Seminar Report

Turing

Mathias Hörtnagl
mathias.hoertnagl@student.uibk.ac.at

17 February 2013

Supervisor: Simon Legner, BSc

Abstract

Turing was originally designed in 1982 to serve as a new teaching language at the University of Toronto. This Pascal-like language was ment to be appealing to the beginner as well as the professional programmer. Today considered a legacy language, Turing was amongst the most frequently chosen languages for computer science courses in highschools of Ontario.
This seminar report touches some of the most prominent features and aspects of Turing and its history.

Inhaltsverzeichnis

1 Einführung **1**
- 1.1 Zielsetzungen der neuen Sprache . . . 1
- 1.2 Anwendungsbereiche und Verbreitung . . . 1
- 1.3 Aufbau der Arbeit . . . 1

2 Turing-Implementierungen und -Derivate **2**
- 2.1 Holtsoft Turing . . . 2
  - 2.1.1 Turing Plus oder System Turing . . . 2
  - 2.1.2 Numerical Turing . . . 2
  - 2.1.3 Objekt-orientiertes Turing . . . 2
- 2.2 OpenT . . . 3
- 2.3 Open Turing . . . 3
- 2.4 Vergleich mit Pascal . . . 3

3 Elementare Sprachkonzepte **4**
- 3.1 Hallo Welt . . . 4
- 3.2 Primitive Datentypen . . . 5
- 3.3 Programmflussstrukturen . . . 5
  - 3.3.1 Schleifen . . . 5
  - 3.3.2 if-Verzweigungen . . . 6
  - 3.3.3 case-Verzweigungen . . . 7
- 3.4 Unterprogramme . . . 7
  - 3.4.1 Parameterlose Subroutinen . . . 8
  - 3.4.2 Aufruf vor Deklaration . . . 8
- 3.5 Strings . . . 9
- 3.6 Types und Records . . . 10
- 3.7 Arrays . . . 11
- 3.8 Module . . . 11
- 3.9 Sound, Grafiken und GUIs . . . 12

4 Systemprogrammierung **13**
- 4.1 Zeiger . . . 13
- 4.2 Prozesse . . . 13

5 Objekt-orientierte Programmierung **14**
- 5.1 Klassendefinitionen . . . 14
- 5.2 Objekterzeugung . . . 14
- 5.3 Zugriff auf Elemente eines Objekts . . . 15
- 5.4 Vererbung . . . 15
  - 5.4.1 Die Wurzelklasse anyclass . . . 16
  - 5.4.2 Klassenidentifikation zur Laufzeit . . . 16
- 5.5 Objektzerstörung . . . 17

6 Zusammenfassung und Ausblick **17**

Literaturverzeichnis **18**

1 Einführung

`Turing` entstand im Jahre 1982 an der Universität Toronto aus dem Versuch heraus, die dazumal verwendete, aber nicht mehr zeitgemäße, Programmiersprache `PL/1` im universitären Unterricht zu ersetzen.
`Pascal` oder die ebenfalls an der Universität entwickelte Programmiersprache `Euclid` sollten `PL/1` ursprünglich ersetzen. `Euclid` stellte sich aber als zu komplex für den Programmierneuling dar und `Pascal`, obwohl leicht zu erlernen, wies einige Schwächen auf, die die Universität dazu veranlassten eine neue Programmiersprache zu entwerfen [1].

1.1 Zielsetzungen der neuen Sprache

Beim Design der Sprache orientierten sich die beiden Entwickler Ric Holt und James Cordy stark an Pascal und Euclid, versuchten aber eine einfachere Syntax zu erzielen, die es Einsteigern ermöglichen sollte, `Turing` schnell und effizient zu erlernen. [6].
Die neue Sprache sollte aber auch für professionelle Entwickler nutzbar sein. Um den Anforderungen aktueller Programmiersprachen gerecht werden zu können, unterstützte `Turing` unter anderem dynamische Strings und Arrays und die Organisation großer Projekte in Modulen. Mit den Erweiterungen zu System Turing und objekt-orientiertem Turing kamen die Aspekte der systemnahen, parallelen und der objekt-orientierten Programmierung hinzu [2].

1.2 Anwendungsbereiche und Verbreitung

Besonders häufig fand `Turing` aber Anwendung in universitären Kursen nicht nur als Einführungssprache in die grundlegenden Konzepte, sondern auch in Kursen aus den Gebieten der Betriebssysteme, Compilerentwicklung und Systemprogrammierung.
Außerhalb der Universität fand `Turing` breiten Anklang an den Highschools des Bundesstaates Ontario, Kanada, an der sie auch heute noch zuweilen im Informatikunterricht angewendet wird [1].

