Feder – Eine Programmiersprache

(von Fionn Langhans)

Kurze Beschreibung

Das Ziel des Projekts ist es, eine einfache, kompilierbare Programmiersprache zu definieren, die einige Aufgaben auf dem Computer erledigen kann, wie herkömmliche Sprachen, so z.B. C, C++ oder Java, es auch machen. Die Programmiersprache soll nicht nur kompilierbar sein, sondern auch möglichst wenig Laufzeitressourcen verbrauchen. Zudem sollte die definierte Syntax es erlauben, die Größe des Quelltextes von Programmen möglichst kurz zu halten. „Fremdsprachen“ (von Feder aus gesehen), wie C, sollen auch einfach implementiert werden können, um verschiedene Bibliotheken, die in Fremdsprachen geschrieben wurden, in Feder benutzen zu können. Außerdem sollte ein Benutzer dieser Sprache auch eigene „Basisbibliotheken“ erstellen können (man kann in Feder Grundtypen wie int oder double selbst erstellen). Darüber hinaus soll die Sprache einsteigerfreundlich sein, deswegen sollte sie nicht zu viele Operatoren benutzen oder viele Sonderfälle haben. Damit Objekte auch einfach zu handhaben sind, wurde ein „Garbage Collector“ (eng. für „Müllsammler“) für Objektlebenszeiten (automatisches Löschen) implementiert. Zusätzlich wurden alle Quelltextdateien und Quelltextdokumentationen in Englisch abgefasst, sodass eine größere Zielgruppe erreicht werden kann. Es gibt noch keine statischen oder dynamische Bibliotheken, da dies den Zeitrahmen sprengen würde.

Zum Ausprobieren der Programmiersprache wurde ein Compiler entwickelt, damit Feder in eine maschinennähere Sprache übersetzt und somit auf üblichen Betriebssystemen (z.B. Linux, BSD, Windows) ausgeführt werden kann. Auch wurden Bibliotheken in Feder geschrieben, damit Typen, wie z.B. (u)int(16|32|64), double oder String, verwendet werden können oder Interaktionen mit Input/Output gemacht werden können (Dateisystem, I/O im Terminal). Auch wurden für den Konsoleneditor „vim“ und das GUI Programm „gedit“ Skripte geschrieben, damit spezielle Schlüsselwörter oder Ausdrücke markiert werden, weil man somit mehr Übersicht über Feder - Quelltext hat.

Das komplette Projekt befindet sich auf GitHub: https://github.com/codefionn/feder. In dem Verzeichnis „doc/tutorials“ sind erklärte Beispiele, wie man bestimmte Sachen in Feder macht (auf Englisch).

Inhaltsverzeichnis

1. Einleitung

1.1. Wahl des Themas

Nachdem ich schon länger einige Programmiersprachen lerne und einige Programme entwickelt habe, wollte ich diese in einer eigenen Programmiersprache schreiben können. Zudem wollte ich ausprobieren, wie man denn eine Programmiersprache definiert und einen Compiler dafür programmiert.

1.2. Hilfsmittel

Um einen Compiler zu erstellen, wurde das Buch „Einführung in die Informatik“ von Heinz-Peter Gumm und Manfred Sommer, 10. Auflage, benutzt, insbesondere das 9. Kapitel „Theoretische Informatik und Compilerbau“ [1, S. 695]. Auf Grundlage dieses Kapitels wurden Teile des Compilers geschrieben. Zum anderen wurde die Programmiersprache Java zur Entwicklung des Compilers benutzt und die Programmiersprache C, da der Compiler Feder zu C umwandelt. Des Weiteren wurde die freie Software „valgrind“ [3] und der GNU Compiler [4] und clang [5] benutzt. Weitere Hilfsmittel werden im Text genannt. Auch werden hier reguläre Ausdrücke nach dem POSIX Standard von „The Open Group“ [2] verwendet.

2. Die Programmiersprache

2.1. Konzepte

Feders Ziel ist es, wie schon oben genannt, einfach und kompilierbar zu sein und durch die Syntax möglichst überschüssige Zeichen zu vermeiden. Um dies zu erreichen, wurden verschiedene Konzepte entwickelt, die hoffentlich die genannten Prinzipien verwirklichen können.

