Lehrstuhl für Programmiersysteme
Prof. Gert Smolka

Proseminar Programmiersysteme

Vortrag : Frank Bauer
Betreuer: Andreas Rossberg

Ausarbeitung zum
Vortrag

„Strukturierte Programmierung: Algol, Pascal, Modula“

Inhaltsverzeichnis

1. Einführung

In dieser Ausarbeitung soll die Entstehungsgeschichte der Strukturierten Programmierung mit den uns bekannten Konzepten, die heute in den gängigsten Programmiersprachen realisiert sind, nachvollzogen werden. Insbesondere wird dabei auch anhand von Beispielen aufgezeigt, wie schwierig die Umsetzung und Etablierung mancher Ideen war.
Zunächst wird eine Schilderung der Situation vor etwa 1960 aufzeigen, wie wesensfremd die Programmierung von damals verglichen mit der heutigen war. Dann folgt eine genauere Untersuchung am Beispiel der Sprachen Algol, Pascal und Modula – 2.

1.1 Situation zur Zeit vor 1960

Zu dieser Zeit wurden im Wesentlichen die Programmiersprachen FORTAN und Assembler benutzt. Damals hatte man Programme lediglich als eine bloße Folge von Anweisungen, die einen einzigen monolithischen Block bildeten, verstanden. D.h. eine strukturelle Trennung und Aufteilung in mehrere Teile bzw. Dateien, wie es heute zumindest bei größeren Projekten der der Fall ist, war nicht gegeben. Die Programme waren überdies hinaus auch schwer zu lesen und für einen Außenstehenden kaum nachvollziehbar. Ein Konzept, das diesen Umstand begleitete, war die selbstverständliche Benutzung von globalen Variablen, die die ganze Lebensdauer des auszuführenden Programms überdauerten. Aus heutiger Sicht ist klar, dass die Verwendung dieser Variablen ein Programm nur unnötig unübersichtlich machen und es zu Missverständnissen bzgl. anderer Variablen bzw. deren Namen kommen kann. Dieses Konzept wird heute kaum mehr angewendet. Zu jener Zeit gab es auch noch nicht den Begriff der Prozedur, wie er uns heute von vielen verbreiteten Programmiersprachen bekannt ist. Es gab unter FORTRAN lediglich den Begriff der „Subroutine“ , der das Programm zwar syntaktisch aufgliederte, aber nicht den Funktionsumfang bot, wie wir es heute gewohnt sind. Dies ist darauf zurück zu führen, dass eine exakte wissenschaftliche Analyse von Prozeduren damals noch nicht vorgenommen bzw. umgesetzt wurde. Die Benutzung von Prozeduren und Funktionen als „first-class citizens“ ist ein Beispiel dafür. Es war somit nicht möglich eine Prozedur als Argument zu übergeben, noch eine Prozedur als Resultat zu erhalten.
Ein Umstand, der noch öfter in diesem Papier erwähnt werden wird, ist das Auftreten des GoTo – Statements, von dem damals sehr häufig gebraucht gemacht wurde und das heute keine Rolle mehr bei der Programmierung spielt. Anhand von Sprungmarken (Labels), die auch durch Zahlen dargestellt werden konnten, war es möglich beliebig Sprünge innerhalb eines Programms zu vollführen. Dieses macht aber Code unübersichtlich, schwer verständlich und legt nicht unbedingt den wirklichen Programmfluß offen. Außerdem sind mit diesem Konstrukt bei der Einführung der Blockstruktur und der Schleifen sowie bei der Deklarierung von Variablen einige Probleme verbunden, erwähnt sei hier z.B. die Rolle von lokalen Variablen eines Blockes.
Dem FORTRAN zugrunde liegenden Ausführungmodell fehlen ebenfalls heute entscheidende Konzepte. Es wurde gebildet durch
a) Eine Flachregistermaschine
b) Keine Stacks, keine Rekursion und
c) Speicher als lineares Feld angeordent **[Mitchel 2003]**

Die Situation bei Assembler war ähnlich zu FORTAN; hierbei fehlte auch eine höhere Abstraktionsebene , wie sie seit der Einführung der Blockstruktur bekannt ist. Es wurde direkt mit Maschinenregistern gearbeitet.

