Pascal, Ada and Modula-2
A System Programmer's Comparison

eine Ausarbeitung für die Technische Universität Wien von Peter Calvache (Matrikelnummer 0425140)

❍ ❍ ❍

Pascal, Ada and Modula-2 A System Programmer's Comparison 1
Abstract 1
1 - Systemprogrammierung 2
2 - Die Programmiersprache Pascal 2
3 - Ziel dieser Arbeit 2
4 - Pascal 2
4.1 - Strong Typing und Arrays 2
4.1.1 - Array-Größe als Teil des Datentyps 2
4.2 - Separate Funktionalität 3
4.3 - Evaluierungsreihenfolge bei Ausdrücken 3
4.4 – Limitationen der Sprachkonstrukts 3
4.4.1 – Break, Continue und Return 3
4.4.2 – Implementierung des Break 3
4.4.3 – Case 4
4.5 – Keine Command-Line-Parameter 4
4.6 – Probleme bei Speicher-Überlauf 4
4.7 – Keine Bitmanipulation 4
4.8 – Kein Präprozessor 4
5 – Ada 5
5.1 – Laufzeitsicherheit 5
5.2 – Exception Handling 5
5.3 – Bitmanipulation 5
5.4 – Ausbrechen aus Schleifen 5
5.5 – Repariertes Case 6
5.6 – Type Casts 6
5.7 – Generics 6
5.8 – In- und Out-Parameter 6
6 – Modula-2 6
6.1 – Separate Kompillierung 6
6.2 – Case Sensitivity 7
6.3 – Exit und Return 7
6.4 – Zeigerunterstützung 7
6.5 – Type Casts 7
6.6 – Problemlösung bei Char-Arrays 7
6.7 – Evaluierungsreihenfolge in Ausdrücken 7
7 – Schlussfolgerung 7
8 – Literatur 7

❍ ❍ ❍

Abstract

Ein Systemprogrammierer beschäftigt sich mit maschinennaher Programmierung (oft an Hardware-Schnittstellen) und unterliegt somit einer etwas anderen Arbeitsweise als ein Anwendungsprogrammierer. Somit legt der Systemprogrammierer großen Wert auf direkte Speichermanipulation, flexible Sprachkonstrukte und die Freiheit, den Rechner uneingeschränkt von den Möglichkeiten der Sprache zu steuern. In dieser Arbeit werden die Programmiersprachen Pascal, Ada und Modula-2 unter Beachtung dieser Kriterien analysiert und verglichen.

1 - Systemprogrammierung

Im Gegensatz zur Anwendungsprogrammierung beschäftigt sich Systemprogrammierung mit Low-Level entwickeln von hardwarenahen Projekten wie Betriebssystemen oder Controller-Software. Die ersten Systemprogrammierer entwickelten Software in der entsprechenden Assembler-Variante ihres Zielprozessors. Inzwischen haben sich Compiler so weit entwickelt, dass man für Systemprogrammierung fast ausschließlich Sprachen der 2. Generation (eine Abstraktionsstufe über Assembler) wie C oder Pascal verwendet. Während überwiegende Teile von Anwendungsprogrammen plattformunabhängig realisiert werden, kann man bei Systemprogrammierung davon ausgehen, dass die entwickelte Software für einen einzigen Prozessor mit definierten periphären Elementen zugeschnitten ist.

2 - Die Programmiersprache Pascal

Basierend auf Algol-Syntax wurde Pascal im Jahre 1970 vom schweizer Informatiker Niklaus Wirth entwickelt, und hatte den Zweck, das Erlernen strukturierter Programmierung für Studenten zu erleichtern. Da diese Programmiersprache ursprünglich nur auf Schulen und Universitäten verwendet werden sollte, konnte sie wegen zahlreichen Restriktionen (die im Verlauf dieser Ausarbeitung näher beschrieben werden) nur zu einfachen Zwecken eingesetzt werden. Mit der Zeit wuchs Pascal's Popularität und unabhängige Entwicklergruppen begannen, ihre eigene Versionen von Pascal zu implementieren, um die Limitationen der Sprache mit eigenen Lösungen zu umgehen. Es resultierten mehrere unterschiedliche Versionen von Pascal (unter anderem auch das populäre Turbo Pascal mit OOP-Unterstützung) sowie mehrere Sprachen, die auf Pascal-Syntax aufbauen (wie z.B. Ada, Modula-2 und neuerdings Delphi).

