ALGOL 60

Jodok Huber
Jodok.Huber@student.uibk.ac.at

18 February 2013

Abstract

This article is about ALGOL 60, a programming language build mainly to implement algorithms for numeric problems. The language was developed around 1960 and was one of the first programming languages. Nevertheless it was very advanced for its time and had a great impact on many of the ensuing programming languages. This article will give you a brief background information of the history and the new features of ALGOL 60 and introduce the basics of programming in this language.

Contents

1 Einleitung

Dieser Artikel gibt einen Überblick über die Programmiersprache ALGOL 60. Es wird die Geschichte der Sprache, von der Entstehung, über die aktuelle Verwendung bis hin zum Einfluss auf darauf folgende Programmiersprachen beschrieben. Die wichtigsten Neuerungen, wie zum Beispiel die Blockstruktur mit ihren verschachtelten Gültigkeitsbereichen, rekursiv aufrufbare Prozeduren und einige andere werden erklärt. Des weiteren wird in diesem Artikel eine kurze Einführung in den Aufbau und die Programmierung der Sprache gegeben. Um noch einmal einen guten Überblick über die Funktionsweise der Sprache zu geben, befindet sich am Ende des Artikels ein Codebeispiel, das das Sieb des Eratosthenes implementiert.

2 Die Geschichte von ALGOL 60

ALGOL 60 steht für ALGOrithmic Language 1960 und ist eine international entwickelte, von kommerziellen Interessen unabhängige, prozedurale Programmiersprache, die hauptsächlich für wissenschaftliche Anwendungsgebiete gedacht war. Sie wurde in den Jahren von 1958 bis 1963 von einem Komitee aus europäischen und amerikanischen Informatikern geschaffen und war Vorreiter vieler anderer Programmiersprachen wie zum Beispiel: BCPL, B, Pascal, Simula und C.

Die Zahl 60 im Namen steht für das Jahr der "Fastfertigstellung". Die endgültige Fassung, die im Revised Report veröffentlicht wurde, stammt aus dem Jahre 1963.

ALGOL 60 gehört zur Familie der ALGOL Sprachen, die mit ihrem Vorgänger ALGOL 58 ins Leben gerufen wurde. ALGOL 58 wurde zu einer Zeit entwickelt, in der FORTRAN, eine Programmiersprache von IBM, die gängigste Sprache war. Aufgrund der mangelnden Freiheit, die bei der Entwicklung von FORTRAN gegeben war, entschied sich eine Gemeinschaft von Informatikern eine neue Programmiersprache zu entwickeln. Der Diskussionsentwurf wurde dann unter dem Namen IAL (*International Algorithmic Language*) bekannt. Später wurde die Sprache in den einfacheren Namen ALGOL umbenannt.

Das Hauptziel der neuen Sprache war es eine Syntax zu entwickeln, die so nahe wie möglich an der mathematischen Standardnotation lag, sodass diese auch für die Beschreibung von Algorithmen in schriftlichen Publikationen verwendet werden konnte. 2 Jahre später wurde dann der Nachfolger ALGOL 60 entwickelt mit dem Ziel aus AGLOL eine vollwertige Programmiersprache für wissenschaftliche Berechnungen zu machen.

Niklaus Wirth schrieb die, auf ALGOL 60 basierende Programmiersprache ALGOL W. Die Sprache war im Großen und Ganzen eine sauberere und vereinfachte Version von ALGOL 60 und war als deren Nachfolger geplant. Das ALGOL Komitee entschied sich dann aber für die komplexere und erweiterte Version ALGOL 68. Aus ALGOL W wurde später die Programmiersprache Pascal. ALGOL 68 ist ein kompletter Neuentwurf und unterscheidet sich grundle-

gend von ALGOL 60. Der Nachfolger wurde dafür auch kritisiert, sodass im Allgemeinen mit „Algol“ die Dialekte von ALGOL 60 gemeint sind.

3 Tatsächliche Verwendung

ALGOL 60 war jahrelang die führende Programmiersprache für Algorithmen und wurde auch zum Standard für die Publikation von diesen. Auch an Universitäten war es eine populäre Sprache für den Programmierunterricht, da sie für damalige Verhältnisse einfach zu nutzen war.

Trotz des starken Einflusses im akademischen Bereich, fand die Sprache im wirtschaftlichen Bereich nur wenig Verwendung.