1.3 Aufbau der Arbeit

Im nächsten Kapitel werden die wichtigsten Erweiterungen und Entwicklungen in der Geschichte von `Turing` kurz Erwähnung finden.
Kapitel 3 widmet sich der heute gültigen elementaren Syntax. Neben Datentypen, Kontrollstrukturen und Subroutinen, geht diese Kapitel besonders auf Strings und Arrays ein.
Prozesse und Zeiger finden in Kapitel 4 Erwähnung und Kapitel 5 widmet sich vollends den objekt-orientierten Mechanismen, die `Turing` zur Verfügung stellt.
Das letzte Kapitel schließt die Diskussion mit einigen zusammenfasssenden Bemerkungen.

2 Turing-Implementierungen und -Derivate

Seit seiner erstmaligen Entwicklung in den frühen 1980er Jahren unterwarf sich `Turing` einiger Neuerungen und Erweiterungen, die die Sprache weiterhin attraktiv für Anfänger wie Professionelle gestalten sollten. Besonders wichtig waren dabei die Erweiterung hin zu einer Systemprogrammiersprache um hardwarenahe Lösungen zu entwickeln, und der Schritt in Richtung einer objektorientierten Sprache um Entwicklungen großer Anwendungen übersichtlicher zu halten.

Neben diesen Erweiterungen kam es auch zur Implementierung offener Compiler, während und besonders nachdem der Quellcode des ursprüngliche `Turing` 2007 veröffentlicht und die Weiterentwicklung von Seiten der Firma Holtsoft eingestellt wurden [3].

2.1 Holtsoft Turing

Holtsoft Turing war die proprietäre Implementierung der `Turing`-Programmiersprache. Die Fortführung, der an der Universität Toronto begonnenen Arbeit, geschah in dem von Dr. Ric Holt gegründeten Unternehmen *Holtsoft*, das auch die kommerzielle Verbreitung des Compilers vorantrieb.
Einer der Kunden war der Bundesstaat Ontario, der `Turing` an allen Schulen im Informatikunterricht einsetzte.

2.1.1 Turing Plus oder System Turing

Mit *System Turing* (*Turing Plus*) betrat `Turing` die Bühne der systemnahen Programmiersprachen und betrachtete sich als Alternative zu bereits etablierten Sprachen wie C oder Ada.
Zu den neuen Features zählten unter anderem Zeiger und entsprechende Zeigerarithmetik, vorzeichenlose Zahlen unterschiedlicher Größen, explizite Typkonvertierungen, Prozesserzeugung und -synchronisation sowie Interruptbehandlungsmechanismen [1]. Kapitel 4 stellt einige der Konzepte näher vor.

2.1.2 Numerical Turing

Numerical Turing war eine experimentelle Erweiterung der `Turing`-Programmiersprache um numerisch-wissenschaftliche Berechnungen bequemer implementieren zu können [5]. Die Erweiterungen wurden allerdings nie Teil der `Turing`- Spezifikation und werden von heutigen Compilern nicht unterstützt.

2.1.3 Objekt-orientiertes Turing

Im Jahr 1992 gelang `Turing` der Schritt hin zu einer objekt-orientierten Sprache. System Turing erhielt die Fähigkeit Klassen zu definieren, Objekte zu erzeugen und Mittel der Vererbung [6].

In Kapitel 5 werden die Möglichkeiten der objekt-orientierten Programmierung detailliert vorgestellt.

2.2 OpenT

OpenT ist eine offene Implementierung mit annähernd vollständiger Kompatibilität zu objekt-orientiertem Turing. Die letzte Aktivität im OpenT-Repository datiert zurück auf den 1. August 2009 [8].

2.3 Open Turing

Open Turing beruht auf dem von Holtsoft freigegebenen Sourcecode der letzten unveröffentlichten Version des Holtsoft-Turing- Compilers. Der offene Compiler bietet vollständige Kompatibilität zur `Turing` Spezifikation und kann mit neuen Erweiterungen, wie Unterstützung für 3D-Grafik und einer nativen HashMap-Implementierung aufwarten. `Turing`-Code soll im Vergleich zur originalen Implementierung auch bis zu 50 Prozent schneller ausgeführt werden [7].
Das Projekt steht unter der MIT Lizenz und kann von `github`[1] bezogen werden.

