Backus Naur form and its influence on programming languages

Viktor Gsteiger
University of Basel
Matriculation Number: 18-054-700

October 10, 2020
Seminar: 58826-01 - Turing Award Winners and Their Contributions

Abstract

The contribution of Peter Naur to the programming language Algol 60 and his preliminary work in the field of programming language description was a milestone of great importance to the field of computer science and the field of programming language design in particular. The simple but powerful Backus Naur form, an extension of Naur to the already existing Backus form, has shown to be the new standard in language description from its use in the Algol 60 report on. The design and logic behind Algol 60 proved to be the foundation on which a large subset of all programming languages still in use today has been built upon. Naur's thinking ahead and in larger meta-questions rather than small technicalities have demonstrated to be fundamental for the strength of Algol based languages. Since the importance of Naur's work, it is a worthwhile undertaking to restate the most important parts of Naur's contribution to the field of programming language description, to Algol as well as his influence on the field of compiler design. This report is aimed at bachelor level students of Computer Science, however, we hope to appeal to a wider audience as well.

1 Introduction

The content of this paper will present a fairly detailed review of the development of formal notations of languages. The main focus of the first half of the report will lie in the development leading up to the notation used in the Algol 60 report and the notation used in the report itself. The subsequent parts will establish the link between Naur's further contributions to the topic of formalization with applications like the Gier Algol Compiler and beyond. We will wrap up this report with the impact of Naur on the field of computer science in a wider sense.

2 History of Formal Descriptions for Languages

The goal of defining a language by a formal system of variables and rules to derive the language has been of interest to grammarians and mathematicians since the verge of civilizations. Panini, a Sanskrit philologist and grammarian which has worked between the 6th and 4th century BCE had, according to the biography *Panini* by Bhate (2002), already described a notation structure for the language of Sanskrit. This shows that the problem of formally describing a language has existed even long before the topic of computer languages was even thinkable. The topic however only became of interest in the west at the beginning of the 20th century. With the advance of interest in new, artificial languages like Esperanto or Ido and especially the rapid advances in calculating machines and the first version of computers the need for more formal language descriptions arose. The advance of new technologies posed new challenges in human-machine as well as human-human interactions which first had to be resolved before the underlying technological advances could be made usable for a larger audience than scientists.

2.1 Noam Chomsky's introduction of phrase-structure grammar

Noam Chomsky, an American linguist teaching at MIT combined the new interest from mathematicians and linguists in his teaching groups. Chomsky gave lectures to information technology students and combined past research on the topic of string rewriting rules and formalism to develop his theory of formal logical systems. His main idea was to define the grammar structure of English as a finite-state Markov process. To achieve this idea, Chomsky (1956) introduced in his paper *Three Models for Description of Language* a notation for the rewriting of strings according to a grammar. A grammar can be viewed as a machine operating on strings and giving certain output to an input. Chomsky also defined several different levels of grammars that are used in theoretical computer science until today.

The importance of Chomsky's work can not be understated, even though he was not the original inventor of the idea. He built upon work from other mathematicians as well as computer scientists. Most notably *Axel Thue* as well as *Alan Turing*. From the perspective of this paper, it is of great importance to understand the concept of phrase-structure grammars, also called context-free grammars.

Chomsky defined a language L is a (finite or infinite) set of sentences of finite length. Furthermore, a string is defined as a concatenation of symbols of an Alphabet A. The Alphabet consists of a finite set of terminal symbols. A grammar is then defined as a device that produces all of the strings that are sentences of L. A properly formulated grammar should define unambiguously the set of grammatical sentences of the language it represents. This means that a sentence which is clearly part of the language should be handled by the grammar in a fixed and predetermined manner.

A phrase-structure grammar consists of a finite vocabulary V_p, a finite set of initial strings$E$ in V_p, and a finite set of rules F of the form $X \rightarrow Y$, where X and Y are strings in V_p. The rules are interpreted as an instruction to the grammar device where X has to be rewritten by Y. So any Y_i is formed from a X_i by replacement of a single symbol of X_i by some string. A derivable string from this language is a string that can be derived by the rules F. A terminal string from this language is a string which is the last line of a derivation where there is no further string to be derived.