Die kommerzielle Nutzung blieb aufgrund fehlender Standardisierung der Ein- und Ausgabe und mangelnder Unterstützun von größeren Computerfirmen aus. Eines der wenigen konkreten Anwendungsgebiete war zum Beispiel das Betriebssystem der Burroughs-B5000-Rechner, das in einer ALGOL-Version (ESPOL) programmiert wurde.

Eine weitere Beobachtung, die man bezüglich der Verwendung der Programmiersprache machen konnte war, dass ALGOL 60 in Europa viel stärker verbreitet war als in den USA. Dies könnte damit zusammenhängen, dass in Amerika zu dieser Zeit die Computer-Industrie schon weiter fortgeschritten war als in Europa. Dies bedeutet jedoch nicht, dass ALGOL 60 weniger fortschrittlich war als andere Programmiersprachen zu dieser Zeit, sondern vielmehr, dass der Wechsel von einer Sprache, die bereits in Verwendung war zu einer neuen einen zu großen Aufwand bedeutet hätte. Ohne zuverlässige Implementierungen und ohne der Unterstützung von einem Computerhersteller war ein Umstieg so gut wie unmöglich. Offizielle Unterstützung für algorithmische Sprachen gab es in den USA auch nicht. In Deutschland hingegen wurde die Verwendung von ALGOL gefördert, indem für jeden von der Universität angeforderten Rechner eine ALGOL-Implementierung vorausgesetzt wurde.

Ein weiteres Problem waren die vielen unterschiedlichen Übersetzungen von verschiedenen Firmen. Die Programmiersprache FORTRAN im Vergleich dazu hatte zwar auch mehrere Übersetzungen, jedoch waren alle von der Firma IBM, die diese Programmiersprache damals unterstützte und zum größten Computerhersteller in den 60er Jahren wurde.

Obwohl ALGOL 60 der Eintritt in die Industrie erschwert bis verhindert wurde, hatte sie, nicht zuletzt wegen dem starken Verbreitung im akademischen Bereich, einen großen Einfluss auf die Entwicklung und die Geschichte der Programmiersprachen.

4 Merkmale und Einfluss der Sprache

ALGOL 60 war eine sehr fortschrittliche Programmiersprache und hatte einen großen Einfluss auf darauf folgende:

ALGOL hatte einen klaren und einfachen Sprachkern, der die meisten darauf folgenden Programmiersprachen inspirierte. Außerdem war der Code nicht

mehr an eine bestimmte Form gebunden und konnte frei formatiert werden, was die Lesbarkeit von Programmen enorm vereinfachte.

Mit ALGOL 60 begann die saubere Definition von Spracheigenschaften unabhängig von der Implementierung. Es wurde mit Hilfe der Backus-Naur-Form die formale Definition der Syntax eingeführt. Die Backus-Naur Form, eine formale Metasprache zur Darstellung kontextfreier Grammatiken, wurde von John Backus speziell für ALGOL 58 zur Beschreibung der Programmiersprache entwickelt. Peter Naur erweiterte diese dann für ALGOL 60. Die Backus-Naur Form wurde daraufhin bald allgemein üblich für die Definition von Programmiersprachen.

Ein Ausschnitt der Definition der Syntax von ALGOL 60 die mit Hilfe der Backus-Naur-Form im Revised Report definiert ist sieht zum Beispiel so aus:

<Operator>::= <arithmetischer Operator>| <Vergleichsoperator>| <logischer Operator>| <Folgeoperator>

<arithmetischer Operator>::= + | - | × | / | ÷ | ↑

<Vergleichsoperator>::= <| ≤ | = | ≥ | >| ≠

<logischer Operator>::= ≡ | ⊃ | ∨ | ∧ | ¬

<Folgeoperator>::= **goto** | **if** | **then** | **else** | **for** | **do**

Wie man bei dieser Definition eines Operators sieht, wurden sehr viele spezielle Symbole verwendet. Diese wurden dann meist, je nach Implementierung, durch Wörter ersetzt.

Eine weitere Besonderheit in ALGOL waren die vorgeschriebenen Laufzeitprüfungen. So wird zum Beispiel ein Fehler während des Programmablaufs ausgegeben, falls eine Division durch Null auftritt. Dies war für die damalige Zeit noch sehr unüblich und wurde erst mit der Zeit gewürdigt und übernommen.

