Compiler Construction
The Art of Niklaus Wirth

Hanspeter Mössenböck
University of Linz
moessenboeck@ssw.uni-linz.ac.at

Abstract

Niklaus Wirth is not only a master of language design but also a pioneer of compiler construction. For four decades he has refined his techniques for building simple, efficient and reliable compilers. This paper tries to collect some general principles behind his work. It is not a paper about new compilation techniques but a reflection about Wirth's way to write compilers.

1 Introduction

Among the numerous research topics that Niklaus Wirth approached during his professional career, programming languages and their compilers clearly was the dominating theme. There is hardly anybody in Computer Science who has such a deep insight into language design as Niklaus Wirth. Since the early sixties he has been active in this field and has contributed such seminal languages as Algol-W, Pascal, Modula-2 and Oberon. And being an engineer, he was always as much concerned about the simplicity and efficiency of his compilers as about the elegance of his languages.

In this paper I will try to summarize some principles of Wirth's art of compiler construction. Although I have no personal recollections from Wirth's early projects I have studied his compilers from the early eighties on and was lucky to have the chance to work with him for several years during the Oberon project. So this paper is more about what I learned from him and what I appreciate about his work. Wirth himself gave a much more thorough description of his compiler technology in his recent compiler book [22], in his Turing Award lecture [2] and in his recollections at the second conference on the History of Programming Languages [21].

When I write about Wirth's art of compiler construction I also want to honour the work of his colleagues and assistants [1, 3, 7, 9, 10, 11, 12]. However, although many of the well-known compilers from ETH Zurich were implemented by his bright PhD students, it was always Wirth who set the direction and determined the style of ETH compilers.

I will start this paper with some general principles that I learned from Wirth and then look at more detailed guidelines and techniques for the individual compilation phases.

2 Guiding Principles

Besides using specific techniques, data structures and algorithms it is also important that a compiler designer follows some general guidelines, which set the framework for the whole development. Here are a few principles, which I believe to be central in Wirth's compilers. The list is perhaps incomplete and somewhat arbitrary but hopefully still some food for thought.

- *Use straightforward techniques*
  Compiler technology is one of the most intensively researched fields in Computer Science. There are highly sophisticated algorithms for syntax analysis, semantic analysis, optimization and code generation. However, more sophisticated techniques do not necessarily lead to better compilers. Wirth's compilers use only the simplest techniques such as recursive descent parsing, context-free semantic analysis and single-pass code generation with only a few optimizations. Nevertheless these compilers generate sufficiently fast code and run much faster than most commercial compilers. A compiler writer should have the heart to refrain from using highly sophisticated algorithms if he can see that more straightforward techniques are perfectly sufficient. In his Turing Award lecture [2] Wirth wrote: "Within the Euler, Algol-W and PL360 projects, much consideration was given to the development of table-driven, bottom-up syntax analysis techniques. Later I switched back to the simple recursive-descent, top-down method, which is easily comprehensible and unquestionably sufficiently powerful, if the syntax of the language is wisely chosen."

- *Simple compilers are more reliable compilers*
  As in any software project it is essential that a single person can grasp the whole design of a compiler. This is what Brooks calls the *conceptional integrity* of a software system [6]. It can only be achieved if the compiler is simple enough. A compiler that cannot be understood as a whole by a single software engineer cannot be guaranteed to be correct and cannot be maintained without introducing new errors. In his Turing Award lecture Wirth confessed: "… that this Modula compiler is already at the limits of comprehensible complexity, and I would feel utterly incapable of constructing a good compiler for Ada."

- *Generate good code for the frequent case; don't optimize the exceptional case*
  It is much more important that frequent language constructs such as procedure calls, array accesses, or general control structures are compiled efficiently than to put too much effort into optimizations such as loop unrolling or induction variable analysis that can only be exploited in rare cases. Wirth's compilers don't use any of the more advanced optimization techniques. Instead they

generate good code for ordinary programs by fully exploiting the machine architecture and its addressing modes.

