1 Benutzerdokumentation

Die Benutzerdokumentation soll eine allgemeine Anleitung zur Verwendung von Sunset aus der Sicht eines Benutzers liefern und allfällige Fragen beantworten. Deshalb ist sie in zwei Teile geteilt. Einmal der allgemeine Vorgang zum Erstellen eines Programmes innerhalb Sunset und einmal eine FAQ-Sektion.

1.1 Vorgang zum Erstellen eines Programmes mit Sunset

1. Begin des Programmes mit
   ```
   program NAME {
   ```
2. Definition von Konstanten nach folgendem Schema
   ```
   const varName: Type := val;
   ```
3. optional deklarieren eigener Funktionen/Prozeduren
   ```
   function FuncName (varName : Type; weitere Variable): ReturnType {...}
   procedure ProcName (varName : Type; weitere Variable){...}
   ```
4. Definition von Variablen
   varName,[weitere Variablen] : Type;
5. festlegen der Variablenwerte und Programmlogik dannach
6. Abschluss des Programmes mit
   }

Syntax für setzen eines Variablenwertes varName := val;

1.2 Frequently Asked Questions (FAQ)

Frage: Kann eine Funktion mehrere Rückgabewerte liefern?
Antwort: Nein, Sunset beschränkt die Anzahl der Rückgabewerte von Funktionen auf genau einen um BestPractices zu forcieren.

Frage: Welche Programmierkonstrukte werden unterstützt?
Antwort: Unterstützt werden, `if else`; `for to [step]`; `while`; `break` und `new`; ;

Frage: Welche Operatoren werden unterstützt?
Antwort: Neben den Logischen Operatoren `AND` und `OR` werden die gängigen Vergleichsoperatoren, Grundrechenarten, ein mathematisch Korrekter Modulo `MOD`, der HashTag `#` als Operator für die Abfrage der Arraylänge und der Punkt um auf Record Elemente zuzugreifen unterstützt.

Frage: Wie kann ich `Z(p)[x]` verwenden?
Antwort: Man verwendet es in dem man gleich bei der Deklarierung angibt, in welcher Restklasse das Polynom liegt und bei der Initialisierung der Variable ein Polynom oder einen Integer zuweist.

Frage: Kann ich unterschiedliche Restklassen miteinander mulitplizieren / addieren / subtrahieren?
Antwort: Nein.

Frage: Was ist der Datentyp Record?
Antwort: Record ist ähnlich wie ein Struct in C und wird mittels Record und EndRecord deklariert. Zwischen Record und Endrecord stehen beliebige weitere Typausdrücke. In der aktuellen Version ist eine Verwendung des Datentyps Record allerdings nicht vorgesehen.

FRAGE: Was liefert die Funktion leadingCoefficient()?
ANTWORT: Diese Funktion liefert den Koeffizienten der Stelle mit dem höchsten Exponenten welcher ungleich 0 ist

FRAGE: Wie kann ich den Datentyp Polynom initialisieren?
ANTWORT: Die Initialisierung kann auf mehrere Arten erfolgen:

- mittels Zuweisung eines Integerwertes an das Polynom. Dieser kann auch in der Form `e^x` erfolgen.
- mittels Zuweisung eines Polynoms komplett in eckigen Klammern. Beispielsweise `[]x^3 + x^2 + x - 123]`
- mittels Zuweisung eines Polynoms mit nur **x** in Klammern `[x]^2 + [x]`

wichtig ist hier nur, dass das **x** in der eckigen Klammer steht und somit als unbestimmte Variable genutzt wird.

FRAGE: Kann ich Matrizen anlegen?
ANTWORT: Ja, Matrizen können über multidimensionale Arrays angelegt werden, wobei die Länge der Subarrays jeweils gleich sein muss.

FRAGE: Wie müssen Variablennamen (Konstantennamen) beschaffen sein?
ANTWORT: Variablennamen müssen immer mit einem Buchstaben (Groß/Klein) oder dem Unterstrich "_"beginnen, sonst gibt es hier keine Einschränkungen.

FRAGE: Kann ich die Länge einer Arraydimension bei der Deklarierung bestimmen?
ANTWORT: Nein, die Arraylänge wird erst bei der Initialisierung festgelegt.