2.4 Vergleich mit Pascal

`Turing` sollte als eine Weiterentwicklung von `Pascal` verstanden werden [9]. Hier sollen einige syntaktische Verbesserungen durch `Turing` aufgezeigt werden. Listing 1 zeigt ein einfaches Programm in `Pascal`.

```
program test (output); {redundant header, semi-colon}
  var n,m integer;     {grouped dcl'ns at top of program}
begin                  {redundant begin}
  if n > 0 then        {no semi-colon, no un-init. check}
    if m < 0 then
      writeln('n>0, m<0'); {semi-colon changes meaning}
    else                   {applies to outer-if, NOT inner}
      writeln('n&m>0');    {wrong!!!, semi-colon redundant}
end.                   {period, NOT semi-colon}
```

Listing 1: Ein einfaches Programm in `Pascal` [9].

Listing 2 implementiert das selbe Programm in `Turing`. Die Syntax in `Turing` verzichtet vollständig auf Semikolons. Auch wurden einige Strukturen, wie der **program**-Kopf und der **begin** ... **end**-Block entfernt. Vereinfacht wurden damit auch Nuancen in der Syntax, wie der Punkt am Ende eines `Pascal` Programmes oder die Regeln zur korrekten Verwendung der Semikolons.

Variablen können in `Turing` an jeder beliebigen Stelle im Quellcode deklariert werden. In `Pascal` stehenn alle Variablendeklarationen im Programmkopf. Die **if**-Struktur wird mit einem **end if** abgeschlossen und verhindert so Fehler bei verschachtelten **if**-Blöcken, wie sie in Listing 1 auftreten. Der **else**-Block gehört in diesem Fall zum äußeren **if** und nicht, wie in der Formatierung angedeutet, zum inneren Block.

[1] https://github.com/Open-Turing-Project/OpenTuring

```
                        % no header
var n, m : int          % declarations may appear anywhere
if n > 0 then
  if m > 0 then
    put "n>0, m<0"      % simpler I/O
  else
    put "n&m>0"
  end if                % clearly delimits statement
end if
                        % no redundant end, no period
```

Listing 2: Ein einfaches Programm in `Turing` [9].

3 Elementare Sprachkonzepte

Dieses Kapitel widmet sich der grundlegenden Syntax der `Turing`-Programmiersprache. Alle erwähnten Strukturen sind Teil der heute gültigen `Turing`-Spezifikation, in die die Erweiterungen von System Turing und objekt-orientiertem Turing einflossen. Die vorgestellten Inhalte sind [6] entnommen oder fußen auf eigenen Erfahrungen mit dem Compiler.

3.1 Hallo Welt

Wie eingangs bereits erwähnt, wurde `Turing` auch entwickelt, um einen leichten Einstieg in die Programmierung zu bieten. Dabei sollten einfache Programme auch einfach umzusetzen sein. Listing 3 zeigt ein erweitertes "Hello, World!"-Programm in `Turing`.

```
/*
  Hello, World! in Turing.
*/
put "Enter your first name: " ..
var firstName: string
get firstName : *                   % Read name.
put "Hello, ", firstName, "!"
```

Listing 3: "Hello, World!" Programm in `Turing`.

Das obige Programm ist bereits vollständig. `Turing` benötigt wie Skriptsprachen keine `main`-Methode.

Mit **put** können Daten auf der Konsole ausgegeben werden. Dabei rückt der Cursor nach jeder **put**-Anweisung eine Zeile nach unten. Um das zu unterbinden, müssen am Ende der Anweisung zwei Punkte angefügt werden.
Es können auch mehrere Daten in einer **put**-Anweisung ausgegeben werden, indem diese durch Komma getrennt aneinander gereiht werden. **put** wird diese

hintereinander ausgeben. Sollte eine der Variablen eine Nummer sein, wird diese automatisch für die Ausgabe konvertiert.

Die Operation **get** liest und konvertiert Eingaben entsprechend des Datentyps der Variable, in die gespeichert wird.
Werden mit **get** Strings gelesen, so kann mit einem nachfolgenden Doppelpunkt die Anzahl der zu lesenden Zeichen angegeben werden. Möchten wir die gesamte Zeile lesen, so wird anstelle der Anzahl ein * angegeben.

Einzeilige Kommentare werden mit `%` eingeleitet, mehrzeilige mit `/* ... */` umschlossen.

3.2 Primitive Datentypen

`Turing` unterstützt die fünf in Tabelle 3.2 aufgelisteten primitive Datentypen.