- *Generate good code from the beginning; don't generate bad code and optimize later*
  Some compilers first generate poor intermediate code because it is optimized later anyway. For such compilers it is often reported that optimizations achieve substantial speedups. However, this is not surprising since the unoptimized code was rather inefficient. If a compiler generates good code from the beginning there is much less room for speedups. Brandis and others [3] report that non-optimizing Oberon compilers for the Motorola 68020, for the Sun SPARC and for the DEC MIPS architectures generated code that was as efficient as the code generated by the standard C compiler with the optimization option switched on (cc -O). Only for the IBM RS/6000 the optimizing C compiler was able to generate more efficient code than the Oberon compiler.

- *Don't let the compiler repair poor programming style*
  Some people expect a compiler to smoothen inefficiencies caused by poor programming style. This includes some cases of common subexpression elimination, loop invariant code motion and copy propagation. It is, however, simpler and often also clearer if the programmer does this transformations in the source code instead of relying on the compiler to do them behind the scenes. Although not all of these transformations can be expressed in source code, most of them can and the remaining ones usually don't cause major inefficiencies, except in some frequently executed inner loops.

- *Generate predictable code*
  Optimizations often have the effect that minor changes in the source code cause major differences in the run time of the program, because a small change in the code may allow or disallow an optimization. Programmer feel uneasy if they cannot predict the behaviour of the generated code. It impedes debugging and benchmarking.

- *What is difficult to compile is usually also difficult to understand*
  Like a programmer, a compiler has to analyze (i.e., to "understand") a program before it can translate it. If a language construct is difficult to analyze for a compiler, it is often difficult to understand for a human, too. For example, in C the construct
  ```
  x (*y) ();
  ```
  declares a function pointer *y* whereas the syntactically similar construct
  ```
  x (*y);
  ```
  denotes the call of a procedure *x*. Since both constructs can occur at the same place in a program, it is difficult to keep them apart syntactically, both for the compiler and for the programmer.

- *Instead of applying more sophisticated compilation techniques, simplify the language*
  Of course, this advice can only be followed if the compiler writer is at the same time also the language designer (as Wirth always was for his compilers). If the implementers of C did not rely on powerful compiler construction tools such as *Yacc*, the syntax of C would probably be simpler.

- *Strive for short compilation times*
  It is not only important that the compiler generates efficient code but that it also runs efficiently itself. These two goals are often in conflict and one has to find a reasonable trade-off. Unfortunately, many commercial compilers emphasize code efficiency and accept long compilation times. However, even a compilation time of 5 to 10 seconds per module can be perceived as unduly long, especially if several modules have to be recompiled after a program modification. During software development, the productivity of programmers can be boosted significantly if their work is not disrupted by unduly long compilation pauses.

- *A compiler is also an educational tool; it should inhibit bad programming style*
  Clearly a programming language can have an educational effect by providing only clean and structured language features. But a compiler can have a similar effect by introducing certain restrictions, which limit the size of a module, the number of imported modules, or the number of global variables to a reasonable maximum. It can also penalize questionable programming constructs such as the access to non-local variables by not optimizing such cases. Although these measures are not very popular with programmers they can have an educational value and may prevent major style deficiencies. Such restrictions are a problem, however, when source code is generated by a tool. In that case the length of names or the size of modules cannot be influenced by the programmer.

- *Write the compiler in its own language*
  For Wirth, language design and compiler construction always went hand in hand. He writes [21]: "I never could separate the design of a language from its implementation, for a rigid definition without the feedback from the construction of its compiler would seem to me presumptuous and unprofessional." Writing a compiler in its own language not only proves that the language is powerful enough but it has also the advantage that any improvement of the generated code will make the compiler faster, too.

Other people will probably find even more principles. Wirth himself published some general compiler guidelines in an early paper [16] but I wish he would do it again today in the light of the Modula and Oberon projects.

In the remaining sections I will look at the individual compilation phases and try to extract some general principles from them.

3 Scanning

There is not much to say about scanning. Wirth's scanners are hand-crafted and optimized since the scanner is one of the most time consuming parts of a compiler. Token positions for error messages are recorded as character positions and not as line/column pairs, which simplifies book-keeping. Attributes and token positions are returned in global variables to avoid parameter overhead for information that is not needed with every token. The length of names is restricted to a reasonable amount (24 or 32 characters). This can be seen as an educational means since unduly long names can inhibit the readability of programs as much as cryptically short names do.