3 - Ziel dieser Arbeit

Ich werde die Sprachen Pascal, Ada sowie Modula-2 aus der Sicht eines Systemprogrammierers analysieren und gegenüberstellen. Dabei werde ich zuerst die „Basissprache" Pascal genauestens behandeln und anschließend die Vorteile der abgeleiteten Sprachen Ada und Modula-2 erörtern.

4 - Pascal

Dieses Kapitel bezieht sich (da von meinem Auftraggeber nicht anders spezifiziert) auf das „ursprüngliche" im Jahre 1970 definierte Pascal. Es sei vermerkt, dass jegliche Limitationen, die ich hier beschreiben werde, seitdem in neueren, unabhängigen Versionen (Turbo Pascal, Free Pascal, P4) eliminiert wurden. Seit Version 5 unterstützt Turbo Pascal (im Moment bei Version 7.0) unter anderem auch objektorientierte Programmierung und kann somit als Alternative zu C++ für generelle Softwareentwicklung verwendet werden.

4.1 - Strong Typing und Arrays

Pascal ist eine „strongly typed Language". Dies bedeutet, dass jede Variable an jedem Punkt im Programm einen vordefinierten Datentyp besitzen muss. Diese Tatsache allein ist nicht ungewöhnlich (und meitens sogar erwünscht), jedoch bringt Pascal's strong Typing einen großen Nachteil mit sich: Es ist unmöglich, mit Hilfe eines Casts eine Variable zu einem andren (kompatiblen) Datentyp umzuwandeln. Dies hat verheerende folgen für Systemprogrammierer: Die einfache Programmierung eines generalisierten Heap Allocators ist nicht möglich. Wenn ein Pascal-Systemprogrammierer ein Speicherverwaltungssystem implementieren will, müssen ihm alle Datentypen, die mit dem Endprodukt zusammenarbeiten sollen, bekannt sein. Ausserdem muss für jeden Datentyp eine separate Version des Systems entwickelt werden.

4.1.1 - Array-Größe als Teil des Datentyps

Pascal enthält einen weitaus schlimmeren Designfehler, der von vielen als Pascal's schlimmstes „Verbrechen" gegen Programmierer gesehen wird. Wenn man ein statisches Array definiert, ist die Größe des Arrays ein Teil des Datentyps. Wenn man versucht, eine Funktion zu entwickeln, die ein Array als Parameter einliest, wird diese Limitation zum Horror. Es ist in Pascal unmöglich, eine Funktion zu schreiben, die ein Array beliebiger Größe akzeptiert. In der Praxis bedeutet dies, dass sich nicht einmal eine generalisierte Int-Sortierfunktion programmieren lässt.

```
var
        a : array [ 1 .. 6 ] of char;
        b : array [ 1 .. 7 ] of char;
```

Obwohl Array „b" nur ein Character länger ist als „a", besitzt es einen anderen Datentyp und kann nur von einer Funktion akzeptiert werden, die explizit ein Char-Array der Größe 7 akzeptiert. Zumindest in der ursprünglichen Version von Pascal ist dieses Problem unumgänglich. Der Pascal-Programmierer kann sich Helfen, in dem er z.B. jedes char-Array mit Leerzeichen auf eine vordefinierte Länge ausgleicht:

```
„Ein String.                 "
„Ein zweiter String.         "
„Und noch ein weiterer String."
```

Diese Methode umständlich zu nennen wäre eine Untertreibung. Weiters besitzen statische char-Arrays in Pascal keine Terminierung mit einem Null-Character, folglich lässt sich das Ende eines Strings nicht bestimmen. Die einzige Abhilfe ist eine Pascal-Funktion zur Längenbestimmung eines char-Arrays, aber diese zu verwenden wäre für einen C-Programmierer nur

selten der schnellste Weg. Der eingebaute „string"-Datentyp löst diese Probleme zumindest für char-Arrays, doch für andere Datentypen bleiben sie erhalten.

