Arbeitsgruppe Programmiersprachen und Übersetzerkonstruktion
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Masterarbeit

Überprüfung von Stilrichtlinien für deklarative Programme

Katharina Rahf

Juli 2016

betreut von
Prof. Dr. Michael Hanus
M. Sc. Björn Peemöller

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe.

Kiel, 4. Juli 2016

Katharina Rahf

Zusammenfassung

Guter Programmierstil leistet einen wichtigen Beitrag zur Lesbarkeit und damit zur Wartbarkeit und Erweiterbarkeit von Programmen. Um einen universellen Standard zu schaffen, existieren für die meisten Programmiersprachen sogenannte Style Guides mit Richtlinien, die den bevorzugten Programmierstil beschreiben. Die automatische Überprüfung von Quelltext sowie die automatische Korrektur von dabei gefundenen Stilverletzungen nehmen einem Programmierer Arbeit ab.

In dieser Thesis wird ein Werkzeug zur Überprüfung von Stilrichtlinien mit eingebauter Fehlerkorrektur für die logisch-funktionale Programmiersprache Curry entwickelt. Dazu wird auf die Tokenfolge des zu überprüfenden Quelltextes und seinen abstrakten Syntaxbaum zugegriffen. Aus diesen beiden Datenstrukturen entsteht ein erweiterter abstrakter Syntaxbaum, der die Positionen eines jeden Tokens aus dem Quelltext enthält. Die Überprüfung funktioniert mittels prädikatenlogischer Validierungsfunktionen. Schlägt eine solche fehl, wird ein Hinweis auf die Stilverletzung mit Angabe von Position und Fehlerbeschreibung ausgegeben. Die zu überprüfenden Kriterien sowie die automatische Korrektur von Stilverletzungen lassen sich in einer Konfigurationsdatei ein- und ausschalten.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

1. Einleitung

Gute Lesbarkeit ist eine der wichtigsten Eigenschaften von geschriebenem Programmcode. Sie ist die Grundlage für die Wartbarkeit und Erweiterbarkeit von Programmen. Sogenannte Code-Konventionen oder Stilrichtlinien sorgen dafür, dass Menschen Programmcode besser lesen können. Durch übersichtlichen Code können auch Programmierfehler leichter vermieden werden. Für Compiler hingegen ist es nicht wichtig, wie der Code geschrieben ist, solange er syntaktisch korrekt ist.

1.1. Motivation

Ein Beispiel für die Bedeutung von Programmierstil ist in folgenden semantisch äquivalenten Curry-Programmen auf der linken und der rechten Seite zu sehen. Beide Seiten werden fehlerfrei kompiliert.

```
condAbs p x = if p  then  abs x
  else           (   x)



abs n
   | n <   0 = (( - n ) )
 | n    > 0 = n
     | otherwise   = 0
```

```
-- Compute absolute value of
-- input if predicate is True
condAbs :: Bool -> Int -> Int
condAbs p x = if p
                then abs x
                else x


-- Return absolute value of Int
abs :: Int -> Int
abs n
  | n < 0     = -n
  | n > 0     =  n
  | otherwise =  0
```

Listing 1.1.: Semantisch äquivalente Programme: links unleserlich, rechts lesbar

Auf der linken Seite erkennt ein menschlicher Leser auf den ersten Blick keine Strukturen. Alle Elemente sind völlig unformatiert und es gibt überflüssige Klammern. Auf der rechten Seite hingegen befinden sich Zusatzinformationen in Kommentaren und

Typsignaturen. Außerdem sind die Sprachkonstrukte so formatiert, dass ihre Bedeutung leicht durchschaubar ist.

In den meisten Programmiersprachen gibt es festgelegte Richtlinien, die bestimmen, wie „gut“ geschriebener Code auszusehen hat. Darunter kann beispielsweise die allgemeine Zeilenlänge fallen, oder auch die Formatierung von bestimmten Sprachkonstrukten, wie etwa die Einrückung von if-then-else-Ausdrücken. Oft gibt es verschiedene Möglichkeiten, Code im Sinne des Compilers syntaktisch korrekt aufzuschreiben und es ist Geschmackssache, welche Formatierung gewählt wird. Damit aber innerhalb eines Programms, das von einem Team verschiedener Programmierer entwickelt wird, der gleiche Programmierstil verwendet wird, ist die Festlegung eines gemeinsamen Standards nötig. Für die Programmiersprache Curry, mit der im Laufe dieser Thesis gearbeitet wird, existiert beispielsweise von der Arbeitsgruppe *Programmiersprachen und Übersetzerkonstruktion* der Christian-Albrechts-Universität zu Kiel ein Dokument mit Stilrichtlinien, an denen sich alle Entwickler orientieren sollten [14].

1.2. Anforderungen

Wie eingangs erwähnt, wird die Einhaltung von Stilrichtlinien nicht vom Compiler überprüft. Es wäre also für Programmierer angenehm, wenn eine Möglichkeit der automatisierten Überprüfung von Stilrichtlinien bestehen würde. Deshalb ist die Entwicklung eines Werkzeugs, das automatisiert die Einhaltung von Stilrichtlinien in deklarativem Programmcode überprüft, Gegenstand dieser Thesis. Konkret geht es um Stilüberprüfung bei der logisch-funktionalen Programmiersprache Curry. Ein Aufruf des Programmes soll für geschriebenen Curry-Code eine Reihe von Meldungen liefern, die auf Verletzungen der Stilrichtlinien aufmerksam machen, beispielsweise in der Form

```
line x, column y: check indentation of keywords!
```

Wünschenswert wäre auch eine Möglichkeit, diese Richtlinienverletzung gegebenenfalls automatisch beheben lassen zu können.

Weil Stilrichtlinien geändert und gegebenenfalls auch erweitert werden können, sollte das zu entwickelnde Tool das Einpflegen von Änderungen leicht ermöglichen. Dazu bietet sich eine Modularisierung an, die die Richtlinien von der ausführenden Logik trennt. Außerdem sollten die Richtlinien so einfach aufgeschrieben sein, dass jeder Nutzer Erweiterungen an ihnen vornehmen kann, ohne die genaue Implementierung des Werkzeugs zu kennen.

1.3. Überblick über diese Arbeit

Zunächst wird in Teil I ein Überblick über den aktuellen Stand von Möglichkeiten der Stilüberprüfung in deklarativen Sprachen gegeben. Es folgt eine Einführung in die logisch-funktionale Programmiersprache Curry, in deren Rahmen besonders auf wichtige Unterschiede zur funktionalen Programmiersprache Haskell eingegangen wird. Dies sind insbesondere der Nichtdeterminismus, Functional Pattern Matching, und freie Variablen. Anschließend werden die Curry-Systeme PAKCS und KiCS2 vorgestellt. Zuletzt folgen einige Grundlagen über Token und den abstrakten Syntaxbaum, die zum Verständnis der allgemeinen Vorgehensweise in dieser Arbeit wichtig sind.

Teil II behandelt den praktischen Teil der Arbeit – die Implementierung des entwickelten Werkzeugs. Nach Kapitel 4, das Anforderungen und den Entwurf des Systems enthält, folgen die beiden Hauptabschnitte der Implementierung. Zunächst wird in Kapitel 5 die nötige Vorarbeit, also die Beschaffung aller nötigen Informationen, erläutert. Anschließend wird in Kapitel 6 das entwickelte Werkzeug zur Stilüberprüfung vorgestellt.

In Teil III endet die Thesis mit einer abschließenden Bewertung, einer Zusammenfassung sowie einem Ausblick auf mögliche Weiterentwicklungen des Werkzeugs.

I.

Grundlagen

2. Stilüberprüfung

Automatisierte Stilüberprüfung ist, unabhängig von der Programmiersprache, ein nützliches Werkzeug. In diesem Kapitel wird ein Überblick darüber gegeben, welche Möglichkeiten der automatisierten Stilüberprüfung bereits existieren und wie man diese Ansätze benutzt. Für Curry selbst gibt es noch keine vergleichbaren Werkzeuge, daher werden zunächst die bekanntesten Vertreter deklarativer Programmiersprachen, an die Curry angelehnt ist, betrachtet: Prolog und Haskell. Dabei wird auffallen, dass es zwar verschiedene Kriterien gibt, die in beiden Sprachen überprüft werden können, jedoch kein Werkzeug die Möglichkeiten bereitstellt, die wir uns für Curry wünschen und die im Laufe dieser Thesis umgesetzt werden sollen. Die Stilüberprüfung in der vielseitigen Sprache Ruby ist zusätzlich interessant für den Entwurf eines solchen Systems. Anschließend wird der Curry Style Guide von der Arbeitsgruppe *Programmiersprachen und Übersetzerkonstruktion* der Christian-Albrechts-Universität zu Kiel vorgestellt und dahingehend analysiert, welche der Stilkriterien sich gut für eine automatische Überprüfung eignen und welche nicht.

2.1. Prolog

Prolog ist eine logische Programmiersprache, deren Name sich vom Französischen „*Pro*grammation en *Log*ique“, zu deutsch „Programmieren in Logik“ ableitet. Prolog-Programme bestehen aus Fakten und Regeln, der sogenannten Wissensdatenbank. Der Benutzer stellt Anfragen an diese Wissensdatenbank und der Prolog-Interpreter benutzt Fakten und Regeln, um die Antwort logisch abzuleiten.

Das Werkzeug zur Stilüberprüfung `style_check`[1] ist ein eingebautes Prädikat der Prolog-Implementierung SWI-Prolog. Mittels der Befehle `style_check(+Spec)` und `style_check(-Spec)` werden einzelne zu überprüfende Kriterien an- beziehungsweise ausgeschaltet.

Werden Verletzungen einzelner eingeschalteter Kriterien festgestellt, werden sie beim Kompilieren des überprüften Programms automatisch als Warnung ausgegeben [16].

[1] `http://www.swi-prolog.org/pldoc/man?predicate=style_check/1`

Die folgenden Kriterien sind eingebaut, wobei `Spec` im Befehl durch den Namen des Kriteriums ersetzt wird:

- **`singleton:`** Variablen, die nur ein einziges Mal in einer Klausel vorkommen, werden *Singleton* genannt und können immer durch die anonyme Variable `_` („don't care“) ersetzt werden. Häufig sind diese Variablen Tippfehler. Um dem System mitzuteilen, dass eine Singleton-Variable wirklich eine Singleton-Variable sein soll und kein Tippfehler, beginnt sie mit einem Unterstrich. Ist das Kriterium `singleton` also eingeschaltet, wird eine Warnung ausgegeben, wenn eine Variable, die nur einmal in einer Klausel vorkommt, nicht mit einem Unterstrich beginnt.
- **`no_effect:`** Diese Warnung wird ausgegeben, wenn der Compiler beweisen kann, dass die Nutzung eines eingebauten Prädikats bedeutungslos ist.
- **`var_branches:`** Wenn eine Variable in einem Zweig eingeführt wird und nach diesem Zweig benutzt wird, überprüft dieses Kriterium ob sie in allen anderen Verzweigungen auch eingeführt wird. So wird Fehlern vorgebeugt, die durch den Aufruf ungebundener Variablen entstehen können.
- **`atom:`** Weil Atome oder Strings mit mehr als sechs „unescaped“ neuen Zeilen zu Fehlern mit dem eingebauten Prädikat `read` führen, kann durch dieses Kriterium vor solchen gewarnt werden.
- **`discontiguous:`** Bei diesem Kriterium wird eine Warnung ausgegeben, wenn sich die Klauseln für ein Prädikat nicht alle in derselben Quelldatei befinden.
- **`charset:`** Warnt, wenn Atom- oder Variablennamen Zeichen beinhalten, die keine ASCII-Zeichen sind

2.2. Haskell

Haskell ist eine funktionale Programmiersprache, an die die logisch-funktionale Programmiersprache Curry, die im weiteren Verlaufe der Arbeit eine große Rolle spielen wird, eng angelehnt ist. Daher bietet es sich an, die Möglichkeiten der Stilüberprüfung für diese Sprache zu betrachten.

2.2.1. HLint

HLint[2] ist ein open-source-Projekt des Haskell-Entwicklers Neil Mitchell. Es handelt sich hierbei um ein Werkzeug, das stilistische Verbesserungen, insbesondere Vereinfachungen, für Haskell-Quellcode vorschlägt. Es lässt sich über den ghc-Paketmanager mit dem Befehl `cabal install hlint` installieren, über `hlint checkMeHlint.hs` benutzen und kann auch ganze Ordner, die Haskell-Dateien enthalten, überprüfen. Dabei werden eine Vielzahl möglicher Stilverletzungen berücksichtigt; eine vollständige Liste mit Beispielen ist im HLint-Report[3] einsehbar.

Ein Beispiel für eine mögliche Vereinfachung ist die Benutzung der Funktion `concatMap` anstelle der Funktionen `concat` und `map` . In Abbildung 2.1 ist zu sehen, wie die entsprechende HLint-Ausgabe dazu aussieht.

```
checkMeHlint.hs:1:1:  Warning:  Use concatMap
Found:
  concat $ map (++ "!  ") ["one", "two", "three"]
Why not:
  concatMap (++ "!  ") ["one", "two", "three"]

1 hint
```

Abbildung 2.1.: Ausgabe von HLint für `concatMap`

Weitere Vereinfachungen, die HLint ebenfalls erkennt, sind beispielsweise redundante `$`-Operatoren, wenn ein Unterstrich anstelle des Camel-Case benutzt wird, oder redundante Klammern.

Durch die Option `--refactor` lassen sich die meisten Vorschläge automatisch auf den Quellcode anwenden [12]. Dazu muss `refactor` aus dem Paket `apply-refact`[4] ebenfalls installiert sein. Die Funktion `refactor` benutzt die GHC API um eine gegebene Liste von Überarbeitungen in Quellcode einzupflegen und wird von HLint direkt aufgerufen. Hierbei existieren beispielsweise die beiden nützlichen Optionen `-i`, bei der die Originaldatei durch die Überarbeitung überschrieben wird, und `-s`, bei der vor der Anwendung einer jeden Vereinfachung nachgefragt wird.

HLint lässt sich vom Nutzer konfigurieren und erweitern. Manche der Vereinfachungen, auf die das Werkzeug hinweist, sind subjektiver Natur und lassen sich durch Einträge wie `ignore = "Name of Hint"` in der Datei `HLint.hs` im Arbeitsverzeichnis abschalten.

[2] `https://github.com/ndmitchell/hlint`
[3] `http://community.haskell.org/~ndm/hlint/hlint-report.htm`
[4] `https://github.com/mpickering/apply-refact`

Dort kann auch festgelegt werden, ob Hinweise als Vorschlag, Warnung oder Fehler anzuzeigen sind.

Das Hinzufügen eigener Hinweise ist ohne großen Aufwand durch einfache Syntax möglich [12]. Der Hinweis auf die Benutzung von `concatMap` beispielsweise ist folgendermaßen definiert:

```
warn = concat (map f x) ==> concatMap f x
```

Diese Zeile wird gelesen als „kommt die linke Seite irgendwo im Quellcode vor, schlage vor, sie nach dem Vorbild der rechten Seite zu verändern“ und lässt sich leicht für beliebige Ersetzungen nachbauen. Dabei werden alle Variablen, die aus Einzelbuchstaben bestehen, als Substitutionsparameter interpretiert.

2.2.2. Style Scanner

Der Haskell Style Scanner ist ein Programm zur Stilüberprüfung, das vom Python-Skript `haskell_style_check`[5] und von HLint inspiriert wurde. Es ist in Haskell geschrieben und lässt sich über den GHC-Paketmanager durch den Befehl `cabal install scan` installieren [8].

Der Haskell Style Scanner überprüft die stilistisch gute Nutzung von Leerzeichen anhand der folgenden fünf Kriterien:

- Kein Vorkommen von Tabulatoren
- Zeilenlänge beträgt nicht mehr als 80 Zeichen
- Leerzeichen nach Komma
- Leerzeichen vor und nach Operatoren
- Leerzeichen nach dem Eröffnen eines Kommentars

Eine Überprüfung der Datei aus Listing 2.1 mit dem Aufruf `scan checkMeScan.hs` produziert die in Abbildung 2.2 abgebildete Ausgabe auf dem Terminal:

Durch die Option `-i` werden gefundene Fehler automatisch korrigiert. Hierbei gibt es verschiedene Möglichkeiten, den Pfad der Ausgabedatei festzulegen.

[5] `www.cs.caltech.edu/courses/cs11/material/haskell/misc/haskell_style_guide.html`

```haskell
import Data.List (intercalate)

--this is a comment

sentence        = intercalate " " ["these","are","words"]

plus3 x         = x+3

showAsTuple x y = show(x, y)
```

Listing 2.1.: Beispiel zur Überprüfung durch Haskell Style Scanner

```
checkMeScan.hs:3:3:  put one blank after -- in --th
checkMeScan.hs:5:9:  up to column 17 multiple (8) blanks
checkMeScan.hs:5:44:  put blank after ,
checkMeScan.hs:5:50:  put blank after ,
checkMeScan.hs:7:8:  up to column 17 multiple (9) blanks
checkMeScan.hs:7:20:  put blank before +
checkMeScan.hs:7:21:  put blank after +
checkMeScan.hs:9:23:  put blank before (
```

Abbildung 2.2.: Ausgabe vom Haskell Style Scanner für Beispiel aus Listing 2.1

2.3. Ruby

> *Often people, especially computer engineers, focus on the machines. They think, [...] „By doing this, the machine will run more effectively.“ [...] But in fact we need to focus on humans, on how humans care about doing programming or operating the application of the machines. We are the masters. They are the slaves.*
>
> — Yukihiro Matsumoto, *Google Tech Talks, Februar 2008* [11]

Ruby ist eine vielseitige Programmiersprache, die verschiedene Programmierparadigmen zur Verfügung stellt. Sie ist unter anderem funktional, objektorientiert und imperativ. Bei einem Google Tech Talk am 20. Februar 2008 stellte der Chef-Designer und -Entwickler die Ruby-Version 1.9 vor (aktuelle Version: 2.3 von Dezember 2015) und sagte sinngemäß übersetzt:

> Insbesondere Informatiker konzentrieren sich oft nur auf die Maschine. Sie denken: „Wenn wir das so machen, läuft die Maschine effektiver.“ Stattdessen

> sollten wir uns auf den Menschen konzentrieren, darauf wie Menschen programmieren und Maschinen benutzen wollen. Wir sind die Herren, sie die Sklaven.

Ruby ist also vor allem auf angenehme Benutzung durch den Programmierer, Produktivität und Spaß ausgelegt. Das merkt man auch an den oft fantasievollen Namen für Sprachelemente; Bibliotheken etwa heißen passenderweise *Gems* (aus dem Englischen: *Edelsteine*).