4 Parsing

The design of a parser is mainly determined by the selection of the parsing technique and the error recovering strategy. Since the times of Pascal Wirth's compilers use top-down recursive descent parsing. Recursive descent is the simplest and fastest parsing technique. Although there are more powerful techniques, such as bottom-up LALR parsing, recursive descent is powerful enough if the syntax of the language is properly designed.

Recursive descent parsing requires the grammar of a language to be LL(1), that means it must be possible to analyze the language top-down from left to right with a single lookahead symbol. Whenever there is an alternative in the grammar, the lookahead symbol must lead the way and tell the compiler which alternative to select.

This is probably also the way how humans read and analyze a program. Programmers read source code from left to right. Whenever they read a symbol (keyword, name, number, etc.) they want to know which language construct it belongs to. In contrast, bottom-up parsers read symbols without already deciding to which language construct they belong. They simply push the symbols on a stack. As soon as the end of the stack contains an appropriate right-hand side of a production they reduce this symbol sequence to the corresponding nonterminal symbol. Thus, bottom-up parsers delay the recognition of language constructs in a program as long as possible, which makes them more powerful but also less natural for humans. Programmers don't have large stacks in their heads. They do not want to read many symbols ahead until they know what a program construct means. Programs are notations for humans and not so much for machines. They should be easy to read and therefore suitable for top-down parsing because it is more natural than bottom-up parsing.

Wirth always designed his languages so that recursive descent parsing could be applied. This did not compromise the flexibility or the power of the languages. I know of no language feature that must inherently be expressed in a way that is not amenable to recursive descent parsing. I even think that the availability of powerful bottom-up parser generators such as *Yacc* are responsible for the cryptic syntax of C. If the designers of C were forced to make their language LL(1) it would have probably become more readable.

Unfortunately even Wirth's languages contain a few constructs which are not LL(1). To analyze such constructs, it is not sufficient to rely on syntactic information. One must also take semantic information into account. For example, a type guard *x(T)* in Oberon cannot be distinguished from a procedure call syntactically. One has to consult the symbol table to find out if *x* is a procedure or a variable. This ambiguity could have easily been avoided, for example by using curly brackets instead of round brackets.

The second design decision in a parser is the selection of an appropriate error handling technique. Although good error recovery is more difficult for hand-coded recursive descent parsers than for table-driven parsers, there are a number of suitable standard techniques. The general goal of error handling is to find as many errors as possible in a single compilation and to avoid spurious errors. In addition, error recovery should not slow down error-free parsing and should not blur the parsing code with too many error handling statements.

The standard error recovery technique, which was also applied in the ETH Pascal and Modula-2 compilers, is to pass to every parsing procedure the set of follower symbols of the nonterminal that the procedure is to recognize. If an error is detected, the input is skipped until a symbol is found that is a legal follower of a currently active parsing procedure. The procedure stack is then unwound to the caller of this procedure and parsing can continue there.

Although this technique can be systematically applied, it requires a large amount of set operations and additional parameters of the parsing procedures. This slows down error-free parsing and tends to hide the parsing actions between the error recovery statements. Therefore, Wirth selected a different error recovery strategy for his Oberon compiler:

Given the fact that computers become faster and faster, it is not necessary any more to find *all* errors in a single compilation. If a compilation takes less than a second, it is acceptable to find just *some* errors, correct them, and then compile the program again to find the remaining errors. Thus the Oberon compiler does not synchronize with the followers of every nonterminal symbol but selects a few synchronization points in the grammar where recovery is particularly easy and safe. These synchronization points are the beginning of a statement, a declaration or a type where symbols such as "IF", "WHILE", "TYPE" or "RECORD" are expected. These symbols are considered particularly safe, because they are unlikely to be encountered in the wrong context. If the parser detects an error, it reports it and continues up to the next synchronization point (i.e., to the beginning of the next statement, declaration or type). There it skips input symbols until one of the expected safe symbols occurs. Parsing can be resumed with that symbol. To avoid spurious error messages, an error is only reported, if a certain number of tokens (e.g. three) has been recognized successfully since the last error.

This simple error recovery technique can be implemented in about ten lines of code. It does not slow down error-free parsing and does not burry the parsing statements between error recovery actions. It is another example of Wirth's search for simplicity and adequacy.