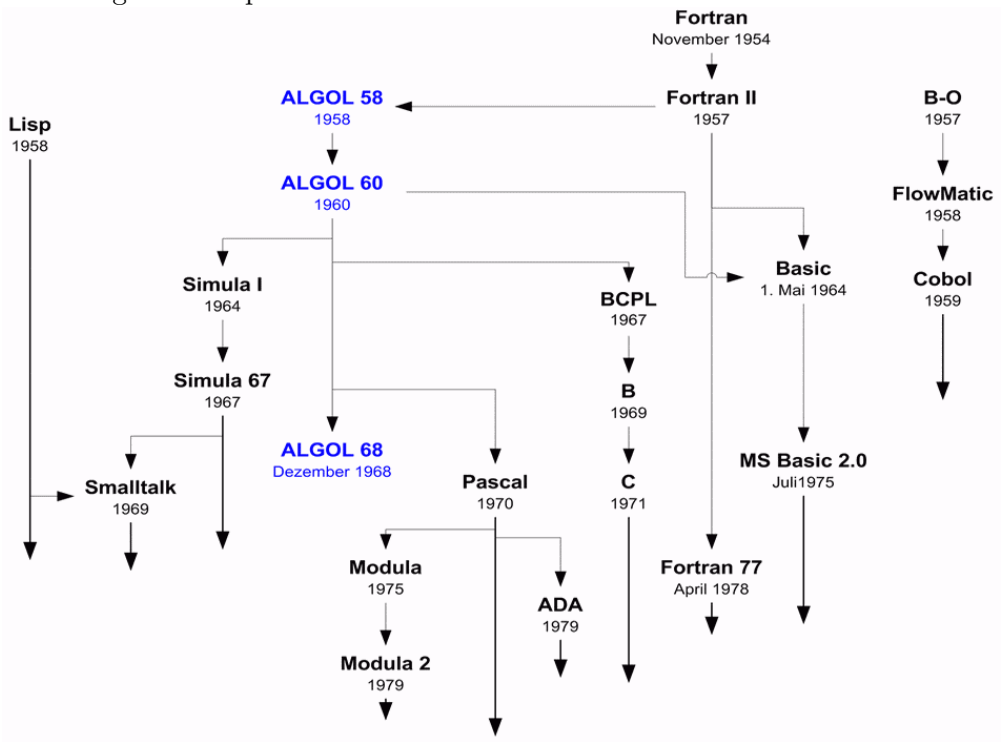
Die Sprache selbst hat im Vergleich zu seinen Vorgängern viele Neuerungen, die bei den meisten aktuellen Programmiersprachen Anwendung finden. Die wesentlichen neuen Sprachmerkmale waren:

- Schlüsselwörter sind reserviert.
- Man kann Funktionen und Prozeduren definieren und man hatte die Möglichkeit eine Funktion rekursiv aufzurufen.
- Die Sprache ist block-strukturiert und Variablen und Bezeichner haben verschachtelte Gültigkeitsbereiche(nested functions).
- Die schon aus FORTRAN bekannte **if**-Abfrage wird um das **if-then-else** Konzept erweitert.
- Mit der Einführung von **while**-Schleifen sind flexible Schleifen möglich.

- Die **switch**-Anweisung wird eingeführt.
- Die dynamische Deklaration von Arrays ist möglich.

Ein großes Problem der Programmiersprache war hingegen die Ein- und Ausgabe. Diese war nicht geregelt und deshalb von Compiler zu Compiler verschieden. Eine Folge davon ist, dass es zum Beispiel kein globales "Hello World" Programm gibt, das auf allen Implementierungen läuft.

Die folgende Grafik gibt einen Überblick über den Einfluss von ALGOL auf andere Programmiersprachen.



Quelle: `http://www.manthey.cc/sites/seminars/src/Sprachen.png`

5 Aufbau der Sprache

Jedes ALGOL-Programm wird durch das Wortsymbol **begin** eingeleitet und endet mit einem **end**. Nach dem **begin** kann ein Vereinbarungsteil folgen, der aus durch Strichpunkte getrennte Deklarationen besteht. Darauf folgt dann der Anweisungsteil der wiederum durch Strichpunkte getrennte Statements enthält.

Wenn der Bereich von **begin** bis **end** keinen Vereinbarungsteil enthält nennt man ihn Verbundsanweisung, anderenfalls einen Block.