1.2 Die Einführung neuer Konzepte

Die Strukturierte Programmierung bringt nun bedeutende, neue Konzepte ins Spiel, die die Situation entscheidend veränderte.
Ein bedeutender Punkt der hiermit verbunden ist, ist der Begriff des (lexikalischen) Scopings bzw. des Scopes. Darunter versteht man die Textregion eines Programms, innerhalb der eine Variablendeklaration sichtbat ist, also der Sichtbarkeitsbereich. Auf diesen Aspekt wird später in bezug auf die Blockstruktur noch näher eingegangen werden.
Der Begriff der Prozedur nimmt seit Entstehen der Strukturierten Programmierung einen immer wichtigeren Platz ein. Er wird zum Gegenstand zahlreicher wissenschaftlicher Arbeiten. Diese tragen wesentlich zu der Gestalt der Prozeduren bei, die wir heute von den meisten Programmiersprachen her kennen. Insbesondere der Begriff von lokalen Variablen, die auf dem Stack allokiert und innerhalb einer Prozedur deklariert werden, trägt zum Bild der heutigen Prozeduren bei.
Eine weitere Strukturierung innerhalb der Programmierung wird später durch die Einführung des sogenannten Modulkonzeptes erreicht, das das Modell der monolithischen Programme aufgibt. Dabei werden bestimmte Teile eines Programms etwa in andere Dateien ausgelagert. Dieser Aspekt wird später im Abschnitt unter Modula – 2 genauer untersucht werden. Letzlich mündet er in die „Objektorientierte Programmierung“, die eine Art Weiterentwicklung des Modulkonzepts darstellt. Diese wird allerdings nicht näher in diesem Artikel behandelt werden.

2. Algol

Algol 60/68 lieferten wesentliche Beiträge zur Strukturierten Programmierung. In den nächsten Abschnitten sollen die Konzepte und deren Entstehung sowie die damit verbundenen Problematiken genauer untersucht werden.

2.1 Algol 60 und die Blockstruktur

Algol 60 war eine der ersten Sprachen, die die im letzten Abschnitt genannten Konzepte zumindest teilweise einführte. Sie wurde zwischen 1958 und 1963 durch ein Komitee renomierter Wissenschaftler entwickelt **[Mitchel 2003]**.
Gegnüber FORTRAN bot sie zum ersten Mal ein einfaches statisches Typsystem, das später unter Algol 68 und Pascal verbessert wurde. Es bot bessere Wege Datenstrukturen darzustellen und erweiterte die Anzahl der Grundtypen, die in FORTRAN sehr begrenzt waren.
Einer der wichtigsten Beiträge von Algol 60 zur Strukturierten Programmierung muss ganz klar in der Einführung der Blockstruktur gesehen werden. Blöcke werden mit **begin**..**end** markiert. Bei jedem Eintritt in einen Block wird auf dem Stack Speicher für die lokalen Variablen alloziiert und beim Verlassen des Blocks wieder deallokiert. Die allokierten Variablen sind nun innerhalb des Blockes sichtbar, wobei Blöcke auch geschachtelt werden können. Dabei kann der Sichtbarkeitsbereich (Scope) einer lokalen Variable überdeckt werden:

```
begin
  int x = ...;
  ...
  begin
    int x = ...;
  end
  ...
end
```

Die zuerst deklarierte Variable x ist erst wieder nach dem Verlassen des zweiten Blocks sichtbar.
Mit dieser Blockstruktur konnten auch komplexere Kontrollstrukturen realisiert werden, d.h. insbesondere die heute zum Standardwortschatz gehörenden Schleifen, wie etwa die FOR – oder auch die WHILE – Schleife, in deren Rumpf Variablen deklariert und die selber ineinander geschachtelt werden können.
Allgemein bot sich damit eine neue Ebene, auf der Programme formuliert werden konnten, und zwar ist dies eine abstraktere Ebene, die durch die Blöcke dargestellt wird. Im Gegensatz hierzu wurde in Assembler direkt auf Maschinenebene agiert.
Durch die Allokierung auf dem Stack wurde es auch möglich Funktionen ähnlich wie in LISP rekursiv in der Sprache zu formulieren. Dies ermöglichte es z.B Hoare seinen Quicksort – Algorithmus elegant in Algol zu verfassen **[Hoare 1980].**
Die Versionen von FORTRAN während der 1960er Jahre hatten keine Blockstruktur und konnten auch keine Rekursion verwirklichen. Jeder Variable und jedem Argument wurde ein fester Bereich im Speicher zugeweisen. Rekursive Funktionen hätten sich somit selbst überschrieben!
Ein weiterer Punkt war sicherlich, dass Algol es erlaubte auch allgemeine Ausdrücke innerhalb von Arraygrenzen zuzulassen.

Neben all diesen Vorzügen gab es allerdings auch einige Probleme mit der Sprache, die erst später behoben wurden. Ein wichtiges Beispiel hierfür ist die Parameterübergabestrategie, die in Algol 60 durch Call-by-value und Call-by-name umgesetzt wurde. Die letztere ist eindeutig ein Fehler im Design der Sprache. Hierbei werden die formalen Parameter der Prozedur in den Körper substituiert. Man nennt dies auch Algol 60 Copy-rule **[Mitchell 2003]**. In rein funktionalen Programmen stellt dies kein Problem dar, allerdings gibt es Probleme mit Seiteneffekten.