FRAGE: Kann ich mir die ArrayLänge anzeigen lassen?
ANTWORT: Ja mittels dem # Operator welcher einfach vor das Array geschrieben wird. Natürlich gilt, das selbe auch für tiefere Dimensionen. Der Zugriff erfolgt hierbei gleich wie man es von Java gewohnt ist (z.B. #arr[0])

FRAGE: Kann ich Arrays miteinander multiplizieren?
ANTWORT: Nein, dass ist in der aktuellen Version nicht möglich.

FRAGE: Wie kann ich auf Elemente in SubArrays zugreifen?
ANTWORT: Normal wie du es von Java gewohnt bist.

FRAGE: Wie unterstützt Sunset Arrays?
ANTWORT: Obwohl Sunset an sich Arrays bis zu einer beliebigen Dimensionstiefe unterstützt, werden von den Methoden innerhalb Sunset nur Arrays mit einer maximalen Dimension von 4 unterstützt.

FRAGE: Kann ich bei for-Schleifen die Sprungweite verändern?
ANTWORT: Ja, dies ist sowohl bei aufsteigenden (zB. 0-20) als auch absteigenden(zB.: 20-0) Schleifen möglich. Hierzu gibt man optional `step` an, wobei dieser größer Null sein muss.

FRAGE: Akzeptiert Sunset unäre Operatoren wie i++?
ANTWORT: Nein, Sunset bzw. FFapl bieten derzeit keine Unterstützung für diese Operatoren an.

FRAGE: Was ist der Datentyp EC?
ANTWORT: EC repräsentiert eine Elliptische Kurve welche sich als Grundtyp aus einem Polynom oder einem Galois-Feld zusammensetzen kann und noch zusätzlich die Parameter a1-a6 ohne a5 benötigt. Beispiele für gültige / ungültige Deklarationen sind im nachfolgenden CodeFragment gelistet:

```
//Gültige Deklarationen
c1: EC(GF(2,[x^2+x+1]), a3 := [x+2], a4 := [x^3-4x+1], a6 := [5]);
c2: EC(GF(2,[x^2+x+1]), a4 := [x^3-4x+1], a6 := [5], a3 := [x+2]);
c3: EC(Z(13), a4 := 12, a6 := 4, a3 := -4);
c4: EC(Z(11), a4 := 10, a6 := 5, a3 := 14);

//Ungültige Deklarationen
c5: EC(GF(2,[x^2+1]), a4 := [x^3-4x+1], a6 := [5], a3 := [x+2]);
c6: EC(GF(2,[x^2+x+1]), a4 := [x^3-4x+1], a6 := 5, a3 := [x+2]);
c7: EC(Z(7), a2 := [4], a3 := 3, a6 := 5);
c8: EC(Z(10), a2 := 4, a3 := 3, a6 := 5);
```

Listing 1: Beispieldklarationen für Elliptische Kurven

FRAGE: Wie kann ich eine Elliptische Kurve Initialisieren
ANTWORT: Für das Initialisieren einer Elliptischen Kurve gibt man 2 Punkte an welche innerhalb « und » stehen und mit einem Komma getrennt sind, wobei die Punkte auf der Kurve liegen und dem Grundtyp entsprechen müssen. Hier gilt beispielsweise « PAI » immer PAI steht hierbei für Point At Infinity. Im folgenden CodeFragment sind 2 gültige Initialisierungen der gleichen Kurve gelistet:

```
  ec, ec2 : EC(Z(37), a4:=-5, a6:=8);
  ec := << 8,6 >>; //gueltiger Kurvenpunkt
  //ec := << 4,3 >>; Kurvenpunkt liegt nicht auf der Kurve
  //ec := << [x^2], [x^4] >>; Hier passt der Kurvenpunkt nicht zum Grundkörper
  ec2 := << PAI >>; //PAI liegt immer auf der Kurve
```

Listing 2: Beispielinitialisierungen für Elliptische Kurven

FRAGE: Welche Operationen sind mit Elliptischen Kurven möglich?
ANTWORT: Sie unterstützen Additionen mit anderen Elliptischen Kurven, sowie Multiplikation mit Integer werten.