Überall wo eine Anweisung stehen kann, darf auch ein weiterer Block eingefügt werden. Dieser muss dann immer mit einem Strichpunkt abgeschlossen werden.

Deklarationen werden nicht wie in den meisten heute gängigen Programmiersprachen mit einem "=", sondern mit einem ":=" zugewiesen und Gleichheitsabfragen funktionieren nicht mit einem doppelten, sondern mit einem einfachen Gleichheitszeichen.

Nach dem Auftreten einer Anweisung darf vor dem **end** keine Deklaration mehr auftreten. Die letzte Deklaration vor dem **end** benötigt keinen Strichpunkt zum Abschluss der Anweisung. Freiräume oder Zeilenumbrüche haben keinen Einfluss auf die Funktionalität des Programms und dienen rein zur Lesbarkeit für den Programmierer.

Beispiel 5.1. :

```
1 begin;
2   integer a;
3   integer b;
4   integer c;
5   a := 1;
6   b := a + 1;
7   c := a + b;
8 end
```

Der Vereinbarungsteil kann auch, wie aus anderen Programmiersprachen bekannt folgendermaßen geschrieben werden:

```
  integer a, b, c;
```

äquivalenter C-Code:

```
1 int main(void) {
2   int a, b, c;
3   a = 1;
4   b = a + 1;
5   c = a + b;
6   return 0;
7 }
```

5.1 Datentypen und Arrays

In ALGOL 60 gibt es vier Datentypen: Den Datentyp **integer**, zur Darstellung ganzer Zahlen, **real** für reelle Zahlen, **Boolean** für die logischen Konstanten `true` und `false` und **String** zur Darstellung einer Folge von Zeichen.

Bei der Definition von Arrays muss immer zuerst der Datentyp geschrieben werden gefolgt von dem Schlüsselwort **array**. Danach folgt der Name des Arrays und die untere und obere Grenze der Indexwerte innerhalb von eckigen Klammern und getrennt durch einen Doppelpunkt. Eine Besonderheit hierbei ist, dass bei der Angabe der Grenzwerte auch Variablen oder arithmetische Ausdrücke stehen können, womit die Möglichkeit zur dynamischen Deklaration von Arrays gegeben ist.

Eine Arraydeklaration sieht beispielsweise folgendermaßen aus:

```
  integer array a[untergrenze:obergrenze];
```

5.2 Standardfunktionen

In ALGOL 60 gibt es die Standardfunktionen **abs**(x) für den absoluten Betrag von x, **sqrt**(x), **sin**(x), **cos**(x), **arctan**(x), **ln**(x), **exp**(x), **sign**(x) (1 falls $x \geq 0$, 0 falls $x = 0$, -1 falls $x \leq 0$) und **entier**(x) für die größte ganze Zahl die kleiner oder gleich x ist.

Alle Standardfunktionen mit Ausnahme von **sign** und **entier**, die ein **integer** zurückgeben, liefern einen Wert vom Typ **real**.

Die Funktion **entier** wird auch zur Konvertierung einer Zahl von **real** zu **integer** verwendet, da der Funktionswert der Funktion für beliebige Argumente vom Typ **integer** oder **real** stets vom Typ **integer** ist. Das übliche Runden einer Zahl x kann nun folgendermaßen realisiert werden: **entier**($x + 0.5$)

5.3 Sprunganweisungen und Marken

In ALGOL 60 können im Programm Marken gesetzt werden. Diese stehen, durch einen Doppelpunkt getrennt, vor der Anweisung:

```
M: x:= x + 1;
```

Mit der Sprunganweisung goto kann dann im Programm an die Stelle einer Marke gesprungen werden, von wo aus das Programm dann weiter ausgeführt wird. Dies ist unabhängig davon ob die Marke vor oder nach der goto-Anweisung steht. Mit `goto M;` würde das Programm also zur Marke M springen.

5.4 Bedingte Anweisungen

Die bedingten Anweisungen sind in ALGOL 60 wie folgt aufgebaut:

```
if <Vergleich> then <Anweisung>;
```

oder:

```
if <Vergleich> then <Anweisung> else <Anweisung>;
```

Es wird der Vergleich ausgewertet und, wenn dieser `true` ist wird die Anweisung nach dem **then** ausgeführt. Bei `false` wird die Anweisung nach dem **then** übersprungen, und falls vorhanden, die nach dem **else** ausgeführt.