`char`	Repräsentiert einen ASCII-Character
`string`	Zeichenketten bis zu einer Länge von 255 Zeichen
`int`	Vorzeichenbehaftete 32 Bit Ganzzahl
`real`	32 Bit Fließkommazahl
`boolean`	Entweder `true` oder `false`

Tabelle 1: Die fünf primitiven Datentypen in `Turing`.

Die strikte Typisierung erlaubt keine impliziten Konvertierungen. Mit der Erweiterung durch Turing Plus kamen vorzeichenlose Ganzzahlen und die Möglichkeit explizit zu konvertieren hinzu.

3.3 Programmflussstrukturen

In diesem Kapitel widmen wir uns den in `Turing` vorhandenen Programmflussstrukturen. Unterstützt werden die üblichen Schleifen und Verzweigungen.

3.3.1 Schleifen

`Turing` unterstützt die üblichen Schleifenkonstrukte. Zum einen die unendliche Schleife (Listing 4):

```
const pi : real := 3.14159
loop
  var radius: real
  put "Enter radius " ..
  get radius
  exit when radius < 0
  put "Area is ", pi * radius ** 2
end loop
```

Listing 4: Die unendliche Schleife [6].

In einem Schleifenblock können an jeder beliebigen Stelle **exit when**-Bedingungen zum Schleifenabbruch eingebracht werden. Ein Block kann auch beliebig viele dieser Abbruchbedingungen enthalten.
Des Weiteren werden auch **for**-Schleifen zur Verfügung gestellt (Listing 7).

```
var mark : int
var count : int
var sum : int := 0

put "How many marks: " ..
get count

for i : 1 .. count
  put "Mark ", i, ": " ..
  get mark
  sum := sum + mark
end for
put "Average is ", round(sum / count)
```

Listing 5: Die zählende Schleife [6].

Die klassische `for`-Schleife kann darüber hinaus in absteigender Folge durch iterieren. Die hierfür nötige Änderung am Schleifenkopf sähe wie folgt aus:

```
for decreasing i : count .. 1
```

Listing 6: Die zählende Schleife [6].

Zudem kann das Zählverhalten der Schleifenvariable geändert werden. Möchten wir zum Beispiel `i` bei jedem Durchlauf um 2 erhöhen so käme das folgendermaßen zum Ausdruck:

```
for i : 1 .. count by 2
```

Listing 7: Die zählende Schleife [6].

Wie in der infiniten Schleife, können auch in der `for`-Schleife beliebig viele **exit when** Ausdrücke platziert werden.

3.3.2 if-Verzweigungen

Das Beispiel in Listing 8 zeigt eine if-Verzweigung. Es können beliebig viele **elsif**-Blöcke zwischen dem **if** und **else** Blöcken stehen.

Bedingungen können alle Ausdrücke sein, die zu **boolean** ausgewertet werden können. Dies sind insbesondere Vergleichsoperatoren wie =, >= oder **not**=. Hinzu kommen die üblichen logischen Verknüpfungen **and**, **or** und ein eigener Operator für die Implikation =>.

```
var x : int
get x
if x = 0 then
  put "Zero"
elsif x > 0 then
  put "Positive"
else
  put "Negative"
end if
```

Listing 8: Die if-Struktur.

3.3.3 case-Verzweigungen

Die **case**-Struktur erlaubt eine kompaktere Darstellung einer Mehrwegverzweigung. In `Turing` können mit **type** definierte Enumeratoren, Zahlen, einzelne Zeichen und darüber hinaus auch Strings als Entscheidungsquellen herangezogen werden.

```
var x : string
loop
   get x : *
   case x of
      label "stop": exit
      label       : put "Type 'stop' to exit."
   end case
end loop
```

Listing 9: Die case-Verzweigung.

Nicht behandelte Fälle können mit einer leeren **label**: Klausel behandelt werden.

3.4 Unterprogramme

`Turing` unterscheidet strikt zwischen Funktionen und Prozeduren und stellt zum Teil Schlüsselwörter zur Verfügung, die nur im jeweiligen Kontext einer Funktion oder Prozedur syntaktisch korrekt sind.
Listing 10 zeigt ein Beispiel zur Deklaration einer Prozedur. Die Rückkehr aus einer Prozedur ist an jeder Stelle im Prozedurrumpf mit **return** möglich.

```
procedure triangles(size : int)
    if size < 1 then
        put "Size must be at leat 1."
        return
    end if
    % Outputs a triangle of size asterisks
    for i : 1 .. size
        put repeat ("*", i )
    end for
end triangles
```

Listing 10: Ein Beispiel für eine Prozedur [6].