4.2 - Separate Funktionalität

Pascal unterstützt weder statische Variablen, noch die Möglichkeit einer Initialisierung. Würde man in C eine Aufgabe folgendermaßen lösen ...

```
void f()
{
        static int i = 0;

        i++;
}

int main()
{
        f();
}
```

... wäre dasselbe Beispiel in Pascal wesentlich aufwendiger:

```
program p;
var
        i : int;

procedure f;
begin
        i := i + 1;
end;

begin
        i := 0;

        f;
end;
```

Die Funktionalität der Variable „i" ist über 3 Teile des Programms aufgeteilt. Was im oberen Beispiel noch relativ übersichtlich wirken mag, wird bei größeren Programmen eher nervtötend, besonders wenn zwischen den einzelnen Referenzen zu „i" jeweils mehrere hundert Zeilen dazwischenliegen. Die Definition von „i" im global Scope bringt auch potentielle Namenskonflikte mit sich. Im Gegensatz dazu wird in C jegliche mit „i" zusammenhängende Funktionalität perfekt zusammengehalten. Man sieht auch, dass es in Pascal nicht möglich ist, paramterlose Prozedur-Aufrufe von Variablen zu Unterscheiden. Wenn man also unbekannten Programmcode analysiert, kann man auf den ersten Blick nicht feststellen, ob „f" eine Variable oder eine Prozedur ist.

4.3 - Evaluierungsreihenfolge bei Ausdrücken

Ein häufig auftretender Fall in der Programmierung ist der Zugriff auf Array-Elemente in einer Schleife bei „gleichzeitiger" Überprüfung, ob der aktuelle Index zulässig ist. In C werden die Terme verknüpfter boolescher Ausdrücke von links nach rechts evaluiert. Man kann somit schreiben ...

```
while( i < arraySize && a[ i ] < 256 )
```

... ohne zu befürchten, dass man einen unzulässigen Speicherbereich einliest. In Pascal ...

```
while( i < arraySize ) and ( a[ i ] < 256 ) do
```

... ist dies nicht ratsam, da die Terme in undefinierter Reihenfolge ausgewertet werden.

4.4 – Limitationen der Sprachkonstrukts

Da Systemprogrammierer viel häufiger mit unübersichtlichen Algorithmen zu tun haben als die meisten anderen Programmierer, sollten Sprachkonstrukts dem Entwickler mit großer Flexibilität assistieren. Unglücklicherweise gibt es auch in diesem Punkt ein paar gravierende Designfehler im Ur-Pascal.

4.4.1 – Break, Continue und Return

Aus Schleifen und Funktionen kann man mit den „gewohnten" Methoden nicht ausbrechen. Tatsächlich gibt es weder „break", „continue", noch „return". Funktionsrückgabe wird realisiert, in dem eine pseudo-Variable, die den Funktionsnamen trägt, gesetzt wird (a la BASIC), aber nach einer solchen Operation erfolgt kein Ausstieg aus der Funktion.