5.5 Schleifen

In ALGOL wurden erstmals Schleifen eingeführt, was im Vergleich zu den anderen Programmiersprachen dieser Zeit einen immensen Vorteil in der Lesbarkeit und der Schwierigkeit der Programmierung verschaffte. Die Schleifen waren folgendermaßen aufgebaut:

```
for <Laufvariable> := <Laufliste> do <Anweisung>;
```

Die Laufvariable bekommt nun nach der Reihe die einzelnen Werte der Laufliste zugewiesen und für jeden dieser Werte wird die Anweisung nach **do** ausgeführt.

Beispiel 5.2. :

```
for x:= 1, 2, 3 do y := y + x;
```

äquivalenter C-Code:

```
for (int x = 1; x <= 3; x++;){
  y = y + x;
}
```

Dieser C-Code bewirkt zwar das Gleiche, ist aber prinzipiell anders aufgebaut. Eine Schleife, wie die im ALGOL 60 Code, die einen Codeteil für eine Variable mit mehreren Vordefinierten Werten ausührt gibt es in C nicht.

Die Lauflistenelemente können folgendermaßen angegeben werden:

- als arithmetischer Ausdruck
- `a` **`step`** `b` **`until`** `c`
  wobei a, b und c arithmetische Ausdrücke sind.
  In dieser Form fängt die Laufvariable mit dem Wert von a an und schreitet mit der Schrittweite b solange fort wie der Wert von c noch nicht überschritten ist.
- `d` **`while`** `e`
  wobei d ein arithmetischer und e ein logischer Ausdruck ist.
  Hier bekommt die Laufvariable immer den Wert d zugewiesen und die Anweisung wird so lange ausgeführt wie e true ist.

Wenn man nach **do** mehrere Anweisungen ausführen will kann man diese mit einer Verbundsanweisung zusammenfügen. Wird die Laufanweisung durch ein **goto** im Anwendungsteil unterbrochen merkt sich die Laufvariable den Wert bei dem sie stehen geblieben ist und setzt gegebenenfalls an dieser Stelle fort.

5.6 Blöcke

Blöcke können überall stehen, wo auch eine Anweisungen stehen darf und haben folgende Form:

```
begin
  <Vereinbarung(en)>
  <Anweisung(en)>
end
```

5.6.1 Die Gültigkeit von Variablen

In einem Block deklarierte Variablen sind nur in diesem Block und in den untergeordneten Blöcken sichtbar. Eine Variable aus dem selben Block nennt man lokal, eine übergeordnete global.

Falls in einem Block eine Variable mit einem Namen erstellt wird, der schon global vorhanden ist, wird sie für diesen Block und für alle Untergeordneten mit dem Wert der lokalen Variable überschrieben.

5.6.2 Die Gültigkeit von Marken

Man kann mit einer Sprunganweisung nur zu einer Marke im selben, oder in einem übergeordneten Block springen. Es ist aber nicht möglich mit einem **goto** in einen anderen untergeordneten Block hinein zu springen.

Bei mehreren gleichnamigen Marken wird derjenige genommen, der im eigenen Block liegt. Falls dieser nicht vorhanden ist wird so lange im darunterliegenden Block nachgeschaut, bis die Marke gefunden wird. Innerhalb eines Blocks selbst dürfen allerdings keine Namenskonflikte auftreten.

Wenn ein Block keine globalen Größen mehr hat, also der äußerste Block ist, wird er als Programm bezeichnet.

5.7 Prozeduren

Der Aufbau einer Prozeduranweisung beginnt mit dem Schlüsselwort **procedure** gefolgt von dem Namen der Prozedur und in Klammern eine Liste der Parameter, die übergeben werden sollen.

Danach muss die vollständige Liste der lokalen Variablen inklusive ihrer Datentypen angegeben werden und erst dann folgt das Statement, beziehungsweise der Block mit dem auszuführenden Code.

Zusätzlich kann man nach den Übergabewerten eine Liste von Parametern angegeben, die dann mittels dem `call by value` Prinzip aufgerufen werden. Diese Parameter müssen lediglich mit dem Schlüsselwort **value** eingeleitet werden. Die Übergabeparameter die nicht mit value gekennzeichnet werden, werden automatisch mittels `call by name` aufgerufen. Diese beiden call Methoden werden in der nächsten Untersektion erklärt.