Das folgende kleine Beispiel zum Tauschen zweier ganzzahliger Variablen zeigt die Schwäche von Call-by-name auf:

```
procedure swap(a,b); integer a,b;
begin integer temp;
        temp:= a;
        a:=b;
        b:=temp
end;
```

Im Falle swap(a[i],i]) funktioniert dies problemlos, denn man erhält durch Einsetzen:
temp:= a[i]; a[i]:= i; i:= temp;
Aber durch den Aufruf mit vertauschten Parametern funktioniert der Austausch nicht mehr:
temp:= i; i:= a[i]; a[i]:= temp
Setzt man hier i = 3 und a[3] = 4, so versagt die Regel...
Ein letzter wesentlicher Aspekt beim Design der Sprache war die erstmalige Verwendung der Backus-Naur-Form (**BNF**), die zur Beschreibung wohlgeformter Programme diente. BNF ist der Standard zur Formulierung der Syntax einer Programmiersprache. Beim Entwurf von Algol 60 setzte man sich zum ersten Mal systematisch mit diesem Thema bezüglich einer Programmiersprache auseinander

2.2 Entstehung von Algol 68

Historisch gesehen entstand Algol 68 aus Algol 60. Anfang der 1960er Jahre wurde eine Arbeitsgruppe renomierter Wissenschaftler (WG 2.1) ins Leben gerufen. Diese sollte eine Weiterentwicklung von Algol 60 entwerfen, um eine ausruckskräftigere Sprache mit weniger umstrittenen Konzepten zu schaffen. Jedoch kam es häufig zu Interessenskonflikten bezüglich der zu verwirklichenden Ideen einzelner Wissenschaftler, so dass zum Teil fragwürdige Kompromisse in die Sprache einflossen. Letzlich hat sich der Entstehungsprozess über Jahre hingezogen, so dass der „Revised Report“ der Sprache erst Mitte der 1970er Jahre veröffentlicht wurde. Zudem war die Verfügbarkeit der Compiler sehr begrenzt.
Dennoch beinhaltet die Sprache auch grundlegende Ideen, die heute als Selbstverständlichkeit in modernen Programmiersprachen implementiert sind.
Insgesamt gesehen brachte Algol 68 eine Systematisierung des Typsystems, wie man in den nächsten Abschnitten näher sehen wird. Damit verbunden ist eine Reihe neuer Typen , die ohne größere Meinungsverscheidenheiten in die Sprache einflossen; zu nennen sind hier etwa die „long“ Versionen von **int** und **real**, der Typ **compl** für komplexe Arithmetik und weitere mehr. Eine Besonderheit ist hierbei die Definition von Strings. Nach **[Lindsey 1993]** ist Algol 68 immernoch die einzige Hochsprache mit Stringvariablen, die keiner maximalen Länge bedarfen. Strings wurden als flexibles Array definiert:

```
mode string = flex [1:0] char;
```

wobei **mode** die Typdefinition einleitet.
Ein unstrittiger Punkt damals war das FOR-Statement, das aus unserer heutigen Sicht zu überladen scheint:

for a **from** b **by** c **to** d **while** e **do** f **od**

Ein wesentlich wichtigerer Aspekt ist jedoch die Tatsache, dass Algol 68 eine imperative Sprache war, die besonderen Wert (zur damaligen Zeit) darauf legte, keinen Unterschied zwischen Ausdrücken (expressions) und Anweisungen (statements) zu machen, d.h. alle Formen von Konditionalausrücken, Case - Ausdrücken und Blöcken liefern Werte. Dies wird am folgenden Beispiel veranschaulicht:

x:= (**real** a= p*q; **real** b=p/q; **if** a>b **then** a **else** b **fi**) + (y:= 2*z);

Darüberhinaus gab es noch einige andere Punkte, die in neueren Sprachen nicht unbedingt auftauchen, aber auch wenig bedeutsam sind, wie etwa das Slicing von Arrays mit dem man Reihen und Spalten extrahieren konnte.
Ein weiterer Sachverhalt der in Verbindung mit Algol 60 zu sehen ist, beinhaltet den Versuch der exakten Definition der Semantik von Algol 68. Algol 60 führte zur Definition der Syntax die BNF ein, wohingegen hier versucht wurde systematisch die Semantik zu beschreiben, wie in **[Lindsey 1993]** ersichtlich wird.