An example from Chomsky for a phrase-structure grammar would be the following. The symbol _ is introduced for concatenation and will be replaced in the terminal string by a space. The # denominates a meta-linguistic symbol to determine the boundaries of the given string:

$$\begin{aligned} \sum &: \#Sentence\# \\ F &: \\ Sentence &\rightarrow NP\_VP \\ VP &\rightarrow Verb\_NP \\ NP &\rightarrow \text{the man, the book} \\ Verb &\rightarrow \text{took} \end{aligned} \tag{1}$$

A derivable and terminal string from this grammar would be "the man took the book".

Interesting in the scope of this report is the elaboration of Chomsky that a language consisting of basic sentences that are derived from terminal strings of a phrase-structure grammar and a set of optional transformation rules applied on these terminal strings may create a language similar to the modern English. The distinction of generative rules of grammar and transformation rules, which are applied after generating the language may remind some of the way functions and similar constructs in programming languages work today. The insight of Chomsky provided further researchers as well as people interested in the structure of formal notations as an entry point.

2.2 Backus's introduction of a formal programming language description

John Backus, an American Computer Scientist who received the ACM Turing award in 1977 for his contribution to the design of practical high-level programming systems and for publication of formal procedures for the specification of programming languages. The latter part is interesting in the scope of this report as Backus built upon the idea of

Chomsky. Backus was also one of the most influential computer scientists working on the then newly established programming language *International Algebraic Language* (IAL). Backus (1959) authored the report *The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference* which was the defining paper of the programming language now known as Algol 58 which was a direct precursor to Algol 60 to which Peter Naur contributed substantially.

Backus, with the knowledge of the power of phrase-structure grammars, introduced a meta-linguistic formula to formally describe the rules of the IAL programming language. The formulae could describe the phrase-structure grammars and was a more approachable notation to the idea behind these grammars. The meta-linguistic formula introduced by Backus was a reference language to describe the functions of the language on a meta-level to allow for later hardware implementation to only differ with regards to word length, overflow conditions, and the like. The idea was to allow the different stakeholders in the development of a programming language to communicate with a common, formal notation. The main goal of the reference language introduced by Backus was to be as precise as possible without seeming out of touch. Backus first describes the reference language in an informal notation before he proceeds to show an example of it in the new notation.
The syntax of the reference language introduced by Backus was the following:

- Sequences of character symbols enclosed in "< >" are meta-linguistic variables whose values are strings of symbols,
- The marks ":≡" and "*or*" are meta-linguistic connectives,
- Any mark that is neither a variable nor a connective denotes itself,
- The combination of marks and/or variables signifies the combination of the strings denoted

This is best illustrated by an example. The _ is introduced for the combination of two marks and/or variables and will in the derivation ultimately be replaced by a space:

$$\begin{aligned} < sentence > &:\equiv < NP > \_ < VP > \\ < VP > &:\equiv < Verb > \_ < NP > \\ < NP > &:\equiv \text{the man\_}or\text{\_the book} \\ < Verb > &:\equiv \text{took} \end{aligned} \tag{2}$$

As one may observe, it is the same example used in the previous sections example (1) and should show the power of the notation. Furthermore, as previously, one possible derivation would be "the man took the book". One notable difference to the notation used by Chomsky is that Backus introduced the mark *or*, however, every string in the Backus notation consisting an *or* could easily be rewritten into one or more strings without using the connective. It is therefore relevant to note that the notation introduced by Backus is exactly as strong as the phrase-structure grammars. The notation used by Backus was too clumsy and did not catch on to the wider field of computer science and would have almost been forgotten, even though the need for an easily understandable and strong notation was greater than ever.