2.1.1. Keine Hauptmethode

Um die Quelltextdateien einfach zu halten, wurde statt der klassischen „void main (int lenargs, char ** vargs)“ Hauptfunktion ein Konzept eingeführt, das meist in Skriptsprachen (z.B. Python, JavaScript) vorzufinden ist: Objekte, Funktionen und Klassen können so aufgerufen oder erstellt werden, indem man sie einfach irgendwo in einem Namensraum oder gleich im Hauptkörper (nicht in einer Klasse, Funktion, usw.) aufruft.
Somit sieht das traditionelle „Hello World“ - Programm so aus:

```
include "stdio.fd"
io.println ("Hello, World!")
```

2.1.2. Statische Objekte

Ein anderes Konzept, das Feder durchsetzt, sind statische Objekte: Diese können in Feder mit einem Typen, also einer Klasse oder einer Schnittstelle, deklariert werden, damit sie „gezwungen“ werden diesen Typen zu vertreten. Dies wird in Funktionsargumenten oder Objekten von Klassen vorausgesetzt. Bei Objekten, die als Typen eine Schnittstelle haben, ist es zwingend notwendig, dass das Objekt statisch deklariert wird.
Statische Objekte haben große Vorteile bei Funktionsargumenten, da der Compiler einen Fehler ausgibt, wenn diese nicht übereinstimmen und somit weniger fehlerhaften Aufrufe geschehen.

```
String s = 100   # Fehler: Typen sind nicht gleich
io.println (100) # Fehler: Keine Funktion
                 # passt zu der aufgerufenen
                 # (Natürlich nur wenn „func println (int32 n)“
                 # in io nicht erstellt wurde)
```

2.1.3. Dynamische Objekte

Jedoch sind statische Objekte manchmal hinderlich oder überflüssig, so hat Feder auch ein dynamisches Objektsystem. Ein großer Vorteil eines dynamischen Objekts ist, dass Typen automatisch zugewiesen oder auch einfach geändert werden können. Nur Objekte, die mit mit ener Klasse erstellt wurden, können dynamisch sein.

```
s = "Hello, World!" # s ist nun ein String
s = Integer32.set (100) # Typ von s wird von String zu
                        # Integer32 geändert
```

2.1.4. „Include“ Operation

Eine „include“ (übersetzt: „einschließen“) ähnliche Operation darf natürlich nicht fehlen in einer Programmiersprache. Mit dem include Operator kann man andere Dateien („Bibliotheken“) in ein Programm miteinbeziehen, dadurch wird die Fehlersuche vereinfacht und es wird die Möglichkeit geboten, Aufgaben auf verschiedene Quelltextdateien aufzuteilen.

```
include "stdc.fd" # Fügt die „Feders C Standardbibliotek“
                  # in den Compiler ein
```

2.1.5. Garbage Collection

Da eine einfache, einsteigerfreundliche Programmiersprache bereitgestellt werden soll, „muss“ ein automatischer Prozess, der nicht mehr verwendete Objekte löscht, implementiert sein, also eine „Garbage Collection“. Der momentane Compiler benutzt einen Referenzzähler, um nicht mehr benutzte Objekte zu löschen. Dieser muss erhöht werden, wenn ein Objekt einen neuen „Alias“ oder „Namen“ erhält (durch Funktionsaufruf oder dem Zuweisungsoperator „=“). Und dieser muss verringert werden, wenn ein Objekt im Kontext nicht mehr benutzt wird (z.B. bei dem Ende eines Körpers, Funktionsaufruf ohne Zuweisung). Wenn der Referenzzähler eines Objekts Null ist, wird dieses Objekt gelöscht (Optimal wären die Stellen, wo es nicht mehr benutzt werden kann).

2.1.6. Schnittstellen

Schnittstellen (eng. „Interface“) sind sehr nützlich, um Funktionen mit einem Objekt darzustellen. Dies findet zum Beispiel besonders Anwendung bei grafischen Benutzeroberflächen, damit es möglich ist, dem Benutzer zuzuhören (eng. „Listener“), was er gerade macht.

2.1.7. Funktionen höherer Ordnung

Ein Nebenprodukt von Schnittstellen (eng. „Interfaces“), die Objekte deklarieren können und auch von Funktionen zurückgegeben werden können, sind Funktionen höherer Ordnung [8]. Diese erlauben es, Funktionen als Parameter zu besitzen und/oder eine Funktion zurückzugeben.

```
int32 interface integerInterface (int32 v)
int32 twice (integerInterface fn, int32 n)
     return fn(fn(n))
;

int32 addThree (int32 v)
     return v.add(3)
;

twice (addThree, 7)
```

2.1.8. Namensräume

Feder stellt auch Namensräume bereit, wodurch Namenskonflikte einfacher vermieden und Funktionen von Bibliotheken oder sonstiges besser unterteilt werden können.

Es gibt auch zwei spezielle Namensräume: „h_intern[[:alpha:]_[:digit:]]*“ und „c_intern[[:alpha:]_[:digit:]]*“ (reguläre Ausdrücke). Namensräume, die mit h_intern starten, legen ihren generierten Code in einer C Headerdatei ab und Namensräume die mit c_intern starten, lassen ihren Code in eine C Quelltextdatei einfügen.