2.3 Orthogonalität

Ein bedeutender Beitrag zur Systematisierung des Typbegriffs - auch in anderen Programmiersprachen - war die Einführung des Konzepts der Orthogonalität in Algol 68.
Allgemein meint man damit die beliebige Kombinierbarkeit von verschiedenen Konstrukten. Eines dieser Konstrukte bilden die Records, in Algol 68 als Strukturen eingeführt, die nach zahlreichen Diskussionen innerhalb der WG 2.1 die Gestalt annahm, wie wir sie heute ähnlich von C her kennen. Sie wurden ins Typsystem integriert und sollten als Variablen, Parameter und Resultate auftreten. Ähnlich verhält es sich mit Prozeduren, die ebenfalls als Variablen, Parameter und Resultate realisiert sein sollten.
Das dritte wichtige und umstrittene Konzept war das der Referenzen. In Algol 68 werden Variablen und Referenzen als synonyme Konzepte angesehen, so dass eine **real** Variable eigentlich vom Typ **ref real** ist. Auch Referenzen konnte man kombinieren; d.h., wenn **ref** t ein Typ ist, dann ist auch **ref ref t** ein Typ. Insbesondere subsummieren Referenzen zu den Variablen auch noch Pointer (ref ref var). Mit dieser Grundlage war es nun möglich diese drei Punkte beliebig zu kombinieren, etwa zu Arrays aus Pointern zu Prozeduren, wie in **[Mitchell 2003]** angeführt.
Der Begriff der Orthogonalität war nicht unumstritten. So beschreibt zum Beispiel Hoare in [**Hoare 1980**] seine Bestürzung über die „Vorherrschaft der Referenzen“ und anderer komplexer Eigenschaften, die zu dieser Zeit zu der Sprache hinzugefügt wurden. Dennoch setzte sich das Konzept durch und wurde später in anderen Sprachen angepasst. Besonders die **structs** und Pointertypen von C wurden durch Algol 68 beeinflusst, wie in [**Lindsey 1993**] beschrieben. Das Referenzkonzept hielt zum Beispiel auch Einzug in SML, wo die veränderlichen Werte durch Referenzen repräsentiert werden.

2.4 Das Typsystem von Algol 68

Nach einer kurzen Darstellung der Entwicklung der Parametübergabe werden wichtige Aspekte des Typsystems und deren spätere Einbindung in andere Sprachen beschrieben.

2.4.1 Parameterübergabe

Bei der Übergabe der Parameter orientierte man sich zunächst an Algol 60 mit call-by-value und call-by-name. Die weitere Entwicklung war jedoch sehr kontrovers und es gab etliche Vorstellung seitens der verschiedenen Wissenschaftler. So wurde der call-by-name Mechanismus zum Beispiel in zwei Fälle aufgespalten:
1. call-by-full-name, bei der der tatsächliche Parameter ein Ausruck war und
2. der Fall, bei dem er eine Variable war.
Die Entwicklung gelangte später durch die Einbindung der Referenzen zu dem endgültigem Mechanismus, wie er in der Endversion der Sprache vorhanden ist, d.h. call-by-value mit call-by-reference, realisiert durch Pointertypen **[Mitchell 2003]**.

2.4.2 Strukturelle Typäquivalenz

Ein interessanter Punkt des Typsystems von Algol 68 ist die Behandlung der Typäquivalenz. Sie wird dort als strukturell aufgefasst. Dazu ein Beispiel:

```
mode a = struct (int val, ref a next);
mode b = struct (int val, ref struct (int val, ref b next) next);
```

Beide Typdeklarationen beschreiben denselben Graphen und sind somit in Algol 68 als äquivalent anzusehen. In Pascal wurde schließlich eine Regel zur Namensäquivalenz eingeführt, wie in **[Lindsey 1993]** erwähnt.

2.4.3 Unions

Eine Variable eines Typs, der als Union deklariert wurde, kann gewissermaßen zwei Typen haben. In Algol 68 gibt es sogar ein Konstrukt mit dem man dann je nach tatsächlichen Typ bestimmte Anweisungen selektiv ausführen kann. Hierzu ein Beispiel:

```
mode man = struct (string a,...), woman = struct (string b, ...);
mode person = union (man, woman);
```

Wurde body als „person“ deklariert, x als „man“ und y als „woman“, kann man nun folgerndermaßen durch einen Case – Ausdruck selektieren:

```
case body in
      (man m): x:= m;
      (woman w): y:= w;
esac
```

Für unions gilt Kommutativität und Assoziativität, wie in **[Lindsey 1993]** aufgezeigt.

2.4.4 Prozeduren

Da wie schon früher ausgeführt, Prozeduren als formale Parameter auftreten und gemäß dem Prinzip der Orthogonalität mit anderen Typen „vermischt“ werden konnten, kann man den Prozeduren in Algol 68 den Status von „first-class-citizens“ zuordnen, wie es **[Lindsey 1993]** ausführt. Der Begriff „first-class-citizens“ ist heute in funktionalen Programmiersprachen üblich.

2.5 Coercion

Der Begriff Coercion wurde erstmals als systematisches Konzept in Algol 68 eingeführt und ist in vielen Programmiersprachen in ähnlicher Weise umgesetzt. Man versteht darunter den impliziten Wechsel eines Typs (a priori) eines Ausdrucks um den Typ des umgebenden Kontextes (a posterio) zu entsprechen.
Die wichtigsten sollen nun aufgrund ihrer weiten Verbreitung näher erläutert werden:

a) Widening

Dieser Coercion ist unumstritten und in den meisten Sprachen etabliert. Damit ist einfach die Zuweisung eines Wertes gemeint, der zuvor „geweitet“ wird. Konkret kann man so einer Varibale vom Typ **real** einen Wert vom Typ **int** zuweisen.