3 Programming language description, formalization and Algol 60

> Was sich überhaupt sagen lässt, lässt
> sich klar sagen; und wovon man nicht
> reden kann, darüber muss man schweigen.
>
> (Ludwig Wittgenstein)

3.1 The Biography of Peter Naur

Peter Naur, born in 1928 to a family of artists and business-minded parents, had an early interest involving astronomy. He was allowed to work at the local observatory even as a very young boy. Naur published his first scientific paper with 15 and he had attained many technical skills of mathematics early on from professional astronomers which took him on as a young prodigy. After finishing his astronomy degree in Copenhagen he was recommended to research at King's College, Cambridge where he focused on astronomy and the emerging field of computer science. Due to weather constraints, Naur had to divert his time from astronomy and had more time to spend programming the Electronic Delay Storage Automatic Calculator (EDSAC). Peter Naur, used to do complicated computational calculations by hand focused his energy mostly on the limitations of the EDSAC such as the limited number range and other meta-questions. After leaving Cambridge, Naur researched at Harvard University and Princeton, where he learned the state of the art in computing. The now more established field of computer science would be the focus of Naur, who returned to Denmark in 1953.

He joined the computer center of Copenhagen and was asked to participate in the development of an algorithmic programming language, later called ALGOL. Naur's contribution to ALGOL lay in selecting the right forms of description to define the language. This was in line with his later research as he was more interested in the meta aspects of the language rather than a specific implementation. Naur's main concern was that the often overly formal reports led to a split between the industry and academia as the two interacted ever less due to different ideas on formalities and the proofs of algorithms. It was an important question for him to make the field of computer science comprehensible to a wider audience.

Naur also contributed to the establishment of computer science as an academic field in Denmark where he continued to advocate for an applied form of computer science not only for his students but also for the field in general hereby opposing Dijkstra and Wirth structured programming agenda.

3.2 Backus Naur Form

The Backus Naur form was introduced and used by Backus et al. (1960) in the *Report on the algorithmic language ALGOL 60* published in 1960. It was introduced as the reference language of the report. A reference language as defined by the report is the defining language that is the basic reference and guide for compiler builders, for all hardware representations, for transliterating from the language used in the report to

any appropriate hardware representation and should be used by the main publications concerning the ALGOL language itself. Furthermore, the reference language should allow for all characters to be determined by ease of mutual understanding and should not be defined by any hardware limitations, code notation, or pure mathematical notation. The reference language therefore should be understandable to both the wider public as well as special interest groups as compiler or hardware builders.
As the name Backus Naur form (BNF) already suggests was the form an extension of the previously introduced notation form by Backus. Naur introduced the notation as Backus Normal form, however, it has since been renamed. Especially after Knuth (1964) pointed out in his letter to the editor of the ACM communications *Backus Normal Form vs. Backus Naur Form* that the name normal form should be reserved for true normal forms, which refer to some sort of special representation that is not necessarily a canonical form. According to Knuth, who in the end coined the name Backus Naur Form, the Backus Naur form did have a reason for distinction from the Chomsky phrase-structure grammar as the syntax from Chomsky represents a production while the Backus Naur form has a quite different form.
The syntax of the Backus Naur form is described as follows and it is always also noted how it differs from the production form of Chomsky:

- Sequences of character symbols enclosed in "<>" are meta-linguistic variables whose values are sequences of symbols, which is a difference from production form as there is a clearer distinction between terminal and non-terminal symbols.
- The marks "::=" and "|", the latter with the meaning or, are meta-linguistic connectives. The connective "|" groups together all alternatives of a production system and is introduced to make the notation shorter, while the connective "::=" separates left from right.
- Any mark that is neither a variable nor a connective denotes itself.
- The combination of marks and/or variables signifies the combination of the strings denoted.
- Full names indicating the meaning of the strings being defined are used for non-terminal symbols, which was quite different from the production form as well as from the previous form of Backus. This contribution, according to Knuth, is probably the most important from Naur to the notation as it made the notation more applicable and usable.

An example of the notation being used in the Algol report is how an integer is defined in the Algol 60 report:

$$\begin{aligned} &< \text{digit} > ::= 0|1|2|3|4|5|6|7|8|9 \\ &< \text{unsigned integer} > ::= < \text{digit} > \mid < \text{unsigned integer} >< \text{digit} > \\ &< \text{integer} > ::= < \text{unsigned integer} > \mid + < \text{unsigned integer} > \mid \\ &\qquad - < \text{unsigned integer} > \end{aligned} \tag{3}$$