Das open-source-Programm zur statischen Code-Analyse für Ruby heißt *rubocop*. Dieser Name ist angelehnt an den Film „RoboCop“ von 1987, das Logo ist der stilisierte Kopf der Hauptfigur RoboCop, einem Roboter-Polizisten. Rubocop setzt den von der Ruby-Community entwickelten Style Guide[6] um, indem Kriterien überprüft und gefundene Fehler mittels der Option `-a` teilweise automatisch behoben werden. Mit dem Befehl `gem install rubocop` lässt sich rubocop installieren und über eine ausführliche Konfigurationsdatei[7] konfigurieren. Unter anderem besteht hier wie in HLint für Haskell auch die Möglichkeit, Fehler als Vorschläge, Warnungen oder Fehler anzuzeigen.

Eine einzelne Überprüfung, die rubocop vornimmt, heißt „Cop“ (aus dem Englischen: *Polizist*). Wenn Stilfehler gefunden werden, werden diese „Offenses“ (aus dem Englischen: *Vergehen*) genannt.

```
1 a = 10
2 b = 3 * a + 2
3 printf("%d %d\n", a, b);
4
5 b = "A string"
6 c = 'Another String'
7 print b + ' and ' + c + "\n"
```

Listing 2.2.: Beispiel zur Überprüfung durch rubocop

Ein Beispiel für die Benutzung von rubocop ist in Abbildung 2.3 zu sehen. Dort ist die Ausgabe für den Aufruf `rubocop checkMeRubocop.rb`, dem Programm aus Listing 2.2, abgebildet. Die Ausgabe ist farbig und die genaue Position der gefundenen Fehler wird durch Unterstreichung mit einem oder mehreren Zirkumflex-Symbolen ^ angezeigt.

[6] `https://github.com/bbatsov/ruby-style-guide`
[7] `https://github.com/bbatsov/rubocop/blob/master/config/default.yml`

```
Inspecting 1 file
C

Offenses:

checkMeRubocop.rb:1:1:   C: The name of this source file
                            (checkMeRubocop.rb) should use
                            snake_case.
a = 10
^
checkMeRubocop.rb:3:24:  C: Do not use semicolons to terminate
                            expressions.
printf("%d %d\n", a, b);
                       ^
checkMeRubocop.rb:5:5:   C: Prefer single-quoted strings when
                            you don't need string interpolation
                            or special symbols.
b = "A string"
    ^^^^^^^^^^

1 file inspected, 3 offenses detected
```

Abbildung 2.3.: Ausgabe von rubocop für Beispiel aus Listing 2.2

Interessant an rubocop ist für diese Arbeit insbesondere die Abgrenzung der Cops (Checks) vom restlichen Programm. Ein Cop besteht immer aus einem oder mehreren Paaren aus Validierung (dem erwarteten Verhalten für das überprüfte Element) und automatischer Korrektur. Wie in Kapitel 6 beschrieben, werden wir uns diese Struktur teilweise zum Vorbild nehmen.

2.4. Curry Style Guide

Der Curry Style Guide enthält eine Anleitung mit Beispielen für den bevorzugten Curry-Programmierstil in der Arbeitsgruppe „Programmiersprachen und Übersetzerkonstruktion“ der Christian-Albrechts-Universität zu Kiel [14] und ist angelehnt an den Haskell Style Guide von Johann Tibell [17].

Folgende Kategorien werden im Curry Style Guide behandelt:

1. Generelle Formatierung
2. Formatierung einzelner Sprachkonstrukte
3. Kommentare

4. Namensgebung
5. Compiler-Warnungen

Unter dem Punkt „Generelle Formatierung“ befinden sich Vorgaben zur maximalen Zeilenlänge und zur Benutzung von Leerzeichen und -zeilen. Die „Formatierung einzelner Sprachkonstrukte“ behandelt, wie der Name schon sagt, die Formatierung von beispielsweise dem Modulkopf, von Pattern Matching, von `if-then-else`-Ausdrücken und von Listen und Tupeln. Der Abschnitt „Kommentare“ schreibt das Kommentieren sowie das Angeben einer Typsignatur bei Top-Level-Funktionen vor. Für Kommentare sollte dabei die CurryDoc-Syntax[8] verwendet werden. Unter „Namensgebung“ befinden sich Hinweise dazu, wie welche Parameter typischerweise heißen. Beispielsweise bezeichnet ein `f` oftmals eine Funktion vom Typ `a -> b` und die Buchstaben `n, m` natürliche Zahlen vom Typ `Int`. Generell werden Namen, die aus mehreren Worten bestehen, durch Binnenmajuskel („CamelCase“) aneinandergefügt. Der letzte Punkt „Compiler-Warnungen“ besagt, dass jeder Code durch Angabe der Parser-Option `:set parser -Wall` ohne jegliche Warnungen kompiliert werden können soll.

Die meisten dieser Stilkriterien eignen sich gut zur automatischen Überprüfung, weil sie allgemeine Formatierungsvorgaben beinhalten. Für die Überprüfung von Formatierungen werden in der Regel nur Positionsangaben für die entsprechenden Elemente benötigt. Auch die Zeilenlänge sowie das Kompilieren ohne Warnungen sind leicht zu überprüfen.

Schwieriger wird es hingegen bei der Überprüfung der Regeln für Leerzeichen. Vor und hinter zweistellige Operatoren wie `(++)` ist jeweils ein Leerzeichen zu setzen. Ob das hintere Leerzeichen gesetzt ist, lässt sich durch Hinzuziehen der Position des Folgeelements leicht herausfinden. Da jedoch nur die Anfangsposition des vorigen Elements bekannt ist, muss auch seine Länge bekannt sein, um herauszufinden, ob das erste Leerzeichen gesetzt ist. Auch das Überprüfen der korrekten Nutzung von Leerzeilen ist nicht trivial. „Zwischen zwei Top-Level-Definitionen sollte jeweils eine Leerzeile stehen, zwischen Typsignatur und Implementierung gehört keine Leerzeile. Kommentare zu Top-Level-Definitionen werden ebenfalls nicht abgegrenzt.“ [14] Abgesehen von der Leerzeile zwischen Typsignatur und Implementierung würden für diese Überprüfungen die Positionen von Kommentaren sowie die Anzahl von Zeilen, über die sie sich erstrecken, benötigt. Das ist, wie im späteren Verlauf dieser Thesis zu sehen sein wird, nicht einfach umzusetzen.

[8] `https://www-ps.informatik.uni-kiel.de/currywiki/tools/currydoc`

Gar nicht automatisiert überprüfbar hingegen ist beispielsweise die Namensgebung. Ohne ein Wörterbuch ist bei Namen wie `thisisafunctionname` nicht ersichtlich, ob der CamelCase korrekt benutzt wird. Es könnte aber beispielsweise nach Unterstrichen in Namen gesucht und dadurch der sogenannte Snake_Case durch den erwünschten CamelCase ersetzt werden:

```
this_is_a_function_name -> thisIsAFunctionName.
```

Außerdem könnte man überprüfen, ob etwa Parameternamen „sehr kurz“ [14] sind, aber oft können auch sprechendere Namen sinnvoller sein als einzelne Buchstaben. Die Zuordnung von bestimmten Buchstaben zu bestimmten Typen oder das Anhängen von Häkchen wie `x'` für veränderte oder aktualisierte Werte zu überprüfen ist auch nicht sinnvoll, da es häufig genug vorkommt, dass Namen sprechender sind und der Code damit für den Programmierer besser lesbar ist, wenn er sich nicht an diese Konventionen hält.

Zusätzlich zu den im Curry Style Guide [14] aufgeführten Richtlinien sind noch andere Kriterien denkbar, deren Überprüfung realisierbar ist. Bestimmte Stellen, an denen Vereinfachungen möglich wären, lassen sicht leicht erkennen und anmerken. Ein Beispiel für eine solche Stelle ist

```
concat $ map (++ "! ") ["one", "two", "three"].
```

Hier werden die Funktionen `concat` und `map` benutzt, anstelle der kombinierten Funktion `concatMap`. Auch Ausdrücke mit überflüssigen Klammern wie in

```
seven = (1 + (2 * 3))
```

wären leicht erkennbar und automatisch zu vereinfachen.

3. Curry

CURRY ist eine deklarative Programmiersprache, die von einem internationalen Team insbesondere für Forschung und Lehre entwickelt wird. Sie ist logisch-funktional und vereint damit zwei der wichtigsten deklarativen Programmierparadigmen.

Im Gegensatz zu imperativen Programmiersprachen, wo der Weg zur Lösung eines Problems im Vordergrund steht, wird in einer deklarativen Sprache das Problem selbst beschrieben. Dadurch sind deklarative Programme oft kürzer, kompakter und dadurch besser verständlich sowie weniger fehleranfällig als vergleichbare imperative Programme.

Curry kombiniert Aspekte der funktionalen Programmierung wie geschachtelte Ausdrücke, Funktionen höherer Ordnung oder die verzögerte Auswertung mit Aspekten der logischen Programmierung, zum Beispiel logische Variablen, partielle Datenstrukturen und eingebaute Suche. Des Weiteren ist auch die nebenläufige Auswertung von Ausdrücken mit Synchronisation auf logischen Variablen möglich.

Durch die Kombination der logischen und funktionalen Aspekte bietet Curry sowohl mehr als rein funktionale Sprachen, in denen beispielsweise keine Berechnungen mit partiellen Informationen möglich sind, als auch mehr als rein logische Sprachen, etwa durch nachfragegesteuerte und damit effizientere Auswertung.

Der funktionale Teil von Curry ist an die rein funktionale Programmiersprache Haskell angelehnt. Deshalb verwendet Curry weitgehend dieselbe Syntax und dasselbe Typsystem wie Haskell. Der logische Teil von Curry ähnelt der logischen Programmiersprache Prolog. Eine detaillierte Beschreibung von Curry kann im aktuellen Curry-Report [6] nachgelesen werden. Zusätzlich dazu befinden sich viele praktische Beispiele im Curry-Tutorial [2].

Im weiteren Verlauf dieses Kapitels werden zunächst einige Eigenschaften von Curry genauer vorgestellt, die Curry-Systeme PAKCS und KICS2 erklärt sowie eine Einführung zu Token und dem abstrakten Syntaxbaum gegeben.

3.1. Eigenschaften von Curry

Weil eine detaillierte Vorstellung von Curry hier den Rahmen sprengen würde, wird im weiteren Verlauf dieses Kapitels davon ausgegangen, dass der Leser mit Haskell vertraut ist. In diesem Abschnitt werden trotzdem einige wichtige Eigenschaften von Curry vorgestellt, die die Sprache von Haskell unterscheiden. Dies sind unter anderem der Nichtdeterminismus, Funktionales Pattern Matching sowie freie Variablen.

3.1.1. Nichtdeterminismus

In Curry wird eine Funktion, wie in deklarativen Programmiersprachen üblich, durch eine Menge von Regeln definiert. In Haskell werden beim Pattern Matching solche Regeln von oben nach unten im Programm ausprobiert. Sobald eine Regel anwendbar ist, wird sie angewandt und das Programm wird weiter abgearbeitet [10]. Anders jedoch in Curry: Hier geschieht die Auswertung eines Programms nichtdeterministisch. Das bedeutet, dass wenn eine Regel anwendbar ist, trotzdem alle weiteren Regeln ausprobiert werden und jede angewandt wird, die passt. Ein Beispiel dafür ist die Definition von `last`:

```
last :: [a] -> a
last [x]    = x
last (_:xs) = last xs
```

Diese Definition von `last` ist nichtdeterministisch, weil für eine Liste mit einem Element sowohl die erste als auch die zweite Regel anwendbar ist. Trotzdem liefert ein Aufruf von `last` das erwartete Ergebnis:

```
> last [1..5]
5
```

Im letzten Schritt der Auswertung passt der Ausdruck `last [5]` auf beide Regeln. Die Anwendung der ersten Regel liefert als Ergebnis 5. Die zweite Regel ruft die Funktion rekursiv mit der Restliste, also der leeren Liste, auf. Der Ausdruck `last []` passt jedoch auf keine Regel, daher schlägt der Aufruf fehl. Weil ein fehlgeschlagener Aufruf kein Ergebnis hat, ist nur das Ergebnis der ersten Regel zu sehen.

Wird in der interaktiven Umgebung eines Curry-Compilers einen Ausdruck ohne Ergebnis wie `last []` direkt aufgerufen, wird dem Benutzer entsprechend signalisiert, dass es kein Ergebnis gibt. Im Compiler KiCS2 geschieht dies durch ein einfaches Ausrufezeichen `!` und in PAKCS durch die Meldung **** No value found!*.

3.1.2. Pattern Matching

Pattern Matching (aus dem Englischen: *musterbasierte Suche*) ist ein Konzept, das in einigen funktionalen und logischen Programmiersprachen dazu genutzt wird, Daten mithilfe ihrer Struktur zu verarbeiten. Werte werden passenden Mustern zugeordnet (*to match*) und die Variablen aus dem Pattern bei einem „Match“ an die Aufrufparameter gebunden. Im vorigen Abschnitt wurde beschrieben, dass Funktionsdefinitionen in Curry aus mehreren Regeln bestehen können. In Haskell werden Pattern von oben nach unten und von links nach rechts ausprobiert, um die anzuwendende Regel auszuwählen. Wie im vorigen Abschnitt beschrieben, gilt dies wegen der Eigenschaft des Nichtdeterminismus nicht für Curry – hier werden alle Regeln ausprobiert, auch wenn schon ein passendes Muster gefunden wurde.

Ein anschauliches Beispiel dafür ist die Definition von `map`, einer Funktion, die eine Funktion `f` auf alle Elemente einer Liste anwendet:

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

Der Unterstrich in der ersten Regel wird Wildcard (aus dem Englischen: *Platzhalter*) genannt und bedeutet, dass dieser Teil des Musters in der rechten Gleichungsseite nicht referenziert wird – so ist es nicht von Interesse, welche Form er hat. Der zweite Parameter der ersten Regel ist die leere Liste. Diese Regel wird angewandt, wann immer `map` mit einer leeren Liste aufgerufen wird. In der zweiten Regel ist die Form der Funktion `f` nicht näher spezifiert und die Liste besteht aus mindestens einem Element `x` und einer Restliste `xs`, die auch leer sein kann.

In Pattern dürfen in Haskell zwar Konstruktoren, jedoch keine Funktionen vorkommen. Ein Pattern der Art `startsWithS ("s" ++ restString) = True` ist also unzulässig.

Für Curry wurde jedoch die Erweiterung „Functional Patterns“ vorgestellt, die genau das möglich macht. So wird die direkte Repräsentation von Spezifikationen in deklarativen Programmen ermöglicht [3]. Die Funktion `last` aus dem vorigen Abschnitt kann mithilfe funktionaler Pattern einfach auf folgende Weise definiert werden:

```
last (xs++[x]) = x
```

Anfang 2016 wurde dazu die Möglichkeit eingeführt, sogenannte „Default Rules“ (aus dem Englischen: *Standard-Regeln*) zu benutzen. Weil durch den in Curry vorherrschenden Nichtdeterminismus immer alle Regeln ausprobiert werden, ist dies eine hilfreiche

Erweiterung. Sie ermöglicht, solche Regeln zu definieren, die nur angewandt werden sollen, wenn keine andere Regel anwendbar ist [4]. Ein Beispiel dafür ist die Funktion `doubleWord`, die prüft, ob in einer Liste von Wörtern das gleiche Wort zweimal nacheinander steht:

```
doubleWord :: [String] -> Bool
doubleWord (_ ++ [x] ++ [x] ++ _) = True
doubleWord _                      = False
```

Die erste Regel benutzt ein funktionales Pattern und wird zu `True` ausgewertet, wenn das Muster auf der linken Seite zur Aufruf-Liste passt. Das Platzhalter-Muster aus der zweiten Regel jedoch passt auf jede denkbare Aufruf-Liste. Weil Curry Vollständigkeit zusichert, würde der Aufruf von `doubleWord` also immer zu `True` *und* `False` ausgewertet werden. Durch die Standard-Regel jedoch wird die Funktion deterministisch, weil die zweite Regel nur dann zur Anwendung kommt, wenn keine andere Regel passt.

```
doubleWord'default _ = False
```

3.1.3. Freie Variablen

Mit freien Variablen lässt sich, ähnlich wie in Prolog, mit einer Art Wissensdatenbank programmieren. In dieser Arbeit wird zwar rein funktional programmiert, aber zum Überblick über Curry als logisch-funktionale Programmiersprache gehört dieser Aspekt dazu.

Ein Beispiel für logische Programmierung ist die Verwandschaftsbeziehung `mother` beziehungsweise `grandmother`. Zunächst wird den Datentyp `Person` definiert, anschließend werden durch Gleichungen Beziehungen zwischen den Personen hergestellt.

```
data Person = Johanna | Mareike | Gesa | Gisela

mother Johanna = Gesa
mother Mareike = Gesa
mother Gesa    = Gisela
```

Auf die gleiche Art und Weise kann unter Verwendung von `mother` die Großmutter-Beziehung definiert werden:

```
grandmother x = mother $ mother x
```

Durch eine Anfrage an das Programm können Informationen aus der Wissensdatenbank extrahiert werden, auch wenn diese – wie die Großmutter-Beziehung – nicht explizit aufgeführt sind:

```
example> mother Mareike
Gesa
example> grandmother Johanna
Gisela
```

Durch Verwendung von sogenannten freien Variablen ist es uns ebenfalls möglich, alle Kinder einer Person zu berechnen. Dazu wird folgende Gleichung gelöst:

```
example> mother x == Gisela where x free,
```

was folgende Ausgabe produziert:

```
{x=Johanna} False
{x=Mareike} False
{x=Gesa} True
```

Bei der Ausführung des Codes mit der freien Variable wird `x` entsprechend der obenstehenden Definition von `mother` nacheinander an die Werte `Johanna`, `Mareike` und `Gesa` gebunden. Diese Bindung wird bei der Ausgabe in KiCS2 in geschweiften Klammern vor dem Ergebnisausdruck dargestellt.

Um nur solche Ergebnisse angezeigt zu bekommen, für die die Gleichung mit der freien Variable erfüllt ist, existiert in der Curry-Prelude die Funktion `solve`. Sie ist definiert als `solve True = True`, so dass die Berechnung fehlschlägt, wenn das Argument zu `False` ausgewertet wird. Um also nur wahre Lösungen anzeigen zu lassen, wird `solve` vor die Berechnung aller Kinder von `Gisela` geschrieben:

```
example> solve $ mother x == Gisela where x free
{x=Gesa} True
```

3.2. PAKCS und KiCS2

Der Compiler PAKCS (Portland Aachen Kiel Curry System) übersetzt nach Prolog [7]. KiCS2 (Kiel Curry System, Version 2) ist ein neuerer Compiler, der nach Haskell übersetzt [5]. Prolog und Haskell sind genau die Sprachen, an die Curry mit seinen logischen und funktionalen Elementen angelehnt ist.