5 Symbol Table Handling

A symbol table maps the names in a program to their attributes such as their types and their addresses. It consists of several subtables (scopes), one for every module, procedure and record type. As a consequence of single pass compilation, which Wirth preferred in his later compilers, the scopes can be organized in a stack-like way. A new scope is opened at the top of the stack whenever the parser encounters a procedure, and it is closed when the parser has finished with this procedure. This has the advantage that most nodes of a scope can be discarded (or left for the garbage collector) after the corresponding procedure has been parsed. Only the nodes for the formal parameters have to be kept.

For the implementation of the symbol table, Wirth usually chose rather simple data structures such as binary search trees or even linear lists. Although a hash table would be more efficient for large scopes, it is more complicated to handle (e.g. to traverse, to enlarge or to shrink) and consumes more memory. Since too many names in a scope are often an indication of a questionable programming style, Wirth felt that large scopes should not be encouraged. Measurements showed that for procedure scopes, which usually don't contain more than ten names, a simple linear list is sufficient.

Symbol tables also store the types of objects. For type compatibility checks, Wirth again preferred the simpler technique (name equivalence) over the more powerful but also more complicated technique (structural equivalence). With name equivalence two types are the same if they are denoted by the same type name, i.e. by the same type node in the symbol table. This can be checked with a simple pointer comparison. In contrast, with structural equivalence (as it is used in Modula-3) two types are the same if they have the same structure, i.e. if they have the same components in the same order and if these components are structurally equivalent pairwise. This requires a recursive comparison of the type structures and is not only more complicated but also more time consuming.

A symbol table contains different kinds of objects (e.g., constants, variables, procedures, etc.) as well as different kinds of structures (e.g., basic types, arrays, records, etc.). With type extension, which was introduced in Oberon, it would seem natural to organize the objects and structures in type hierarchies, i.e. in an object-oriented way. There would be a base type with the common data of all object kinds as well as several extensions with the special data for constants, variables or procedures. Although this organization is elegant and is in fact recommended in some compiler books, it is complicated to handle in practice. Since a compiler often works with general objects, their actual extension types would have to be checked using run-time type tests, and their fields would have to be accessed with guarded type conversions. Alternatively, all operations on objects could be defined in terms of dynamically bound methods, but this would require a large number of methods and would compromise the efficiency of the compiler. Besides, there are some operations which depend on more than one operand. This would require methods that are dynamically bound depending on two or more objects (so-called multi methods). Thus, Wirth chose to represent the fields of all objects in a single flat record type and to use only those fields that belong to the respective object variant. Although less elegant, this technique makes the access to object fields simple and efficient. It is another example of an engineering trade-off, which can be found so often in Wirth's compilers.

6 Separate Compilation

A major achievement of Modula-2 was the concept of *separate compilation*, which emerged in the seventies in languages such as Mesa [13] and later also Ada. In contrast to independent compilation (e.g. as in Fortran and C), where subprograms can be compiled independently, separate compilation performs type checking across module boundaries, thus guaranteeing that separately compiled modules can be safely fit together by the linker. When a module is compiled the symbol tables of the imported modules have to be made available as well so that the compiler can access the types of the imported modules. In order to do that, the module interfaces have to be described in so-called definition modules, i.e. in files that contain the declarations of the exported objects.

Instead of recompiling the interface of a module every time this module is imported, Wirth chose to store the precompiled symbol tables in so-called symbol files [10]. Importing a module then simply requires the compiler to read the respective symbol file and to add the imported symbol table to the symbol table of the currently compiled module. Since symbol files are more compact than source files they usually fit into a single disk block and can be read with a single disk access.

A special problem is that the interface of a module can depend on the interfaces of other modules. This would require to read several symbol files when a module is imported. In order to avoid this, symbol files are made self-contained, i.e. they contain also parts of the symbol tables of those modules on which they depend. Thus every import requires just one symbol file to be read. However, it can happen now, that a partial symbol table of the same module gets imported twice from different symbol files. One has to make sure that these repeatedly imported symbol tables are consistent. This can be done by attaching to every interface a unique version number (either a time stamp derived from the compilation time of the interface or a fingerprint derived from the source code of the definition module). By comparing these version numbers, it can easily be checked, if repeatedly imported symbol tables are the same. The version numbers of the imported modules are also stored in the object file of the importing module so that the linker can check if the object files to be linked match the interfaces that were seen during compilation. Thus separate compilation not only performs type checking across module boundaries but also makes sure that consistent versions of modules get bound together by the linker.