The Backus Naur notation was introduced to give the report a formal reference language. It has since been adapted and changed for multiple uses and an extended version of the Backus Naur form remains used by most programming languages nowadays. The notation proved to be essential for further language and compiler design. The clear vision and simplicity has remained and has allowed further generations of computer scientists to work in the field of language design.
An interesting quality of the Backus Naur form is also that with just some slight modification the form can describe itself. Rohl (1968) has shown that in his text *A note on Backus Naur form*. The goal of this was to allow compilers to accept the Backus Naur form as input for their design. The approach taken by Rohl was for the brackets to surround not the meta-linguistic variables but the basic symbols. This allows the Backus Naur form to describe the meta-linguistic formulae themselves. The brackets are then a device for over-riding any meta-linguistic significance of the symbol they enclose. Rohl also introduced the . to signify concatenation to allow for better readability.
The definition of the meta-linguistic formulae goes as follows:

$$\begin{aligned} \text{meta formula} &::= \text{meta variable. } <::=> \text{ .definiens} \\ \text{definiens} &::= \text{construction. } < \mid > \text{ .definiens} \mid \text{construction} \\ \text{construction} &::= \text{component. } < . > \text{ .construction} \mid \text{component} \\ \text{component} &::= \text{meta variable} \mid <<> \text{ .symbol. } <>> \end{aligned} \tag{4}$$

This definition now allows for a more automatic compiler generation as proposed by Brooker et al. (1963) in the paper *The compiler compiler* where they proposed a compiler that was itself able to generate compilers for a certain hardware architecture, then the Atlas computers, with the language specification of the compilers being a phrase-structure grammar. By defining the meta-linguistic formulae themselves it would be possible to compile a compiler for Algol with the compiler compiler.

3.3 The Algol 60 Report

The Algorithmic Language of 1960, short Algol 60, was an important step in the history of programming languages as it had introduced many new aspects and formalized many older concepts to make them more applicable. Many other important programming languages have been developed with the concepts of Algol in mind with Simula, Pascal and most importantly C being the most widely used ones.

The effort of Algol 60 started after several scientists, including Peter Naur, studied and worked on the *Algol 58* report by Bauer et al. (1958) where they defined an early version of Algol. Peter Naur, working together with Jørn Jensen, found the report to be lacking and they organized a conference in Copenhagen during the year 1959. From this conference, the most notable outcome was that Naur started the *Algol Bulletin* to concentrate the international effort on Algol and to distribute the different ideas. This led to Naur being the leading European mind working on the Algol effort.

According to Naur (1968) in his talk *Successes and failures of the ALGOL effort* which took place in 1968, ten years after the first version of Algol 58 was published, a strong understanding of languages as well as compiler problems led to a need for a sound, common programming language. Together with developments in the United

States on Fortran as well as the developments of the Amsterdam school which supplied powerful and general ideas about the central features of programming languages, there was the solid groundwork for an improved programming language. Furthermore, the development of Backus on the reference language provided a suitable notation for a formal description of the resulting language.

The resulting *Report on the algorithmic language ALGOL 60* by Backus et al. (1960) defined the language Algol 60 in a mix of formal, Backus Naur form, and informal syntax. The influence of the report was wide and we will try to discuss some important developments of the language and its syntax description. The most important aspect of a programming language is the programming of computers, in which Algol was a partial success. As a core language with its formal description and the newly introduced nested function definition as well as a strong focus on the lexical scopes of methods, the language could provide the developers of the time with new and strong concepts for their work. However, the missing input and output facilities proved to be detrimental to a wider field of programming applications.

One aspect of the language which directly coincided with its rather formal notation was the publication of algorithms. The very non-ambivalent description of the language made it a favorite for the development of algorithms and allowed it to be used theoretically in many applications. A further aspect of importance was on the development of formal languages and has acted as a strong stimulus for compiler designers. However, according to Naur, this development could be seen from two sides and the over formalization would not help to draw in further interest from the public.

The form of the description was of great interest to Naur and would prove to be the topic of his interest in his later work. He was interested mostly in that a language could not only be understood by a computer but also by a human reader. This was also why he as the sole editor tried to use about the same amount of formal as well as in informal notation to describe the language. The mixture of both was important to Naur and he later even proposed that the report should have included a more informal introduction. Naur himself criticized his work on Algol 60 strongly. Even though he was content with the formal description of the language, he was not content with how the report was received by the larger public. The Algol 60 report, as said by Naur, was very hard on the uninitiated reader.