Beispiel 5.3. :

```
procedure sum(a, b, c);
integer a, b, c;
c := a + b;
```

Prozeduren können auch Rückgabewerte haben. Dies wird erreicht indem zusätzlich zu Beginn der Prozedur der Rückgabetyp angegeben wird und im Rumpf dem Namen der Prozedur der gewünschte Wert zugewiesen wird.

Beispiel 5.4. Prozedur mit Rückgabewert:

```
integer procedure sum(a, b);
integer a, b;
sum := a + b;
```

In ALGOL 60 können Prozeduren auch als Übergabeparameter für Prozeduren dienen. Diese möglichkeit gab es in FORTRAN beispielsweise noch nicht.

Beispiel 5.5. ein Prozeduraufruf konnte also auch folgendermaßen aussehen:

```
x := sum(a, sum(b,c));
```

5.7.1 Call by value und call by name

Wenn Variablen mittels `call by value` aufgerufen werden, wird der Übergabeparameter sofort ausgewertet, und der lokalen Variable in der Prozedur zugewiesen. Wenn diese lokale Variable nun verändert wird hat dies keinen Einfluss auf die Variablen außerhalb.

Bei `call by name` wird der tatsächliche Übergabeparameter an jede nötige Stelle im Code eingesetzt bevor dieser ausgeführt wird und wird dann evaluiert wenn man die Stelle erreicht.

Die beiden Konzepte gehören zu den schwierigen Bereichen von ALGOL 60 und spielen in heutigen Programmiersprachen so gut wie keine Rolle mehr.

Beispiel 5.6. Hier ein Beispiel um den Unterschied deutlich zu machen:

```
integer procedure f(x, y)
value x, y;
integer x, y;
begin
  x:=10;
  f:=y;
end
```

Diese Prozedur bekommt zwei Werte, ändert den ersten und gibt den zweiten wieder zurück.

Wenn wir diese Prozedur mit f(i, i/2) aufrufen und i den Wert 2 besitzt, bekommen wir 1, also wieder i/2 zurück. Hätten wir aber bei der Deklaration der Prozedur die Zeile **value** x, y; weg gelassen würde die Prozedur mittels `call by name` aufgerufen werden und wir bekommen mit f(i, i/2) einen anderen Wert zurück. Bei `call by name` wird nämlich jedes x im Code durch i, und jedes y durch i/2 ersetzt. Somit wird mit x:=10 der Wert von i auf 10 gesetzt und bei der Zuweisung des Rückgabewerts i/2, also schlussendlich 5 zurück gegeben.

5.8 Die Ein- und Ausgabe

Aufgrund der Tatsache dass die Ein- und Ausgabe im "Revised Report" nicht festgelegt ist, erfolgt diese auf verschiedenen Rechenanlagen und Implementierungen oft recht verschieden. Die bis 1964 erstellten ALGOL Compiler hatten meist vier Ein- und Ausgabeprozeduren. Diese wurden auf späteren Compilern mitunter neben weiteren Prozeduren auch noch zur Verfügung gestellt.

- read

  Mittels `read`(x) können Werte eingelesen werden. Es werden so viele Zahlen oder logische Werte eingelesen und zugewiesen, wie Variablen hinter dem `read` aufgeführt sind.

- print

  Die Ausgabe von Werten erfolgt mit `print`(x). Auch hier können Werte angegeben werden, jedoch auch arithmetische oder logische Ausdrücke.

```
print(x, sin(x), sin(x)/cos(x));
```

- type

  Mit `type`(x) wird der Wert des Ausdrucks x ohne festem Format ausgegeben.

  Dies ist vor allem in Kombination mit der Prozedur `write` sinnvoll.

- write

  Die Ausgabe einer Zeichenfolge erfolgt mittels `write`("Zeichenfolge")

Beispiel 5.7. :

```
1 begin
2   integer x;
3   read(x);
4   write("x has value ");
5   type(x);
6   write(" and is of type int.")
7 end
```

äquivalenter C-Code:

```
1 #include <stdio.h>
2 int main(void){
3   int  x;
4   scanf("%d", &x);
5   printf("x has value %d\n and is of type int.", x);
6   return 0;
7 }
```

5.9 Kommentare

Ein Kommentar wird mit dem Schlüsselwort **comment**, gefolgt von dem Kommentar eingeleitet, und wird mit einem Strichpunkt abgeschlossen. Er kann nach einem Strichpunkt, **begin** oder **end** stehen wobei der Kommentar selbst keine Strichpunkte enthalten darf und nach einem **end** auch kein **else** oder **end** enthalten darf. Ein korrekt eingefügter Kommentar wird vom Compiler überlesen.

