Modula

A Language for Modular Multiprogramming

Daniel Ivancic
0425234

email: e0425234@student.tuwien.ac.at

Modula :
A Language for Modular Multiprogramming

Abstract:
Modula wurde 1976 von Niklaus Wirth vorgestellt. Seine Sprache sollte vor allem für die Systemprogrammierung und für die Entwicklung von Prozess-Kontrollen auf kleinen Rechnersystemen eingesetzt werden und somit die Vormachtstellung des Assemblerprogrammierens in diesen Bereichen brechen.
Da die Sprache auf Pascal basiert, findet man viele Elemente dieser im Bereich der Syntax wieder. Jedoch wurde Modula um das Konzept des *Multiprogrammings* und um das *Module* Konzept erweitert um den Anforderungen in den genanten Bereichen zu entsprechen.
Modula selbst wurde praktisch nie eingesetzt. Der Nachfolger, Modula 2, erfreute sich aber von großer Beliebtheit wegen der Eignung für fast alle Programmieraufgaben.

Inhalt:

1) Wie bereits erwähnt wurde Modula von Niklaus Wirth entwickelt. Wirth in der Schweiz geboren, arbeitete an zahlreichen Projekten an Technischen-Universitäten in Canada und den USA mit, bis er schließlich zurück in die Schweiz auf die ETH kam. Dort entwickelte er Pascal, später Modula und Oberon. Wirth wurde 1984 mit dem ACM-Turing Award ausgezeichnet.

Wirth wollte mit Modula ein Konzept einer höheren Programmiersprache verwirklichen, das im Bereich der Systemprogrammierung und Prozess-Kontrolle die Dominanz des Assemblerprogrammierens bricht.
Für diese Anforderung musste ein Konzept für *multiprogamming* entwickelt werden und ein Konzept für das Operieren auf spezifischer peripherer Hardware.
Multiprogamming nennt man die Technik, welche, während Prozesse auf Eingabe warten, andere Ausführt, um so die CPU-Auslastung zu optimieren.

Für diese Anwendungsbereiche wurde Modula, erstens, mit *Multiprogramming facilities* ausgestattet . Das bedeutet, dass es die Möglichkeit gibt mehrere Prozesse konkurrierend, nebeneinander aufzurufen. Diese Möglichkeit manifestiert sich in den Konstrukten *processes*, *interface modules* und *signals*, die ich noch später erörtern werde.

Um auf der peripheren Hardware zu operieren , die von Computersystem zu Computersystem verschieden ist und von der jeweiligen Hardware und den Hardwareeinstellungen abhängt ist, musste für Modula ein Konstrukt entwickelt werden, dass es erlaubt diese maschinenabhängigen Operationen, in unabhängigen kleinen Programmteilen zu definieren. Diese Konstruktion trägt bei Wirth den Name *module* . Diese *modules* sind in Modula von solch einer Wichtigkeit, dass sie der Sprache ihren Namen verleiten.

Für die sequenzielle Programmierung wurde, wie wir später bei der Erläuterung des Syntax zu sehen werden , das meiste von Pascal übernommen.

Die Sprache wurde bewusst klein gehalten, um eine leichte Implementierung auf Rechnern mit nur wenig Run-Time Support zu gewährleisten. So ist zum Besipiel das Fehlen von jeglicher Input/Output Unterstützung zu verstehen. Diese soll der Programmierer selbst mit Hilfe von Modulen, im Zuge der Hardwareansteuerung entwickeln.

2)Allgemein ist zu sagen, dass der Syntax, dadurch dass vieles, wie bereits erwähnt, von Pascal übernommen wurde stark an diese Sprache erinnert. Was vor allem heraussticht, ist die sehr gute Lesbarkeit des Quelltextes und die Eigenschaft, dass der Code selbsterklärend wirkt.

+	(	div	until	const
-	)	mod	while	var
*	[	or	do	type
/	]	and	loop	array
=	.	not	when	record
<>	,	if	exit	procedure
<	;	then	begin	process
<=	:	elsif	end	module
>	‘	else	with	interface
>=		case	value	device
	(*	of	xor	use
:=	*)	repeat		define

Operat undReservierte Wörter in Modula

2.1)Es gibt vier Grunddatentypen in Modula:

Boolean, mit den Werten *true* oder *false*

Integer, ganzzahliger Typ, der Wertebreich ist abhängig vom System

Char, Zeichen aus dem vom System verwendeten Zeichensatz

Bits, enspricht einem Array von w Boolean Elementen; w entspricht der Wortlänge/Registergröße des Systems auf dem Modula implementiert wird.