4.4.2 – Implementierung des Break

Es ist natürlich möglich, aus einer Schleife auszusteigen, allerdings muss man dies mittels einer boolschen Variable mühselig „manuell" implementieren. Während man in C schreiben könnte ...

```
while( i < arraySize )
{
        if( a[ i ] > 255 ) break;

        counter += a[ i ];
}
```

... würde das Pascal-Äquivalent folgendermaßen aussehen:

```
breakLoop := false;

while( i < arraySize ) and ( breakLoop = false ) do
        if( a[ i ] > 255 ) then
                breakLoop := true
        else
                counter := counter + a[ i ]
        end
end
```

Zweifelsohne ist die C-Variante leichter lesbar. Der Pascal-Nachteil wird zusätzlich problematischer, wenn der zu implementierende Algorithmus ohnehin bereits dutzende If-s und For-s inenander schachteln muss.

4.4.3 – Case

Das Case-Statement ist in Pascal praktisch nutzlos, da das Verhalten undefiniert ist, wenn ein Zustand eintritt, der nicht explizit definiert wurde. Wenn man also 10 ausgewählte States einer 0...255 Int-Variable behandeln möchte, muss man 246 leere States definieren, sonst ist das Programm unberechenbar. Ein „default"-Case gibt es nicht.

4.5 – Keine Command-Line-Parameter

Es gibt keine Möglichkeit, im Ur-Pascal Command Line Parameter einzulesen. Dies ist für Systemprogrammierer kritisch, da System-Tools, die häufig durch die Command Line gesteuert werden, nicht realisierbar sind.

4.6 – Probleme bei Speicher-Überlauf

In modernen PCs eine Seltenheit, war der Stack-Overflow zu Pascal's anfänglichen Zeiten eine große Gefahr, wegen der man seine Programme mühselig optimieren musste. Desshalb ist es auch überraschend, dass Pascal keinen Recovery-Mechanismus für Stack Overflows enthält. Wird mehr Speicher angefordert als vorhanden, verhält sich das Programm undefiniert.

4.7 – Keine Bitmanipulation

Man kann davon ausgehen, dass der Mikrokontroller-Programmierer den Großteil seiner Arbeit damit verbringt, Bits zu verschieben, setzen, löschen und zu invertieren. Auch dies gehört zum Bereich der Systemprogrammierung, somit wird Bitmanipulation zum kritischen Punkt in der Wahl einer Programmiersprache. Umso enttäuschender ist es, dass Pascal keine Bitmanipulationsoperatoren unterstützt. Während man bitwise Shifts durch einfache Arithmetik realisieren kann ...

```
byte := byte * 2         { left shift }
byte := byte / 2         { right shift }
```

... wird es bei booleschen Operationen komplizierter. Weder AND, noch OR, noch XOR entsprechen einer arithmetischen Operation und können daher nur mit großem Programmieraufwand (sowie mit relativ viel Rechenaufwand) realisiert werden.

4.8 – Kein Präprozessor

Schliesslich kommen zu einem der wichtigsten Gründe, warum Pascal für große Projekte unbrauchbar ist: Dem Fehlen eines Präpprozessors. Selbst das einfache Inkludieren separater Dateien ist unmöglich und die Sprache für Teamarbeit unbrauchbar.
Die wahrscheinlich häufigste Verwendung von #define unter C und C++ ist die Definition einer Makrokonstante. Glücklicherweise enthält Pascal ein ähnliches Feature: Die const-Definition akzeptiert leider keine Argumente (für Ausdrücke).

```
program p;
const
        arraySize = 10; { Konstanten haben keinen Datentyp! }
var
        a : array [ 1 .. arraySize ] of int;

procedure f;
var
        i;
begin
        i := 0;

        while( i < arraySize ) do
                { … }

                i := i + 1
        end
end;
```

5 – Ada

In den frühen 70er Jahren zeigte sich das US Verteidigungsministerium besorgt von der großen Anzahl an firmeneigenen Programmiersprachen, auf denen die Software ihrer Rechner entwickelt wurde. Um diesem Problem gegenüberzutreten, wurde die Higher Order Language Working Group erstellt (HOLWG), und damit beauftragt, eine eigene Programmiersprache zu entwickeln. Das Resultat ihrer Anstrengungen war die im Jahre 1979 veröffentlichte auf Pascal-Syntax basierende Sprache „Ada“ und fand überwiegend auf embedded und real-time Systems Anwendung. Im Jahre 1995 wurde Ada 95 veröffentlicht und erweiterte Ada-Programmierung um objektorientierte Konzepte.
Ada ist noch heute sehr populär und wird wegen ihrer Verlässlichkeit in Flugzeugen und Space-Shuttles verwendet.