2.1.9. Mehrzeilige Zeichenketten

Mehrzeilige Zeichenketten (auch bekannt als „Strings“) sind besonders hilfreich, um längere Zeichenketten zu bilden, ohne jedesmal eine andere anhängen zu müssen. So ist die Lösung der Aufgabe auf https://projecteuler.net/problem=13 (zuletzt abgerufen am 05.01.2018) in C relativ schwer, weil man jede Zeile extra bearbeiten muss, um die vielen Zahlen einfügen zu können (natürlich könnte man z.B. awk [8] benutzen um den Prozess zu beschleunigen), jedoch ist die Lösung in z.B. Python sehr einfach (hat auch andere Gründe, dies sei hier aber mal ignoriert).

```
mehrzeilige_zeichenkette = "Ich bin
eine sehr, sehr,
sehr, sehr,
sehr, sehr,
sehr, sehr,
sehr, sehr
lange Zeichenkette"
```

2.1.10. Vererbung

Feder stellt auch ein Vererbungsprinzip (nur einfache Vererbung möglich!) bereit, damit man einige objektorientierte Prinzipien verwirklichen kann. In Feder wird eine Vererbung folgendermaßen gekennzeichnet:

```
class object
;
class AnotherClass object  # AnotherClass erbt von object
;
```

2.1.11. Kommentare

Kommentare dürfen in einer Programmiersprache nicht fehlen, da mit diesen anderen bzw. sich selbst mitgeteilt werden kann, was der Quelltext eigentlich machen soll. Ein Kommentar, der bis Zeilenende gilt, fängt mit einem „#“ an. Ein mehrzeiliger Kommentar muss mit „##“, also zwei „#“, angefangen werden und auch wieder mit zwei „#“ abgeschlossen werden. Ein kurzes Beispiel:

```
# Ich bin ein Kommentar bis zum Zeilenende
Code # Dasselbe wie vorhin nur mit Code davor
## Ich
bin ein
mehrzeiliger
Kommentar ##
```

2.1.12. „Global“ Schlüsselwort

Das „global“ Schlüsselwort ermöglicht es, globale Objekte deklarieren zu können. Dies ist in Feder bei einer normalen Deklaration nicht möglich (in einem Namensraum), da man normalerweise in

die Hauptmethode des Programms schreibt (bzw. zu einer Funktion, die die Bibliothek lädt). Dies würde einen dazu zwingen, keine globalen Objekte zu benutzen. Jedoch wäre das bei z.B. Konstanten sehr hinderlich.
Wenn man z.B. die Konstante „Array_reserve_length“ deklarieren und definieren wollte, müsste man folgendes schreiben:

```
global int32 Array_reserve_length = 100
```

2.1.13. Präprozessor

Feder besitzt auch einen „Präprozessor“. Dieser wird aufgerufen, wenn eine Zeile mit „::“ startet. Es ist eigentlich kein Präprozessor nach [6], da nicht die „Eingabedaten vorbereitet [werden] und zur weiteren Bearbeitung an ein anderes Programm [weitergegeben]“ [6] werden, sondern gleich während der Kompilierung verwendet werden, zumindest im „jfederc“ - Compiler. Dies wurde dennoch Präprozessor genannt, da erstens auch eine Präprozessor-Implementierung nahe liegt und zweitens die Verwendungsart ähnlich ist zu „C“s Präprozessor. Der Präprozessor von Feder ist jedoch nicht so vielseitig wie „C“s.

Mit dem Präprozessor in Feder kann man Code von der Kompilierung ausschließen: Man kann mit dem Schlüsselwort „def“ ein Makro ohne Parameter erstellen. Dies ist wiederum hilfreich für das das Schlüsselword „if“ oder „elif“, diese starten einen Körper und diese werden in die Kompilierung eingeschlossen, wenn das nachstehende Makro existiert. Abgeschlossen werden kann ein Körper, der von „if“, „elif“ oder „else“ gestartet wurde, mit „fi“. Die Körper, die durch „if“ oder „elif“ gestartet wurden, können mit „elif“ oder „else“ auch beendet werden. Wenn an „if“ oder „elif“ ein „n“ angehängt wird, wird der Körper in die Kompilierung eingeschlossen, wenn das Makro nicht existiert.

Ein kleines Beispiel:

```
::def BEISPIEL_MACRO
::if BEISPIEL_MACRO
     # Code der ausgeführt wird
::else
     # Code der nicht ausgeführt wird
::fi

::ifn BEISPIEL_MACRO_DAS_NICHT_EXISTIERT
     # Code der ausgeführt wird
::
```

2.1.14. Regeln

Ein weiteres Prinzip, das in Feder Verwendung findet, wird „Regeln“ (eigens ausgedachte Bezeichnung) genannt. Mit Regeln kann man in Feder bestimmte Operatoren nutzen, um z.B. arithmetische Operationen durchzuführen. Regeln sind Teil des Präprozessors. Es gibt zwei verschiedene Arten von Regeln: die einen rufen Funktionen auf, die anderen eine bestimmte Zeichenabfolge, bei der Teile der Abfolge durch jeweils die linke und rechte Seite des Operators ersetzt werden.