An example of the formal notation one may have a look at the definition of the basic symbols of Algol 60 as introduced by the *Revised report on the algorithmic language ALGOL 60* by Backus et al. (1963). The revised report does not differ in notation and consists of different error corrections at different parts of the report:

The reference language is built up from the following basic symbols:

$$< \text{basic symbol} > ::= < \text{letter} > | < \text{digit} > | < \text{logical value} > | < \text{delimiter} > \tag{5}$$

Some problems in the documentation by Naur (1963) later pointed out in his article *Documentation Problems: Algol 60* were mostly in the reliance of language description on formal notations. However, even though the Backus Naur form helped in the documentation, it provided no safety against errors. Writing a report on a programming language, according to Naur, differed not substantially from any other form of

communication and the integral part was that the description must be complete and unambiguous. It is therefore also important to stress that even though there has been substantial progress in constructing mechanical and formal metalanguages, the natural languages will always remain the ultimate metalanguage.

The effort in the Algol 60 report was thus not futile, however, the over formalization proved to be a thorn in the eyes of Naur and would be of particular interest to him in his further work. Generally, it can be said that the report and the corresponding language of Algol 60 proved to be strongly influential, however, far from perfect.

3.4 Notes on Formalization

Peter Naur's opinions on formalization in program development were twofold. Naur was on one side one of the influential co-creators of the most famous formal notation language, on the other he was a strong opponent of overly formalizing programming language. In his work, Naur always tried to use a twofold of notations. An informal one in the standard language used for human communication and a formal language. However, his interest in the formal notation was mainly to provide informal notations with additional brevity and clarity. Naur (1994) examined the usage of formal and informal modes of expression in his paper *Proof versus formalization* where he compared several proofs and their notation. Most notably, he compared the proof of an algorithm by Warshall from 1962.

Naur showed in his comparison that the common doctrines related to proof and formalization were most probably false. Especially the claim that the usage of formal modes for proving algorithms and mathematical theorems prevent human errors have proven to be a very false claim. Proofs, according to Naur, have the purpose of conveying knowledge from the writer to the reader. This means that proofs should be, as every other form of documentation, with the reader in mind. If a proof is overly formal, this may result in the reader not being able to understand the proof as the normal mode of communication for most humans is informal.

Naur even more strongly opposed to an over formalization in the mode of communication when it came to programming language description. Naur (1982) proved in his paper *Formalization in program development* that an over-formalization is not only making the language less appealing for a wider audience but also that a forced formalization may be detrimental to the preciseness and clarity of description. Again, a formal mode should mostly be introduced for brevity and convenience. Naur especially made clear that the formal mode should not be regarded as an alternative to the informal mode but rather as a freely introduced part of the basic informal mode. Furthermore, every formal argument can be transcribed into an informal one but not the opposite. The formal mode is thus a subset of the informal mode rather than another, independent, set altogether.

The overpromotion of formal notation, according to Naur, will likely be rather harmful to effective program development especially because the focus on the formal notation will lead authors to believe that there are no principles nor practices of good informal expression. Naur proposed that there should be certain principles involved in the usage of formality. The principle consists is that one should choose each specialized word to identify unmistakably a concept that is useful and relevant in the context,

that one should introduce the concepts and designated word where they are used and that the same concept introduced must be referred to invariably by the same designation. Furthermore, every description of real matters must fundamentally be based in an informal mode.

The notation introduced by Naur himself in the Algol 60 report does, according to Naur, qualify to justify these rules and is thus a good example of a formal description rooted in an informal description. Especially the description of the formalism used in the report is a good example. It consists of 16 lines of informal prose, one example of a meta-syntactic formula, and four examples of symbol sequences generated by it. It is thus also easy to read and to understand, which was another important point to Naur. Notation should be designed to make it easier for programmers to use the language and also to write better programs, it should not make the development process harder which would result in the language being used less.