5.1 – Laufzeitsicherheit

Im Gegensatz zu Pascal (und C, und C++) eliminiert Ada eines der häufigsten Probleme in der Programmierung: Die illegale Überschreitung von Array-Grenzen durch einen falschen Index. Jeder indexbasierende Zugriff wird auf Gültigkeit überprüft, was einen geringen Performance-Overhead mit sich bringt. Da diese Methode zusätzlich vor Buffer Overruns schützt, ist Ada zumindest aus dieser Hinsicht vor Hackern sicher, da man Programme nicht dazu bringen kann, illegalen Code auszuführen.

5.2 – Exception Handling

Ausnahmenbehandlung trägt massiv zur Verlässlichkeit der Ada-Programmiersprache bei. Wie in den meisten modernen Programmiersprachen kann man eine Exception manuell auslösen (in dem Man die „raise“-Anweisung ausführt) oder eine automatische Ausnahme (wie z.B. bei einer Division durch Null oder bei einem OutOfMemory-Fehler) abfangen.

5.3 – Bitmanipulation

Einer der größten Pascal-Kritikpunkte war die nicht eingebaute Bitmanipulation. Glücklicherweise wurde diese Limitaiton behoben und ausserdem ein mächtiges, für Systemprogrammierer sehr praktisches Konzept der Zahlendarstellung eingeführt.

```
2#1010_0101#     { Basis 2 }
16#AB#           { Basis 16 }
016#0BC#         { Basis 16 }
8#123#           { Basis 8 }
```

Wie nur in wenigen anderen Programmiersprachen kann man Bitmasken direkt definieren und muss sie nicht auf eine Dezimalzahl umrechnen. Auch hexadezimale und oktale Zahlen können mit einem übersichtlichen Syntax problemlos angegeben werden. Für Controller-Programmierer ist dies ein geradezu göttlicher Vorteil.
In Pascal waren binäre Operatoren ebenfalls stark vermisst, die aber bei Ada komplett vorhanden sind.

```
a := b and c
a := b or not c
a := b xor 2#0000_1000#
```

Wie man erkennen kann, ist die Negation (not) sehr einfach und das übersichtliche Invertieren selektiver Bits ein Kinderspiel.

5.4 – Ausbrechen aus Schleifen

Die Abwesenheit einer „break“-Anweisung war in Pascal ein großes Problem, doch Ada definiert das „exit“-Keyword, das zusätzlich zur „break“-Funktionalität einige interessante Eigenschaften aufweist.

```
exit LoopName when Condition;
```

In Ada ist es möglich, Schleifen (im Allgemeinen Code-Blöcke) zu benennen. Wenn sich das Programm in verschachtelten Schleifen befindet, kann man mit Hilfe des optionalen LoopNames genau festlegen, aus welcher Schleife das Programm aussteigen soll. Die optionale „when“-Kondition verknüpft exit mit einer if-Anweisung und macht den Code leichter lesbar.

```
MyLoop1:
loop
        MyLoop2:
        loop
                …
                exit MyLoop1 when i < 50;
                exit MyLoop2 when i < 100
        end loop MyLoop2
end loop MyLoop1
```

Es ist erkenntlich, dass durch die Flexibilität der exit-Anweisung mehrere häufig auftretende Probleme gelöst werden können, die in anderen Sprachen das Einführen einer booleschen Variable und/oder eines if-Statements verlangen würden.