FRAGE: Sind Negative Exponenten erlaubt?
ANTWORT: Nein, Negative Exponenten sind nicht erlaubt, da Sunset nur ganzzahlig arbeitet

FRAGE: Was liefert die Funktion Hash?
ANTWORT: Die Funktion Hash liefert einen Hash als Integer (genauer SHA-256) übergebenen Typs. Hier wird die Charakteristik bzw. der Typ mitgehasht. Was soviel heißt wie Z(100) := 40 liefert einen anderen Wert als Integer := 40; um hier einen gleichen Hashwert zu erhalten müssen beide durch die str Methode gefiltert werden, da einfach 40 als String überbleibt.

2 Technische Dokumentation

2.1 Einbauen von Built-In Funktionen

Hier wird eine Schritt für Schritt Anleitung aufgebaut wie man Built-In Funktionen nachrüsten kann.

Built-In Funktionen finden sich alle im Java Package `ffapl.java.predefined.function` und implementieren das Interface `IPredfinedProcFunc` Beispiele für solche Funktionen sind CoefficientAt, ConvertToInteger und Print (Println).

IPredefinedFunc verlangt eine Prozedur `public void execute(IVm interpreter)throws FFaplAlgebraicException;` In dieser Methode werden die angegebenen Parameter beginnend beim letzten mittels `interpreter.popStack();` vom Interpreter Stack geholt. Die dabei erhaltenen Rückgabewerte sind vom Typ Object müssen demnach noch in den als Eingabeparameter spezifizierten Typ explizit umgewandelt, werden. Beispiel `BInteger x; x = (BInteger)interpreter.popStack();`, bei mehreren möglichen Typen eines Inputparameters muss man als Zwischenschritt, das Object in den Typ IJavaType umwandeln und danach mittels .typeID() auf den Typen prüfen und entsprechend die Aktionen setzen.

Nach der Berechnung des Rückgabewertes wird dieser mit `interpreter.popStack( Object o)` auf den Stack gelegt. Als Abschluss wird noch `return` mittels `interpreter.funcReturn();` simuliert. Vorausgesetzt es hat alles gepasst, ist damit die Prozedur execute abgehandelt. Falls allerdings ein Fehler auftritt muss die Exception noch geworfen werden und, wenn nötig, eine neuer Fehlertyp hinzugefügt werden. Wobei mit Fehlertyp nur eine neue Meldung gemeint ist. Beispielsweise wirft GCD eine Exception, falls einer der Eingabeparameter, bei Integer oder Prime, < 0 ist.

Die Klassen, welche die Built-In Funktionen beinhalten haben zusätzlich durch die vom Interface geforderte Methode noch eine Methode `public static void registerProcFunc(ISymbolTable symbolTable)throws FFaplException` bzw. im Fall von Print eine Zusatzoption Logger. Wobei die FFaplException nicht innerhalb der Built-In Funktion geworfen wird sondern nur weitergeleitet wird und diese Exceptions Compiler Fehler darstellen. Diese Methode ist dafür zuständig, das neue Funktionssymbol an der API anzumelden. Bei vielen unterstützten Datentypen empfiehlt es sich, die Anmeldung in ein eigene private Methoden zu kapseln und hier nur die privaten Methoden aufzurufen. Dies geschieht mit symbolTable.addSymbol(ISymbol s).

```
1  /**
2   * Registers predefined Function in Symbol table
3   * @param symbolTable
4   * @throws FFaplException
5   */
6  public static void registerProcFunc(ISymbolTable symbolTable)
7      throws FFaplException {
8    FFaplPreProcFuncSymbol s;
9    //isPrime(a)
10   s = new FFaplPreProcFuncSymbol("isPrime",
11             null,
12             new FFaplBoolean(),
13             ISymbol.FUNCTION);
14   s.setProcFunc(new IsPrime());
15   symbolTable.addSymbol(s);
16     symbolTable.openScope(false);
17     //for Parameter
18     symbolTable.addSymbol(
19        new FFaplSymbol("_t1",
20             null,
21             new FFaplInteger(),
22             ISymbol.PARAMETER));
```

```
23      symbolTable.closeScope();
24  }
```