Listing 11 zeigt eine Funktion. Im Gegensatz zu Prozeduren wird zusätzlich im Kopf der Funktion der Typ der Rückgabe angegeben. Ein Ergebnis dieses Typs kann an jeder Stelle mit **result** an die Aufruferroutine zurückgegeben werden.

```
function roundCent(amount : real) : real
    % Round amount to nearest cent
    result round (amount * 100) / 100
end roundCent
```

Listing 11: Ein Beispiel einer Funktion [6].

3.4.1 Parameterlose Subroutinen

Spezielle Syntax erlaubt `Turing` bei parameterlosen Subroutinen.

```
procedure print
    put "Print"
end print

print()
print
```

Listing 12: Parameterlose Subroutinen.

Diese verlangen weder bei der Deklaration der Routine noch beim Aufruf eben dieser keine Klammern. Die Aufrufe print und print() sind gleichwertig.

3.4.2 Aufruf vor Deklaration

Um Subroutinen verwenden und aufrufen zu können, müssen deren entsprechende Deklarationen in der Sourcedatei vorher aufscheinen. `Turing` stellt zwei

Schlüsselwörter zur Verfügung die es ermöglichen Funktionen vor deren eigentlichen Implementierung anzuwenden.

```
forward procedure print()

print()

body procedure print()
  put "Print"
end print
```

Listing 13: Aufruf einer Routine vor der Implementierung.

Ähnlich C wird in Listing 13 erst der Prototyp der Routine bekanntgegeben und ein **forward** vorangestellt. An entsprechender Stelle wird der Implementierung das Schlüsselwort **body** beigefügt um klarzustellen, dass dies die Implementierung einer bereits definierten Routine ist.

3.5 Strings

`Turing` verfährt mit Strings auf eine etwas angenehmere Art und Weise als das große Vorbild Pascal. Prinzipiell muss die Länge des Strings bei der Deklaration nicht bekannt sein. Auch ist keine explizite Speicherallozierung notwendig. Um einen String zu kreieren genügt also:

```
var message : string
```

Listing 14: Deklaration eines Strings.

Wichtig bei der Verwendung von Strings ist es zu beachten, dass sie eine maximale Länge von nur 255 Zeichen annehmen können. Kommt es dennoch zu einer Konkatenation von mehr als 255 Zeichen, reagiert `Turing` mit einem Laufzeitfehler.

Besonders angenehm gestalten sich die wichtigsten Operationen auf Strings. Zur Konkatenation zweier Strings genügt der +-Operator:

```
message := "This is a " + "split" + " message."
```

Listing 15: Stringkonkatenation.

`Turing` bietet auch einfache Möglichkeiten Substrings zu extrahieren.

```
put message(1 .. 4)
put message(*-7 .. *)
```

Listing 16: Zugriff auf Teile eines Strings.

Die erste Anweisung gibt “This” aus. Zu beachten ist dabei, dass Strings beginnend bei 1 indiziert werden.
Das Symbol * verweist auf den Index des letzten Zeichens. In der zweiten Anweisung werden die letzten sieben Zeichen des `message` Strings also “message.” ausgegeben.

Daneben gibt es noch zahlreiche vordefinierte Funktionen, wie

- length(**string**) : **int** gibt die Länge des Strings zurück.
- index(**string**, **string**) : **int** findet die erste Position eines Substrings in einem String.
- strok(**string**) : **boolean** ist wahr, wenn der String fehlerlos in eine Nummer konvertiert werden kann.
- intstr (**string**) : **int** konvertiert einen String in einen Integer.

3.6 Types und Records

Typdefinitionen erlauben es, benutzerdefinierte Datentypen zu kreieren.

```
type nameType : string(30)
type range    : 0 .. 150
```

Listing 17: Benutzerdefinierte Typen.

Hier wird ein Stringtyp mit einer fixen Länge von 30 Zeichen und ein numerischer Typ, der Werte auf einen Teilbereich einschränkt, definiert.
Die Datentypen können dabei auch komplexere Strukturen annehmen, indem Records verwendet werden. Listing 18 zeigt ein Beispiel.

```
type entry : record
  name : nameType
  age  : int
end record
```

Listing 18: Strukturierung mit Records.