5.5 – Repariertes Case

Die Case-Anweisung in Ada wird bei einem undefinierten Fall ignoriert (und nicht undefiniert behandelt, wie bei Pascal). Der optionale „other"-Zustand entspricht „default" in C und tritt ein, falls kein definierter Zustand auf die Variable zutrifft.

```
case i is:
        when 1 => …
        when 2 => …
        when 3 => …
        when others => …
end case;
```

5.6 – Type Casts

Sprachen wie C unterstützen implizierte Datentypkonvertierung zwischen numerischen Typen wie int, float und double.

```
int i = 2;
float f = 3;

i = f / i;
f = i / f;
```

Beide folgenden Anweisungen wären in C zulässig. Da implizite Casts bei komplexeren Ausdrücken zu Problemen führen können, wurde diese Art der Typkonvertierung nicht in Ada implementiert. Der Type Cast muss explizit erfolgen:

```
i := INT( f ) / i
f := FLOAT( i ) / f
```

Der „einfache" explizite Cast ist nur zwischen Typen von verwandtem Aufbau möglich. Wenn man die Runtime-Überprüfung eines Casts umgehen will, muss man die „unchecked_conversion" Funktion verwenden.

5.7 – Generics

Generics erlauben das Definieren eines Typenunabhängigen („parametrisierten") Datentyps. Obwohl man die meisten Generics-Problemstellungen auch traditionell lösen kann, ist eine generische Variante wesentlich entwicklerfreundlicher und sicherer, da man z.B. nicht auf „unchecked_conversion" zurückgreifen muss.

```
generic
        type T is private;

package Stack is
        procedure
                Push( data : in T );
        procedure
                Pop( data: out T );
end Stack;
```

Die oberen Stack-Funktionen (falls implementiert) wären kompatibel mit jedem Datentyp, inklusive benutzerdefinierter Typen.

5.8 – In- und Out-Parameter

Ada unterstützt aus Gründen der Laufzeitsicherheit keine Zeiger, bietet aber einen Mechanismus für Parameterübergabe, der bei vielen Zeiger- Problemstellungen einsetzbar ist. Es gibt vier Arten von Argumenten:

```
function f( a: int, b : in int, c : out int, d : in out int )
```

Parameter „a" ist eine gewöhnliche Parametervariable und entspricht einer Kopie der vom Caller übergebenen Variable.
Parameter „b" ist eine in-Variable und darf nur gelesen, aber nicht verändert werden.
Parameter „c" ist eine out-Variable und darf nur verändert, aber nicht gelesen werden.
Parameter „d" ist eine in-out-Variable und darf sowohl gelesen als auch verändert werden.
Der Unterschied zwischen „a" und „d"? Die letzten drei Parameter werden als Referenz übergeben und sind, im Unterschied zu „a", keine Kopien der vom Caller übergebenen Variable. Damit hätte Ada einen großen Teil der Zeiger-Verwendung abgedeckt.

6 – Modula-2

Nach Pascals Erfolg wurde die erste Modulo-Version von Pascal-Vater Niklaus Wirth definiert (um Pascals Limitationen zu lösen), allerdings nie implementiert. Die zweite Version (Modula-2) wurde sehr populär. Die meisten Kritikpunkte von Pascal konnten erfolgreich beseitigt werden um eine Sprache zu schaffen, die in der professionellen Programmierung einsetzbar ist.

6.1 – Separate Kompillierung

In Pascal musste der gesamte Programmcode in einer Datei residieren, was Teamarbeit praktisch unmöglich machte. Modula-2 definierte das Konzept von mehreren, parallel kompillierten Modulen (Daraus ergab sich der Name der Sprache). Der Programmcode eines Moduls wurde in ein Definitionsmodul (ähnlich einer Header-File) und ein Implementationsmodul

(ähnlich einer Source-File) aufgeteilt. Im Definitionsmodul wurde der strukturelle Aufbau von Typen, Paketen und Prozeduren angegeben und im Implementationsmodul implementiert. Die Definitionsmodule wurden verwendet, um parallel kompillierte Module mit Informationen über die gegenseitigen Softwareschnittstellen zu versorgen.