Listing 3: Beispielimplementation der Methode registerProcFunc label

Dieses Listing aus der Klasse IsPrime im Packet ffapl.java.predefined.function zeigt eine mögliche Implementation der Funktion, welche nun genauer erklärt wird. Zeilen 10 bis 13 sind die ersten Interessanten. In Zeile 10 wird ein neues PreProcFuncSymbol angelegt. Der erste Parameter ist der Name der Funktion, wie er in Sunset angesprochen wird, hier somit isPrime. Der nächste Parameter ist ein Token, welcher üblicherweise `null` bleibt. Im Dritten Parameter wird der Typ des Rückgabewertes spezifiziert, in diesem Fall Boolean, wobei hier die von der FFaplAPI bereitgestellten Wrapperklassen zu verwenden sind. Der letzte Parameter gibt an von welchem Typ das neue Symbol ist. Erlaubte Werte sind PROGRAM, PROCEDURE, FUNCTION, VARIABLE, CONSTANT, PARAMETER, BLOCK. Relevant sind für das hinzufügen von Built-In Funktionen natürlich nur PROCEDURE, FUNCTION. Wobei Function einen Rückgabewert liefert und Prozedur keinen und ein möglicherweise angegebener Rückgabewert einfach ignoriert wird, falls der Typ auf Procedur gesetzt wurde.

In Zeile 11 wird das Funktionssymbol zur symbolTable hinzugefügt.

Zeile 12 spezifiziert, dass der Scope der nachfolgenden Symbole nicht global ist bis der Scope der Symbol Table geschlossen wird. In der Regel wird man dies auch immer auf `false` setzen. In diesem Fall heißt das, dass der definierte Parameter nur für die Funktion selbst gültig ist, da der Scope erst nach hinzufügen des Parameters in Zeile 23 geschlossen wird.

In den Zeilen 18 bis 22 wird ein neuer Parameter für die Methode angelegt, welcher den Typ Integer besitzt und den Symboltyp Parameter besitzt. Solange der Scope nicht geschlossen ist können so neue Parameter hinzugefügt werden. Sunset unterstützt das Überladen von Funktionen insofern, dass für jede Parameterkonfiguration dieser Vorgang wiederholt werden muss.

Nachdem das abgeschlossen ist, muss die Prozedur registerProcFunc nur noch in der Klasse FFaplPredefinedProcFuncDeclaration im Package ffapl.lib über den statischen Aufruf ausgeführt werden und einem ersten Testlauf steht nichts mehr im Weg.

Achtung! diese Funktion taucht nach diesen Schritt noch nicht in der API Dokumentation auf der Seite auf, dazu sind weitere Schritte nötig.

Und zwar muss damit die Funktion dort auftaucht die Datei äpi.xml" im Package sunset.gui.api.xml erweitert werden.

```
1   <ff:function>
2       <ff:name>isPrime</ff:name>
3       <ff:description>DESC_FUNC_ISPRIME</ff:description>
4       <ff:regex>isPrime</ff:regex>
5       <ff:parameterList>
6         <ff:parameter>
7           <ff:name>a</ff:name>
8           <ff:type>Integer</ff:type>
9         </ff:parameter>
10      </ff:parameterList>
11      <ff:returnType>Boolean</ff:returnType>
12    </ff:function>
```

Listing 4: Funktionsdefinition in api.xml

Der XML-Tag description beinhaltet hierbei eine Konstante welche je nach Spracheinstellung ersetzt wird. Diese Konstante setzt man in den .properties Datei-

en im package sunset.bundles. Wobei _ de für deutsch _ en für englisch steht und die dritte Datei als Rückfalllösung genutzt wird, welche wiederum Deutsch ist.

Der Tag regex definiert, welcher Text im Editorfenster kursiv angezeigt wird und somit als vordefinierte Funktion hervorgehoben wird. Parameterlist definiert die möglichen Parameter mit Namen, wobei pro Parameter mehrere Typen erlaubt sind. Am Ende kommt noch der returnType.

Built-In Funktion Hash Bei der Funktion hash, trat ein kleines Problem mit immer negativen Hashwerten auf. Da Hashing immer ein ByteArray zurückliefert und Java intern, bytes immer signed nimmt und somit nur werte zwischen -128 und 127 zulässt, was natürlich bei zufälligen Bytefolgen nicht positiv zu bewerten ist. Die derzeit verwendete Methode um dies zu verhindern ist, den intern verwendeten BigInteger mit dem „Vorzeichenkonstruktor“auf Positiv zu setzen.