Der „+“ - Operator für den primitiven Datentypen „int32“

```
::rule pattern + "{0} + {1}" int32 int32 int32
```

Hierbei zeigt „pattern“ dass wir eine Zeichenabfolge haben. Das „+“, zeigt, welcher Operator das Ziel ist und „{0} + {1}“ ist die Zeichenabfolge. Bei dieser wird die linke Seite „{0}“ ersetzen und „{1}“

wird mit der rechten Seite getauscht. Das erste „int32“ ist der Rückgabetyp und das zweite der erwartete Typ der linken Seite und das letzte „int32“ ist der erwartete Typ auf der rechten Seite.

Mit dem „rule“ - Operator kann man zudem bestimmen, wie mit Zeichenketten, ganzen Zahlen oder Zahlen mit Kommastellen umgegangen werden soll. Der „rule“ Operator bzw. der „rule“ Befehl kann noch wesentlich mehr und bekommt immer mehr Kompetenzen, jedoch würde es den Rahmen dieser Arbeit sprengen, wenn er hier beschrieben werden würde.

2.1.15. Arrays

Um bessere Ressourceneffizientz (wenn ein mit Feder geschriebenes Programm ausgeführt wird) zu bieten, stellt Feder Felder (eng. „array“) bereit. Diese sind vor allem sinnvoll, wenn man eine Ansammlung primitiver Datentypen benötigt. Im Folgenden wird ein Feld mit dem Typen „int32“ erstellt. Die Länge des Feldes soll 200 betragen (Größe des benötigten Speichers in byte (Octett): `200 * sizeof (int32_t)`, also 200 * 4 = 800 [byte]).

```
ar = int32[200]
```

Um ein Objekt an der Position x des Feldes zu bekommen, würde man Folgendes schreiben:

```
ar[x]
```

Die Länge eines Feldes kann durch den reservierten Funktionsnamen „len“ bekommen werden:

```
laenge = len (ar)
```

Wenn man ein Objekt hinten an das Feld anfügen möchte, muss man den reservierten Funktionsnamen „append“ benutzen:

```
append (ar, 20)
```

Man kann auch mehrere Objekte auf einmal anfügen:

```
append (ar, 20, 21, 102)
```

Die Funktion „append“ gibt zudem das Feld zurück, dies ist dasselbe wie das 1. Argument (hier: „ar“).

2.2. Typen erstellen und aufrufen

Im Folgenden wird aufgezeigt, wie man Typen in Feder deklariert und definiert. Wenn eine Deklaration einen „Körper“ startet, muss dieser wieder mit „;“ abgeschlossen werden (Zwischen dem Start des „Körpers“ und „;“ wird definiert). Auch muss ein Name, also etwa eine Klasse, Funktion, Schnittstelle, ein Objekt oder Namensraum, dem regulären Ausdruck „[[:alpha:]_][[:alpha:]_[:digit:]]*“ entsprechen.

2.2.1. Namensräume

Namensräume werden in Feder mit dem Schlüsselword „namespace“ (eng. „Namensraum“) deklariert, diese starten auch einen Körper. Nach dem Schlüsselwort muss der Name des Namensraumes stehen: ‚namespace‘ [Name]. Der Quelltext würde, wenn man den Namensraum „std“ erstellen will, folgendermaßen aussehen:

```
namespace std
      class Klasse
      ;
;
```

Um die Klasse „Klasse“ im Namensraum „std“ aufzurufen, würde man folgendes schreiben:

```
std.Klasse
```

Also kann man Objekte, Klassen, Namensräume, Funktionen oder Schnittstellen in einer Klasse, Objekt oder Namensraum, mit einem „ . “ (Punkt) aufrufen.

2.2.2. Funktionen

Funktionen werden mit dem Schlüsselwort „func“ (kurz für „function“, English für „Funktion“) deklariert und starten danach einen Körper. Eine einfache Deklaration einer Funktion würde wie folgt aussehen: ‚func‘ [Name].
Jedoch können Funktionen in Feder auch einen Rückgabetypen besitzen, dieser wird vor dem Schlüsselwort genannt: [Rückgabetyp] ‚func‘ [Name]. Der Rückgabetyp kann hierbei eine Klasse oder eine Schnittstelle sein. Funktionen können aber auch Argumente besitzen, diese würde man am Schluss der Deklaration anhängen: ([Rückgabetyp]) ‚func‘ [Name] (‚(‘[Argumente],)‘). Zu beachten ist, dass die Argumente einer Funktion nur statische Typen sein können.