Der Übersetzungsvorgang mit KiCS2 dauert in der Regel aufgrund zahlreicher Optimierungen länger als mit PAKCS, die fertigen Programme laufen dann aber aus dem selben Grund meist schneller. Aus diesem Grund wurde hier in der Implementierungsphase zunächst PAKCS verwendet. Die fertigen Programme wurden dann später nochmal mit KiCS2 kompiliert. In der Bewertung in Kapitel 7 werden die Laufzeiten von den verschiedenen Kompilaten des entwickelten Werkzeugs miteinander verglichen.

Ein Compiler besteht im Allgemeinen aus zwei voneinander abgegrenzten Teilen. Diese nennt man Frontend und Backend (Englisch: *vorderes* beziehungsweise *hinteres Ende*). Im Frontend wird der Quelltext analysiert und in eine hardwareunabhängige Zwischensprache (*IR*, intermediate representation) übersetzt. Im Backend wird aus der Zwischensprache das Zielprogramm erzeugt und für eine bestimmte Zielarchitektur optimiert. Diese Aufteilung sowie die Nutzung der Zwischensprache sorgen dafür, dass für eine Übersetzung von m Sprachen auf n Maschinen nicht, wie in Abbildung 3.1 zu sehen, $m \cdot n$ verschiedene Übersetzer benötigt werden.

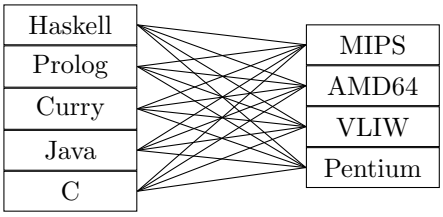

Abbildung 3.1.: Benötigte Übersetzer ohne Zwischensprache

Mithilfe der Zwischensprache genügt es, wie in Abbildung 3.2 erkennbar, m Frontends sowie n Backends zur Verfügung zu stellen [1].

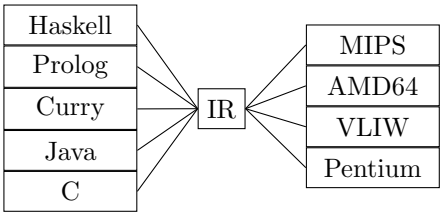

Abbildung 3.2.: Benötigte Frontends und Backends mit Zwischensprache

So können verschiedene Compiler dasselbe Frontend benutzen, wie es auch bei PAKCS und KiCS2 der Fall ist. In Abbildung 3.3 ist ein schematisches Ablaufdiagramm der Vorgänge zu sehen, die im Curry-Frontend stattfinden.

Zunächst wird der Curry-Quellcode lexikalisch analysiert – das heißt, er wird von einem lexikalischen Scanner (kurz *Lexer*) bearbeitet. Dies ist ein Programm zur Zerle-

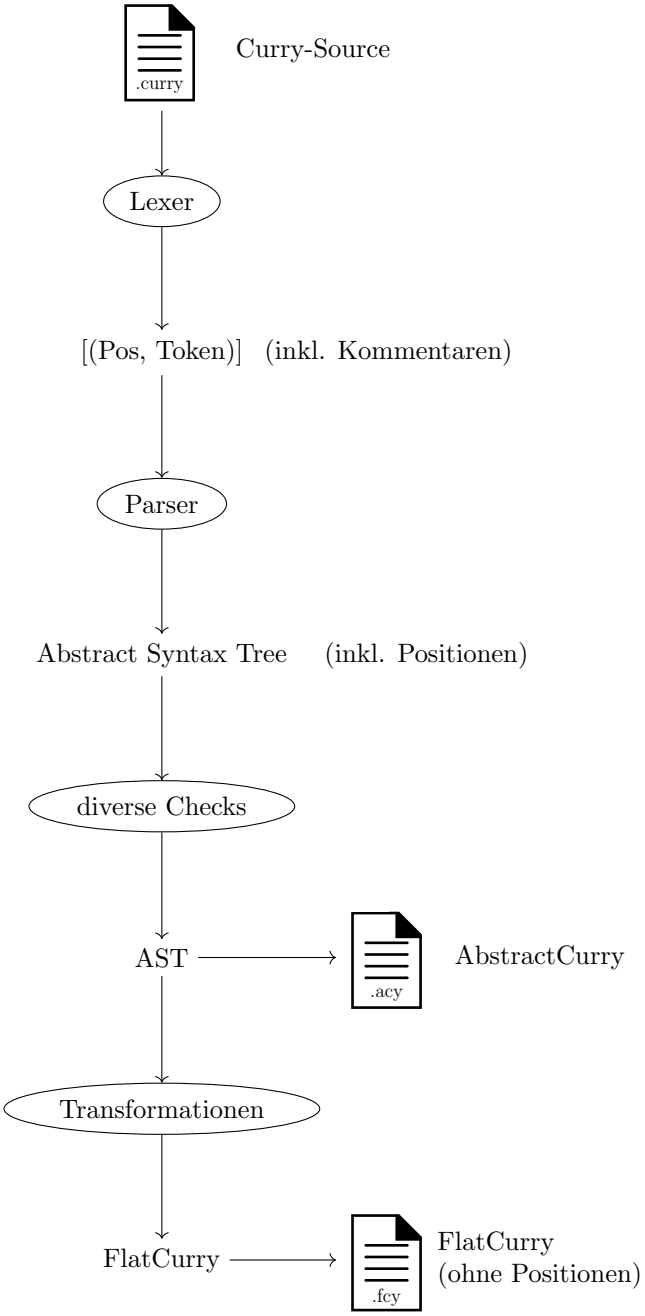

Abbildung 3.3.: Schematische Abbildung des Frontends

gung von Text in Folgen logisch zusammengehöriger Einheiten, die Token (aus dem Englischen: *Zeichen, Kürzel*) genannt werden. Die Zerlegung geschieht typischerweise nach Regeln regulärer Grammatiken und der lexikalische Scanner ist durch eine Menge endlicher Automaten realisiert. Im Curry-Frontend liefert der Lexer eine Liste von Paaren aus Positionsangaben und Token, auf die im nächsten Abschnitt genauer eingegangen wird. Die Liste von Paaren wird weitergereicht an den Parser (*to parse,* aus dem Englischen: *analysieren*), der die syntaktische Analyse übernimmt. Das Ziel ist hier das Erkennen der Programmstruktur, die zum Beispiel bedingte Anweisungen oder geklammerte Ausdrücke beinhalten kann. Um diese Programmstruktur darzustellen, wird ein abstrakter Syntaxbaum mit Positionsangaben benutzt. Es folgen verschiedene Checks, beispielsweise der SyntaxCheck und der TypeCheck, nach denen aus dem abstrakten Syntaxbaum eine AbstractCurry-Datei mit der Endung .acy erzeugt wird. Hier sind keine Positionsangaben mehr vorhanden. Anschließend wird der abstrakte Syntaxbaum nach FlatCurry transformiert. Diese Transformation wird in einer Datei mit der Endung .fcy festgehalten. FlatCurry ist die gemeinsame Zwischensprache der beiden Compiler PAKCS und KiCS2, aus der verschiedene Backends nach Haskell beziehungsweise Prolog übersetzen.

3.3. Token

Das Ziel der lexikalischen Analyse eines Programmes ist seine Zerlegung in eine Folge von Token oder Lexemen. Ein Lexem ist eine lexikalisch atomare Einheit (*lexis*, aus dem Griechischen: *Wort*).

Als Beispiel betrachte man in Listing 3.1 das Ergebnis der lexikalischen Analyse von folgender Curry-Funktion:

```
double :: Int -> Int
double x = x + x
```

Die lexikalische Analyse produziert eine Liste des Typs `[(Position,Token)]`, die wir ab hier als *Tokenfolge* bezeichnen. Die Position eines Tokens ist als Tupel von Zeile und Spalte angegeben und gibt seinen Startpunkt im Quelltext an.

Zudem ist zu sehen, dass die Lexeme in verschiedene Symbolklassen eingeteilt sind. Eine Symbolklasse ist eine Menge von Lexemen, die für die syntaktische Analyse als gleichartig anzusehen sind. Hat eine Symbolklasse mehrere Elemente, wie beispielsweise die Klasse `Id`, wird ein konkretes Lexem als Attribut dargestellt [1].

Eine vollständige Definition der Curry-Token ist in Anhang C zu finden.

```
[ ((1,1 ), (Id "double"))
, ((1,8 ), DoubleColon  )
, ((1,11), (Id "Int")   )
, ((1,15), RightArrow   )
, ((1,18), (Id "Int")   )
, ((2,1 ), (Id "double"))
, ((2,8 ), (Id "x")     )
, ((2,10), Equals       )
, ((2,12), (Id "x")     )
, ((2,14), (Sym "+")    )
, ((2,16), (Id "x")     )
, ((3,1 ), EOF          )
]
```

Listing 3.1.: Token der Funktion `double`

3.4. Abstrakter Syntaxbaum

Ein abstrakter Syntaxbaum (AST) ist die abstrakte Darstellung eines Computerprogramms in Form eines Baumes. In der abstrakten Syntax gibt es keine semantisch irrelevanten Informationen wie beispielsweise Kommentare oder überflüssige Klammern. Auch Positionsangaben haben keinerlei semantische Bedeutung, daher wird in der abstrakten Syntax auch weitgehend auf diese verzichtet. Stattdessen wird ein genauer Überblick über semantische Konstrukte geboten. Um das Beispiel aus dem vorigen Abschnitt weiterzuentwickeln, ist in Listing 3.2 der abstrakte Syntaxbaum abgebildet, der die Funktionsdefinition `double` repräsentiert.

Die Struktur des Baumes wird durch die Einrückung illustriert. Der Wurzelknoten des AST heißt `Module`. Die ersten vier Parameter sind an dieser Stelle noch nicht wichtig.

Die Signatur ist unter dem Knoten `TypeSig` wiederzufinden. Sie beinhaltet den Namen der zugehörigen Funktion – `Ident (1,1) "double"` – und besteht aus zwei Konstruktoren vom Typ `Int`, die durch einen `ArrowType` verknüpft sind.

Auch die Funktionsdefinition unter `FunctionDecl` beginnt wieder mit ihrem Namen, dann folgt eine Gleichung mit einer linken Seite `FunLhs` und einer rechten Seite `SimpleRhs`. Auf der rechten Gleichungsseite steht ein `InfixApply`, das die Variable `x` durch Addition mit sich selbst verknüpft.

Es fällt auf, dass sich immer noch Positionsangaben im AST befinden – jedoch nur solche, die etwa für Fehlermeldungen sinnvoll sein können. Beispielsweise ist zu erkennen,

```
(Module [],(ModuleIdent (0,0) ["Test"]) Nothing
  [(ImportDecl (0,0)
     (ModuleIdent (0,0) ["Prelude"]) False Nothing Nothing)]
  [(TypeSig (1,1)
     [(Ident (1,1) "double" 0)]
     (ArrowType
       (ConstructorType (QualIdent Nothing (Ident (1,11) "Int" 0)) [])
       (ConstructorType (QualIdent Nothing (Ident (1,18) "Int" 0)) [])))
  ,(FunctionDecl (2,1)
     (Ident (2,1) "double" 0)
     [(Equation (2,1)
       (FunLhs (Ident (2,1) "double" 0)
               [(VariablePattern (Ident (2,8) "x" 1))])
       (SimpleRhs (2,12)
         (InfixApply (Variable (QualIdent Nothing (Ident (2,8) "x" 1)))
                     (InfixOp  (QualIdent Nothing (Ident (2,14) "+" 0)))
                     (Variable (QualIdent Nothing (Ident (2,8) "x" 1))))
         []))])])
```

Listing 3.2.: AST der Funktion `double`

dass die Typsignatur an Position `(1,1)` beginnt. Daraus ist aber nicht ersichtlich, dass sich der Pfeil `->` an Position `(1,15)` befindet. Gleiches gilt für die Funktionsgleichung: Es ist bekannt, wo sie beginnt, nämlich an Position `(2,1)`, die Position `(2,10)`, an der das Gleichheitszeichen steht, ist jedoch verloren gegangen.

Der AST ist in einem Modul definiert, das eine Reihe von Datenstrukturen enthält. In der Implementierung des Frontends ist dieses Modul wie der Rest des Frontends in Haskell programmiert; wegen der Verwandschaft der Programmiersprachen Haskell und Curry sieht die hier benötigte Curry-Implementierung jedoch sehr ähnlich aus.

Die vollständige AST-Definition ist in Anhang D abgedruckt. Die Reihenfolge der definierten Knoten des AST entspricht ebenenweise dem möglichen Vorkommen im AST. So beginnt das Modul `AST.AST` beispielsweise mit der Definition des Datentyps `Module`:

```
-- |Curry module
data Module = Module [ModulePragma] ModuleIdent (Maybe ExportSpec)
                     [ImportDecl] [Decl]
```

Anschließend werden in der ungefähren Reihenfolge ihres möglichen Vorkommens die Typen `ModulePragma`, `ExportSpec`, `ImportDecl` und `Decl` sowie deren Kindknoten definiert.

Zur Veranschaulichung ist in Listing 3.3 einen Ausschnitt aus der Definition des AST zu sehen, an dem erkennbar ist, wie die Definitionen der einzelnen Knoten aufeinander aufbauen.

```
-- |Equation
data Equation = Equation Pos Lhs Rhs

-- |Right-hand side
data Rhs
  = SimpleRhs Pos Expression [Decl]
  | ...

-- |Expression
data Expression
  = Literal    Literal
  | Paren      Expression
  | IfThenElse Expression Expression Expression
  | ...

-- |Literal
data Literal
  = Char   Char
  | Int    Ident  Int
  | Float  Float
  | String String
```

Listing 3.3.: Ausschnitt aus der AST-Definition

Ganz oben ist die Beschreibung eines Gleichungs-Knotens `Equation` zu sehen. Eine Gleichung besteht aus einer Startposition `Pos`, einer linken Seite `Lhs` und einer rechten Seite `Rhs`.

Die nächste Definition ist die der rechten Gleichungsseite `Rhs`. Sie kann eine `SimpleRhs` sein und besteht in diesem Fall aus einer Startposition `Pos`, einem Ausdruck `Expression` sowie einer Deklarationsliste `[Decl]`. Die andere Gestalt, die eine rechte Gleichungsseite annehmen kann, wird hier der Übersichtlichkeit halber ausgelassen.

Es folgt die Definition eines Ausdrucks `Expression` mit drei typischen Beispielen. Insgesamt kann der `Expression`-Knoten eine von 24 Formen annehmen. Das `Literal`

wird in der nächsten Definition aufgeschlüsselt. Der Ausdruck `Paren` beschreibt die Umschließung eines Ausdrucks von Klammern (Englisch: *parentheses*) und hat deshalb in der Definition erneut einen Ausdruck `Expression`. Der Ausdruck `IfThenElse` besteht aus drei Unterausdrücken, nämlich denen, die jeweils hinter den Schlüsselwörtern `if`, `then` und `else` stehen.

Abbildung 3.4 verdeutlicht auf anschauliche Art und Weise, wie man sich einen AST auch vorstellen kann, also welche Teile einer Funktionsdefinition mit welchen der obigen Begriffen gemeint sind. Dabei wurden die Strukturen der Übersichtlichkeit halber jeweils nur bis zur Ebene des `Pattern` beziehungsweise `Expression` aufgeschlüsselt.

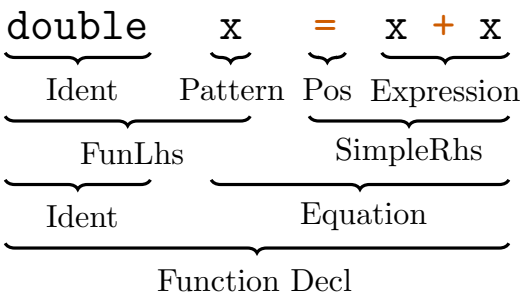

Abbildung 3.4.: Veranschaulichung des AST anhand des Beispiels `double`

II.

Implementierung

4. Entwurf

Ziel dieser Arbeit ist die Entwicklung eines Werkzeugs, das Curry-Quelltext auf die Einhaltung bestimmter Stilrichtlinien überprüft. Die Liste der zu überprüfenden Stilrichtlinien soll konfigurierbar und leicht erweiterbar sein.

In diesem Kapitel wird ein grober Überblick über die Struktur des Programmes, sowie ein Beispiel für das gewünschte Verhalten gegeben. Das zu entwickelnde Werkzeug soll ausschließlich in Curry programmiert sein. So können alle vorkommenden Strukturen als gültiges Curry eingelesen und verarbeitet werden. Dazu müssen die Datenstrukturen der Tokens und des AST aus dem in Haskell geschriebenen Frontend nach Curry übertragen werden. Durch die nahe Verwandschaft der beiden Sprachen ist dies jedoch leicht möglich. So befinden sich als Grundlage zunächst Beschreibungen der Token und des AST im Unterordner `src/AST`.

4.1. Gewünschter Ablauf

Im Hauptprogramm `casc` werden zunächst die Optionen, mit denen das Programm über die Kommandozeile aufgerufen wurde, ausgelesen und verarbeitet. Es soll möglich sein, das Programm sowohl mit einer einzigen Datei, als auch mit mehreren Dateien auf einmal aufzurufen. Wenn statt eines Dateinamens ein Ordnername übergeben wird, sollen alle `.curry`-Dateien innerhalb des Ordners geprüft werden. Für ganze Ordnerstrukturen existiert die Option `-r`, mit der rekursiv auch die `.curry`-Dateien in allen Unterordnern des übergebenen Ordnernamens geprüft werden können.

Nachdem eine Liste der zu überprüfenden Dateien erstellt wurde, werden für jede Datei einzeln folgende Schritte nacheinander ausgeführt:

- Generierung der Tokenfolge
- Generierung des AST
- Erweiterung des AST mithilfe der Tokenfolge
- Aufruf der Checkfunktionen
- Aufruf der Korrekturfunktionen

Dabei werden bei der Verbosity-Option „Status“ kurze Statusmeldungen ausgegeben, um den Nutzer darüber zu informieren, was gerade passiert. Auch die Ausgabe einzelner Zwischenprodukte wie der Tokenfolge und dem AST ist durch die Verbosity-Option „Debug“ möglich.