Die Deklaration von Variablen benutzerdefinierten Typs geschieht wie gewohnt. Auf Elemente eines Records wird mit dem Punkt-Operator zugegriffen.

```
var me : entry
me.name := "Mathias Hoertnagl"
me.age  := 26
put "Name: ", me.name
```

Listing 19: Verwendung von Records.

3.7 Arrays

Die Definition von Array geschieht in `Turing` wie in folgendem Beispiel.

```
var marks : array 1 .. 100 of int
```

Listing 20: Definition eines Arrays.

Dies erzeugt ein Array der Größe 100 vom Datentyp **int**. Erlaubt sind ebenfalls Variablen als Bereichsgrenzen. Die Zellen des Array sind nach der Deklaration nicht initialisiert. Dies muss gesondert für jedes Element des Arrays erfolgen.

```
marks(1) := 1
put marks(1)
```

Listing 21: Zugriff auf Elemente eines Arrays.

In `Turing` ist es erlaubt, Arrays nach Deklaration verlustfrei in ihrer Ausdehnung zu vergrößern. Wird das Array verkürzt, so gehen die Inhalte der betroffenen Elemente verloren.

```
var marks : flexible array 1 .. 100 of int
```

Listing 22: Definition eines dynamischen Arrays.

Mit dem Schlüsselwort **flexible** wird dem Compiler mitgeteilt, dass das Array während der Laufzeit seine Größe ändern kann.

```
new marks, 200
```

Listing 23: Vergrößerung eines dynamischen Arrays.

In Listing 23 wird das vorher auf 100 Elemente limitierte Array auf 200 erweitert. Bereits vorher gespeicherte Elemente bleiben erhalten. Diese flexible Behandlung von Arrays ist allerdings auf die erste Dimension beschränkt.

3.8 Module

Zur Organisation großer Projekte bietet `Turing` die Möglichkeit der Strukturierung und Kapselung von logischen Komponenten mittels Modulen an.
Listing 24 zeigt eine Implementierung eines Stacks für Strings. Die Moduldefinition verbirgt dabei alle Variablen, Prozeduren und Funktionen, die nicht explizit in **export** aufgezählt werden.

```
module stack
  export push, pop, var top

  var top : int := 0
  var contents : array 1 .. 100 of string

  procedure push(s : string)
    top := top + 1
    contents(top) := s
  end push

  procedure pop(var s : string)
    s := contents(top)
    top := top - 1
  end pop
end stack
```

Listing 24: Organisation einer Stackimplementierung mittels Modulen [4].

Sollen Variablen global sichtbar sein, so werden diesen, wie am Beispiel von `top` ersichtlich, ein **var** in der Exportaufstellung vorangestellt.
Verwendet werden kann das Stack-Modul wie folgt:

```
stack.push("Satan")
put stack.top
var name : string
stack.pop(name)
put stack.top
```

Listing 25: Exemplarische Verwendung des Stack-Moduls.

Besonders wichtig ist hier der Unterschied zu Objekten, da Module nicht instanziiert werden und demzufolge in diesem Beispiel immer nur ein Stack existiert auf dem operiert wird.

3.9 Sound, Grafiken und GUIs

Eine Bibliothek bietet Unterstützung bei der Entwicklung von grafischen Benutzeroberflächen als auch beim Zeichnen von Grafiken und Sprites. Zusätzlich steht noch eine rudimentäre Bibliothek zur Wiedergabe einfacher Tonsequenzen bereit.
Andere vordefinierte Bibliotheken, die etwa grundlegende Datenstrukturen implementieren, fehlen in `Turing` gänzlich.

Dies schließt die Diskussion der grundlegenden Sprachkonzepte von Turing. Im nächsten Kapitel werden die systemnahen Erweiterungen von Turing Plus näher erörtert.

4 Systemprogrammierung

Mit Turing Plus erfuhr die Programmiersprache eine Erweiterung hin zur systemnahen Sprache. Hinzu kamen unter anderem vorzeichenlose Datentypen, unsichere Typkonvertierungen, Pointer, Pointerarithmetik. Zusätzlich konnten nun auch Prozesse erzeugt und synchronisiert werden [1].

4.1 Zeiger

Die wichtigste Neuerung stellt dabei die Unterstützung von Zeigern dar. Der Beispielcode aus Listing 26 deklariert dabei einen Zeiger auf eine natürliche Zahl auf zwei Arten.

```
var pNat  : pointer of nat
var pNat2 : ^nat
```

Listing 26: Definition eines Zeigers.