So würde man in Feder eine Funktion mit dem Namen „sync“, die keinen Rückgabetypen und keine Argumente besitzt, wie folgt erstellen:

```
func sync
;
```

Die Funktion „addto“, die keinen Rückgabetypen besitzt, jedoch ein Argument mit dem Name ‚n‘ und dem Typen ‚int‘, würde man folgendermaßen erstellen:

```
func addto (int32 n)
;
```

Zuletzt kommt noch eine Funktion mit dem Namen „add“. Diese hat den Rückgabetypen ‚int‘ und zwei Argumente mit jeweils dem Namen ‚n0‘ und ‚n1‘. Beide Argument sind von Typen ‚int‘.

```
int add (int32 n0, int32 n1)
;
```

Um die obige Funktion aufzurufen, müsste man Folgendes schreiben:

```
result = add (1, 2) # Die Nummern 1 und 2 sind vom Typen ‚int32‘
# Oder:
add (1, 2) # Man muss nicht „result = “ schreiben,
           # damit die Funktion aufgerufen wird.
```

Oben sieht man, dass mehrere Argumente mit „ , “ getrennt werden.
Um die Funktion „sync“, die oben erstellt wurde, aufzurufen, würde man Folgendes machen:

```
sync ()
```

So sieht man, dass [name] ‚(‘ [argumente] ‚)‘, eine Funktion aufruft.
Funktionen können in derselben Datei, wo sie deklariert wurden, auch erweitert werden:

```
func sync
     # Code …
;
```

```
func sync
	# Code ..
;
```

2.2.3. Schnittstellen

Schnittstellen werden in Feder fast genauso wie Funktionen deklariert, nur dass das Schlüsselword „func“ durch „interface“ ersetzt wird. Auch startet eine Schnittstellendeklaration keine Definition, also keinen neuen Körper, somit braucht man auch diesen nicht zu beenden. Die Definition sieht also wie folgt aus: ([Rückgabetyp]) ‚interface‘ (‚(‘ [Argumente] ‚)‘)

Die Schnittstelle „int_equals“ mit den Argumenten „object obj0“, „object obj1“ und dem Rückgabetyp „bool“ sieht folgendermaßen aus:

```
bool interface int_equals (object obj0, object obj1)
# Kein ;, weil kein neuer Körper gestartet wurde
```

2.2.4. Klassen

Klassen werden mit dem Schlüsselword „class“ (übersetzt „Klasse“) deklariert, danach kommt der Name der Klasse: ‚class‘ [Name].
Klassen können jedoch auch von anderen Klassen erben, dies ist in Feder möglich, indem man einen bereits bekannten Klassennamen hinten anhängt: ‚class‘ [Name] ([Klasse von der diese erbt]).
Wenn man also eine Klasse mit dem Namen „Person“ erstellen will, würde man Folgendes tun:

```
class Person
;
```

Die „;“ schließen die Definition der Klasse ab. Jedoch besitzt Feder auch die Möglichkeit, dass man die Klasse auch weiter definieren kann:

```
class Person
	func method0
	;
;

# Nochmals etwas hinzufügen
class Person
	func method1
	;
;
```

Man kann überall, also in jeder Quelltextdatei, Funktionen zu einer Klasse hinzufügen, jedoch ist es nur in der Datei möglich, wo die Klasse zuerst deklariert wurde, weitere Objekte der Klasse anzufügen. Dies funktioniert folgendermaßen (eigentlich genauso wie oben):

```
class Person
	String name
;

# Nur möglich in der Datei,
# wo die Klasse Person deklariert wurde !
class Person
	int32 age # Objekt age wurde der Klasse
```

```
                    # „Person“ angefügt
;
```

2.2.5. Objekte

In Feder gibt es drei grundlegend verschiedene Arten von Objekten: Objekte die mithilfe einer Klasse initialisiert wurden, Objekte die mit einer Funktion zugewiesen wurden und eine Schnittstelle als Datentyp haben oder Objekte die einen primitiven Datentypen vertreten.