In einer Konfigurationsdatei kann der Nutzer durch eine Liste aller verfügbaren Checks und den Schaltern {0, 1, 2} angeben, welche Checks durchgeführt werden sollen (1) beziehungsweise bei welchen Kriterien eine automatische Korrektur durchgeführt werden soll (2). Außerdem kann festgelegt werden, wie viele Zeichen eine Zeile maximal beeinhalten darf, und der Pfad zu PAKCS- beziehungsweise KiCS2-Bibliotheken angegeben. Dieser ist wichtig für das Generieren von Tokenfolge und AST durch `cymake`, dem Build-Tool des Frontends.

Die Checkfunktionen befinden sich im Unterordner `src/Check`. Sie sind thematisch aufgeteilt und bestehen jeweils aus einem Check und einer Validierungsfunktion. Die Checks werden bei der Traversierung des erweiterten AST für jeden Knoten aufgerufen. Ein Check wird aktiviert, wenn das entsprechende Kriterium in der Konfigurationsdatei eingeschaltet ist. Falls ja, gibt er die Struktur an die entsprechende Validierungsfunktion weiter. Dies ist eine prädikatenlogische Gleichung, die besagt, wie sich bestimmte Positionen aus der zu überprüfenden Struktur zueinander verhalten müssen, damit die Struktur nach den Stilrichtlinien valide ist. Als Beispiel dient diese Validierungsfunktion für das If-Then-Else-Konstrukt, die die Positionen der Schlüsselwörter als Parameter erhält:

```
validITE :: Pos -> Pos -> Pos -> Bool
validITE pi pt pe =
     (line pi == line pt && line pt == line pe)
  || (col pt == ((col pi) + 2) && col pt == col pe)
```

Die Funktion `validITE` beschreibt genau die beiden gültigen Formatierungen eines If-Then-Else-Konstruktes, nämlich in der ersten Zeile diese:

```
if ifExpression then thenExpression else elseExpression
```

und in der zweiten Zeile diese, bei der die Fälle `then` und `else` um zwei Zeichen eingerückt untereinander unter dem `if` stehen:

```
if ifExpression
  then thenExpression
  else elseExpression
```

Die Ergebnisse der Überprüfung werden farblich hervorgehoben auf der Konsole ausgegeben. So kann der Nutzer sich schnell einen Überblick über die zu korrigierenden Stilfehler verschaffen, oder wenn gewünscht das Ergebnis auch direkt in eine Datei schreiben, zum Beispiel mittels `./casc "File.curry" > result.txt`.

Eine Übersicht über den gewünschten Ablauf des Stilüberprüfungs-Programms (automatische Korrektur ausgenommen) bietet das Flussdiagramm in Abbildung 4.1:

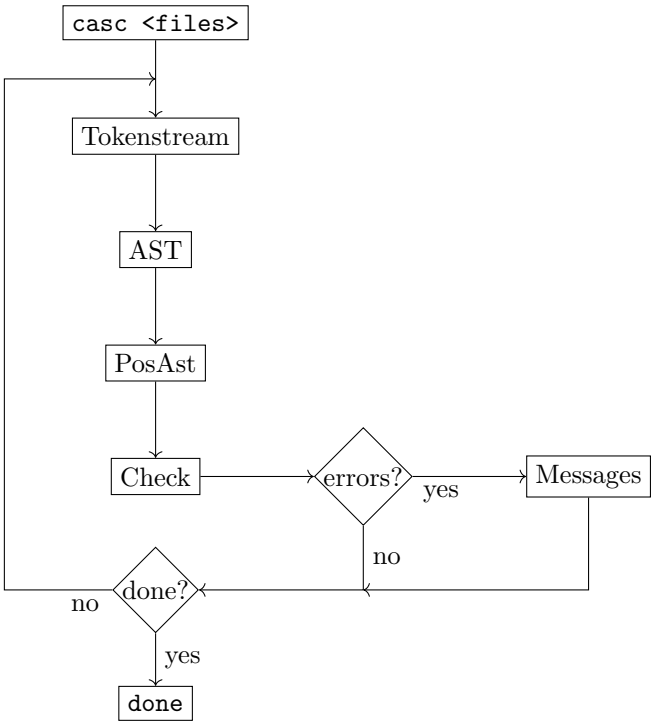

Abbildung 4.1.: Schematischer Ablauf eines Aufrufes von `casc`

Die automatische Korrektur der Stilfehler wird als letzte Phase für eine Datei eingeleitet, sofern in der Konfigurationsdatei durch eine 2 Kriterien zur automatischen Korrektur angegeben sind. Außerdem gibt es in der Konfigurationsdatei noch den Parameter `autoCorrect`. Nur, wenn dort „yes“ eingetragen ist, wird die automatische Korrektur eingeleitet. Wie schon bei den Checks muss der gesamte erweiterte AST durchlaufen werden und Korrekturen auf die entsprechenden Strukturen angewandt werden. Dazu werden Funktionen benutzt, die einen „falschen“ Ausdruck nehmen und ihn mit korrigierter Position zurückgeben. Um dem Nutzer diese Korrekturen zur Verfügung zu stellen, sollte ein exakter Pretty Printer existieren, der den erweiterten AST exakt mit allen Elementen an den angegebenen Positionen als Curry-Quelltext in die `.curry`-Datei zurückschreibt.

Um den Überblick über das Programm abzurunden, ist in Abbildung 4.2 ein Aufruf des fertigen Programms mit seinen Statusausgaben und den Nachrichten zu gefundenen Stilfehlern zu sehen.

4.2. Überblick über das Thesis-Repository

Der im Laufe dieser Arbeit erstellte Code ist in dem Git-Repository `2015-krah-ma`[9] verfügbar. Am Curry-Frontend[10] und Curry-Backend[11] vorgenommene Veränderungen befinden sich in den entsprechenden Repositories. Das Repository `2015-krah-ma` ist in die Ordner `latex`, `src` und `Test` gegliedert. Der Ordner `latex` ist an dieser Stelle nicht weiter relevant, in ihm befinden sich die Folien für den Proposal- und den Abschlussvortrag sowie die Thesis selbst. Der Code für das entwickelte Werkzeug befindet sich im Ordner `src` und ist in die Unterordner `AST`, `AutoCorr`, `Check` und `Config` unterteilt. Ihre Bedeutung wird in den folgenden Kapiteln erklärt werden. Der `Test`-Ordner enhält einige Testdateien, geordnet nach verschiedenen Elementen des AST. Die Testdateien sind zur Überprüfung gedacht, sie sollen möglichst viele verschiedene Sprachkonstrukte in unterschiedlichen Varianten enthalten, so dass anhand einer Überprüfung aller Dateien im Ordner `Test` festgestellt werden kann, ob alle Elemente des AST erweitert und alle Überprüfungen ausgeführt werden.

[9] `https://git-ps.informatik.uni-kiel.de/theses/2015-krah-ma`
[10] `https://git.ps.informatik.uni-kiel.de/curry/curry-frontend`
[11] `https://git.ps.informatik.uni-kiel.de/curry/curry-base`

```
krah @ krahbuntu  /Repositories/2015-krah-ma/src$
casc Testfiles

Welcome to casc, the Curry Automatic Style Checker.
The following files will be checked:
Test/Expressions.curry, Test/TestGuardedRhs.curry

Processing file:  Test/Expressions.curry
Invoking:  ./cymake --tokens --parse-only -W none -q
           -i  /bin/pakcs/lib/ Test/Expressions.curry
Extending abstract syntax tree...  Checking...

    (3,0) Line is longer than 80 characters.
    (6,5) Let:  Keywords 'let' and 'in' are not aligned.
    (6,9) Let:  Equality signs are not aligned.
    (13,14) IfThenElse:  Wrong indentation of keywords or
            subexpressions.

Done.

Checking file:  Test/TestGuardedRhs.curry
Invoking:  ./cymake --tokens --parse-only -W none -q
           -i  /bin/pakcs/lib/ Test/Expressions.curry
Extending abstract syntax tree...  Checking...

    (4, 3) GuardedRhs:  Bars are not aligned.
    (4,15) GuardedRhs:  Equality signs are not aligned.

Done.
Done with all files.
```

Abbildung 4.2.: Ausgabe von casc

5. Erweiterung des abstrakten Syntaxbaums

Im abstrakten Syntaxbaum sind, wie in Kapitel 3.4 beschrieben, nicht mehr viele Positionsangaben vorhanden. Für unser Vorhaben, Sprachkonstrukte auf eine vorgegebene Art der Einrückung zu prüfen, sind diese jedoch essentiell. Auch die Zwischenrepräsentationen von Curry – Abstract Curry und Flat Curry – sind dafür ungeeignet, weil auch dort nicht genügend Informationen vorhanden sind. Aus diesem Grund muss zunächst eine weitere Datenstruktur geschaffen werden, die dem AST ähnelt, ihn jedoch um alle Informationen ergänzt, die zum Überprüfen der gewünschten Stilrichtlinien benötigt werden. Diese Datenstruktur nennen wir *PosAST*, kurz für „um Positionsangaben erweiterter abstrakter Syntaxbaum“. In diesem Kapitel wird zunächst am Beispiel von `double` betrachtet, wie so ein PosAST aussehen sollte, anschließend werden einige Besonderheiten in seiner Definition sowie die Erweiterung des AST zu einem PosAST erläutert.

5.1. Beispiel `double`

Das Ziel ist im Folgenden ein PosAST für das in den vorigen Kapiteln betrachtete Beispiel `double`. Zum in Listing 3.2 abgebildeten AST der Funktion könnte der PosAST aussehen, wie in Listing 5.1 angegeben.

Im Unterschied zum originalen AST gibt es hier einige weitere Positionen und einige Stellen, an denen jetzt ein zusätzliches `Nothing` steht. Der `ArrowType`-Knoten hat beispielsweise drei statt nur zwei Blätter. Das zusätzliche Blatt, die Position `(1,15)`, zeigt die Stelle an, an der der Rechtspfeil `->` in der Signatur steht. Außerdem hat sich die Position im Knoten `SimpleRhs` von `(2,12)` auf `(2,10)` verändert. Position `(2,10)` ist die Position des Gleichheitszeichens, die uns interessiert. An Position `(2,12)` steht die Variable `x`.

Das `Nothing` kennzeichnet Stellen, an denen in der Definition des PosAST `(Maybe Pos)` steht, es also sein kann, dass noch etwas folgt, dessen Position auch festgehalten werden

```
(Module [] Nothing (ModuleIdent (0,0) ["Test"]) Nothing Nothing
  [(ImportDecl (0,0) Nothing (ModuleIdent (0,0) ["Prelude"])
     Nothing False Nothing Nothing)]
  [(TypeSig
    [(Ident (1,1) "double" 0)] [] (1,8)
    (ArrowType
      (ConstructorType Nothing
                       (QualIdent Nothing (Ident (1,11) "Int" 0))
                       [] Nothing)
      (1,15)
      (ConstructorType Nothing
                       (QualIdent Nothing (Ident (1,18) "Int" 0))
                       [] Nothing)))
  ,(FunctionDecl (2,1)
     (Ident (2,1) "double" 0)
     [(Equation
        (FunLhs
          Nothing
          (Ident (2,1) "double" 0)
          Nothing
          [(VariablePattern (Ident (2,8) "x" 1))])
        (SimpleRhs
          (2,10)
          (InfixApply (Variable
                        (QualIdent Nothing (Ident (2,12) "x" 1)))
                      (InfixOp Nothing
                               (QualIdent Nothing (Ident (2,14) "+" 0))
                               Nothing)
                      (Variable
                        (QualIdent Nothing (Ident (2,16) "x" 1))))
          Nothing []))])])
```

Listing 5.1.: Erweiterter AST für das Beispiel double

muss. Ein Beispiel für eine solche Stelle ist das `Nothing`, das neben dem `InfixApply` auch ein Blatt des `SimpleRhs`-Knotens ist.

Um das genauer zu verstehen, betrachte man den Unterschied in der Definition der Datenstrukturen von `Rhs` an, die entweder wie im obigen Beispiel eine `SimpleRhs`, oder eine `GuardedRhs` sein kann. Oben steht die Definition der rechten Gleichungsseite im originalen AST, darunter wie sie im erweiterten AST definiert ist:

```
data Rhs
  = SimpleRhs  Pos Expression [Decl]
  | GuardedRhs [CondExpr] [Decl]

data Rhs
  = SimpleRhs  Pos Expression (Maybe Pos) [Decl]
  | GuardedRhs Pos [CondExpr] [Pos] (Maybe Pos) [Decl]
```

Eine `SimpleRhs` besteht in der originalen Definition aus ihrer Anfangsposition, einem Ausdruck und einer Liste von Deklarationen. Diese Liste kann auch leer sein, was beim Beispiel `double` auch der Fall ist. Ist sie nicht leer, steht hinter dem Ausdruck noch das Schlüsselwort `where` und es folgen eine oder mehrere Deklarationen.

Diese Definition wurde so verändert, dass die Position nicht mehr den *Beginn* der rechten Gleichungsseite kennzeichnet, sondern die Position des *Gleichheitszeichens* oder *Pfeils* (der auch in einfachen rechten Gleichungsseiten auftauchen kann). Das `(Maybe Pos)` bildet den Umstand ab, dass ein `where` folgen kann, aber nicht muss. Gibt es ein `where`, soll seine Position gespeichert werden, gibt es keines, steht dort `Nothing` und die Deklarationsliste ist leer.

Gleichermaßen wurde auch die `GuardedRhs` erweitert. Sie stellt eine andere Variante der `SimpleRhs` dar. Die Positionen der Bars `|`, die vor den konditionalen Ausdrücken stehen, müssen festgehalten werden, außerdem kann auch hier das optionale Schlüsselwort `where` folgen.

5.2. Besonderheiten des PosAST

Auf die im Beispiel in Kapitel 5.1 beschriebene Weise wurde der gesamte abstrakte Syntaxbaum erweitert. An vielen Stellen wurden Positionsangaben hinzugefügt und teilweise wurde die Bedeutung von Positionen verändert. Sprachkonstrukte, bei denen optionale Zusätze wie das Schlüsselwort `where` oder Klammern stehen können, deren Positionen gespeichert werden sollen, werden mithilfe des aus Haskell bekannten `Maybe`

abgebildet. In den folgenden Abschnitten werden einige Aspekte des PosAST behandelt und auftretende Besonderheiten erläutert.

5.2.1. Optional auftretende Elemente

Auch im nicht erweiterten AST kommen optionale Elemente schon vor. Ein Beispiel dafür ist die Präzedenz bei der Deklaration von Infix-Operatoren: Es ist gültiges Curry wenn beispielsweise ein links-assoziativer Infix-Operator als `infixl 5 ‘fun‘` deklariert wird. Dabei wurde eine Präzedenz, also Bindungsstärke, von 5 angegeben. Genauso wäre die Deklaration aber ohne Angabe einer Präzedenz gültig: `infixl ‘fun‘`. Im AST wird dies folgendermaßen modelliert:

```
data Decl
  = InfixDecl Pos Infix (Maybe Precedence) [Ident]
  | ...
```

Wird im Quellcode eine Präzedenz angegeben, enthält der `InfixDecl`-Knoten ein `(Just Precedence)`, ansonsten ein `Nothing`.

Diese Methode lässt sich – wie im Beispiel aus dem vorherigen Kapitel schon erwähnt – auch sehr gut für optionale Schlüsselwörter benutzen, deren Position gespeichert werden soll, falls sie im Quellcode vorkommen. Ein Beispiel für solche Schlüsselwörter ist schon im Header eines jeden Curry-Programms zu finden – oder eben auch nicht: An den Anfang eines Curry-Programms kann, aber muss nicht, `module Module.Name where` geschrieben werden. Dies spiegelt sich im Unterschied zwischen den Definitionen des Wurzelknotens `Module` in AST und PosAST wider:

```
-- Module in AST
data Module = Module [ModulePragma] ModuleIdent
                     (Maybe ExportSpec) [ImportDecl] [Decl]

-- Module in PosAST
data Module = Module [ModulePragma]
                     (Maybe Pos) ModuleIdent (Maybe Pos)
                     (Maybe ExportSpec) [ImportDecl] [Decl]
```

Die beiden den `ModuleIdent` umschließenden `(Maybe Pos)`, die im PosAST hinzugefügt wurden, stehen für die Positionen der Schlüsselwörter `module` und `where`, die gespeichert werden sollen.

Ein weiteres Beispiel für ein optionales Schlüsselwort ist wie schon erwähnt `where`, das am Ende von Funktionsdefinitionen stehen kann, um die Definition lokaler Funktionen einzuleiten. Auch hier ist der Unterschied zwischen den Definitionen in AST und PosAST im direkten Vergleich zu sehen:

```
-- Rhs in AST
data Rhs
  = SimpleRhs  Pos Expression [Decl]
  | ...

-- Rhs in PosAST
data Rhs
  = SimpleRhs  Pos Expression (Maybe Pos) [Decl]
  | ...
```

Das im PosAST hinzugefügte `(Maybe Pos)` beinhaltet die im einfachen AST nicht gespeicherte Position des optionalen Schlüsselwortes `where`. Die darauf folgende Deklarationsliste `[Decl]` zeigt eine weitere Möglichkeit auf, optionale Elemente zu verpacken: Folgt in einer rechten Gleichungsseite `Rhs` kein `where` mit lokalen Funktionsdefinitionen, bleibt diese Liste einfach leer.

5.2.2. Veränderung der Bedeutung von Positionen

In der Definition der rechten Gleichungsseite `Rhs`, die schon im vorigen Abschnitt betrachtet wurde, befindet befindet sich auch direkt ein Beispiel für die Veränderung der Bedeutung einer Position. Wie im Beispiel in Kapitel 5.1 gesehen, beinhaltet die erste Position in der `SimpleRhs` im AST noch den Beginn der rechten Gleichungsseite (also allem, was rechts vom Gleichheitszeichen steht). Im PosAST steht sie für die Position des Gleichheitszeichens (bei einer Gleichung) oder des Pfeils nach rechts (bei der Alternative einer Fallunterscheidung):

```
-- Rhs in PosAST
data Rhs
  = SimpleRhs  Pos Expression (Maybe Pos) [Decl]
  | GuardedRhs Pos [CondExpr] [Pos] (Maybe Pos) [Decl]
```

Die zweite Variante einer `Rhs` ist die `GuardedRhs`, also eine rechte Gleichungsseite mit einem sogenannten Guard. Die rechte Gleichungsseite wird nur dann ausgeführt, wenn

die Bedingung im Guard `True` ist. Solche `GuardedRhs` können in Curry folgendermaßen verwendet werden:

```
abs :: Int -> Int
abs x | x > 0     =  x
      | x < 0     = -x
      | otherwise =  0
```

Die erste Position in der `GuardedRhs` ist wie bei der `SimpleRhs` der Beginn der Struktur, nämlich das erste Bar-Symbol `|`. Die Positionen der übrigen Bars werden in der Positionsliste `[Pos]` gespeichert. Die Positionen der Gleichzeitszeichen werden jeweils im entsprechenden `CondExpr` gespeichert:

```
-- |Conditional expression in AST
data CondExpr = CondExpr Pos Expression Expression

-- |Conditional expression in PosAST
data CondExpr = CondExpr Expression Pos Expression
```

Auch in einem `CondExpr`, einem konditionalen Ausdruck, wurde die Bedeutung der Position verändert: Beschreibt sie im AST noch den Beginn des gesamten `CondExpr`, wird im PosAST stattdessen die Position des Gleichheitszeichens gespeichert.