6.2 – Case Sensitivity

Sämliche Schlüsselwörter in Modula-2 sind case sensitive und müssen mit Großbuchstaben angegeben sein. Dies war eine fragliche Design-Entscheidung, da viele Programmierer zweifelsohne lowercase-Keywords bevorzugten.

6.3 – Exit und Return

Mit „exit" kann man aus der unmittelbar umschließenden Schleife aussteigen, jedoch ist ein Sprung zu beliebigen Schleifenenden (wie in Ada) nicht möglich. Dafür kann man mit Hilfe von „return" die momentane Funktion beenden.

6.4 – Zeigerunterstützung

Ein riesiger Vorteil für Systemprogrammierer ist die Unterstützung von Zeigern, die es weder im Ur-Pascal noch im Ada gibt.

```
pi : POINTER TO INTEGER;
```

Die obere Zeile definiert einen Int-Zeigers in Modula-2.

6.5 – Type Casts

Die Umwandlung eines Datentypen ist in Modula-2 über zwei Mechanismen realisierbar: Conversion Functions und Type Transfer. Beim Ersten wird (ähnlich eines expliziten Ada-Casts) der Wert der Variable neu berechnet. Aus einem int-Wert wird ein numerisch identischer float-Wert, der aber einer anderen binären Representation entspricht. Der Type Transfer entspricht einer „unchecked_conversion" in Ada und ist als bitweises Kopieren einer Variable einen kompatiblen Speicher zu verstehen.

6.6 – Problemlösung bei Char-Arrays

Wie bei Ada sind (auch inline-definitierte) char-Arrays mit einem Null-Character am Ende versehen, sodass Algorithmen das Ende einer Zeichenfolge bestimmen können. Zum Umschließen der inline-Definition kann man sowohl ´ als auch `` verwenden, um das jeweils andere Anführungszeichen in der Zeichenfolge verwenden zu können. Somit wären folgende Beispiele legal:

```
„Die ´char´ Variable ist 1 Byte gross."
´Die "char" Variable ist 1 Byte gross.´
```

6.7 – Evaluierungsreihenfolge in Ausdrücken

Mit booleschen Operatoren verknüpfte Ausdrücke werden in Modula-2 von links nach rechts evaluiert.

7 – Schlussfolgerung

Aus meiner Ausarbeitung resultiert zweifelsohne, dass Ada die mächtigste der drei Sprachen ist. Sie unterstützt heutzutage bei der Entwicklung großer Projekte unverzichtbare Features wie Generics und Objektorientierte Programmierung. Die durch zahlreiche Laufzeit-Überprüfungen resultierende Verlässlichkeit der Sprache macht sie bei kritischen Aufgaben zur einzigen Lösung. Der Vorteil von Modula-2 ist die Unterstützung von Zeigern, die wiederum für viele Systemprogrammierer von sehr hoher Bedeutung ist. Ansonsten ist Modula-2 eine empfehlenswerte Alternative für prozedural entwickelte Projekte. Pascal, die Basissprache beider Technologien, ist in der ursprünglichen Form nutzlos und kann nur zu experimentellen Zwecken empfohlen werden. Sämtliche direkten „Ableitungen" dieser Sprache wie Turbo Pascal oder Delphi sind jedoch sogar mit Ada konkurrenzfähig und auch für große Projekte eine sehr gute Alternative zu vergleichbaren Technologien.

8 – Literatur

- **Programmieren mit Ada und C: Eine beispielorientierte Gegenüberstellung**
  Annette Weinert – Braunschweig, Wiesbaden – Vieweg, 1992
  ISBN 3-528-05240-6

- **TopSpeed Modula-2: Erweiterte Modula-2 Version für den IBM-PC**
  Jan Bielecki – Heidelberg – Hüthig, 1992
  ISBN 3-7785-1967-0