Zuerst werden die Objekte beschrieben, die als Wert eine Funktion haben können. Solche Objekte können nur statisch sein, können sich also nicht automatisch dem Typen nach „=“ anpassen oder den Typen ändern. Der Typ von diesen Objekten wird durch ein Interface festgelegt: [Schnittstelle] [Name]. Einem solchen Objekt können auch nur Funktionen zugewiesen werden, die genau zu der Schnittstelle passen, d.h. Argumente und Rückgabetyp müssen gleich sein. Im Folgenden wird die Funktion „sync_frame (int n)“ erstellt und eine passende Schnittstelle „sync (int n)“. Danach wird ein Objekt „sync_fn“ erstellt, das als Typ „sync“ haben soll und als Wert „sync_frame“.

```
# Funktion sync_frame (int32 n)
func sync_frame (int32 n)
;

# Interface sync (int32 n)
interface sync (int32 n)

# Objekt „sync_fn“
sync sync_fn = sync_frame
# Objekt „sync_fn“ aufrufen als Funktion
sync_fn (100)
```

Die andere Art von Objekt, also die, die Daten speichern kann, kann statisch oder dynamisch sein. Ein Objekt wird erstellt, wenn es zum ersten Mal mit einem Typen deklariert oder zum ersten Mal einem Typen zugewiesen wird.

```
# Erstes Beispiel: Objekt wird mit Typen erstellt
# => statisches Objekt
String s0
# Oder:
String s1 = "" # Der Typ wird überprüft andere
                  # Werte vom Typen ‚int32‘ zuzuweisen
                  # ist nicht zulässig

# Zweites Beispiel: Objekt wird durch Zuweisung erstellt
# Dies funktioniert nur bei Objekten, die mit einer Klasse erstellt
# wurden
# => dynamisches Objekt
number1 = Integer32.set(0)
# Typen des Objekts ändern
number1 = "Ich bin eine Nummer"
```

Wenn man ein Objekt, das durch einen Datentypen beschrieben wird, zum ersten mal nennt, kann man darauf verzichten, den Typen vor der Nennung zu schreiben (also „n = 0“ statt „int32 n = 0“), wenn man ein Zuweisungszeichen verwendet, weil dieser automatisch erkannt wird. Jedoch ist n, weil es sich um einen primitiven Datentypen handelt, kein dynamisches Objekt, sondern ein statisches.

2.3. Ein Beispielprogramm

Nun wird noch kurz ein kleines Programm vorgestellt, das in der Sprache Feder programmiert wurde. Benutzt wurde die „Feder Standardbibliothek“ und natürlich „jfederc“, der Feder Compiler, um das Programm auszutesten.

Das Beispielprogramm soll Fahrenheit zu Celsius umrechnen. Den Eingabewert „Fahrenheit“ muss man als Argument dem Programm übergeben.

```
include "stdio.fd"

if !isEqual (args.length (), 2)
      # Es wurde erwartet, dass die Länge von „args“ 2 ist
      # Index 0 ist die ausgeführte Datei und alle Indices
      # die einen größeren Wert als 0 haben, sind dem Programm
      # als Argument übergeben worden
      io.print ("Error: Expected: ")
      io.print (String from args.at(0))
      io.println (" [double]")

      # „Falsch“ zurückgeben, da ein Fehler entdeckt wurde
      return false
;

# Nun wird das Argument bei Index 1 zu einem „double“
# konvertiert
fahrenheit = (String from args.at (1)).todouble ()

# Nun wird die Formel für Fahrenheit zu Celsius angewendet
# In Feder wird (momentan) die „Punkt vor Strich“ - Regel
# nicht beachtet
celsius = (fahrenheit - 32.0) * (5.0 / 9.0)

# Die Fahrenheit ausgeben
io.print (fahrenheit)
# Ein wenig Information
io.print (" °F convert to °C are: ")
# Und nun die Celsius ausgeben
io.println (celsius)

# Keine Fehler sind im Programm aufgekommen! Wahr zurückgeben
return true
```

3. Der Compiler

Damit man auch Programme ausführen kann, die mit der Programmiersprache Feder geschrieben wurden, benötigt es ein anderes Programm, das entweder die Sprache interpretiert oder kompiliert. Feder wird kompiliert, damit es in der Laufzeit „schneller läuft“. Der Compiler für Feder wurde in Java (Version 8) geschrieben und wurde „jfederc“ (Java Feder Compiler) genannt. Der Kompilierprozess funktioniert folgendermaßen: Der Quelltext wird durch eine lexikalische Analyse laufen gelassen, danach durch eine Syntaxanalyse, um zu prüfen, ob grobe grammatikalische Fehler vorliegen. Danach wird versucht Feder zu C zu übersetzen. Dieser Prozess prüft auch auf Syntaxfehler, die bei der ersten Syntaxanalyse nicht auffindbar waren (z.B. Objekt soll Klasse eines anderen Objekts sein, ist nicht zulässig, kann jedoch nicht von der groben Syntaxanalyse erkannt werden). Der Compiler ist ein top-down Compiler, der auch für Operatoren die „operator-precedence parse“ - Methode benutzt, um zum Beispiel „Punkt vor Strich“ zu können.