5.2.3. Weglassen von redundanten Positionen

Im Beispiel `CondExpr`, das im vorigen Abschnitt betrachtet wurde, wurde die Bedeutung der gespeicherten Position verändert. Wieso aber wurde nicht die Position des Gleichheitszeichens als `data CondExpr = CondExpr Pos Expression Pos Expression` hinzugefügt; wieso ist die Position, die im AST den Beginn des `CondExpr` beschrieben hat, überflüssig?

Das liegt daran, dass im PosAST zwar die Position jedes einzelnen Elements vorkommen soll, jedoch das Speichern von redundanten Positionen nicht erwünscht ist. In diesem Fall heißt das, dass der erste Ausdruck `Expression` im `CondExpr` eine Position hat. Auf diese Weise ist die Startposition des `CondExpr` bekannt, sie ist nämlich die gleiche wie die Position des ersten Ausdrucks.

Ein zweites Beispiel für redundante Positionen ist die `ConstrDecl`, die Konstruktordeklaration für algebraische Datentypen. Im AST wird hier für alle drei Varianten die Startposition gespeichert.

```
-- |Constructor declaration for algebraic data types in AST
data ConstrDecl
  = ConstrDecl Pos [Ident] Ident [TypeExpr]
  | ConOpDecl  Pos [Ident] TypeExpr Ident TypeExpr
  | RecordDecl Pos [Ident] Ident [FieldDecl]

-- |Constructor declaration for algebraic data types in PosAST
data ConstrDecl
  = ConstrDecl [Ident] Ident [TypeExpr]
  | ConOpDecl  [Ident] TypeExpr Ident TypeExpr
  | RecordDecl [Ident] Ident Pos [FieldDecl] [Pos] Pos
```

Dies ist jedoch nicht nötig, da jeweils eine Liste mit Identifiern `[Ident]` folgt. Ein `Ident` hat immer eine Position:

```
-- |Simple identifier
data Ident = Ident
  { idPosition :: Pos      -- ^ Source code 'Position'
  , idName     :: String   -- ^ Name of the identifier
  , idUnique   :: Int      -- ^ Unique number of the identifier
  }
```

Wird also jeweils die Position des ersten `Ident`s aus der Liste `[Ident]` betrachtet, so ist dies auch gleichzeitig die Startposition der gesamten `ConstrDecl`.

Es ist sinnvoll, keine redundanten Positionen zu speichern. Zum einen soll der Rechenaufwand und die Menge der gesammelten Daten möglichst klein gehalten werden; zum anderen führt es bei der automatischen Korrektur zu Problemen, wenn eine Quelltext-Position an verschiedenen Stellen im PosAST gespeichert ist.

5.2.4. Geklammerte und andere Identifier

In diesem Abschnitt wird eine Schwierigkeit beim Parsen der Tokenfolge betrachtet. Da für die Erweiterung des AST die Tokenfolge Stück für Stück konsumiert wird, während passende Regeln dafür aufgerufen werden, muss immer genau bekannt sein, für welche AST-Konstrukte welche Token erwartet werden. Dies wird zum Problem, sobald nicht eindeutig festgelegt ist, wie ein AST-Konstrukt im Quelltext aussieht. Bei Identifiern ist dies zum Beispiel der Fall, wenn sie nach der Curry-Syntax geklammert werden dürfen, aber nicht müssen. Auch Backticks um Identifier herum wie in `` `ident` `` kommen in

bestimmten Sprachkonstrukten vor und in anderen nicht. Also wird, wenn ein `Ident` im AST erweitert wird, in der Tokenfolge nur ein Token `Id "ident"` erwartet. Ist ein Name dann trotzdem geklammert, liegen noch die Token `LeftParen` und `RightParen` mit in der Tokenfolge. Diese werden dann nicht konsumiert und das weitere Erweitern des AST wird unmöglich, da Tokenfolge und AST nicht mehr synchron sind; gleiches gilt für die Token `Backtick`.

Eine Lösung für dieses Problem ist die Einführung von besonderen Identifiern, die es nur im PosAST gibt:

```
data SymIdent     = SymIdent     (Maybe Pos) Ident     (Maybe Pos)
data SymQualIdent = SymQualIdent (Maybe Pos) QualIdent (Maybe Pos)
```

So können im Zuge der Erweiterung des AST solche `Ident`s, beziehungsweise `QualIdent`s, die möglicherweise von Klammern, Backticks oder anderen öffnenden und schließenden Symbolen umschlossen sind, direkt in `SymIdent`s, beziehungsweise `SymQualIdent`s, übersetzt werden. In dieser Übersetzung werden dann die umschließenden Symbole berücksichtigt und ebenfalls von der Tokenfolge konsumiert.

Beispiele für die Verwendung dieser neu eingeführten Datentypen sind in Listing 5.2 zu sehen. Sie werden immer dann benötigt, wenn nicht bekannt ist, ob Klammern oder andere öffnende und schließende Symbole in der Tokenfolge liegen oder nicht. In der folgenden Auflistung befinden sich Beispiele und Erläuterungen in derselben Reihenfolge, wie die entsprechenden PosAST-Elemente in Listing 5.2 gezeigt werden.

- Werden Infix-Operatoren importiert, müssen sie mit Klammern versehen werden, Funktionen aber nicht: `import List (intercalate, (\\))`.
- Das gleiche gilt für den Export: `module Example (myFun, (>=<)) where ...`
- Werden Präfix-Funktionen in Infix-Schreibweise verwendet, müssen sie durch Backticks gekennzeichnet werden: `" - " `intercalate` ["one", "two", "three"]`
- Umgekehrt können Infix-Operatoren auch in Präfix-Schreibweise verwendet werden, müssen dafür aber zusätzlich geklammert werden: `(+) 11 4`

5.2.5. ParenType

Im Gegensatz zu den im letzten Abschnitt beschriebenen `SymIdent` und `SymQualIdent`, die ausschließlich im PosAST vorhanden sind, beschreibt dieser Abschnitt ein Element,

```
data Import
  = Import SymIdent
  | ...

data Export
  = Export SymQualIdent
  | ...

data InfixOp
  = InfixOp SymQualIdent
  | ...

data Expression
  = Variable SymQualIdent
  | ...
```

Listing 5.2.: Beispiele für die Verwendung von `SymIdent` und `SymQualIdent`

das auch im AST als Erweiterung eingeführt wurde. Dies war nötig, weil Klammern in Typsignaturen im AST nicht abgebildet wurden. Klammern in Typsignaturen sind wichtig für Funktionen höherer Ordnung, die als Parameter eine Funktion übergeben bekommen, beispielsweise `applyTwice`:

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

Im AST geht die Information verloren, dass die ersten beiden `a`s von Klammern umschlossen sind. Dies wird uns bei der Erweiterung zum PosAST zum Verhängnis, weil wieder Klammern in der Tokenfolge liegen, über deren Existenz anhand der Informationen aus dem AST nichts bekannt ist. Auch für die automatische Korrektur von Fehlern ist solch eine fehlende Information ein Problem, denn werden die Klammern im AST nicht erwähnt, können sie auch beim Zurückschreiben eines korrigierten AST in Quellcode nicht berücksichtigt werden.

Die Lösung dafür ist die Einführung eines neuen Elements der Datenstruktur `TypeExpr`, mit der Typsignaturen im AST dargestellt werden. Hier war eine Anpassung des Curry-Basis-Pakets nötig, so dass der Parser beim Erstellen des AST geklammerte Strukturen in Typsignaturen als `ParenType` darstellt:

```
-- |Type expressions in AST
data TypeExpr
  = ParenType TypeExpr
  | ...
```

```
-- |Type expressions in PosAST
data TypeExpr
  = ParenType Pos TypeExpr Pos
  | ...
```

5.3. Erweiterung

Die Erweiterung des AST findet auf Grundlage der Tokenfolge und des originalen AST statt. Das Curry-Frontend wurde so modifiziert, dass der Aufruf von `cymake` dem Build-Tool des Frontends, mit der Option `--tokens` eine Liste aller Token mit zugehöriger Positionsangabe des gewünschten Quelltextes ausgibt. Diese wird im Ordner `/.curry` mit der Dateiendung `.tokens` gespeichert. Durch die eingebaute Option `--parse-only` wird der AST des gewünschten Quelltextes erzeugt und ebenfalls im Ordner `/.curry` mit der Dateiendung `.cy` gespeichert. Zunächst erfolgt also ein Systemaufruf:

```
cymake --tokens --parse-only CheckMe.curry
```

Weil der AST jedoch in verschiedenen Punkten von der Tokenfolge abweicht, müssen an ihm Transformationen vorgenommen werden, bevor durch die Kombination der Informationen aus beiden Elementen der PosAST zusammengesetzt werden kann. Diese Transformationen betreffen zum einen die Reihenfolge der Deklarationen, die in der Tokenfolge trivialerweise dieselbe ist wie im Quelltext. Im AST jedoch stehen zuerst alle `data`-Deklarationen und anschließend die Funktionsdeklarationen. Zum anderen wurde in Kapitel 3 erläutert, dass Funktionsdeklarationen aus mehreren Gleichungen bestehen können. Diese Gleichungen dürfen im Curry-Quelltext beliebig verteilt sein und werden trotzdem korrekt derselben Funktion zugeordnet. Im AST jedoch hat ein `FunctionDecl`-Knoten als Blatt eine Liste der Gleichungen, die die entsprechende Funktion definieren:

```
data Decl
  = FunctionDecl Pos Ident [Equation]
  | ...
```

Aus diesem Grund müssen `FunctionDecl`-Knoten so aufgeteilt werden, dass zu jedem dieser Knoten nur eine Gleichung gehört. Anschließend werden alle `Decl`-Knoten

entsprechend der Reihenfolge in Quelltext und Tokenfolge durch das Quicksortverfahren [9] sortiert. Diese zweistufige Transformation wird vom Modul `AST.SortSplit` durchgeführt.

Ist das erledigt, sind der transformierte AST und die Tokenfolge synchron, so dass sie parallel durchlaufen werden können, um die Informationen aus der Tokenfolge in den AST einzupflegen. Der Wurzelknoten eines AST heißt `Module`. Wir brauchen also eine Konvertierungsfunktion

```
addPosModule :: AST.Module -> PosAST.Module,
```

die vom Wurzelknoten aus jeden beliebigen AST traversiert, also `addPos`-Funktionen für jeden beliebigen Teilbaum aufruft. Dabei muss immer die Tokenfolge mit übergeben werden. Wie bei einem Parser wird immer das nächste Tupel aus Position und Token konsumiert, das heißt aus dem Tokenfolge entfernt. Der Übersichtlichkeit halber wird die Tokenfolge in die **A**dd-**P**osition-**M**onade verpackt.

Monaden wurden 1991 von Eugenio Moggi erstmals als allgemeine Möglichkeit zum Strukturieren von Programmen vorgestellt [13]. Darauf aufbauend stellten Philip Wadler und Simon Peyton Jones – die beide auch an der Spezifikation von Haskell beteiligt waren – 1993 ein auf Monaden basierendes Modell für die Durchführung von Input- und Output-Aktionen (*IO-Aktionen*) in einer nicht-strikten, rein funktionalen Sprache vor [15].

An dieser Stelle wird die Monade `APM` zum Verstecken der Tokenfolge in einem monadischen Zustand benutzt. Die Definition der Add-Position-Monade sowie alle zugehörigen Funktionen befinden sich im Modul `AST.APM`. In Listing 5.3 ist die Definition der Monade `APM`, des bind-Operators `(>>=)` und der Einheitsfunktion `returnP` zu sehen.

```
type APM a = PosTokens -> (a, PosTokens)
type PosTokens = [(Pos, Token)]

(>+=) :: APM a -> (a -> APM b) -> APM b
(m >+= f) ts = let (a, ts') = m ts in f a ts'

returnP :: a -> APM a
returnP x ts = (x, ts)
```

Listing 5.3.: Monade mit bind-Operator und Einheitsfunktion

5.3.1. Beispiele

Als Beispiel für eine `addPos`-Funktion dient in Listing 5.4 ein Ausschnitt der Definition der Funktion `apExpr` („add position to `Expression`“), die einen getypten Ausdruck `Typed` erweitert. In der oberen Definition ist die tatsächliche Curry-Notation zu sehen und in der unteren wurde die Funktion in die aus Haskell bekannten `do`-Notation übersetzt. Weil in Curry jedoch bisher keine Typklassen implementiert sind, ist der syntaktische Zucker der `do`-Notation nur bei der `IO`-Monade anwendbar. Daher muss mit dem oben gezeigten bind-Operator `(>>=)` gearbeitet werden.

Im AST befindet sich der Knoten `Typed` mit den Parametern `e` und `te`, also einem Ausdruck und einem getypten Ausdruck. Im Curry-Quelltext hat ein getypter Ausdruck die Form `expr :: typeExpr`. In der Tokenfolge befindet sich also zunächst alles, was zum Ausdruck `e` gehört, dann ein doppelter Doppelpunkt – dargestellt durch das Token `DoubleColon` – und anschließend alles, was zum Typausdruck `te` gehört. Es muss also zunächst die Tokenfolge mit durchgereicht werden, um den Ausdruck `e` zu erweitern. Anschließend muss das Token `DoubleColon` von der Tokenfolge konsumiert und seine Position gespeichert werden. Danach wird die so gekürzte Tokenfolge wieder durchgereicht, damit der Typausdruck `te` erweitert werden kann. Diese Methode entspricht deterministischem Parsen der Tokenfolge mit einem Orakel für die Regelauswahl, weil für jeden Knoten des AST genau eine Regel existiert.

Betrachte man in Listing 5.5 für das minimale Beispiel `x :: Int` die Tokenfolge und den relevanten Ausschnitt des AST. Dort sind die soeben beschriebenen Gegebenheiten zu sehen, beispielsweise wie das Token `DoubleColon` in der Tokenfolge, aber nicht im AST auftaucht. Das liegt daran, dass im AST implizit bekannt ist, dass die Struktur von `Typed` die Form `expr :: typeExpr` hat, während die Tokenfolge hingegen eine genaue Abbildung aller Token eines Programms ist.

Mit den in Listing 5.5 abgebildeten Strukturen lässt sich nun die Funktion `apExpr` für den Fall `Typed` aus Listing 5.4 parallel an Tokenfolge und AST genau verfolgen: Für den ersten Ausdruck `Expression` wird erneut `apExpr` aufgerufen, hier tritt der nicht abgebildete Fall ein, dass der Ausdruck eine `Variable` ist. Durch den `bind`-Operator wird das Ergebnis dieses Aufrufs an `e'` gebunden und das entsprechende Token mitsamt seiner Position `((1,1),(Id "x"))` von der Tokenfolge konsumiert. Es folgt der Aufruf von `tokenPos DoubleColon`. Die Definition von `tokenPos` ist in Listing 5.6 zu sehen. Auch hier wird durch den `bind`-Operator das Ergebnis an die Position `p` des Tokens `DoubeColon` gebunden und das Token mit Position aus der Tokenfolge entfernt. Der dritte Aufruf von `apTypeExpr` mit dem AST-Knoten `TypeExpr` sowie der verbliebenen Tokenfolge `[((1,6),(Id "Int")),((2,1),EOF))]`

```
apExpr :: AST.Expression -> APM PosAST.Expression
apExpr (AST.Typed e te) =
  apExpr     e           >+= \ e'  ->
  tokenPos   DoubleColon >+= \ p   ->
  apTypeExpr te          >+= \ te' ->
  returnP (PosAST.Typed e' p te')

apExprDo :: AST.Expression -> APM PosAST.Expression
apExprDo (AST.Typed e te) =
  do e'  <- apExpr     e
     p   <- tokenPos   DoubleColon
     te' <- apTypeExpr te
     return (PosAST.Typed e' p te')
```

Listing 5.4.: Funktion `apExpr` für den Fall `Typed`, tatsächliche und `do`-Notation

```
-- Token
[ ((1,1), (Id "x")   )
, ((1,3), DoubleColon)
, ((1,6), (Id "Int") )
, ((2,1), EOF        )
]

-- Excerpt from AST
(Typed (Expression (Variable (QualIdent Nothing (Ident (1,1) "x" 1))))
       (TypeExpr   (VariableType (Ident (1,11) "Int" 0))))
```

Listing 5.5.: Tokenfolge und AST des Beispiels `x :: Int`

funktioniert analog zu den eben beschriebenen. Zu guter Letzt wird aus den nach und nach berechneten Parametern des AST-Knotens `AST.Typed e te` der erweiterte `PosAST`-Knoten `PosAST.Typed e' p te'` zusammengesetzt und zurückgegeben.

```
-- |Return position of token t
tokenPos :: Token -> APM Pos
tokenPos t = getTokenPos >+= \(p, t') -> ensure (t == t') p

-- |Get the next position, removing it from the state
getTokenPos :: APM (Pos, Token)
getTokenPos (t : ts) = (t, ts)
getTokenPos [] = error "Tokenstream is empty."

-- |Ensure a predicate. If predicate is `False`, fail.
ensure :: Bool -> a -> APM a
ensure b x =
  if b
    then returnP x
    else error $ "(AST.APM): Function `ensure` failed. "
              ++ "This should have been returned: " ++ show x
```

Listing 5.6.: Definition der Funktion `tokenPos` mit Hilfsfunktionen

5.3.2. Optionale Schlüsselwörter

Eine erwähnenswerte Besonderheit bei der Erweiterung des AST ist die Behandlung von optional auftretenden Code-Stücken, zu denen in Kapitel 5.2 schon Beispiele wie `where decl` oder Klammern gezeigt wurden. Wie bereits erwähnt, werden optionale Positionen im PosAST über ein `(Maybe Pos)` modelliert, wie aber funktioniert die Erweiterung?