b) Dereferencing

Hiermit kann man den Wert, der durch eine Referenz referenziert wird, erhalten.
Hat man eine Variable vom Typ **real** deklariert, hat sie den Typ **ref real**. Will man sie einer Variable zuweisen, muss die zugewiesene Variable zunächst dereferenziert werden. Dies geschieht in Algol 68 automatisch.

c) Deproceduring

Dies ist ein Coercion, der gebraucht wird, wenn man parameterlose Prozeduren benutzen möchte. Man benötigt ihn also, um zu unterscheiden, ob man den Wert haben will, den die Prozedur zurückliefert (**real** x:= random) oder ob man den Wert, den die Prozedur selbst darstellt, haben will (**proc** r:= random). Der erste Fall wird bei manchen Programmiersprachen einfach durch das angeben eines Klammernpaares erreicht, zum Beispiel p().

d) Uniting

Das Uniting betrifft nur Sprachen, die auch die im letzten Abschnitt angesprochenen Unions einsetzen. Somit kann etwa ein a priori Typ **int** in einen a posterio Typ **union** (**int**, **real**) umgewandelt werden.

Diese Coercions sind unumstritten und verbreitet. Zwar hat Algol 68 noch weitere realisiert, aber diese sind nicht ganz so leicht und bergen eher Stoff zu Diskussionen.
Es sei hier auch noch auf die Rolle des GoTo – Statements verwiesen, welches auch in Algol 68 noch immer verwendet wurde.

3. Pascal

Pascal hat ebenso wie Algol einen wesentlichen Beitrag zur Strukturierten Programmierung geleistet, wenn auch auf eine andere Art.
Die Sprachdefinition der Programmiersprache Pascal wurde 1970 vom Erfinder der Spache – N. Wirth – veröffentlicht. Wirth selbst war Mitglied der WG 2.1 gewesen, die Algol entwickelte. Jedoch war ihm die Arbeit innerhalb der Gruppe zu mühselig. Die heftigen Debatten und Diskussionen, die den Entwurf von Algol verzögerten, ebenso wie die Kompromisse die geschlossen werden mussten, bewegten ihn dazu eine eigene Sprache zu entwickeln, die quasi als Gegenentwurf zu sehen ist. In dieser wollte er seine alleinigen Vorstellungen umsetzen. So verschob die Komplexität von Algol 68 die endgültige Implementation der Sprache selbst auf Anfang der 1970er Jahre, als viele von Algol 60 auf andere Sprachen umgestiegen waren **[Wirth 1993]**.
Als Wirth 1968 Professor in Zürich wurde, war Algol 60 die erste Wahl unter den Wissenschaftlern. Die Compiler der Sprache waren schlecht designed bezüglich der damals vorhandenen CDC Computer und konnten sich mit den FORTRAN Compiler nicht messen, wie in **[Wirth 1993]** zu sehen.
All dies veranlaßte Wirth eine neue Sprache zu entwicklen. Diese sollte wesentlich einfacher und ausdruckskräftiger sein als Algol 68. Außerdem sollte die Sprace Pascal nicht nur theoretisch die Grundlagen Strukturierter Programmierung und des „Stepwise Refinement“ abarbeiten, sondern sie vielmehr in der Praxis nutzbar machen. Insbesondere hatte er dabei die Vorstellung, die Sprache im Lehrbereich zu etablieren. Später wurde Pascal dann eine weitverbreitete Sprache in genau diesem Sektor. Zu dem Erfolg trugen wesentlich auch die entwickelten Handbücher bei, die mit speziellen Anleitungen und Übungen versehen waren, welche es in einer solch erfolgreichen Ausprägung bis zu jenem Zeitpunkt nicht gab.
Darüberhinaus stand für Wirth der Begriff der Effizienz im Mittelpunkt. So sollte es möglich sein, effiziente Programme zu schreiben, die auf einem möglichst effizienten Compiler basierten, der sich mit den damals vorhandenen FORTRAN Compilern messen lassen sollte. Hierbei gab es jedoch etliche Umsetzungsprobleme um die Effizienz zu erreichen, die Wirth sich vorstellte.
Gerade in Hinsicht auf die Portierbarkeit gab es bei der Weiterentwicklung von Pascal ein erwähnenswertes System, das heute als P-Code-System bekannt ist. Dieses bestand aus einer P(seudo)maschine, dem P-Code und dem P-Compiler und erfuhr bei der Entwicklung von UCSD Pascal um 1975 noch einige Erweiterungen wie einen Programmeditor, ein Filesystem und einen Debugger. Im Wesentlichen entspricht es dem, was heute unter dem Begriff der „Virtuellen Maschine“, vor allem von der Programmiersprache Java, bekannt ist.
So wurde der Sourcecode erst in sogenannten Bytecode übersetzt, der dann auf der Maschine lief. **[Wirth 1993]** weist darauf hin, dass durch dieses System einer weiten Verbreitung der Sprache die Tür geöffnet wurde, da mit dieser Architektur Pascal auf verschiedensten Plattformen realisierbar war.
1978 begann dann eine erste Standardisierung unter der Federführung von IEEE und ANSI. Später folgte eine Arbeitsgruppe der ISO. Unterdessen wurde Pascal von verschiedenen Firmen selbst implementiert und es gab im weiteren Verlauf eine ganze Menge verschiedener Pascal - Dialekte. Unter anderem entstanden auch Erweiterungen wie das bekannte Concurrent Pascal, später folgte dann zum Beispiel auch TURBO - Pascal.
Wirth selbst wandte sich Ende der 1970er Jahre dann einer anderen Programmiersprache zu - nämlich Modula -, die in 4. behandelt wird.