5.10 Code-Beispiel

Folgender Code ist eine Implementierung des "Sieb des Eratosthenes" und soll noch einmal einen groben Überblich über den Aufbau eines ALGOL 60 Programms geben.

Dieser Algorithmus bestimmt Primzahlen in einem definierten Zahlenbereich indem es immer von der nächsten Primzahl alle Vielfachen aus der Lösungsmenge entfernt.

Beispiel 5.8. :

```
begin
  comment die Integer-Werte an einer position im array
      haben folgende Bedeutung:
      1 = Primzahl und 0 = keine Primzahl ;
  integer array candidates[0:1000];
  integer i, j;
  for i := 0 step 1 until 1000 do
  begin
    candidates[i] := 1;
  end;
  candidates[0] := 0;
  candidates[1] := 0;
  i := 0;
  for i := i while i < 1000 do
  begin
```

Die folgende Schleife sucht sich die nächste Zahl, die noch nicht aus der Lösungsmenge entfernt wurde, indem sie `i` solange erhöht bis ein Arrayeintrag mit dem Wert 1 gefunden wird. Diese Zahl ist nun garantiert eine Primzahl.

```
    for i := i while i < 1000 and candidates[i] = 0 do
    begin
      i := i + 1;
    end;
    if i<1000 then
    begin
```

Nun werden alle Vielfachen der aktuellen Primzahl `i` aus der Lösungsmenge entfernt indem immer der Arrayeintrag an der Stelle `k` mit 0 beschrieben wird. `k` wird berechnet, in dem, von `2*i` ausgehend, immer wieder die Primzahl `i` dazu addiert wird.

```
      j := 2*i;
      for j := j while j < 1000 do
      begin
        candidates[j] := 0;
        j := j + i;
      end;
      i := i + 1;
    end ;
  end;
end
```

Nun sind nur mehr die Arrayeinträge mit einer Primzahl als Index mit einer 1 beschrieben.

6 Schluss

Trotz der vielen neuen Ideen die diese Programmiersprache lieferte, war ALGOL 60 nicht sehr erfolgreich. Die Sprache war zwar stark auf numerischem Gebiet, aber schwach bei der Zeichenverarbeitung. Ein weiterer Grund könnte sein, dass die aus Europa stammende Sprache von den großen Computerherstellern in Amerika nicht so sehr wahrgenommen wurde. Nichts desto trotz hatte die Sprache einen großen Einfluss auf die darauf folgenden Programmiersprachen und führte sehr viel ein, was heute standardmäßig in modernen Programmiersprachen verwendet wird.

Der britische Informatiker C. A. R. Hoare untermalte die Bedeutung der Sprache mit folgender Aussage:

> *ALGOL war ein Fortschritt gegenüber den meisten seiner Nachfolger.*

References

[1] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithm language algol 60. *Commun. ACM*, 6(1):1–17, January 1963.

[2] HT de Beer. The history of the algol effort. pages 1–97, 2006.

[3] Rudolf Herschel. *Anleitung zum praktischen Gebrauch von ALGOL 60* -. Oldenbourg, Deutschland, 1969.

[4] Gtz Alefeld, Jrgen Herzberger, and Otto Mayer. *Einfhrung in das Programmieren mit Algol 60* -. Bibliographisches Institut, Gotha, 1972.

[5] Walter: Heinrich. *Programmierung mit ALGOL 60* -. Teubner, Wiesbaden, 1971.

[6] Wikipedia. Algol 60 — wikipedia, the free encyclopedia, 2012. [Online; accessed 6-February-2013, `http://en.wikipedia.org/w/index.php?title=ALGOL_60&oldid=528048408`].

[7] Wikipedia. Algol 60 — wikipedia, die freie enzyklopdie, 2012. [Online; Stand 6. Februar 2013, `http://de.wikipedia.org/w/index.php?title=Algol_60&oldid=110401360`].