Es wurde weder der **real** Typ aus Pascal übernommen, noch findet man einen Pointertyp in Modula .

Eine Variablendefinition wird mit dem reservierten Wort *var* eingeleitet und steht immer nach dem Prozedur- oder Programm- oder Modulkopf.

Über die Grundtypen hinaus gibt es noch *records* und *arrays* mit denen man komplexere Typen definieren kann.

```
Arrays: matrix= array 1 :20 of integer;
        Matrix=array 1:20,0:10 of integer;
account = record
        Username: array 1:20 of char;
                Z:integer;
        end;
```

2.2) Man findet in Modula 3 Typen von Schleifen:

Kopfgesteuert:"*while*" Ausdruck "*do*" Anweisungsblock "*end*"
Fussgesteuert:"*repeat*" Anweisungsblock"*until*" Ausdruck
"*Loop*" Anweisungen;
"*when*" Ausdruck "*do*" Anweisung "*exit*"
"*when*" Ausdruck "*do*" Anweisung "*exit*"
end;

2.3) Kontrollstrukturen/Logische Abfragen :

IF Abfrage
"*if*" Ausdruck "*then*" Anweisungen
"*elsif*" Ausdruck "*then*" Anweisungen
"*else*" Anweisungen
"*End*"

Case Abfrage:
"*case*" Variable "*of*"
Ausdruck":""*begin*"Anweisungen"*end;*"
"*end*"

2.4) *Modules* sind, wie bereits erwähnt, Ansammlungen von Prozeduren, Dateitypen und Variablen, wobei man jedoch die Sichtbarkeit und Verwendbarkeit dieser nach außen hin kontrollieren kann. Typen, die in der *use*-list deklariert sind, werden importiert, Typen in der *define*-list werden exportiert. Außerhalb des Moduls ist der Typ der exportierten Größe nicht erkennbar. Nur die Prozeduren und Prozesse, die innerhalb eines solchen Modules deklariert sind und exportiert werden, können mit diesen operieren.(vergleiche Module mit Classen in Java, AdV)

Man kann in Modula drei Arten von Modulen unterscheiden. *Device Modules,* die zur Implementierung Modulas auf spezifischen Rechnern und deren Hardware dienen, *interface modules,* die speziell für *multiprogramming* verwendet werden und normale *modules*, die für allgemeine Aufgaben verwendet werden können.
Beispiele für solche *module*-Konstruktionen folgen später.

2.5) Eine der erwähnten Besonderheiten Modulas sind die *multiprogramming facilities* die dazu dienen Prozesse konkurrierend ablaufen zu lassen und diese zu koordinieren. Die Konstrukte die dafür verwendet werden sind *interface modules*, *processes* und *signals.*

Interface module sind Module die als "Rahmen" für die Prozeduren und Variablen, die innerhalb konkurrierende Prozesse aufgerufen werden und koordinieren und nur einen aktiven Prozess den Zugriff auf die Typen zulassen und die anderen, wartenden Prozesse, verwaltet.

Processes sind Konstrukte die konkurrierend ablaufen. Sie sind deklariert wie Prozeduren, können selbst aber keinen neuen Prozess starten und müssen in der äußersten Programmstruktur aufgerufen werden, d.h. eine Prozedur kann keinen Prozess enthalten.

Prozesse sind völlig selbständig, wenn sie einmal aufgerufen werden.
Ein Prozess kann drei Zustände annehmen: bereit, laufend oder wartend auf ein *signal*.
Prozesse die laufen oder bereit sind, sind in einer Art Ring verbunden.