3.1 Probleme der Sprache Pascal

Wie schon angesprochen war Pascal auf Einfachheit ausgelegt. Getreu diesem Prinzip wurden auch einige Konstrukte, die in Algol überladen waren, in eine einfachere, effizientere Form gebracht, zum Beispiel das schon erwähnte FOR – Statement.
Wichtig ist hierbei aber, dass es Wirth besonders um die Vielfalt an Daten(typ) – Strukturen ging, wohingegen Algol Wert auf die Einführung von Statements legte.
Deshalb versuchte er auch viele Ideen von Hoare hinsichtlich dynamischer Datenstrukturen und Pointerbindung zu verwirklichen. Um eine möglichst hohe Effizienz zu erhalten, versuchte Wirth am Begriff der frühen Bindung und des statischen Typs in besonderer Weise festzuhalten, was zu einigen schlechten Entscheidungen hinsichtlich des Designs verschiedener Konstrukte führte.
Herausstreichen sollte man etwa, dass bei der Übergabe von Arrays als Prozedurparameter, die Arraygrenzen strikt zu behandeln sind. Folgendes Beispiel verdeutlicht dies:

```
TYPE A1 = ARRAY [1..10] OF INTEGER;
     A2 = ARRAY [1..20] OF INTEGER;
VAR  x: A2;
PROCEDURE P(x: A1); BEGIN...END
```

Hierbei ist es nicht erlaubt, die Prozedur P mit der deklarierten Variable x vom Typ A2 aufzurufen. Ebensowenig ist eine Verallgemeinerung der Arraygrenzen mit einem Parameter erlaubt. Diese Punkte führen dann zum Beispiel zu Problemen bei der Programmierung von allgemeinen Algorithmen.
Ein Beispiel für eine äußerst ungelungene Datenstruktur ist der sogenannte „Variant Record“:

```
VAR R:  RECORD maxspeed: INTEGER;
            CASE  v: Vehicle OF
            truck: (nofwheels: INTEGER);
            vessel: (homeport: STRING);
        END
```

In diesem Fall kann man R.nofwheels nur anwenden, wenn R.v den Wert truck hat und R.homeport kann lediglich verwendet werden, wenn gilt R.v = vessel. Somit erlaubt der Variant Record den Wechsel der Elementtypen zu jeder Zeit, wodurch ein Overhead zur Laufzeit entsteht, da jedes mal der Elementtyp kontrolliert werden muss. Wirth gibt selbst an, dass diese Datenstruktur für etliche Leute ein beliebter Ansatzpunkt war, das Typsystem zu „brechen“. Die Implementation war zudem unsicher im Falle einer Zuweisung, da die einzelnen Felder im Speicher überlagert wurden.
Einen weiteren Kritikpunkt, den [**Kernighan 1981**] anführt, ist die Tatsache, dass Pascal sehr monolithisch ausgrichtet ist und es keine „Separate Kompilierung“ gibt, die zum Beispiel die Realisierung großer Softwareprojekte ermöglicht. Wirth setzt dieses Ziel erst mit der nachfolgenden Sprache Modula – 2 um. Diese führt das Modulkonzept ein, wie im nächsten Abschnitt genauer erläutert wird.
Eine letzte Bemerkung, die aber eher symbolischen Wert hat, ist die Feststellung, dass auch in Pascal noch immer am GoTo – Statement festgehalten wurde, obwohl Dykstra schon auf die damit verbundene Problematik hingewiesen hatte.

4. Modula – 2

Modula – 2 ist der Nachfolger von Pascal und wurde Ende der 1970er jahre durch N. Wirth entwickelt. In Modula - 2 wird ein weiteres Konzept der Strukturierten Programmierung zur Verfügung gestellt, und zwar wird durch eine Modularisierung die bisher noch vorhandene monolithische Struktur eines Programms aufgebrochen.