3.5 Checking operand types in the Gier Algol Compiler

A notation is important for human-human communication, however, it is also important in the process of human-machine interaction. One case of this very direct human-machine interaction can be observed in the development of compilers where a language has to be made understandable to computers. Compilers translate the human-readable programming language into machine code which the computer can execute on. Peter Naur was closely involved in the development of the Gier Algol compiler and he has introduced the operand type checking system into this compiler. The motivation behind an operand checking system is to allow the programmers to detect errors more efficiently and thus make the development process less time-consuming.

The philosophy of error treatment in the Gier Algol compiler, as stated by Naur (1965) in the article *Checking of operand types in ALGOL compilers*, is to not permit any program in which the compiler has detected an error to run. The compiler will try to suppress all secondary consequences of a detected error to continue to process the rest of the program. This results in the need to fully integrate the error detection with the translation process. There will be two levels of error checking with the first being a check if the delimiter structure of the source program is allowed according to the language definition and a second one to check for the compatibility of the declarations in the way they are used to form expressions and statements.

It is important to note that the first level of error detection can work with the description given by the Backus Naur form as it can check if the source code is a terminal and reachable string of the described grammar. The second error checking needs some additional logic.

The problem of the second error checking is the following: Given source code and given a complete description of the types and kinds of identifiers (provided by the language description), check that the usage of the identifiers and source code usages conform to their descriptions.

The solution proposed by Naur was to use the Reverse Polish form, which is a form of evaluating arithmetic expressions by using a stack to hold the values of those operands that have not been operated upon yet. The stack can be described as follows:

$$\textbf{array } \textit{OPERAND STACK}[1 : \textit{some upper limit}] \tag{6}$$

Furthermore, let us introduce an integer, *LAST USED*, to denote the current top value of the stack. The evaluation rules of Reverse Polish form is to evaluate the expression from left to right in a strictly sequential form. When an operand is being encountered, place it on top of the stack by increasing *LAST USED* by one and saving the operand in the *OPERAND STACK* at position *LAST USED*. When encountering an operator, operate and place the result on the stack again. Binary operators remove two items from the stack.

The form of Reverse Polish form employed in the operand checking in the Gier compiler makes usage of both an *OPERAND STACK* as well as an *OPERATOR STACK*. The logic of operand checking now is simply to apply the rules of Reverse Polish notation while each time an operator is placed on top of the stack, one operand may already be checked on compatibility as it is not necessary to have both operands for a binary operator.

An example of the evaluation of an expression and its corresponding declarations is the following:

$$\begin{aligned} &\textbf{Boolean } a;\ \textbf{integer } b,\ d;\ \textbf{real } c; \\ &a \equiv b - c = d \end{aligned} \tag{7}$$

The sequence of actions taken in checking the operand types may be best done in a snapshot as described by Naur (1966) in the article *Proof of algorithms by general snapshots* where Naur provided a diagram to display the behavior of algorithms for proofs by taking a snapshot of the current state of the involved variables at certain predetermined points to compare the algorithms behavior with the theoretically expected behavior. We will present the behavior of the Reverse Polish form algorithm for operand checking applied in the Gier compiler by showing a snapshot of the algorithm after one input character got evaluated.

Action	Input	OPERATOR [1]	STACK [2]	OPERAND STACK [1]	Output
1	a			bo	a
2	≡	≡			
3	b	≡		in	b
4	-	≡	-		
5	c	≡	-	re	c
6	=	≡		re	-
		≡	=		
7	d	≡	=	in	d
8	,	≡		bo	=
				bo	≡

Table 1: Sequence of Operand type checking

Each time an operand gets evaluated, its value gets put into the *OPERAND STACK*, each time an operator gets evaluated, the top operand type gets checked and removed from the *OPERAND STACK* while the operator goes on the *OPERATOR STACK*, furthermore, every time a new operator gets evaluated, it gets checked if an operator may be unstacked. This happens for example when = is the current input and - gets evaluated with the current top operand. This unstacking is done for the operators on top of the *OPERATOR STACK* where every operator on the stack gets compared with the current input operator until there is an operator on the stack with less priority than the input operator. Every operator that is unstacked will be placed in the output and the top operand gets checked against it.