Dafür stellt das Modul `AST.APM` die besondere Funktion `maybeTokenPos` zur Verfügung, die analog zu der in Listing 5.6 abgebildeten Funktion `tokenPos` zu verwenden ist.

```
-- |Return `Just` position of optional token or `Nothing`
maybeTokenPos :: Token -> APM (Maybe Pos)
maybeTokenPos t ts@((p,t') : ts')
  | t == t'   = (Just p , ts')
  | otherwise = (Nothing, ts )
maybeTokenPos _ []  = error "Tokenstream is empty."
```

5.3.3. Klammern

Auch für das Erweitern von geklammerten Strukturen aller Art stehen im Modul `AST.APM` besondere Funktionen zur Verfügung, die jeweils die öffnenden und schließenden Symbole aus der Tokenfolge entfernen und ihre Positionen speichern. Ein Beispiel dafür ist in Listing 5.7 abgebildet, andere Arten von Klammern funktionieren analog dazu.

```
-- |Parse an expression that is surrounded by braces
braces :: APM a -> APM (Pos, a, Pos)
braces f = between (tokenPos LeftBrace) f (tokenPos RightBrace)

-- |Parse an expression that is surrounded
-- |by some kind of opening and closing symbols
between :: APM a -> APM b -> APM c -> APM (a, b, c)
between open f close =
  open  >+= \ po ->
  f     >+= \ pf ->
  close >+= \ pc ->
  returnP (po, pf, pc)
```

Listing 5.7.: Funktionen für das Parsen von geklammerten Ausdrücken

5.3.4. Optionale Klammern

Eine ähnliche Funktion gibt es für optionale Klammern. Hierbei ist es wichtig, dass immer genau das schließende Symbol, das zum vorigen öffnenden Symbol gehört, vom Tokenstream entfernt wird. Als Beispiel dient der Import einer einfachen, nicht geklammerten Funktion: `import List (intercalate)`. Würde hier die Erweiterung ohne eine besondere Funktion, sondern mittels `maybeTokenPos` durchgeführt werden, wie im folgenden Ausschnitt einer add-Position-Funktion, würde dies zu einem Problem führen.

```
  maybeTokenPos LeftParen  >+= \ mpl ->
  apIdent       i          >+= \ i'  ->
  maybeTokenPos RightParen >+= \ mpr ->
```

Beim Beispiel `import List (intercalate)` würde nun zunächst die öffnende Klammer, die zur `Import`-Struktur gehört, aus der Tokenfolge entfernt werden. Dann wäre

der obige Aufruf von `maybeTokenPos LeftParen` an der Reihe. Hier wird festgestellt, dass in der Tokenfolge keine öffnende Klammer folgt. Also wird weitergemacht mit `apIdent i`. Anschließend jedoch findet `maybeTokenPos RightParen` eine schließende Klammer an nächster Stelle in der Tokenfolge und konsumiert sie. Ab hier schlägt das Erweitern des AST fehl, weil die schließende Klammer der `Import`-Struktur nicht mehr in der Tokenfolge vorhanden ist, Tokenfolge und AST also nicht mehr synchron laufen.

Um dieses Problem zu vermeiden, sorgt die Funktion `maybeEnclosedBy` aus Listing 5.8 dafür, dass ein schließendes Symbol nur aus der Tokenfolge entfernt wird, wenn vorher das entsprechende öffnende Symbol gefunden wurde.

```
-- |Parse an expression that might be surrounded by parens
maybeParens :: APM a -> APM ((Maybe Pos), a, (Maybe Pos))
maybeParens f = maybeBetween [(LeftParen, RightParen)] f

-- |Parse an expression that might be surrounded
-- |by any kind of opening and closing symbols
maybeBetween :: [(Token, Token)] -> APM a
                     -> APM ((Maybe Pos), a, (Maybe Pos))
maybeBetween [] f = f >+= \ pf -> returnP (Nothing, pf, Nothing)
maybeBetween ((o, c):ocs) f =
  readToken >+= \t -> case (t == o) of
    True -> between (maybeTokenPos o) f (maybeTokenPos c)
    _    -> maybeEnclosedBy ocs f
```

Listing 5.8.: Funktionen für das Parsen von eventuell geklammerten Ausdrücken

Auf die in diesem Kapitel beschriebene Weise wird der gesamte AST traversiert und Teilbaum für Teilbaum erweitert, während die Tokenfolge Stück für Stück konsumiert wird, bis nur noch das Tupel mit dem Token `EOF` übrig ist. Die Buchstaben „EOF“ kennzeichnen das Ende der Datei (englisch: *End Of File*). Durch dieses Token ist erkennbar, ob die Quelltext-Datei vollständig verarbeitet wurde. Durch diesen Vorgang entsteht ein um Positionsangaben erweiterter AST, auf dessen Grundlage die Überprüfung der Stilrichtlinien möglich ist.

6. CASC - Curry Automatic Style Checker

Das entwickelte Werkzeug `casc` wurde nach seiner Aufgabe benannt: `casc` ist ein Akronym für **C**urry **A**utomatic **S**tyle **C**hecker. Ein erster grober Überblick über das erwünschte Verhalten und die ungefähre Struktur von `casc` wurde schon in Kapitel 4 gegeben. Hier werden einzelne Programmelemente in der Reihenfolge vorgestellt, in der sie bei einem Durchlauf von `casc` benötigt werden.

Das Herzstück des Werkzeugs `casc` ist das gleichnamige Hauptprogramm. Hier ist die `main`-Funktion enthalten, die zunächst die Argumente, mit denen das Programm über die Kommandozeile aufgerufen wurde, an die Funktion `getCheckOpts` im Modul `Opts` übergibt. Dort werden sie geparst. Eventuell dabei auftretende Fehler werden gesammelt, um dem Nutzer Rückmeldung geben zu können. Daher ist die Rückgabe dieser Funktion das Quadrupel `(prog, opts, files, errs)`, das den Programmnamen, Optionen, Datei- beziehungsweise Ordnernamen und die Parse-Fehler enthält.

6.1. Optionen

Mittels der Optionen wird in der Funktion `casc` das weitere Vorgehen entschieden. Sie bestehen aus vier Elementen:

```
-- |Casc options
data Options = Options
  { optMode       :: Mode
  , optVerbosity  :: Verbosity
  , optRecursive  :: Bool
  , optColour     :: ColMode
  }
```

`optMode`: Der `optMode` bestimmt den Ausführungsmodus des Programms. Folgende Modi stehen zur Verfügung:

```
-- |Modus operandi of the program
data Mode
  = ModeHelp
  | ModeVersion
  | ModeCheck
```

`ModeHelp` wird durch `-h`, beziehungsweise `--help` angesteuert. Ist er aktiv, werden Informationen zur Benutzung des Programmes und den verfügbaren Optionen ausgegeben. `ModeVersion`, der über `-v` oder `--version` aufgerufen wird, zeigt die Versionsnummer. Im Hauptmodus `ModeCheck`, der per Default aktiv ist, überprüft `casc` Curry-Quelltext in Hinblick auf Stilrichtlinien.

`optVerbosity`: Die `optVerbosity`, gibt an, wie viele Ausgaben `casc` während eines Programmlaufes machen soll. Es gibt folgende Möglichkeiten:

```
-- |Verbosity level
data Verbosity
  = VerbQuiet
  | VerbStatus
  | VerbDebug
```

Ein Durchlauf des Programmes mit der Verbosity `VerbQuiet`, einzustellen über `-q` oder `--quiet`, erzeugt bis auf die gefundenen Stilfehler keine Ausgaben. `VerbStatus` sorgt dafür, dass während des Durchlaufs kurze Statusmeldungen ausgegeben werden, beispielsweise „Extending abstract syntax tree...“ und „Checking...“. Diese Variante ist per Default eingestellt.

`optRecursive`: Dies ist ein Boolescher Wert. Ist er `True` und wurden Ordnernamen zur Überprüfung übergeben, werden auch alle Curry-Dateien in den jeweiligen Unterordnern überprüft. In den Default-Optionen ist dieser Parameter auf `False` gesetzt. Mittels `-r` oder `--recursive` lässt sich die Option einschalten.

`optColour`: Als letztes besteht noch die Möglichkeit, die farbige Ausgabe von Nachrichten ein- oder auszuschalten. Dazu existieren folgende Modi:

```
-- |Colour Mode
data ColMode
  = ColOn
  | ColOff
  | ColAuto
```

Per `--colour n`, wobei n zwischen 0 und 2 liegt, lassen sie sich ansteuern. Die farbige Ausgabe ist entweder angeschaltet (`ColOn`), ausgeschaltet (`ColOff`), oder wie auch per Default eingestellt, automatisch aktiv (`ColAuto`). „Automatisch“ bedeutet in diesem Fall, dass die Ausgabe nur farbig ist, wenn sie auf ein Terminal erfolgt. Das ist sinnvoll, da in die zu färbenden Strings Ansi-Codes eingefügt werden. Möchte der Nutzer aber die Ausgabe beispielsweise in einer Datei speichern oder direkt in einem Editor anzeigen, sollten keine Ansi-Codes eingefügt werden.

6.2. Vorbereitung

Ist der Ausführungsmodus von `casc` auf `ModusCheck` gesetzt, müssen die in Kapitel 5.1 erläuterten Vorbereitungen getroffen werden. Dies geschieht für jede zu überprüfende Datei einzeln. Zunächst werden also mit dem System-Aufruf

```
cymake --tokens --parse-only -W none -q -i ~/bin/pakcs/lib/ file.curry
```

vom Build-Tool des Curry-Frontends die Tokenfolge und der AST der zu überprüfenden Curry-Datei `file.curry` erzeugt. Dabei sollen keine Warnungen (`-W none`) und keine Statusmeldungen (`-q`) angezeigt werden. Außerdem muss der Pfad zu PAKCS- oder KiCS2-Bibliotheken mit `-i path` angegeben werden. Tokenfolge und AST werden im `.curry`-Ordner gespeichert. Die in Kapitel 5.3 erläuterte Erweiterung des AST um Positionen wird mit dem Aufruf von `apModule` („add position to `Module`“) durchgeführt und das Ergebnis in der Variable `posAST` abgelegt. Dies ist die Grundlage für die Durchführung der später in Kapitel 6.4 näher erläuterten Checks.

6.3. Konfiguration

Um die Benutzung von `casc` für Benutzer angenehmer zu machen, ist das Werkzeug über eine Datei konfigurierbar. Hier lassen sich einzelne zu überprüfende Kriterien ein- und ausschalten. Ein Anwendungsbeispiel dafür könnte sein, dass jemand zwar alle Stilfehler aufgezeigt bekommen möchte, aber das Sprachkonstrukt `let expr in decl` grundsätzlich anders einrückt, als es die in `casc` implementierte Stilrichtlinie vorgibt.

6.3.1. Elemente der Konfigurationsdatei

Die Syntax der Konfigurationsdatei gibt vor, dass alle verfügbaren Kriterien zur Stilüberprüfung (*Checks*) aufgelistet werden. Sie sind nach dem AST-Element benannt, auf das sie sich beziehen, beispielsweise `Data.Bars` oder `Expression.IfThenElse`. Nach dem Namen eines jeden Checks folgt ein Gleichheitszeichen und eine Zahl zwischen 0 und 2. Eine 0 sorgt dafür, dass der entsprechende Check nicht durchgeführt wird. Mit einer 1 wird der Check durchgeführt und gegebenenfalls eine Nachricht ausgegeben. Durch eine 2 wird die automatische Korrektur des Kriteriums nach der Überprüfung aktiviert, sofern auch der weiter unten erläuterte Parameter `autoCorrect` auf „yes“ gesetzt ist. Einen Ausschnitt der Liste mit Checks ist in Listing 6.1 zu sehen. Eine vollständige Konfigurationsdatei befindet sich in Anhang B.

```
Expression.Case               = 1
Expression.IfThenElseKW       = 2
Expression.IfThenElseSubExpr = 1
Expression.Let                = 0
Import.Explicit               = 0
```

Listing 6.1.: Ausschnitt aus der Konfigurationsdatei für casc

Als nächstes kann der Parameter für die gewünschte maximale Zeilenlänge eingestellt werden. Dieser wird in die Zeile `maxLength = ...` eingetragen.

Für den Aufruf vom Curry-Build-Tool `cymake` muss bekannt sein, an welchem Ort PAKCS- beziehungsweise KiCS2-Bibliotheken zu finden sind. Den Pfad dazu wird in Zeile `pakcsLibPath = ...` festgehalten.

Der letzte Parameter der Konfigurationsdatei gibt an, ob `casc` automatisch Korrekturen an den eingegebenen Dateien vornehmen soll. Dies ist natürlich nur möglich, wenn mindestens ein Check mit 2 gekennzeichnet ist. Nur, wenn in der Zeile `autoCorrect = ...` ein „yes“ steht, wird die automatische Korrektur aktiviert.

Das Einlesen der Konfigurationsdatei geschieht im Modul `Config.ReadConfig`. Sie wird mittels der Funktion `lines` in ihre Zeilen zerlegt. Die Liste der Checks wird mit der Funktion `readCfg` eingelesen. Es entsteht eine Liste aus Tupeln, die jeweils den Namen des Checks sowie eine Ziffer zwischen 0 und 2 für die entsprechende Einstellung enthalten. Aus dieser Liste werden die beiden Listen `checkList` und `corrList` erstellt, indem alle Checks mit einer 1 der zu überprüfenden Liste `checkList` und alle Checks mit einer 2 der zu korrigierenden Liste `corrList` zugeordnet werden. Die anderen Parameter werden über eigene Funktionen eingelesen, die mittels `last $ words ln`

jeweils das letzte „Wort“ der Zeile, die mit dem Parameternamen beginnt, lesen. Die Funktion `readConfig` ruft alle Auslese-Funktionen auf und gibt ein Quadrupel zurück, das aus den vier Elementen der Konfigurationsdatei besteht.

6.3.2. Voreingestellte Parameter

Die Konfigurationsdatei wird zunächst in dem Verzeichnis gesucht, von dem aus `casc` aufgerufen wurde. Wird dort keine Datei namens `.cascrc` gefunden, wird auch im Home-Ordner des Benutzers gesucht.

Ist keine Konfigurationsdatei vorhanden, wird eine voreingestellte Default-Konfiguration zurückgegeben. Diese sieht so aus, dass alle Kriterien auf „1“ geschaltet sind, also zwar überprüft werden, jedoch keine automatische Korrektur vorgenommen wird. Die maximale Zeilenlänge beträgt konventionelle 80 Zeichen, nach den Bibliotheken wird im Pfad `~/bin/pakcs/lib` gesucht und die automatische Korrektur ist ausgeschaltet.

Sind in einer Konfigurationsdatei bei den Parametern `maxLength`, `libPath` und `autoCorrect` keine oder unerwartete Werte eingetragen, wird jeweils auf den Wert aus der Default-Konfiguration zurückgegriffen. Wenn sich unerwartete Zeilen in der Konfigurationsdatei befinden, wird eine kurze Hilfe zu ihrer Syntax ausgegeben. Unerwartete Zeilen sind in diesem Fall solche, die nicht wie ein Kommentar mit `--`, oder den Parameternamen `maxLength`, `libPath` und `autoCorrect` beginnen, oder mit einer 0, 1 oder 2 enden.

6.4. Checks

Nachdem der AST wie weiter oben beschrieben um Positionen erweitert wurde, wird in der Hauptfunktion `casc` durch den Aufruf

```
let msgs = checkModule posAST
        ++ condMsg (shallCheck CLineLength) (checkLine src)
```

seine Überprüfung eingeleitet. Dass die Überprüfung der Zeilenlänge zusätzlich einzeln aufgerufen werden muss, hängt damit zusammen, dass sie nicht auf dem PosAST durchgeführt werden kann, sondern die Zeilenlänge direkt in der Quelldatei prüft.

Durch das im vorigen Abschnitt beschriebene Auslesen der Konfigurationsdatei kann eine Liste erstellt werden, die die zu überprüfenden Kriterien enthält. Dies sind genau die Kriterien, die in der Konfigurationsdatei mit 1 oder 2 markiert sind. Über die Funktion `shallCheck` wird überprüft, ob ein Check sich in dieser Liste befindet. Wenn nicht, wird er übersprungen.

Die Funktion `checkModule` im Modul `Check.CheckPosAST` ist der Ausgangspunkt für die Traversierung des zu überprüfenden PosAST. Alle `check`-Funktionen haben eine Signatur der Form

```
checkNode :: Node -> [Message] ,
```

wobei `Node` hier für ein beliebiges Element des PosAST steht und eine `Message` aus einer Quellcode-Position und einer Fehlermeldung besteht:

```
data Message = Message Pos String
```

6.4.1. `check`-Funktion für `Expression`

In Listing 6.2 ist die konkrete `check`-Funktion `checkExpression` zu sehen. Es wird zunächst eine Fallunterscheidung für die verschiedenen Ausdrücke, bei denen noch zusätzliche Knoten durchlaufen werden müssen, benötigt. Außerdem gibt es bei `checkExpression` die Besonderheit, dass ein `Expression` rekursiv definiert ist. Das bedeutet, dass ein Ausdruck aus beliebig vielen Ebenen von Teilausdrücken bestehen kann, die auch alle überprüft werden müssen. Das gleiche Phänomen tritt bei den ebenfalls rekursiv definierten PosAST-Elementen `TypeExpr`, `Lhs` und `Pattern` auf.

```
-- |Signature for check functions
type CheckF a = a -> [Message]

-- |Check Expression
checkExpression :: CheckF Expression
checkExpression e = case e of
  ListCompr _ _   _  sts _  _   -> [ m | e' <- subExprs e, m <- check e' ]
                                   ++ concatMap checkStatement sts
  Do        _ sts _             -> [ m | e' <- subExprs e, m <- check e' ]
                                   ++ concatMap checkStatement sts
  Case      _ _   _ _    alts   -> [ m | e' <- subExprs e, m <- check e' ]
                                   ++ concatMap checkAlt alts
  _                             -> [ m | e' <- subExprs e, m <- check e' ]
where
  check = expressionCheck
```

Listing 6.2.: Funktion `checkExpression`

Die Ergebnisliste von Nachrichten wird jeweils so aufgebaut, dass zunächst der eigentliche Check für die Teilausdrücke aufgerufen wird und das Ergebnis mit der Ergebnisliste des weiteren Baumdurchlaufes konkateniert wird.

Zur Namensklärung: Die Funktionen `checkAstElement` sind die Funktionen, die den AST traversieren und die Checks für die Stilkriterien, die `astElementCheck` heißen, aufrufen.