In a recent dissertation [8], Régis Crelier refined the version checking between separately compiled modules and implemented it on a finer granularity. The problem with the basic version checking technique is that whenever the interface of a module is extended it gets a new version number. This invalidates all modules that import this interface, even if they use only features that did not change. The new idea is to provide distinct version numbers for all exported objects instead of a single version number for the whole interface. If an object is added to the interface or some exported object is changed, the version numbers of the other objects in the interface remain unchanged. This technique gives a programmer much more flexibility in changing interfaces without having to recompile dependent modules. The implementation of this technique is non-trivial and is described in [8].

As a surprise to the software engineering community, Wirth abandoned the definition modules in Oberon. However, he did not abandon the concept of separate compilation, but only the necessity to write down a definition module explicitly. In Oberon, there are just implementation modules. The programmer simply marks those declarations that are to be exported in the source code, and the compiler generates a symbol file from them during the normal compilation. A definition module can still be generated from the symbol file of a module, but it is only documentation. A similar technique is also used in other modern languages such as Eiffel and Java.

The elimination of explicit definition modules caused much debate among Modula-2 enthusiasts. In fact there are good arguments in favour of definition modules, but there are also good arguments against them. A definition module is a clear description of a module interface. It forces a programmer to think on the design of a module first before starting its implementation. This is good software engineering practice. On the other hand, definition modules introduce redundancy into a language. For example, the interfaces of exported procedures have to be described both in the definition module and in the implementation module. Every time a procedure interface is modified, it has to be modified in two places, and the compiler has to check the consistency of these declarations. Definition modules also make it harder to selectively export record fields, because the record with the exported fields would have to be declared in the definition module and the rest of the record in the implementation module. Finally, explicit definition modules duplicate the number of source files and make it harder to maintain a large software system.

Personally, I think that the elimination of definition modules was a good decision, since it simplifies the language and its compiler. One can still start with designing the interface of a module by writing a skeleton of the implementation module that only contains the exported declarations. Later this skeleton can be filled with flesh by adding private declarations and code. Nevertheless, I must admit that it is tempting to mark the exported declarations only after the whole implementation module has been written.

7 Code Generation

The code generation phase is not as systematical to describe and to implement as the earlier compiler phases. It heavily depends on the peculiarities of the target machine, which is often irregular and full of machine-specific details. Nevertheless the code generators of Wirth's compilers show some common characteristics.

One principle that was already mentioned earlier is to generate reasonably good code from the beginning. This can be achieved by delayed code generation using the *Item technique* described below, by exploiting the special instructions and addressing modes of the target machine, and by applying simple optimizations such as constant folding and strength reduction.

The spine of Wirth's code generators is what I call the *Item technique*. It is described in Wirth's compiler book [22] although it is not given a specific name. The general idea is that every value that turns up during code generation is described by a structure that is called an *item*. Items can describe constants, variables, intermediate results, or condition codes resulting from compare instructions. Every parsing procedure processes a language construct and returns an item that describes the value of this construct. Items are thus the attributes of the imaginary syntax tree. While the parsing procedures move top-down, the resulting items are propagated bottom-up thus creating more complex items and generating code alongside.

For example, the code generation for the addition $x + y$ first leads to two items describing the variables x and y. In the next step, *load*instructions are generated for x and y, and the items describing the variables are transformed into items describing register contents, because the values of x and y are now in registers. Finally an*add* instruction is emitted and the two register items are combined into a single register item, which describes the result of the addition. If the target machine supports the addition of memory operands, only x has to be loaded into a register and y is still described by a variable item.

In many cases, items can be constructed without emitting code. For example, a record selection *r.f* results in an item describing the record variable r and an object describing the field *f*. In the next step, a new item is constructed that describes a memory cell with the base address of *r* and the offset of *f*. No code has to be emitted in this step. The transformation just leads to a compile-time addition of an offset. Using such compile-time operations, code generation is delayed as long as possible. An instruction is only generated if the result of the transformation cannot be produced at compile time.