The logic applied in the Reversed Polish form in the Gier Algol compiler also extends to all other Algo 60 delimiters like for example **then**. Every time an error gets detected, the internal operand class, *UNDECLARED*, replaces the current operand and an error message is given.

The operand checking in the Gier Algol compiler is a good example of how a good operand definition of a language coupled with robust algorithms can be coupled to implement a strong error detection which then helps the human programmers to develop software faster. It is also a good example of how a formal language definition is beneficial to the compiler designers as it makes their job clear cut and the edge cases in this example can clearly be handled within the constraints of the language.

4 Peter Naur's Further Work

Peter Naur has been both an advocate of formalization as well as a strong opponent of over-formalization. His influence can be seen in the Algol 60 report as well as in the subsequent development of programming languages. Most programming languages up to today use some extended form of the Backus Naur form in their language specification. Even though Naur himself did not want to be so closely associated with the notation, he did have a strong influence on the design and content of the Algol report. His philosophy of conveying theories rather than brute force information can also be seen in the article *Programming as theory building* from Naur (1985) where he pointed out that the designers' job is not to pass along the design of a program but rather the theories driving the design. The primary job of a programmer is thus to build up knowledge. This philosophy can already be seen in Naur's early work where the documentation seemed to always place more importance on the wider concept than technical details. The description of Algol 60 for example made it such a strong programming language exactly because it was rather a theory conveying than focusing on the technical details of its implementation.

Naur furthermore influenced the field of language description and therefore the future of compiler design by providing not only a good notation to be used by the designers but also by his human-first approach to the field of computer science. The idea of descriptive documents to him was always to allow humans to understand it better and he observed early on that for a language and its system to flourish it must be accessible to many more people than just a small part of academia. Naur (2007) further displayed this interest in the human understanding in his acceptance speech of the ACM's 2005

A.M. Turing Award *Computing Versus Human Thinking* where he displayed a deep interest in the human understanding and the complex processes of thinking. Not only could Naur be considered a computer scientist but also a philosopher, psychologist, and linguist. His fundamental contribution to the field of computer science is vast and the influence of his work can be seen until today for example in form of a larger audience being able to participate and the humanist approach used in most educational courses and literature nowadays and beyond.

References

Backus, J. (1959). The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. In *IFIP Congress*.

Backus, J., Bauer, F., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J., Wijngaarden, A., and Woodger, M. (1963). Revised report on the algorithmic language algol 60. *Comput. J.*, 5:349–367.

Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., et al. (1960). Report on the algorithmic language algol 60. *Numerische Mathematik*, 2(1):106–136.

Bauer, F. L., Bottenbruch, H., Rutishauser, H., Samelson, K., Backus, J., Katz, C., Perlis, A., and Wegstein, J. H. (1958). Algol 58.

Bhate, S. (2002). *Panini*. Sahitya Akademi.

Brooker, R., MacCallum, I., Morris, D., and Rohl, J. (1963). The compiler compiler. In *International Tracts in Computer Science and Technology and Their Application*, volume 3, pages 229–275. Elsevier.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124.

Knuth, D. E. (1964). *Backus Normal Form vs. Backus Naur Form*, chapter 7.12. ACM.

Naur, P. (1963). Documentation problems: Algol 60. *Communications of the ACM*, 6(3):77–79.

Naur, P. (1965). Checking of operand types in algol compilers. *BIT Numerical Mathematics*, 5(3):151–163.

Naur, P. (1966). Proof of algorithms by general snapshots. *BIT Numerical Mathematics*, 6(4):310–316.

Naur, P. (1968). Successes and failures of the algol effort. *algol Bulletin*, (28):58–62.

Naur, P. (1982). Formalization in program development. *BIT Numerical Mathematics*, 22(4):437–453.

Naur, P. (1985). Programming as theory building. *Microprocessing and microprogramming*, 15(5):253–261.

Naur, P. (1994). Proof versus formalization. *BIT Numerical Mathematics*, 34(1):148–164.

Naur, P. (2007). Computing versus human thinking. *Communications of the ACM*, 50(1):85–94.

Rohl, J. S. (1968). A note on backus naur form. *The Computer Journal*, 10(4):336–337.