4.1 Das Modulkonzept

Die Module sind das bedeutenste Konzept, das Modula – 2 von Pascal unterscheidet. Zunächst einmal versteht man unter einem Modul eine Zusammenfassung von verschiedenen Deklarationen, im einzelnen: Konstanten, Variablen, Typen und Prozeduren. Objekte, die man in Modulen zusammengefasst hat, können nun wiederum in anderen Modulen zur Verfügung gestellt werden; man nennt dies Importierung. So werden überlicherweise oft spezielle Standardprozeduren zur Ein-/Ausgabe auf dem Bildschirm benötigt. Ein einfaches Hauptprogramm, das selbst auch ein Modul darstellt, verdeutlicht dies:

```
MODULE Hauptprogramm;
FROM InOut IMPORT Read, Write;
Deklarationen;
BEGIN
Read(ch); Write(ch);
END Hauptprogramm;
```

Somit erspart man sich das erneute Erstellen wichtiger Prozeduren und lagert sie aus dem Hauptprogramm aus. Dieses gliedert sich somit in verschiedene Module, die ihrerseits wiederum Prozeduren enthalten können. Die Module können somit separat aufbewahrt werden und unabhängig vom Programm kompiliert werden. Bei Bedarf werden diese dann mit dem Hauptprogramm gelinkt. Man nennt dies Separate Kompilierung **[Wirth 1982]**. Somit kann man eine ganze Hierarchie an Modulen erhalten, an deren Spitze das eigentliche Programm steht; dies kennt man in ähnlicher Form von der Objektorientierten Programmierung. Diese indirekten Importe unter den Modulen können somit eine Art Umgebung schaffen.
Neben der Wiederverwendbarkeit spielt außerdem die Unabhängigkeit der Erstellung eine wesentliche Rolle. So können an einem Projekt verschiedene Aufgaben in verschiedenen Modulen durch mehrere Programmierer bewältigt werden und somit zum ersten Mal große Projekte umgesetzt werden. Dies war durch die schon vorgestellten Sprachen in dieser Art nicht möglich.
Außerdem eignet sich die Aufspaltung in Module dazu, eine gewisse Abstraktionseben zur Verfügung zu stellen. Und zwar in dem Sinn, dass man als Benutzer die Funktionalität ausnutzen möchte, nicht aber unbedingt mit den genauen Implementierungsdetails konfrontiert werden will. Darüberhinaus bietet sich durch die Abstraktion nach Außen, die die Module letzlich darstellen, die Möglichkeit Informationen auch gezielt zu schützen, um die Konsistenz von Programmen sicherzustellen und Fehler einzudämmen, wie in **[Wirth 1982]** beschrieben. Die Enkopplung der Module ermöglicht es zudem, auf eine Recompilierung oder Änderung eines importierenden Moduls zu verzichten, wenn das importierte Modul verändert werden muss. Folglich muss eine Trennung des Wesentlichen von der genauen Implementierung erfolgen. Dies wird in Modula – 2 durch zwei verschiedene Module verwirklicht.

4.2 Trennung in Interface und Implementierung

Das „definition module“ stellt das Interface, das der Benutzer zu Gesicht bekommt, dar. Es bietet die Deklarationen der verschiedenen Objekte, d.h etwa von Prozeduren oder Typen, wobei nur der Prozedurkopf angegeben wird und die Implementierung „verschwiegen“ wird. Dieses Modul sorgt nur für die Sichtbarkeit der Funktionalität nach außen hin.
Die eigentliche Implementation bildet das „implementation module“. In ihm werden die genauen Implementierungsdetails des beobachtbaren Verhaltens festgelegt. Konkret heißt dies, dass hier alle Prozedurkörper vorhanden sind und eingesehen werden bzw. verändert werden können.
Beide Teile werden unabhängig voneinander komiliert und bilden somit eigenständige „compilation units“ **[Wirth 1982]**.
Hierzu ein Beispiel:

```
DEFINITION MODULE Punkt;
 TYPE Punkt = ARRAY[1..2] OF REAL;
 PROCEDURE  assign(x,y:REAL):Punkt;
 PROCEDURE  abs(p:Punkt): REAL;
END Punkt.
```

```
IMPLEMENTATION MODULE Punkt;
 PROCEDURE  assign(x,y:REAL): Punkt;
   BEGIN...END assign;
 PROCEDURE  abs(p:Punkt): REAL;
   BEGIN...END abs;
END Punkt.
```

Ähnliche Konzepte kennt man von modernen Programmiersprachen wie SML; dort werden die Begriffe „signature“ und „structure“ verwendet.

4.3 Typabstraktion

Das obige Beispiel bezieht sich auf den transparenten Export von Typen, d.h. die detaillierte Typdeklaration ist nach außen hin in anderen Modulen sichtbar. Will man aber gemäß dem Prinzip des „Information Hiding“ diese Information verbergen, so muss man zum opaken Typexport wechseln. Der Unterschied besteht lediglich darin, dass die komplette Typdeklaration hierbei nur in der Implementierung zu finden ist und nicht etwa im allseits zugänglichen Interface. Die Nutzung von opaken Typexporten ist allerdings auf den Gebrauch von Pointertypen beschränkt. Dadurch wird eine uniforme Typdarstellung ermöglicht. Überträgt man dies auf das obige Beispiel, so erhält man für die Typdeklaration im „definition module“ und im „implementation module“:

```
TYPE Punkt;
```

```
TYPE Punkt = POINTER TO PunktData;
TYPE PunktData = ARRAY [1..2] OF REAL;
```

Somit sieht der Benutzer lediglich, dass ein Typ Punkt deklariert wurde und auch als Argumenttyp bzw. Ergebnistyp vorkommt; die genauen Details sind ihm jedoch verborgen.