Problematisch bei der Compilerentwicklung ist z.B. undefiniertes Verhalten, also Quelltext, der nicht vom der Sprache definiert ist, jedoch vom Compiler erlaubt wird. Auch könnte ein Compiler den Quelltext falsch übersetzen, so würde zwar der Quelltext richtig sein, jedoch der neue Quelltext nicht. Fatal wäre auch, wenn der Compiler einen Fehler findet, der eigentlich nach dem Standard der Sprache kein Fehler ist. Dies sollte jedoch nicht passieren, so geschieht es eher, dass zu viel erlaubt wird und wie oben genannt, undefiniertes Verhalten entsteht.

4. Tests

Der Compiler, kompilierte Programme und Bibliotheken wurden auch getestet. Tests waren besonders nötig, um Feders Garbage Collector zu testen. Die freie Software „valgrind“ war hierbei besonders hilfreich, vor allem das eingebaute „memtest“ Werkzeug, das „allocs“ und „frees“ zählt und ausgibt, ob alles ordnungsgemäß entfernt wurde. Zudem können Speicherfehler (z.B. wenn ein bereits gelöschtes Objekt gelöscht werden soll), damit zurückverfolgt und eventuell behoben werden.

5. Ein Feder – Programmierer werden

Das Ziel dieses Abschnitt ist es, grundlegende Prinzipien von Feder zu zeigen, also wie man Funktionen, Klassen oder Schnittstellen erstellt und aufruft.

Kommen wir zunächst zu einer wichtigen Operation, damit man gewonnene Resultate im Terminal (bzw. im Standard Output, C: „stdout“) ausgeben kann.

```
io.print (x)
# und
io.println (x)
```

Dies ist eine Funktion, die in „stdio.fd“ verfügbar ist (genauer: in „io/console.fd“). Die Funktion wurde in dem Namensraum „io“ (Input/Output) erstellt und kann dadurch in diesem aufgerufen werden. Das Argument „x“ kann hierbei unterschiedlichen Typs sein: (u)int(32|64), double, byte oder String. Die zweite genannte Funktion besitzt ein „ln“ am Ende des Namens. Dies zeigt, dass bei dieser Funktion auch eine neue Zeile angefangen wird.

Nun wird eine eigene Funktion erstellt, diese heißt „summe“ und hat zwei Argumente mit dem Typen „int32“ und gibt „int32“ zurück (wird genauer in 2.2.2 und 2.2.5 erklärt).

```
int32 func summe (int32 n0, int32 n1)
      return n0 + n1
;
```

Wir rufen nun die Funktion auf, speichern ihren Rückgabewert in einem Objekt (primitiver Datentyp) und geben diesen Wert in der Konsole aus (Objekte werden in 2.2.5 genauer erläutert).

```
resultat = summe (5, 7)
io.println (resultat)
```

Nun werden wir eine Klasse Namens „Objekt“ erstellen (mehr in 2.2.4). Diese Klasse wird ein Attribut mit dem Namen „alter“ und Typen „int32“ haben. In der Klasse wird eine Funktion Namens „alterAusgeben“ erstellt, die keine Argumente und keinen Rückgabetypen hat.

```
class Objekt
      int32 alter

      func alterAusgeben
            io.println (alter)
```

```
        ;
    ;
```

Auch erstellen wir eine Klasse Namens „Person“. Diese Klasse erbt von der Klasse „Objekt“ (mehr in 2.1.10). Sie wird ein Attribut mit dem Namen „name“ und dem Typen „String“ besitzen. Die Klasse wird auch eine Funktion besitzen, die den Namen „neu“ trägt. Die Argumente dieser Funktion sind ein „int32“ (das Alter) und ein „String“ (der Name). Die Funktion gibt einen „Person“ - Typen zurück.

```
class Person Objekt
    String name

    Person func neu (int32 alter0, String name0)
        alter = alter0
        name = name0
        return this
    ;
;
```

Nun erstellen wird eine Person und speichern die erstellte Person in einem Objekt. Danach rufen wir die Funktion „neu“ der Klasse Person auf (durch das erstellte Objekt).

```
p0 = Person
p0.neu (18, "Fionn Langhans")
```

In Feder kann ein Objekt, je nach Kontext, nur durch Nennung einer Klasse initialisiert werden. Dies könnte auch mit weniger Quelltext erreicht werden:

```
p0 = Person.neu (18, "Fionn Langhans")
```

Um die Funktion alterAusgeben von dem Objekt „p0“, das als Typen „Person“ besitzt, aufzurufen, würde man dies schreiben:

```
p0.alterAusgeben ()
```

So wie hier „p0“ erstellt wurde, wird es als dynamisches Objekt gehandhabt. Wenn man jedoch ein statisches haben will, müsste man bei der ersten Nennung des Objekts, den Typen vor der Nennung schreiben (siehe 2.2.5):

```
Person p0 = Person
```

6. Die Standardbibliothek

Es war auch nötig, eine kleine Standardbibliothek zu schreiben, die grundlegende Operationen erlaubt. In Feder ist dies besonders nötig, da Typen, wie z.B. int32, double oder String nicht vordefiniert worden sind, nicht einmal der Name ist gegeben. Dies ermöglicht es aber auch unterschiedlichste Standardbibliotheken, so kann zum einem „int32“ ein primitiver Datentyp sein oder auch eine Klasse.