Die eigentliche Hashfunktion wird nur mittels Strings angesprochen und aus diesem mit der Kodierung UTF-8 bzw. im Fehlerfall mit der Standardcodierung des Systems ein ByteArray als Input für die in den Java.Security Klassen verwendete Hash Funktion SHA-256 übergibt. Um daraus den Hashwert zu berechnen, welcher als BInteger zurückgegeben wird. Die HashFunktion akzeptiert ähnlich wie str und print alle Datentypen als Input und liefert einen dazu passenden Hashwert. Wobei bei allen Datentypen außer String derzeit zusätzlich noch die ClassInfo mit aufgerufen wird. Somit ist der Hashwert von Z(6) ein anderer als der Hashwert von str(Z(6)), welches auch so beabsichtigt ist. Nur RandomNumbers sind ausgenommen, da diese für eine Hashfunktion keinen Sinn machen.

2.2 Fehlerquellen

In diesem Kapitel wird auf mögliche Fehlerquellen und Lösungen eingegangen.

Typkompabilitäten Aufzupassen ist bei den Typkompabilitäten insbesondere im Hinblick auf die Rückgabewerte und Parameter. Das heißt man muss immer die eigens für FFapl angelegten Datentypen verwenden.

Unterschieden wird zwischen den Typen in ffapl.types und ffapl.java.classes da es hier sonst zu Fehler kommt. Ein Beispiel für die Kompatibilität liefert String. Als Rückgabewert in der Symbolerstellung muss hier FFaplString benutzt werden und innerhalb execute hat ein JString auf dem Stack zu landen.

Arrays Arrays lassen sich bis Dimension 2 implizit und explizit deklarieren und initialisieren. Darüber hinaus gehende Dimensionen lassen sich Fehlerfrei derzeit nur implizit, mittels `new` initialisieren. Nachfolgend eine Liste der auftretenden Fehler:

Array Dimension 4

```
a : Integer[][][][];
a := {{{{4}}}}; // führt zu
//FFapl Kompilierung: [calculate] CompilerError 213 (Zeile 4, Spalte 14)
//Impliziter Cast von Integer[] zu Integer ist nicht möglich!
```

Array Dimension 3

```
a : Integer[][][];
a := {{{4}}};
//FFapl Kompilierung: [calculate] CompilerError 208 (Zeile 4, Spalte 11)
//Typ Diskrepanz in Zuweisung: -> Integer[][][] := Integer[][][][]
```

Der Zusammenhang zwischen diesen Fehlern ist mir noch nicht klar. Der Parser kann, denke ich, ausgeschlossen werden, da es hier vorher bereits zu einem Fehler kommen müsste und er die Klammern in der Richtigen Menge zurückliefert. Außerdem erkennt er die Dimension richtig, erst bei der Abarbeitung der einzelnen Symbole werden die Dimensionen falsch erkannt bzw. falsch gesetzt. Bei dem Versuch Lösungen für das Problem zu entwickeln, stoße ich derzeit immer auf die selben bzw. ähnliche Probleme. Hier wird jedesmal ein Fehler bezüglich der Typkonvertierung gemacht. Und die Probleme bleiben bestehen. Dies könnte ein Hinweis dafür sein, dass ich an der Falschen Stelle im Programm suche. Wobei die gemachten Änderungen (fast) immer darin resultieren, das Sunset mit der Fehlermeldung

```
Impliziter Cast von Integer zu Integer[] ist nicht möglich!
```

endet. Somit besteht, das eigentliche Problem noch und wird nur verlagert. Eine Änderung des LOOKAHEADS in ffaplParser.jj brachte keine Verbesserung, ebensowenig wie eine Änderung des Amounts in ASTArrayType ArrayType() von 3 auf 10 (Ebenfalls in der Datei ffaplParser.jj).

Ursprünglich trat bei expliziter Angabe überhaupt eine ClassCastException in Array.java Zeile 161 auf, da eine Objektkonvertierung nicht möglich war, dies wurde durch einen zusätzlichen Methodenaufruf behoben.

```
1 if(tmp instanceof Array){   //diese if-Anweisung wurde hinzugefügt
2   tmp = ((Array)tmp).getArray();
3 }
4 result = new Array(_baseType, _dim - pos.size(), (Object[]) tmp, _jType, _thread); //hier
      trat der Fehler auf
```

Primzahlen und Integer Wenn eine Built-In Funktion mit Unterstützung für Primzahlen und Integer implementiert werden soll so muss nur auf die Unterstützung für Integer geachtet werden, da bei mehrmaligem Aufbau die Funktion als bereits deklariert gilt. Wenn es allerdings zur Unterstützung von Arrays der beiden Typen kommt müssen diese getrennt behandelt werden.

