Laboratories, Murray Hill, New Jersey, October 1975.
- [Mallgren83] W.R. Mallgren. *Formal Specification of Interactive Graphics Programming Languages*. M.I.T. Press, Cambridge, Massachusetts, 1983.
- [Marlin85] C.D. Marlin and M.J. Oudshoorn. Using abstract data types in a model of the data control aspect of programming languages. In *Proceedings of the Eighth Australian Computer Science Conference*, Pages 19-1 –□19-10, Melbourne, February 1985.
- [Naur60] P. Naur. Report on the algorithmic language ALGOL 60. *Communications of the A.C.M.* Volume 3, pages 299–314, 1960.
- [Oudshoorn85] M.J. Oudshoorn. *A study of algebraic specification techniques and the implementation of abstract data types*. Technical Report 84–04A, Department of Computer Science, The University of Adelaide, Adelaide, South Australia, February 1985.
- [Oudshoorn88] M.J. Oudshoorn and C.D. Marlin. Describing data control in programming languages. In *Proceedings of the I.E.E.E. International Conference on Computer Languages 1988*, Miami Beach, Florida, pages 100–109, October 1988.
- [Tennent76] R.D. Tennent. The denotational semantics of programming languages. *Communications of the A.C.M.* Volume 19, number 8, pages 437–453, August 1976.
- [Uhl82] J. Uhl, S. Drossopoulou, G. Persch, G. Goos, M. Dausmann, G. Winterstein and W. Kirchgässner. An attribute grammar for the semantic analysis of Ada. *Lecture Notes in Computer Science*. Volume 139, Springer–Verlag, 1982
- [Wirth71] N. Wirth. The programming language Pascal. *Acta Informatica*. Volume 1, number 1, pages 35–63, January 1981.
- [Wirth85] N. Wirth. *Programming in Modula-2*. Springer-Verlag, third, corrected edition, 1985.