The kinds of items depend both on the source language and on the target machine. For every object kind in the symbol table (i.e. for constants, variables, fields, procedures, etc.) an item kind has to be provided that describes that object. These are the *source level items*. Similarly, for every addressing mode of the target machine (i.e. for immediate, register, register relative, indexed, etc.) an item kind has to be provided that describes a value accessed by this addressing mode. These are the *machine level items*. Code generation proceeds by transforming source level items (e.g. variables) into machine level items accessed by a specific addressing mode (e.g. register relative). All items are implemented by a flat record that encodes the item kind and its attributes. Usually some source level items are identical to some machine level items (for example, a constant item corresponds to an immediate item, a variable item corresponds to a register relative item, etc). In this case the two item kinds can be combined into a single item kind.

Having identified the necessary item kinds, one has to implement three sets of procedures: (a) procedures that transform objects into source level items (e.g. variable objects into variable items), (b) procedures that transform one item kind into another (e.g. from register relative to register by loading the value into a register), and (c) procedures that generate the actual machine instructions: for every machine instruction *op a, b* one has to implement a procedure *Op(a, b)* where the parameters *a* and *b* are items. This procedure is implemented by a case statement considering all possible combinations of item kinds that *a* and *b* can assume. By applying these guidelines, item kinds and the corresponding code generation procedures can be developed systematically for any given machine.

Another important issue in code generation is register allocation. It is especially important for RISC machines where a large number of registers are available and should be exploited to have faster access to variables and intermediate values. Good register allocation, however, cannot be done in a single pass. It requires life range analysis of programs and a coverage of the life ranges by registers using a technique that is based on graph colouring. Since Wirth's compilers do code generation in a single pass a trade-off is used. The basic register allocation technique is quite simple. Whenever a new register is needed, the register allocator returns an arbitrary free register and marks it as used. At the end of every statement all registers are returned. This technique is simple and efficient but does only exploit a small subset of the available registers.

As a simple optimization, a compiler can try to keep the first few variables of every procedure in registers. However, this can only be done if these variables are not accessed by nested procedures and if they are not passed to other procedures as reference parameters. Instead of allocating a memory cell for such register variables, the symbol table simply holds the register numbers where these variables are stored. Of course, register variables must not be marked as free at the end of every statement. One also has to save and restore register variables across procedure calls, since the called procedures will use the registers for their own variables. A detailed description of this technique can be found in [3].

8 Single Pass versus Multi Pass Compilers

Wirth designed his languages so that they can be compiled in a single pass: every name has to be declared before it is used; pointers and procedures can be forward declared in order to allow cyclic dependencies. If it was necessary a few years ago to decompose a compiler into several passes due to memory restrictions, this is not an argument any more. Current computers have enough memory to fit a whole compiler into it. The predominant reason for multi-pass compilers is optimization. Advanced optimization requires data flow analysis, which has to be performed on an intermediate program representation such as the control flow graph. Building this graph and analyzing it requires several passes. Wirth's compilers avoid sophisticated optimizations, thus no intermediate program representation is necessary and the code can be generated on the fly during parsing. Although the resulting code is not as efficient as it could be, it compares well with the code generated by optimizing C compilers on most platforms (measurements are given in [3]).

There is, however, another good reason to decompose a compiler into two passes: namely portability. If a compiler is decomposed in a language-dependent front end and a machine-dependent back end, porting the compiler to a new machine only requires to rewrite the back end. This solution was chosen in the Oberon OP2 compiler [7], designed by Régis Crelier and ported to about 10 different platforms by various PhD students. All these compilers use the same front end, which contains the scanner, the parser and the symbol table handler. It generates an intermediate program representation in the form of an abstract syntax tree and a symbol table. These data structures are passed to the back end, which contains the code generator.