Die tatsächlichen Checks befinden sich ebenfalls im Ordner `Check` und sind dort nach Themengebieten in einzelnen Modulen organisiert. Ein jedes dieser Module ist in zwei Abschnitte unterteilt, nämlich einen Check und eine oder mehrere Validierungsfunktionen. Beispielhaft wird auch hier wieder die Überprüfung von Ausdrücken betrachtet, konkret vom Ausdruck `IfThenElse`. Die entsprechenden Funktionen befinden sich im Modul `Check.Expression`.

```
-- |Check Expression
expressionCheck :: CheckF Expression
expressionCheck e = case e of
  IfThenElse pi i pt t pe el -> keywords ++ subexpr
    where
      keywords
        = condMsg (shallCheck CIfThenElseKW && not (validITEkw pi pe pt))
                  [Message pi "IfThenElse: Wrong indentation of keywords."]
      subexpr
        = condMsg (shallCheck CIfThenElseSubExpr
                              && not (validITEsubE pi i pt t pe el))
                  [Message pi
                            "IfThenElse: Wrong indentation "
                       ++ "of subexpressions."]
```

Listing 6.3.: Funktion `expressionCheck`

6.4.2. Check für `if-then-else`

In Listing 6.3 ist der Check `expressionCheck` für den Fall, dass der zu überprüfende Ausdruck `IfThenElse` ist, zu sehen. Die Rückgabe setzt sich aus den lokalen Funktionen `keywords` und `subexpr` zusammen, die jeweils für ein implementiertes Kriterium stehen, das `IfThenElse`-Ausdrücke betrifft. Die Funktion `condMsg` ist eine Hilfsfunktion, die folgendermaßen definiert ist:

```
-- |Conditional Message: return message if predicate is `True`
condMsg :: Bool -> [Message] -> [Message]
condMsg p msg = if p then msg else []
```

Es wird also für beide Kriterien durch `shallCheck` überprüft, ob in der Konfigurationsdatei ihre Überprüfung vorgesehen ist. Dank der in Curry genutzten Lazy

Evaluation, die dafür sorgt, dass Ausdrücke nur so weit ausgewertet werden, wie das Ergebnis gerade benötigt wird, wird der zweite Teil des mit `&&` verknüpften Prädikats nur dann überhaupt geprüft, wenn der erste Teil `True` ist. So werden Checks, die in der Konfigurationsdatei ausgeschaltet sind, in der Auswertung übersprungen. Der zweite Teil des Prädikats ist eine Validierungsfunktion. Ist der Ausdruck nicht valide, wird die entsprechende Nachricht in einer Liste zurückgegeben. Ansonsten ist die Rückgabe eine leere Liste, also keine Nachricht. Auf diese Art und Weise funktionieren alle Checks.

6.4.3. Validierungsfunktionen für `if-then-else`

Die zur lokalen Funktion `keywords` gehörige Validierungsfunktion ist in Listing 6.4 zu sehen.

```
-- |Check indentation of keywords
validITEkw :: Pos -> Pos -> Pos -> Bool
validITEkw pi pt pe =
     (line pi == line pt && line pt == line pe)
  || (col pt == ((col pi) + 2) && col pt == col pe)
```

Listing 6.4.: Validierungsfunktion für `if-then-else`-Schlüsselwörter

Validierungsfunktionen sind möglichst übersichtliche logische Gleichungen, die abbilden, wie das Layout eines gültigen Ausdrucks aussieht. In diesem Fall beispielsweise werden die beiden folgenden Layouts als valide anerkannt:

```
if ifExpr then thenExpr else elseExpr

if ifExpr
  then thenExpr
  else elseExpr
```

Diese Validierungsfunktion reicht allerdings noch nicht aus, um einen kompletten `IfThenElse`-Ausdruck zu validieren, denn werden nur die Positionen der Schlüsselwörter überprüft, wären auch folgende Formatierungen valide:

```
if   ifExpr then     thenExpr else   elseExpr

if  ifExpr
  then      thenExpr
  else   elseExpr
```

Das ist natürlich nicht erwünscht. Um also auch die korrekte Einrückung der Teilausdrücke zu überprüfen, existiert die zweite Validierungsfunktion; abgebildet in Listing 6.5. Durch sie wird sichergestellt, dass die Teilausdrücke mit genau einem Leerzeichen Abstand zu den zugehörigen Schlüsselwörtern beginnen.

```
-- |Check space between keywords and subexpressions
validITEsubE :: Pos -> Expression -> Pos -> Expression
             -> Pos -> Expression -> Bool
validITEsubE pi i pt t pe el =
     (col (exprPos  i) == ((col pi) + 3))
  && (col (exprPos  t) == ((col pt) + 5))
  && (col (exprPos el) == ((col pe) + 5))
```

Listing 6.5.: Validierungsfunktion für `if-then-else`-Teilausdrücke

6.4.4. Implementierte Stilrichtlinien

An dieser Stelle wird in alphabetischer Reihenfolge ein Überblick über alle bisher implementierten Stilrichtlinien gegeben.

`Check.ConstrDecl`**:** In diesem Modul befinden sich vier Validierungsfunktionen zur Überprüfung von Record-Deklarationen. Die Richtlinien für eine korrekt formatierte Record-Deklaration lauten:

- Geschweifte Klammern und Kommata befinden sich untereinander
- Konstruktoren befinden sich untereinander
- Doppelpunkte befinden sich untereinander
- Typen befinden sich untereinander

```
data Person = Person
  { firstName :: String
  , lastName  :: String
  , age       :: Int
  }
```

`Check.Decl`: Das Modul enthält drei Validierungsfunktionen zur Überprüfung von `data`-Deklarationen mit Konstruktoren. Die Richtlinien lauten:

- Gleichheitszeichen und Bars befinden sich untereinander *oder* alle in einer Zeile
- Konstruktoren befinden untereinander *oder* alle in einer Zeile
- Die ersten Komponenten befinden sich untereinander *oder* alle in einer Zeile

```
data example = X ... | Y ... | Z ...

data example = X ...
             | Y ...
             | Z ...
```

`Check.Expression`: In diesem Modul befinden sich Validierungsfunktionen für die Ausdrücke `IfThenElse`, `Let` und Alternativen von Fallunterscheidungen mittels `case`.
Die Richtlinien für `IfThenElse` lauten:

- Schlüsselwörter `if`, `then` und `else` stehen alle in einer Zeile *oder* untereinander, wobei `then` und `else` von `if` aus jeweils um zwei Leerzeichen eingerückt sind
- Teilausdrücke, die hinter den Schlüsselwörtern folgen, haben nur ein Leerzeichen Abstand zum entsprechenden Schlüsselwort

```
if ... then ... else ...

if ...
  then ...
  else ...
```

Die Richtlinien für `Let` lauten:

- Schlüsselwörter `let` und `in` stehen entweder in derselben Zeile *oder* untereinander
- Deklarationen stehen untereinander
- Gleichheitszeichen stehen untereinander

```
let decl in expr

let decl1 = ...
    decl2 = ...
    ...
in ...
```

Die Richtlinie für Alternativen von Fallunterscheidungen mittels `case` sieht vor, dass Alternativen untereinander ausgerichtet werden:

```
case e of E1 -> ...
          E2 -> ...

case e of
  E1 -> ...
  E2 -> ...
```

`Check.ImportExport`: Das Modul enthält Validierungsfunktionen für Imports und Exports.

Die Richtlinie für den Import besagt, dass anstelle des Imports von ganzen Modulen Funktionen möglichst explizit importiert werden sollten.

Beim Export von Funktionen sollten die Klammern und Kommata entweder alle in derselben Zeile stehen *oder* untereinander ausgerichtet sein:

```
module Example (f1, f2, ...) where

module Example ( f1
               , f2
               , ...
               ) where
```

`Check.LineLength`: Wie in Kapitel 6.3 erwähnt, ist die Richtlinie für die maximale Zeilenlänge individuell anpassbar. Normalerweise sollte eine Zeile aber nicht mehr als 80 Zeichen enthalten. Auch Kommentare sollten diese Beschränkung nicht überschreiten.

`Check.Pattern`: Die konkreten Richtlinien für Pattern sind noch nicht implementiert. Da bei Pattern jedoch aufgrund ihrer rekursiven Struktur auch die untergeordneten Sub-Pattern berechnet und überprüft werden müssen, sind die grundlegenden Strukturen

zur Überprüfung von Pattern im Durchlauf des PosAST im Modul `Check.CheckPosAST` schon vorhanden. Die Stilrichtlinien für Pattern sehen vor, dass Pattern bei Funktionen mit mehreren Regeln untereinander angeordnet werden:

```
test [] [] [] = ...
test [] [] _  = ...
test [] _  _  = ...
test xs ys zs = ...
```

`Check.Rhs`: Dieses Modul stellt zwei Validierungsfunktionen für `GuardedRhs`, also bedingte rechte Gleichungsseiten, zur Verfügung. Die Richtlinien legen fest:

- Bars stehen untereinander
- Gleichheitszeichen stehen untereinander

```
abs n
  | n < 0     = -n
  | otherwise = n
```

`Check.TypeExpr`: Dieses Modul implementiert die erste Hälfte einer Richtlinie, die besagt, dass jeweils vor und nach einem Pfeil ein Leerzeichen zu setzen ist. Ob nach einem Pfeil genau ein Leerzeichen folgt, ist leicht zu erkennen und wird deshalb hier überprüft. Ob jedoch vor einem Pfeil genau ein Leerzeichen steht, lässt sich nur herausfinden, wenn die Länge des Elements vor dem Pfeil bekannt ist. Diese Richtlinie erlaubt also Folgendes, wobei das zweite aber eigentlich nicht erwünscht ist:

```
test1 :: Int -> Int

test2 :: Int-> Int
```

6.5. Nachrichten

Die Rückgabe einer `check`-Funktion ist eine Liste von Nachrichten. Während des Baumdurchlaufes mit allen `check`-Funktionen werden alle diese Nachrichten in einer Ergebnisliste gesammelt. Diese Ergebnisliste wird schließlich dem Benutzer mittels einer Pretty-Print-Operation ausgegeben:

```
-- |Pretty print messages
prettyMsg :: Bool -> [Message] -> String
prettyMsg col msgs = "\n" ++ (showMsg col $ sortMsg msgs)
```

Dafür werden zunächst eventuelle Nachrichten von der Überprüfung der Zeilenlänge einsortiert, denn, wie weiter oben erwähnt, muss die Überprüfung der Zeilenlänge einzeln aufgerufen werden. So erscheinen eventuelle Nachrichten zur Zeilenlänge erst nach allen anderen Nachrichten. Zum Sortieren der Nachrichten wird die Funktion `quickSortBy` aus der Bibliothek `Sort` benutzt. Dafür wird ein Sortierkriterium, nämlich eine kleiner-gleich-Operation auf Nachrichten, benötigt:

```
-- |Sort Messages by position to which they refer.
-- |We need this because messages concerning line
-- |length are unordered
sortMsg :: [Message] -> [Message]
sortMsg msg = quickSortBy leqMsg msg

-- |Less or equal function on messages
leqMsg :: Message -> Message -> Bool
leqMsg (Message p1 _) (Message p2 _) = line p1 <= line p2
```

Die sortierten Nachrichten werden von der `show`-Funktion optisch aufbereitet. Hier spielt der Parameter `col` eine Rolle, der von `casc` durchgereicht und als lokale Funktion folgendermaßen bestimmt wurde:

```
where col  = case optColour opts of
               ColOn   -> True
               ColOff  -> False
               ColAuto -> unsafePerformIO $ hIsTerminalDevice stdout
```

`col` ist `True`, wenn es in den Aufrufoptionen gesetzt wurde, oder es auf `ColAuto` steht und die Standardausgabe ein Terminal ist.

```
-- |Show Messages
showMsg :: Bool -> [Message] -> String
showMsg _ []  = ""
showMsg col ((Message p m) :ms)
  | col       = "    " ++ (yellow (show p)) ++ " "
```

```
                ++ (red m) ++ "\n" ++ showMsg col ms
  | otherwise = "    " ++ show p ++ " " ++ m ++ "\n"
                       ++ showMsg col ms
```

Sollen die Nachrichten also farblich hervorgehoben werden, passiert dies in der `show`-Funktion durch die Befehle `red` und `yellow` aus der Bibliothek `AnsiCodes`. Sie fügen Ansi-Codes in die Strings ein, die die bunte Darstellung im Terminal ermöglichen. Zusätzlich werden Nachrichten um vier Leerzeichen eingerückt, damit sie sich auch im schwarz-weiß-Modus besser vom Terminal abheben.

Aus der Nachrichtenliste

```
[
  Message (3,1)   "Import: Used functions should be imported "
             ++   "explicitly unless you are importing many "
             ++   "at once."
, Message (6,0)   "Line is longer than 80 characters."
, Message (9,9)   "Let: Equality signs are not aligned."
, Message (9,5)   "Let: Keywords 'let' and 'in' are not aligned."
, Message (16,14) "IfThenElse: Wrong indentation of keywords."
]
```

wird mit eingestellter Colour-Option auf die beschriebene Art und Weise folgende Ausgabe:

```
(3,1) Import:  Used functions should be imported explicitly
      unless you are importing many at once.
(6,0) Line is longer than 80 characters.
(9,5) Let:  Equality signs are not aligned.
(9,9) Let:  Keywords let and in are not aligned.
(16,14) IfThenElse:  Wrong indentation of keywords.
```

Abbildung 6.1.: Ausgabe von colorierten Nachrichten auf dem Terminal

6.6. Automatische Korrektur

Die automatische Korrektur wird durch die Funktion `correctOne` vorgenommen. Sie wird nur nach der Überprüfung einer Datei aufgerufen, sofern die entsprechende Option

gesetzt ist, in der Datei überhaupt Fehler gefunden wurden und in der Konfigurationsdatei mindestens ein Kriterium durch eine 2 zur automatischen Korrektur gekennzeichnet ist.

Im Modul `AutoCorr.AutoCorrPosAST` befindet sich die Funktion `correctModule`. Von ihr ausgehend wird der vorher erstellte PosAST traversiert, indem `correct`-Funktionen für alle Knoten aufgerufen werden. Bis jetzt ist noch keine vollständige Traversierung implementiert, sondern nur ein Pfad durch den PosAST von `Module` zu `Expression`. Die Funktion `correctExpression` ist für die beiden Ausdrücke `IfThenElse` und `Let` implementiert. In Listing 6.6 wird sie am Beispiel von `IfThenElse` vorgestellt.

```
-- |Correct Expression
correctExpression :: Expression -> Expression
correctExpression expr = case expr of
  IfThenElse pi i pt t pe e
    -> if (shallCorrect CIfThenElseKW)
         then IfThenElse pi                        (correctExpression i)
                         (line pt, (col pt) + 2) (correctExpression t)
                         (line pe, (col pt) + 2) (correctExpression e)
         else IfThenElse pi                        (correctExpression i)
                         pt                        (correctExpression t)
                         pe                        (correctExpression e)
  ...
```

Listing 6.6.: Korrekturfunktion für if-then-else-Schlüsselwörter

Durch die `if`-Abfrage `(shallCorrect CIfThenElseKW)` wird überprüft, ob das Kriterium `Expression.IfThenElseKW` in der Konfigurationsdatei mit einer 2 für die automatische Korrektur markiert ist. Wenn nein, wird das `IfThenElse` mit unveränderten Positionen zurückgegeben und die Funktion `correctExpression` für die Teilausdrücke aufgerufen. Wenn ja, werden die Positionen der Schlüsselwörter `then` und `else` korrigiert, indem ihre Zeile beibehalten wird und die Spalte so gesetzt wird, dass sie vom Schlüsselwort `if` jeweils um zwei Leerzeichen eingerückt sind. Auch hier wird `correctExpression` wieder für die Teilausdrücke aufgerufen. Was hier noch fehlt, ist, dass die Teilausdrücke ebenfalls verschoben werden. Wie das funktionieren könnte, wird im Ausblick in Kapitel 9.2 beschreiben.

Die Rückgabe von `correctModule` ist ein PosAST, in dem Fehler, deren zugehörige Kriterien in der Konfigurationsdatei mit einer 2 markiert sind, korrigiert wurden.

Bis jetzt ist die Ausgabe der automatischen Korrektur aus Ermangelung eines exakten Pretty Printers noch so gelöst, dass der korrigierte PosAST die AST-Datei `file.cy` überschreibt. Dazu muss er jedoch zunächst wieder in einen normalen, nicht-erweiterten AST überführt werden. Dies erledigt das Modul `AST.RemovePositions`, das das Gegenstück zum Modul `AST.AddPositions` darstellt. Dort wurden, wie in Kapitel 5.3 beschrieben, Positionen durch die Traversierung des AST in Synchronisation mit der Tokenfolge hinzugefügt. Ausgehend von der Funktion `rpModule` („remove positions from `Module`“) wird der PosAST durchlaufen und Knoten für Knoten zurück in einen normalen AST überführt. Ein Beispiel dafür ist die Funktion `rpExpression`:

```
-- |Remove positions from Expression
rpExpression :: PosAST.Expression -> AST.Expression
rpExpression e = case e of
  PosAST.IfThenElse _  e1  _  e2 _ e3
    -> AST.IfThenElse (rpExpression e1)
                      (rpExpression e2)
                      (rpExpression e3)
  ...
```

Die Positionen der Schlüsselwörter werden wieder vergessen und die Teilausdrücke ebenfalls von Positionen befreit.

Einen Überblick darüber, wie die automatische Korrektur von Fehlern in Curry-Quelltext in Zukunft funktionieren könnte, befindet sich im Ausblick in Kapitel 9.2.

III.

Schlussbetrachtung

7. Bewertung

In diesem Kapitel wird das entwickelte Werkzeug zur Stilüberprüfung `casc` bewertet. Dazu werden zunächst die bekannten Schwächen des Programms mit möglichen Lösungen aufgeführt. Anschließend werden, um die Benutzbarkeit von `casc` im Alltag zu analysieren, die Ergebnisse einiger Zeitmessungen von Testläufen betrachtet.

7.1. Bekannte Schwächen

Es gibt drei bekannte Umstände, unter denen das fertige Programm nicht einwandfrei funktioniert. Sie werden an dieser Stelle mit Lösungsvorschlag erklärt. Der erste ist der Syntax von Curry geschuldet, die beiden anderen haben mit dem Auffinden des `.curry`-Ordners in bestimmten Fällen zu tun.

7.1.1. `where` mit leerer Deklarationsliste

In Kapitel 5.2.1 kamen bereits optionale Schlüsselwörter vor. Ein Paradebeispiel für ein solches Schlüsselwort ist immer wieder `where`, das am Ende von Funktionen stehen kann, um die Definition von lokalen Funktionen einzuleiten.