Elliptic Curve Zu Beginn lies sich print bzw. println nicht, wie bei den anderen Datentypen zu diesen kombiniert mit + als String an print zu übergeben. Beispiel:

```
1 rc : Z(6);
2 ec : EC(GF(2,[x^2+x+1]), a3 := [x+2], a4 := [x^3-4x+1], a6 := [5]);
3 rc := 3^5;
4 ec := << PAI >>;
5
6 println("Restklasse " + rc); //möglich
7 println("Elliptic Curve " + ec); //nicht möglich
```

Listing 5: Elliptische Kurve und String Konkatenation

So ein Fehler wird damit gelöst indem man in der Klasse FFaplTypeCrossTable die OP2 Kompabilität für String und den Entsprechenden Datentyp in **beiden** Zeilen auf true setzt. Damit wäre das gelöst, falls man zusätzlich noch eine Charakateristik hat und diese hier mit angegeben werden soll ist es nötig in der Klasse FFaplVM in der privaten Funktion castTo den Fall FFaplCrossTable.FFAPLSTRING (ca. Zeile 1966 aktuell) anzupassen.

Obfuscator Verwendet wurde der Obfuscator Proguard, welcher empfohlen wurde, obwohl es Startschwierigkeiten gab, wurden diese erfolgreich überwunden. Der Große Fehler, war die Änderung der Klassensignatur und, dass nicht die richtigen Dateien behalten wurden. Hier kam es beispielsweise zu einer ClassCast Exception, mit der aktuellen Function list und dem XML-Parser, welcher im unobfuscated Code nicht auftritt, die Exception sieht wie folgt aus: java.lang.ClassCastException: com.sun.org.apache.xerces.internal.dom. ElementNSImpl cannot be cast to sunset.gui.api.jaxb.Function. Die Lösung dafür ist die Option -keepattributes signatures welches die Exception auflöst und den Code ausführbar hält. Aufzupassen ist bei den ResourceFiles insbesondere den .properties Dateien, bei diesen habe ich noch keine Möglichkeit gefunden, sie auch zu Verschleiern, womit Sie noch lesbar bleiben. Allerdings werden sonst Resource Files insbesondere in ffapl.bundles entfernt, was den Code zwar ausführbar hält, allerdings keine FFaplCompilermeldungen mehr anzeigt und eine ResourceNotFound Exception wirft. Dies wurde derzeit so gelöst, das diese Dateien einfach unberührt bleiben sollen. Interessant ist auch, dass die Klassen im Unterpackage jaxb erhalten bleiben müssen, da sonst das Verschleiern im Allgemeinen nicht funktioniert. Was nicht, auch laut Proguard FAQ, funktioniert ist die Verschlüsselung von String Konstanten, was diese immer lesbar macht.

Eine Frage ist noch, ob man nicht daran denken sollte inkrementell zu Verschleiern, da man damit relativ einfach Patches nachschießen kann. Was Obfuscation nicht kann, ist, dass der Code wirklich immer unverständlich bleibt, der Obfuscated Code hält nur den Normaluser (und voraussichtlich die meisten Studenten) davon ab den Code auf Anhieb zu verstehen. Wenn sich jemand lange genug, mit dem Obfuscated Code beschäftigt wird er dennoch hinter die Funktionsweise des Programmes kommen.