The syntax tree also allows for more optimizations in the back end such as the elimination of array index checks in certain situations. The back end for the IBM RS/6000 even applies heavy optimizations before code is generated [4]: it turned out that this multiscalar processor requires instruction scheduling to achieve a code quality comparable to the code generated by optimizing C compilers. Therefore the syntax tree is first transformed into a control flow graph using a simplified version of the static single assignment form [5]. Various optimizations such as common subexpression elimination, constant propagation, loop invariant code motion and instruction scheduling are applied to this graph before code is generated.

The intermediate program representation between front end and back end turned out to be of value for still another purpose: mobile code. Michael Franz found a compact encoding of the syntax tree and the symbol table, that can be transferred over a network to some target machine, where it is compiled to machine code by a code generating loader [9]. This is an alternative to the Java bytecodes and allows "write once run everywhere" software. It is interesting to note that this idea was published before the advent of Java.

9 Conclusion

This paper tried to highlight some principles that I believe are central in Wirth's compilers. Maybe my view is biased and incomplete, but I think that it reflects some valuable ideas. I always enjoyed studying Wirth's compilers. Although they contain hardly a line of comment, their structure is so clear, simple and elegant that they are a pleasure to read. The full source code of some of his compilers can be found in his books on compiler construction [22] and on the Oberon Project [20]. The source code of the OP2 compiler is also available online from ETH Zurich as part of the Oberon distribution.

For Wirth, compiler construction is an engineering discipline and not pure science. He prefers well-proven techniques over fashionable results. For him, a compiler is a tool that has to be as useful and practical as possible and not so much a piece of scientific experimentation.

I regard Wirth's way of compiler construction an art, not in the sense of black magic but in the sense of a craft built on four decades of experience.

References

1. Amman U.: The Zurich Implementation, in "Pascal - The Language and its Implementation", Wiley 1978.
2. Ashenhurst R.L., Graham S.: ACM Turing Award Lectures – The First Twenty Years, 1966-1985. ACM Press 1987.
3. Brandis M., Crelier R., Franz M., Templ J.: The Oberon System Family. Software—Practice and Experience 25(12): 1331-1366, 1995.
4. Brandis M.: Optimizing Compilers for Structured Programming Languages. ETH dissertation, 1995.
5. Brandis M., Mössenböck H.: Single-Pass Generation of Static Single Assignment Form for Structured Languages. ACM Transactions on Programming Languages 16(6): 1684-1698, 1994.
6. Brooks F.P.: The Mythical Man-Month—Essays on Software Engineering. Addison-Wesley, 1975.
7. Crelier R.: OP2: A Portable Oberon-2 Compiler. Proceedings of the 2nd International Modula-2 Conference, Loughborough, 1991.
8. Crelier R.: Separate Compilation and Module Extension. ETH dissertation, 1994.
9. Franz M.: Code-Generation on the Fly—a Key to Portable Software. ETH dissertation, 1994.
10. Geissmann L.B.: Separate Compilation in Modula-2 and the Structure of the Modula-2 Compiler on the Personal Computer Lilith. ETH dissertation, 1983.

11. Jacobi C.: Code Generation and the Lilith Architecture, ETH dissertation, 1983.
12. Le V.K.: The Module: A Tool for Structured Programming, ETH dissertation, 1978.
13. Mitchell J.G., Maybury W., Sweet R.: Mesa Language Manual, CSL-79-3, Xerox PARC, 1979.
14. Wirth N.: The Programming Language Pascal. Acta Informatica 1: 35-63, 1971.
15. Wirth N.: The Design of a Pascal Compiler. Software—Practice and Experience 1: 309-333, 1971.
16. Wirth N.: Programming Languages: What to Demand and How to Assess Them. Symposium on Software Engineering, Belfast, April 1976.
17. Wirth N.: Design and Implementation of Modula. Software—Practice and Experience 7(1): 67-84, 1977.
18. Wirth N.: Programming in Modula-2. Springer-Verlag 1982.
19. Wirth N.: The Programming Language Oberon. Software—Practice and Experience 18(7): 671-690, 1985.
20. Wirth N., Gutknecht J.: Project Oberon. The Design of an Operating System and Compiler. Addison-Wesely 1992.
21. Wirth N.: Recollections about the Development of Pascal. Proceedings History of Programming Languages II, Sigplan Notices 28(3): 333-342, 1993.
22. Wirth N.: Compiler Construction. Addison-Wesely 1996.