Die zweite Deklaration ist lediglich eine Kurzform der ersten. Zeiger spielen insbesondere in der objekt-orientierten Programmierung eine Rolle, da alle Objektvariablen Zeiger auf die sie instanziierende Klasse sind. Ein Beispiel für eine solche Instanziierung enthält Kapitel 5.2.
Mit der Einführung von Zeigern wurden auch explizite Konvertierungen von primitiven Datentypen notwendig, um die sonst strikte Typisierung von `Turing` zu lockern, ohne sie gänzlich aufgeben zu müssen.

4.2 Prozesse

Die Erzeugung eines Prozesses erfolgt in zwei Schritten. In Listing 27 wird erst in einem **process**-Block die Funktionalität eines Prozesses implementiert und anschließend mit **fork** ein neue Prozess erzeugt.

```
process speak(word: string)
  loop
    put word
  end loop
end speak

fork speak("Hi") % Start saying: Hi Hi Hi . . .
fork speak("Ho") % Start saying: Ho Ho Ho . . .
```

Listing 27: Ein einfacher Prozess [4].

Zur Synchronisation von Prozessen, stellt `Turing` dem Entwickler ausschließlich Monitore zur Verfügung [9].

Nach dieser kurzen Einführung in die wichtigsten systemnahen Elemente von `Turing`, soll das nächste Kapitel einen Überblick über die objekt-orientierte Programmierung geben.

5 Objekt-orientierte Programmierung

In diesem Kapitel wird die Umsetzung des objekt-orientierten Paradigmas in `Turing` diskutiert. Die Vererbung ist auf eine Superklasse beschränkt. Methoden werden nur anhand ihres Namens, nicht aber anhand der Parametertypen unterschieden. `Turing` besitzt wie C++ keinen Garbage-Collector.

5.1 Klassendefinitionen

In Listing 32 entsprechend [4] soll exemplarisch ein Stack als Klasse implementiert werden. Die Definition ähnelt jener von Modulen. über das Schlüsselwort **export** werden die öffentlich zugänglichen Methoden und Funktionen angegeben.

```
class Stack
  export push, pop

  var top : int := 0
  var contents : array 1 .. 100 of string

  proc push(s : string)
    top := top + 1
    contents(top) := s
  end push

  proc pop(var s : string)
    s := contents(top)
    top := top - 1
  end pop
end Stack
```

Listing 28: Stack als Klasse.

`Turing`-Klassen benötigen keinen Konstruktor, wie er etwa in C++ oder Java üblich ist.

5.2 Objekterzeugung

Erzeugen neuer Instanzen von Klassen geschieht in zwei Schritten. Wie in Listing 29 abgebildet wird zuerst eine Variable als Zeiger auf die Klasse `Stack` erzeugt. Anschließend wird diese Klasse neu instanziiert. Das Schlüsselwort **new** verlangt dabei die Variable, die auf das neu instanziierte Objekt verweist.

```
var stack : ^Stack
new stack
```

Listing 29: Eine Instanz der Klasse Stack.

5.3 Zugriff auf Elemente eines Objekts

Auf mit **export** gelistete Methoden, Funktionen und Variablen kann mit dem `->`-Operator zugegriffen werden. Dieser besitzt die selbe Semantik wie in C++. Erst wird der Zeiger dereferenziert und anschließend die Methode oder Funktion aufgerufen.

```
stack->push("Satan")
```

Listing 30: Aufruf einer Methode mit `->`.

Alternativ kann der Zeiger explizit dereferenziert werden, indem der umklammerten Variable die Klasse vorangestellt wird, als die sie interpretiert werden soll. Anschließend kann mit dem Punkt-Operator die gewünschte Methode aufgerufen werden.

```
Stack(stack).push("Satan")
```

Listing 31: Aufruf einer Methode mit dem Punkt-Operator.

5.4 Vererbung

`Turing` unterscheidet nicht zwischen Klassen, abstrakten Klassen oder Interfaces wie wir es von Java gewohnt sind. Sollen bestimmte Methoden einer Klasse erst von einer Subklasse implementiert werden, so müssen diese mit **deferred** markiert werden.

```
class Stack
  export push, pop

  deferred proc push(s : string)
  deferred proc pop(s : string)
end Stack
```

Listing 32: Stack als Klasse.

Vererbung in `Turing` bietet nur die Möglichkeit von einer Klasse zu erben. Da Interfaces auch nur Klassen sind, kann de facto nur von einer einzigen Klasse geerbt werden. In Listing 33 erbt die Klasse RealStack von Stack. Die Superklasse wird mit dem Schlüsselwort **inherit** angegeben.