Read Methoden Unter read Methoden werden in diesem Kontext jene Methoden verstanden, welche ähnlich wie bei einem rein konsolenbasierten Programm Werte während der Ausführung einlesen und diese weitergeben an das laufende Programm. In diesem Fall, demnach Werte aus dem unteren Panel lesen, welches bis dato nur für Fehlermeldungen und vom FFapl Nutzer gewollte Ausgaben da war. Diese Methoden müssen für jeden Datentyp extra geschrieben werden, damit der jeweilige Rückgabewert gleich passend ist, da FFapl keine Typumwandlung im Sinne von anderen Programmiersprachen kennt. Außerdem müssen diese Methoden mit der unteren Konsole interagieren und an den spezifizierten Stellen auf Inputs warten. Somit müssten Sie ähnlich wie die Print Methoden einen zusätzlichen "Logger"mit übergeben bekommen. Ein weiterer zu Beachtender Punkt, ist bei diesen Methoden, dass die eingelesenen Werte an die SymbolTable übergeben werden müssen damit sie weiterverwendet werden können. Die „Symbole“ sollten Zuweisung an Ihren Typ auf Richtigkeit überprüft werden, wobei dies implizit über die FFaplKlassen passieren kann. Bei einem Fehler muss auch mit der gleichen Fehlermeldung abgebrochen werden, als wäre die Initialisierung im Code geschehen. Der Zugriff auf den FFaplConsoler handler bzw. dessen Console scheint schwierig ohne allzu viele Änderungen im Code nach sich zu ziehen. Das heißt bis jetzt habe ich noch keine Möglichkeit gefunden, dies zu erreichen. Eine andere leichter zu implementierende Lösung ist, den Input mittels Dialogen zu kreieren. Was allerdings den Nachteil mit sich bringt, dass die Benutzbarkeit besonders bei vielen Dialogen eingeschränkt wird und es auch sonst nicht in das Konzept des Sunset Compilers passt, da die Eingaben sonst alle über die Bereitgestellten Konsolen laufen. Der Vorteil hierbei ist es, das

kaum Änderungen am restlichen Code vorgenommen werden müssen und sich die Änderungen lokal beschränken. Da Java Dialogs nur Strings zurückliefern muss bei read-Funktionen anders als String auf die Interne Umwandelbarkeit geachtet werden. Was bei beiden Varianten einfach umsetzbar ist, ist die Unterscheidung des Rückgabetypes somit können alle readFunktionen innerhalb einer einzigen Klasse beispielsweise ReadFunctions.java abgearbeitet werden und es muss nicht für jede dieser Funktionen eine eigene Klasse angelegt werden. Der Abbruch ist eine Falsche Zuweisung wenn beispielsweise eine Integervariable mit readString befüllt werden soll, welche einen String zurückliefert. Die ReadMethoden müssen Zugriff auf die aktuelle symbolTable insbesondere im Hinblick auf die Zuweisungen von komplexen Typen bekommen bzw. direkt mit dem Parser interagieren. Hier geht es darum, dass beispielsweise bei einer Zuweisung von

```
1  h1 : Z(4);
2  h1 := readResidue();
3  println(h1);
```

muss (sollte) der Modulo der Restklasse bekannt sein, da man sonst eine neue Restklasse erzeugt und bei bekanntem Symbol bzw. in diesem Fall modulo des Symbols der Wert richtig gesetzt werden kann. Bei Restklassen lässt sich das noch relativ einfach realisieren indem man den Modulo Parameter mit angibt welcher, für die Zuweisung genutzt wird. Oder man weißt der Restklasse gleich nur einen reinen Integer wert zu. Bei komplexeren Typen wie EC ist keiner der Ansätze praktikabel bzw. durchführbar und hier muss etwas anderes versucht werden. Da eine simple Wertzuweisung nicht möglich ist. Die Optimale Lösung wäre es einen beliebigen Ausdruck einzuhängen und den Rest vom vorhandenen Code übernehmen zu lassen. Das heißt es gehört so dargestellt, als ob an dieser Stelle nie etwas anderes gestanden hat. Bei dieser Lösung ist nur zu beachten, dass der Interpreter richtig reinitialisiert wird. Da der Konstruktor verständlicherweise nur einmal aufgerufen werden darf um nur einen Interpreter zu haben.