Signals dienen der Synchronisierung von Prozessen. Sie werden deklariert wie Variablen als Typ signal. Mit diesen Signalen können zwei Operationen und ein Test ausgeführt werden, nämlich *wait*(s), *send*(s) und *awaited*(s).(s:signal).
Wait(s) bewirkt, dass der Prozess, indem diese Prozedur aufgerufen wird, wartet, bis *s* von einem anderen Prozess in den Ring gesendet wird.
Send(s) sendet s in den Ring, sodass alle Prozesse, die im Ring auf s warten, weiter abgearbeitet werden.
Awaited(s) prüft ob s im Ring der Prozesse erwartet wird.

3)Um den besprochenen Syntax zu verdeutlichen führe ich jetzt ein Beispiel (Für die Formatierung des Textes wurde der Quelltext des Modulabeispieles auf der folgenden Seite plaziert) an.
Dieses soll die Modulkonstruktion und die Variablendeklaration näher erklären und die *multiprogramming fascilities* näher erläutern.

In diesem Beispiel laufen zwei Prozesse "Producer" der von einem Eingabegerät Zeichen in einem Buffer einlest und "Consumer", der die Zeichen aus dem zirkulären Buffer nimmt und auf ein Ausgabegerät ausgibt.

Erläuterung der Besipieles:
Print und Read sind vorausgesetzte Prozeduren, die auf den input und output device operieren. *Consumer* und *Producer* sind zwei Prozesse, welche mittels dem *interface module*, *Bufferhandling*, und deren Prozeduren *Put* und *Get,* durch die Signale *Nonempty* und *Nonfull* in Wechselwirkung stehen.

4)Da Modula nie in der Praxis eingesetzt wurde, weil das Konzept bis 1978/79 noch so oft verändert wurde, bis man sich anscheinend entschloss, Modula 2 erscheinen zu lasssen, kann man nur schwer die Vorteile und Nachteile abschätzen.
Zusammenfassend gesagt bietet Modula ein Konzept einer Sprache mit sehr gut lesbaren Syntax und mit einigen sehr interessanten Konzepten, wie zum Beispiel die *multiprogramming facilities*, die einen einführenden Einblick in die Techniken des Multitasking bietet, oder der Modularenstruktur die als einer der Vorläufer der objektorientierten Programmierung gelten kann

```
module Listinput;
        use Read, Print;
        interface module Bufferhandling;
                define Get, Put;
                const Nmax=256;
                var     N, In, Out: integer;
                        Nonempty, Nonfull: signal;
                        Buf: array 1:Nmax of char;

                (*Insert a character into the circular Buffer*)

        procedure Put (Ch: char);
                begin
                        If N=Nmax then wait(Nonfull) end;
                        Inc(N); (*N:=N+1, increment*)
                        Buf[In]:=Ch;
                        In:= (In mod Nmax)+1;
                        Send(Nonempty)
                end Put;
                (*Removes charcter from the circular buffer*)

        procedure Get(var Ch:char);
                begin
                        If N=0 then wait(Nonempty) end;
                        Dec(N);
                        Ch:=Buffer[out];
                        Out:= (Out mod Nmax)+1;
                        Send(Nonfull);
                end Get;
        begin
                N:=0;
                In:=1;
                Out:=1;
        end Bufferhandling;

        (*Read input characters and store in buffer*)
        process Producer;
                use Read,Put;
                var Ch:char;
        Begin
                        Loop
                                Read(ch);
                                Put(ch);
                        End;
        end Producer;

        (*Remove characters from Buffer and print*)
        process Consumer;
                uses Get,Print;
                var Ch:Char;
        begin
                loop
                        Get(Ch);
                        Print(Ch);
                end;
        end Consumer;
begin
        Producer;
        Consumer;
end Listinput;
```

Bsp ist aus der Arbeit "Euclid and Modula " von David Elliott, David Thompson und David Barnard; Department of Computer Science University of Toronto

Quellen

Wirth, N., Modula: A programming language for multiprogramming, Software - Practice and Experience 7,3-35;
Wirth, N., Modula: The use of modula, Software - Practice and Experience 7,37-65;
Wirth, N., Modula:Design and Implementation of Modula, Software - Practice and Experience 7,67-84;

http://www.computerbase.de/lexikon/Modula-2
http://www.modulaware.com/mdlt52.htm
"Euclid and Modula " von David Elliott, David Thompson und David Barnard; Department of Computer Science University of Toronto; http://portal.acm.org/citation.cfm?id=954380