Jede bereits in der Superklasse exportierte Methode, die erstmalig oder erneut implementiert werden soll, muss mit **body** versehen werden.

```
class RealStack
  inherit Stack
  ...
  body proc push(s : string)
    ...
  end push

  body proc pop(s : string)
    ...
  end pop
end RealStack
```

Listing 33: RealStack erbt von Stack.

5.4.1 Die Wurzelklasse anyclass

An der Wurzel der Vererbungshierarchie in `Turing` steht nicht `Object` mit einigen wenigen nützlichen Methoden wie in Java, sondern das methodenlose **anyclass**. Wie der Name schon verrät, können Variablen dieses Typs, Objekte jeder Klasse in `Turing` halten.

5.4.2 Klassenidentifikation zur Laufzeit

Um die Klasse eines Objekts zur Laufzeit erhalten zu können, stellt `Turing` die Funktion **objectclass**(^**anyclass**) zur Verfügung:

```
var c : ^anyclass
if objectclass(c) = ClassA then
  ...
elsif objectclass(c) >= ClassB then
  ...
end if
```

Listing 34: Klassenidentifikation zur Laufzeit.

Es können die üblichen Vergleichsoperatoren verwendet werden. Insbesondere kann mit ermittelt werden, ob das Objekt c eine Instanz der Klasse ClassB oder einer Subklasse von ClassB ist.

```
objectclass(c) >= ClassB
```

Listing 35: Klassenidentifikation zur Laufzeit.

5.5 Objektzerstörung

Erzeugte Objekte müssen vom Programmierer manuell wieder entfernt werden.

```
var stack : ^Stack
free stack
```

Listing 36: Zerstören eines Objektes.

6 Zusammenfassung und Ausblick

Mitte der 1980er Jahre konnte `Turing` noch mit einigen interessanten Aspekten punkten und in einigen Bereichen den großen Konkurrenten Pascal übertrumpfen. Die Erweiterungen brachten keine nennenswerte Vergrößerung des Wirkungsbereichs der Sprache und besonders die Umsetzung des objektorientierten Paradigmas ist empfindlich unvollständig.
Die von `Turing` bereitgestellten Bibliotheken decken nur enge Teilbereiche, wie etwa die GUI-Programmierung ab, mehrheitlich fehlen aber Implementierungen von wichtigen Datenstrukturen wie HashTables oder Listen. Strukturen, die durchaus von Programmieranfängern rudimentär implementiert werden können, die der professionelle Entwickler aber nicht missen möchte.

Dennoch wird die Sprache an vielen Highschools Ontario's noch heute in Einführungskursen zur Programmierung genutzt.

Literatur

[1] R. C. Holt and J. R. Cordy. The turing programming language. *Commun. ACM*, 31(12):1410–1423, Dec. 1988.

[2] R. C. Holt and T. West. *Object Oriented Turing Reference Manual*. Holt Software Associates Inc., 203 College Street, Suite 305 Toronto, Ontario M5T 1P9, seventh edition, 1999. `www.somewhere.com`.

[3] Holtsoft. Turing 4.4.1. Freigabe. `http://compsci.ca/blog/download-turing-411/`. zuletzt abgerufen am 10.01.2013.

[4] Holtsoft. Turing 4.4.1. Online Documentation. `http://compsci.ca/holtsoft/doc/`. zuletzt abgerufen am 10.01.2013.

[5] T. E. Hull, A. Abrham, M. S. Cohen, A. F. X., Curley, C. B. Hall, D. A. Penny, and J. T. M., Sawchuk. Numerical turing. *SIGNUM Newsl.*, 20(3):26–34, July 1985.

[6] J. N. P. Hume. *Introduction to Programming in Turing*. Holt Software Associates Inc., 203 College Street, Suite 305 Toronto, Ontario M5T 1P9, first edition, 2001. `www.somewhere.com`.

[7] Open Turing. The Open Turing Website. `http://tristan.hume.ca/openturing/`. zuletzt abgerufen am 10.01.2013.

[8] OpenT. OpenT Projekt Homepage. `http://code.google.com/p/opent/source/list`. zuletzt abgerufen am 10.01.2013.

[9] S. Perelgut and J. R. Cordy. Turing Plus: a comparison with C and Pascal. *SIGPLAN Not.*, 23(1):137–143, Jan. 1988.