Derzeit funktionieren die readMethoden für die „primitiven“ Datentypen wie `String`, `Integer` (und der Subtyp `Prime`)und `Boolean`, da hier eine direkte Zuweisung möglich ist. Für die Zufallszahlengeneratoren haben solche Methoden keinen Sinn da diese ohnehin „zufällig“ agieren sollen. Das „Protokoll“ gibt derzeit primitiv einfach nur den zugewiesenen Wert aus. In Zukunft wäre es möglich, wenn, das zu zuweisende Symbol mit auszugeben , im obigen Beispiel h1, eine Ausgabe der Art symbol := Wert wäre wünschenswert. Da ein Polynom auch keine zu beachtende Charakteristik besitzt ist es auch hier möglich den Wert des Polynoms direkt einzulesen. Wobei in diesem Fall, muss der Input String noch auf Validität geparst werden und die Werte entsprechend gesetzt. Dh am Ende muss auch das Polynom gesetzt werden, das der Input vorgibt. Beispielsweise `[x^212 + x - 32];` muss genauso ankommen, als Polynom 212 Grades. Derzeit wird der Inputstring welcher (aktuell) von einem Java Input Dialog geliefert wird mit einem Regex (welcher leider noch nicht alle Fälle abdeckt), geparst und auf Gültigkeit geprüft. Falls gültig wird dieser an einen extra geschaffenen Konstruktor in der Klasse ffapl.java.classes.Polynomial übergeben. Dort wird das Polynom erstellt, indem der Inputstring in seine Bestandteile aufgeteilt wird. Hier muss auf das Vorzeichen bzw. den Operator geachtet werden da ja 3+-x äquivalent zu 3-x ist. Das Polynom wird anhand des bekannten Patterns aufgeteilt und in seine Bestandteile zerlegt um kleine Subpolynome zu erhalten, welche einfach aufaddiert werden um das gesamte Polynom zu erhalten. Die Umwandlung in ein Polynom hat noch Probleme mit dem Minus, da eine simple

Aufsplittung nicht Möglich ist ohne die Information zur Negativität zu verlieren. Die Frage nach dem Return Value von der Konsole ist auch noch nicht ganz geklärt. Da das Observable Pattern nur in eine Richtung funktioniert.

Andere Lösungsmöglichkeit Eine komplett andere Möglichkeit ist, die Funktionen bereits im Parser bekannt zu machen und die Console von dort aus anzusprechen, die Funktion müsste im Prinzip sich selbst als Symbol entfernen und an diese Stelle den vom User eingegebenen Ausdruck setzen, auf welchem weitergearbeitet wird. Der Punkt hier ist dem Parser klar zu machen, dass die Funktion einen Typ zurückliefert und, dass die Funktion selbst nicht zugewiesen wird sondern, der Rückgabewert. Das heißt es müsste gleich ablaufen wie bei den durch die Benutzer erstellten Funktionen, mit dem Unterschied, dass der Input zur Laufzeit bestimmt wird. Für diese Funktionsgruppe müsste eine zweite Inputquelle geschaffen werden, welche nur kurzfristig offen ist und den Input extra parst.

Als Hack wäre eventuell möglich diese Funktion als User definierte Funktion anzugeben um so in der SymbolTable zu landen. Das Problem hierbei ist die Angabe, was die Funktion leisten soll, welches sich als nicht lösbar erweisen könnte.

Für die normale Restklasse wurde das ganze derzeit, so gelöst, dass einfach nur ein Integerwert zurückgebeben wird und das selbe passiert wie bei readInt(). Somit kann derzeit keine neue Restklasse sondern nur noch ein wert zugewiesen werden, wie es verlangt wurde. Für fast alle Datentypen sind die ReadFunktionen implementiert und funktionieren halbwegs nach Wunsch. Bei der Umsetzung von Polynomen und den darauf aufbauenden Datentypen gibt es aktuell noch Probleme mit den Vorzeichen und der Korrekten Parsung bzw. Aufteilung des Input Strings. Bei der Elliptischen Kurve besteht jedoch weiterhin, dass Problem mit der Charakteristik, da ich nicht einfach eine Kurve erstellen kann und ihr 2 Punkte zuweise und eine Punktzuweisung ohne Charakteristik müsste auch fehlschlagen, weil der Kurvenpunkttest ins leere gehen muss. Für die anderen Datentypen ist mir die Charakteristik nicht wichtig, weil ich ohnehin nur eine Zuweisung eines simpleren Datentypes mache. Beispielsweise bei Z(x) weise ich ohnehin nur einen Integer zu, das selbe gilt für Z(p)[x], wobei ich hier entweder Integer oder ein Polynom zuweise, bzw. in beiden Fällen ein Polynom zuweise.