Die Standardbibliothek teilt sich auf verschiedene Dateien auf:

stdc.fd: Grundlegende Operationen, die in C vordefiniert sein müssen und die Klasse „object“
stdio.fd: Einfache Input/Output Operationen, wie z.B. Dateisystem oder das Terminal
stdmath.fd: Ein paar mathematische Operationen
stdsystem.fd: Operationen des Systems, wie z.B. Laufzeit, Zugriff auf den Befehlsprozessor

(Auf allen Betriebssystem gleich)
stdtype.fd: Grundlegende Typen, wie z.B. [u]int[16|32|64], double, byte, Array, List, String

7. Abschließende Bemerkungen

Die Ziele, die das Projekt hatte, konnten zum großen Teil erreicht werden. Die Programmiersprache konnte jedoch nicht „sehr einfach“ gemacht werden (sondern nur „einfach“), da sie auch noch kompilierbar sein und auch möglichst wenig Laufzeitressourcen verbrauchen soll. Dies war jedoch schon am Anfang klar, also kein unerwartetes Problem.

7.1. Probleme der Programmiersprache & des Compilers

Eines der Probleme der Sprache (bzw. eigentlich der momentan vorhandenen, programmierten Ressourcen) ist, dass die Sprache nur eine kleine, eigene Bibliothek besitzt und es so noch ein großer Aufwand wäre, z.B. Musik abzuspielen oder Ähnliches zu machen, da man selber Bibliotheken dafür entwickeln müsste.
Der Compiler sollte bei manchen Quelltexten Fehler ausgeben, tut dies jedoch nicht immer und lässt invalide Syntax manchmal zu. Zudem gibt es ein Problem mit der Fehlerausgabe: Die Position des Fehlers wird nicht genau gezeigt.

7.2. Weitere Entwicklung

Ich werde weiter an der Entwicklung meiner Programmiersprache arbeiten, so müssen noch verschiedene Konzepte, die hier nicht aufgeführt sind, verwirklicht werden. Außerdem werde ich noch einen Paketverwalter (package manager), wie pip [9] (Pythons Paketverwalter) oder Ähnliches erstellen. Dazu muss natürlich die Modulfähigkeit der Sprache gesteigert werden (z.B. ein „import“ Operator ?). Denkbar wäre auch, den „Zwischenschritt“ zu C zu übersetzen auszulassen und stattdessen llvm [5] zu verwenden.

7.3. Persönlicher Rückblick

Wie bei jedem Programmierprojekt, das ich angefangen habe, neige ich dazu, kaum Kommentare zum Quellcode zu schreiben und ich musste mich förmlich zwingen, immer wieder mal den Code zu kommentieren, um seinen Zweck jedermann mitzuteilen. Jedoch hat mich das Resultat, angesichts der kurzen Zeit (~ 4 Monate), sehr zufrieden gestellt.

Abschließend würde ich hier gerne meinem Betreuungslehrer Herrn Stocker danken, der mich erst auf die Idee gebracht hat, bei Jugend Forscht mitzumachen und zudem auch meiner Mutter, die mir geholfen hat, sprachliche Fehler der Arbeit zu korrigieren.

8. Literaturverzeichnis

[1]: Heinz-Peter Gumm, Manfred Sommer: Einführung in die Informatik, 2013; 10. Auflage
[2]: The Open Group Base Specification Issue 7: 9. Chapter: Regular Expressions; IEEE Std 1003.1-2008, 2016 Edition; Aufrufbar im Internet, http://pubs.opengroup.org/onlinepubs/009696899/basedefs/xbd_chap09.html. Abgerufen am 15.01.2018.
[3]: http://valgrind.org/. Abgerufen am 05.01.2018.
[4]: https://gcc.gnu.org/. Abgerufen am 05.01.2018.
[5]: https://clang.llvm.org/. Abgerufen am 05.01.2018.
[6]: https://de.wikipedia.org/wiki/Pr%C3%A4prozessor. Abgerufen am 15.01.2018.
[7]: https://de.wikipedia.org/wiki/Awk. Abgerufen am 18.01.2018.
[8]: https://de.wikipedia.org/wiki/Funktion_h%C3%B6herer_Ordnung. Abgerufen am 05.12.2018.
[9]: https://pypi.python.org/pypi/pip. Abgerufen am 18.01.2018.