4.4 Lokale Module

Lokale Module sind verschachtelte Module, die innerhalb anderer deklariert wurden. Hieraus folgt direkt, dass sie nicht separat kompilierbar sind. Sie stellen einen eigenen Scope zur Verfügung, in dem ihre deklarierten Objekte sichtbar sind. Man kann nun durch IMPORT oder EXPORT Objekte des äußeren Scope nach innen hin bzw. von innen nach außen hin sichtbar machen. Das anschließende selbsterklärende Beispiel veranschaulicht diesen Sachverhalt:

```
VAR a,b: INTEGER;
MODULE M;
  IMPORT a; EXPORT w, x;
  VAR u, v, w: INTEGER;
  MODULE N;
    IMPORT u; EXPORT x, y;
    VAR x, y, z: INTEGER;
   (* u, x, y, z sind hier sichtbar *)
  END N;
  (* a, u, v, w, x, y sind hier sichtbar *)
END M;
(* a, b, w, x sind hier sichtbar *)  [Wirth 1982]
```

Sollte es zu Namenskonflikten bei der Exportierung kommen, so kann man die Variablen mit qualifiziertem Modulnamen exportieren. Dies wird erreicht, indem man EXPORT QUALIFIED benutzt. Dadurch wird ein doppeltes Vorkommen durch Angabe des Modulnamens aufgelöst, zum Beispiel A.x und B.x.
Festzuhalten ist, dass also auch Scopingregeln auf dieses neue, durch Modula – 2 eingeführte Modulkonzept, übertragen wurden - insbesondere auch auf lokale Module.

5. Schlussfolgerung

Zusammenfassend kann man sagen, dass durch die Sprachen Algol, Pascal und Modula – 2 wichtige Konzepte der heute allseits bekannten Strukturierten Programmierung eingeführt wurden, wenngleich dies nicht auf allen Gebieten der Fall ist.
Algol 60 steuerte einen wichtigen Teil dazu bei, indem es durch die Blockstruktur eine neue Abstraktionsebene einführte. Damit verbunden sind natürlich Begriffe wie Scoping, lokale Variablen und Stack. Algol 68 baute das Typsystem von Algol 60 aus und erarbeitete systematisch Konzepte wie Orthogonalität und Coercion. Negativ anzumerken ist hierbei, dass zahlreiche Kompromisse geschlossen werden mussten und der Entwicklungsprozess sich zu sehr in die Länge zog, so dass die Sprache letzlich nie eine relevante Bedeutung in der Praxis einnahm. Außerdem war der Beitrag von Algol 68 selbstverständlich nicht umfassend, wie das Fehlen von Exception Handling oder auch Polymorphismus zeigt.
Pascal wiederum erlangte in der Praxis weite Verbreitung insbesondere im Lehr- und Lernbereich, was auch das erklärte Ziel des Entwicklers N. Wirth war. Die Sprache machte wesentliche Konzepte der Strukturierten Programmierung einer breiten Masse zugänglich. Wie im entsprechenden Abschnitt ersichtlich, hatte auch Pascal seine Probleme bei der Umsetzung der Ziele. Überdies hinaus integrierten sowohl Algol als auch Pascal immernoch das GoTo – Statement.
Schließlich führte Modula – 2 mit dem Modulkonzept ein Instrument ein, mit dem sich Aufgaben nicht nur mit Prozeduren lösen ließen, sondern in eigenständigen Einheiten, den Modulen, unabhängig vom Hauptprogramm verwirklichen lassen konnten. Dabei wurde versucht die bereits enwickelten Konzepte systematisch auf diesen neuen Sachverhalt zu übertragen, zum Beispiel das Scoping. Im Gegensatz zu den anderen Sprachen verbannte Modula – 2 dann auch endlich das dubiose GoTo – Statement.
Man sieht also, dass hier eine große Entwicklung stattgefunden hat, die sich von Algol über Pascal bis zu Modula – 2 hin erstreckt. In ihr wurden wichtige Grundsteine gelegt, die uns heute in vielen Programmiersprachen begegnen.

Referenzen

[Mitchell 2003]

John C. Mitchell, *Concepts in Programming Languages*, Cambridge University Press 2003

[Hoare 1980]

C.A.R. Hoare, *The 1980 ACM Turing Award Lecture*, Communications of the ACM, Februar 1981

[Lindsey 1993]

C.H. Lindsey, *A History of ALGOL 68*, ACM SIGPLAN Notices, März 1993

[Wirth 1993]

N. Wirth, *Recollection about the Developement of Pascal*, ACM SIGPLAN Notices, März 1993

[Kernighan 1981]

Brian W. Kernighan, *Why Pascal is Not My Favorite Programming Language*, April 1981

[Wirth 1982]

N. Wirth, *Programming in Modula – 2*. Springer 1982