Aufgrund der bisherigen Implementierung stößt man bei der Erweiterung des AST für folgende Funktion auf ein Problem:

```
test = 3 where
```

Die Funktion `test` ist wider Erwarten syntaktisch korrektes Curry, obwohl hinter dem `where` keine Deklarationen lokaler Funktionen folgen.

Betrachte man die Implementierung der Funktion, die lokale Deklarationen (ein Element von rechten Gleichungsseiten) erweitert:

```
apLocalDecls :: [AST.Decl] -> APM (Maybe Pos, [PosAST.Decl])
apLocalDecls []       = returnP (Nothing, [])
apLocalDecls ds@(_:_) = tokenPos KW_where >+= \ mpw  ->
                        mapM apDecl ds    >+= \ ds'  ->
                        returnP (Just mpw, ds')
```

Die Funktion `apLocalDecls` wird mit einer Liste von Deklarationen `[Decl]` aufgerufen. Ist diese Liste leer, wird für die optionale Position des `where` ein `Nothing` zurückgegeben, sowie eine leere Liste anstelle der `[Decl]`-Liste mit erweiterten Deklarationen. Hat man es mit einer nicht-leeren Deklarationsliste zu tun, wird das Schlüsselwort `where` von der Tokenfolge konsumiert, seine Position gespeichert und die Deklarationen mit der Funktion `apDecl` erweitert.

Auf diese Weise ist es nicht möglich, bei leerer Deklarationsliste noch das Schlüsselwort `where` von der Tokenfolge zu entfernen. So geht die Synchronität von Tokenfolge und AST verloren und die Erweiterung des AST kann nicht fortgesetzt werden.

Die naheliegende Möglichkeit, die `apLocalDecls`-Regel für den Fall einer leeren Deklarationsliste anzupassen, wurde im Laufe der Arbeit getestet:

```
apLocalDecls [] = maybeTokenPos KW_where >+= \ mpw ->
                  returnP (mpw, [])
```

So würde das Schlüsselwort `where` aus der Tokenfolge konsumiert, seine Position gegebenenfalls in einem `Just Pos` gespeichert und ansonsten ein `Nothing` zurückgegeben. Der AST für den Fall `test = 3 where` könnte damit erfolgreich erweitert werden.

Dies führt aber an einer anderen Stelle zu Problemen, etwa bei folgendem Beispiel von geschachtelten rechten Gleichungsseiten:

```
test xs = case xs of
    [] -> y
  where y = 42
```

Ein Ausschnitt des abstrakten Syntaxbaums, der die Funktion `test` abbildet, ist in Listing 7.1 zu sehen. In den Zeilen 8, 14 und 21 beginnt jeweils eine `SimpleRhs`. Weil der AST nicht unbedingt auf den ersten Blick durchschaubar ist, ist in Abbildung 7.1 ein Überblick über die Schachtelung der `SimpleRhs` gegeben.

Nun dazu, was passieren würde, wenn `apLocalDecls` für die leere Liste so aussehen würde, wie zuletzt beschrieben: Die `SimpleRhs` aus Zeile 14 würde über `apRhs` erweitert

$$\underbrace{\texttt{= case xs of [] } \underbrace{\texttt{-> y}}_{\text{SimpleRhs Z. 14}} \texttt{ where y } \underbrace{\texttt{= 42}}_{\text{SimpleRhs Z. 21}}}_{\text{SimpleRhs Z. 8}}$$

Abbildung 7.1.: Schachtelung der `SimpleRhs` aus Listing 7.1

werden und im Zuge dessen auch `apLocalDecls` aufrufen. Durch `apLocalDecls` würde das im Tokenstream folgende `where` konsumiert werden, obwohl es *nicht* zur `SimpleRhs` aus Zeile 14 gehört, sondern zur `SimpleRhs` aus Zeile 8. So fehlt dann das `where`, wenn die `SimpleRhs` aus Zeile 8 weiter erweitert wird und Tokenfolge und AST sind nicht mehr synchron.

Eine mögliche Lösung des Problems wäre, ein `where` ohne darauf folgende Deklarationen in der Curry-Syntax gar nicht mehr zuzulassen. Ansonsten müsste bei der Erweiterung des AST die Schachtelungstiefe berücksichtigt werden, so dass keine nicht zur aktuellen Struktur gehörigen Schlüsselwörter von der Tokenfolge entfernt werden können. In jedem Fall sollte die Stilüberprüfung aber eine Warnmeldung ausgeben, wenn wie in `test = 3 where` ein überflüssiges `where` in einer Funktion steht.

```
1  [(FunctionDecl
2    (3,1)
3    (Ident (3,1) "test" 0)
4    [(Equation
5      (3,1)
6      (FunLhs (Ident (3,1) "test" 0)
7              [(VariablePattern (Ident (3,6) "xs" 1))])
8      (SimpleRhs
9        (3,11)
10       (Case Rigid
11             (Variable (QualIdent Nothing (Ident (3,16) "xs" 1)))
12             [(Alt (4,5)
13                   (ListPattern [])
14                   (SimpleRhs
15                     (4,11)
16                     (Variable (QualIdent Nothing (Ident (4,11) "y" 2)))
17                     []))])
18       [(PatternDecl
19         (5,9)
20         (VariablePattern (Ident (5,9) "y" 2))
21         (SimpleRhs
22           (5,13)
23           (Literal (Int (Ident (0,0) "_" 4) 42))
24           []))]))])]
```

Listing 7.1.: AST der Funktion `test`: geschachtelte `SimpleRhs`

7.1.2. Modulkopf

Wie in Kapitel 5.3 beschrieben, werden die Dateien mit der Tokenfolge und dem AST vom Curry-Frontend im `.curry`-Ordner gespeichert. In ihm wird die Ordnerstruktur der Curry-Dateien über eine Ebene hinweg nachgebildet. Wo sich dieser Ordner befindet, lässt sich am Anschaulichsten mithilfe der Funktion zeigen, die im Programm `casc` für die Berechnung der Dateipfade von `.curry`- und `.cy`-Dateien zuständig ist:

```
dotCurryPath :: String -> FilePath -> FilePath
dotCurryPath ext f = ante </> ".curry" </> post
  where
    (path, file) = splitFileName f
    dirs         = splitDirectories path
    ante         = joinPath $ init dirs
    post         = last dirs </> replaceExtension file ext
```

Der Pfad des zu einer Curry-Datei gehörigen `.curry`-Ordners wird gebildet, indem „`.curry`“ zwischen den letzten Ordnernamen und die übrigen Ordnernamen eingefügt wird. So entsteht ein Dateipfad der Form `folders/.curry/lastFolder`. Entsprechend sucht `casc` auch genau in diesem Ordner nach dem AST und der Tokenfolge zur weiteren Verarbeitung.

Bei Dateien, die einen Modulkopf der Form `module Module.fileName where ...` haben, landen die erstellten Dateien an der richten Stelle im `.curry`-Ordner, `casc` findet sie und läuft einwandfrei durch. Wenn einer Datei jedoch der beschriebene Modulkopf fehlt, werden AST und Tokenfolge im Ordner `folders/.curry` gespeichert. Anstatt also im `.curry`-Ordner die Ordnerstruktur nachzubilden, erhält jeder Ordner mit Curry-Dateien ohne Modulkopf einen eigenen `.curry`-Ordner. Wird `casc` dabei aus dem Ordner `folders` aufgerufen, ist das kein Problem, AST und Tokenfolge werden unter `.curry/` gesucht und gefunden. Erfolgt der Aufruf jedoch aus einem übergeordneten Ordner, wird unter `.curry/folders/` nichts gefunden und das Programm stürzt mit folgender Fehlermeldung ab:

```
EXISTENCE ERROR:
file "/home/drachi/Repositories/2015-krah-ma/.curry/Test/test.tokens"
does not exist
```

Effektiv bedeutet das, dass `casc` über Ordnerstrukturen hinweg nur Dateien mit Modulkopf bearbeiten kann.

Eine mögliche Lösung wäre, die Erstellung des `.curry`-Ordners und seiner Dateien im Curry-Frontend für Dateien ohne Modulkopf anzupassen. Ansonsten müsste in `casc` eine Art Umgehungslösung implementiert werden, so dass bei Dateien ohne Modulkopf an der für diesen Fall richtigen Stelle nach den `.cy`- und `.tokens`-Dateien gesucht wird. Eine weitere Möglichkeit ist, dass `casc` diese Dateien vor der Phase der AST-Erweiterung an den eigentlich erwarteten Ort kopiert.

7.1.3. Fehlende Interface-Dateien

Versucht man, `casc` außerhalb einer Programm-Hierarchie zu starten, beispielsweise indem es folgendermaßen von außerhalb auf den src-Ordner des Arbeits-Repositories angewendet werden soll, werden die Interface-Dateien für die importierten Module nicht gefunden:

```
krah@krahbuntu ~/Repositories/2015-krah-ma $ casc src -r
...
Invoking: mycymake --tokens --parse-only -W none -q -i ~/bin/pakcs/lib/
          src/Check/Decl.curry

src/Check/Decl.curry, line 12.1: Error:
    Interface for module AST.AST not found

src/Check/Decl.curry, line 13.1: Error:
    Interface for module AST.PosAST not found

...
```

Das liegt daran, dass das zu überprüfende Programm für die Erstellung des AST durch cymake kompiliert werden muss. Dabei wird in dem Ordner, aus dem der Aufruf kam, nach einem `.curry`-Ordner gesucht. Der `.curry`-Ordner befindet sich jedoch im `src`-Ordner selbst.

Eine mögliche Umgehung dieses Verhaltens ist anstelle des obigen Aufrufes folgenden Befehl direkt aus dem Verzeichnis `src` zu starten:

```
krah@krahbuntu ~/Repositories/2015-krah-ma/src $ casc . -r
```

7.2. Performanz

Aus Mangel an anderen Stilüberprüfungs-Werkzeugen für Curry werden hier Laufzeiten von `casc`-Kompilaten der beiden Compiler PAKCS und KiCS2 miteinander verglichen. Alle Testläufe fanden auf einem Laptop mit einer Intel Core i5-4200U CPU (1,6 GHz) und 4 GB Arbeitsspeicher unter Ubuntu 14.04 „Trusty Tahr“ statt.

7.2.1. Laufzeiten von `casc`

Für die Werte aus den Tabellen 7.1 und 7.2 wurden jeweils drei Messungen durchgeführt und der Mittelwert berechnet. Die Zeitmessung wurde mit dem `time`-Befehl durchgeführt.

Zunächst ist in Tabelle 7.1 zu sehen, wie lange die beiden `casc`-Kompilate jeweils für die Bearbeitung von fünf willkürlich bestimmten Dateien[12] ihres eigenen Quelltextes benötigen. Um dabei herauszufinden, wie viel von der gesamten Laufzeit für die Textausgabe benötigt wird, werden die Zeiten für verschiedene Verbosity-Optionen gemessen und verglichen. Außerdem wird der zeitliche Unterschied zwischen den Modi „nur Stilüberprüfung“ und „Stilüberprüfung und -verbesserung“ überprüft. Dabei ist zu beachten, dass bisher nur die Ausdrücke `IfThenElse` und `Let` automatisch korrigiert werden können.

		Compiler	
Mode	Verbosity	PAKCS	KiCS2
Checks only	`VerbQuiet`	11.00	3.65
	`VerbStatus`	12.32	3.68
	`VerbDebug`	18.36	4.84
Checks and corrections	`VerbQuiet`	12.34	3.77
	`VerbStatus`	13.08	3.83
	`VerbDebug`	23.85	6.38

Tabelle 7.1.: Laufzeiten von `casc` für Selbstüberprüfung in Sekunden

Beim Erstellen dieser Tabelle fiel auf, dass es etwa zwei Sekunden länger dauert, wenn zusätzliche Optionen wie `-r` oder Verbosity-Optionen eingelesen werden. Um die Zeiten besser miteinander vergleichen zu können wurde deshalb anstelle von Verwendung der Optionen `--quiet` und `--debug` die Verbosity jeweils direkt im Quellcode definiert, indem die entsprechende Default-Option angepasst wurde.

[12] `AST/AddPositions.curry`, `AST/APM.curry`, `Check/Expression.curry`, `Config/ReadConfig.curry` und `Config/Types.Curry`

In Tabelle 7.1 fällt zunächst auf, dass das PAKCS-Kompilat für jede Aktion etwa drei- bis viermal länger braucht als das KiCS2-Kompilat. Das ist genau das Verhalten, das im Vergleich zwischen PAKCS und KiCS2 erwartet wird und ist auf die Optimierungen zurückzuführen, die KiCS2 beim Kompilieren vornimmt. Außerdem ist zu sehen, dass die Ausführungszeit von `casc` sich verlängert, je mehr Ausgaben die Verbosity-Option fordert. Das ist ein typisches Phänomen bei I/O-Aktionen. Besonders bei der Verbosity `VerbDebug` ist dieser Unterschied gut zu sehen, da hier deutlich mehr Ausgaben getätigt werden als bei `VerbStatus`, während hingegen `VerbStatus` nur unmerklich mehr Ausgaben produziert als `VerbQuiet`. Es ist davon auszugehen, dass wenn bereits mehr automatische Korrekturen implementiert wären, der Unterschied zwischen den beiden Modi gravierender wäre.

In Tabelle 7.2 ist die Leistung des PAKCS- und KiCS2-Kompilats von `casc` für eine einzelne Datei zu sehen. Um zu untersuchen, wie sehr sich mehr Fehler in einer Datei auf die Laufzeit von `casc` auswirken, werden die Laufzeiten für die Programme `ugly.curry` und `notUgly.curry` miteinander verglichen. Auch hier wurde zwischen den beiden Modi „nur Stilüberprüfung“ und „Stilüberprüfung und -verbesserung“ unterschieden. Die zu überprüfenden Programme sind semantisch identisch. Dabei wurde `notUgly.curry` aus Teilen der Beispielprogramme für Curry[13] zusammengestellt und ist 230 Zeilen lang. Um daraus `ugly.curry` zu erstellen, wurden an allen möglichen Stellen Verletzungen der bisher implementierten Stilrichtlinien vorgenommen. Beide Dateien sind im Repository `2015-krah-ma` im Ordner `Test` zu finden.

		Compiler	
Mode	File	PAKCS	KiCS2
Checks only	`notUgly`	3.29	0.69
	`ugly`	3.49	1.02
Checks and corrections	`notUgly`	3.22	0.70
	`ugly`	3.68	0.72

Tabelle 7.2.: Laufzeiten von `casc` für die semantisch identischen Dateien `ugly.curry` und `notUgly.curry` in Sekunden

Tabelle 7.2 ist zu entnehmen, dass es im Vergleich zu einer fehlerfreien Datei minimal länger dauert, eine Datei mit vielen Fehlern zu untersuchen. Hierbei ist jedoch auffällig, dass die Werte für die Datei `notUgly.curry` in den beiden Modi jeweils nahezu identisch sind. Das ist dadurch begründet, dass für eine korrekte Datei keine Fehlerkorrekturen

[13] `https://www.informatik.uni-kiel.de/~curry/examples/`

durchgeführt werden müssen. Auch hier ist wieder die deutliche Überlegenheit des KiCS2-Kompilats zu sehen.

7.2.2. Kompilierungsvorgang

In 7.3 werden die Zeiten miteinander verglichen, die das Kompilieren von `casc` mit PAKCS, beziehungsweise KiCS2 benötigt. In beiden Compilern wird eine zu kompilierende Datei zunächst mit dem Befehl `:load filename` geladen und anschließend mittels `:save` als ausführbare Datei gespeichert. Die Zeitmessung wurde in den interaktiven Umgebungen von PAKCS und KiCS2 durch den Befehl `:set +time` aktiviert und jeweils für beide Teile des Kompilierungsvorgangs durchgeführt.

	Compiler	
Command	PAKCS	KiCS2
`:load`	21.2	16.53
`:save`	1.18	77.10
total	22.38	93.63

Tabelle 7.3.: Zeitaufwand für den Kompilierungsvorgang von `casc` durch PACKS und KiCS2 in Sekunden

Aufgrund der zahlreichen Optimierungen laufen mit KiCS2 kompilierte Programme zwar deutlich schneller als mit PAKCS kompilierte, diese Optimierungen kosten aber beim Kompilierungsvorgang auch entsprechend Zeit. Während man mit PAKCS nach 22.38 Sekunden ein ausführbares Programm erhält, dauert derselbe Vorgang mit KiCS2 93.63 Sekunden, also etwa viermal so lange.

7.2.3. Ergebnis

Aus den beiden vorigen Abschnitten kann entnommen werden, dass das Kompilieren von `casc` mit KiCS2 zwar länger dauert als mit PAKCS, aber dann drei- bis viermal kürzere Ausführungszeiten erzielt. Weil bekannt ist, dass KiCS2 beim Kompilieren eine Reihe von Optimierungen vornimmt und PAKCS nicht, wurde mit diesem Ergebnis bereits gerechnet und daher in der Implementierungs- und Testphase dieser Arbeit mit PAKCS gearbeitet. Möchte man `casc` jedoch im Alltag benutzen, empfiehlt sich die Nutzung einer mit KiCS2 kompilierten Version. Eine mit PAKCS kompilierte Version von `casc` kann außerdem nicht viele Dateien gleichzeitig verarbeiten, sondern stürzte im Test ab circa sechs zu überprüfenden Dateien (je nach Dateigröße mal einige mehr

oder einige weniger) mit folgender Fehlermeldung ab – ein Test auf einem Computer mit mehr Arbeitsspeicher lieferte dasselbe Ergebnis:

```
ERROR: system_error(SPIO_E_TOO_MANY_OPEN_FILES)
! System error
! 'SPIO_E_TOO_MANY_OPEN_FILES'
! System error
! 'SPIO_E_TOO_MANY_OPEN_FILES'
SICStus 4.3.2 (x86_64-linux-glibc2.5): Fri May  8 04:54:53 PDT 2015
Licensed to SP4.3informatik.uni-kiel.de
```

Um ein Beispiel für die realistisch benötigte Dauer eines Überprüfungsvorgangs für ein komplettes Projekt zu liefern, wurde der gesamte Quelltext von `casc` mit einem KiCS2-Kompilat überprüft. Dies dauerte 14.81 Sekunden für 24 Dateien, die auf 4 Unterordner verteilt sind. Da jede Datei einzeln überprüft und jeweils ihr abstrakter Syntaxbaum durchlaufen wird, ist mit einem linearen Anstieg der Laufzeit für mehr beziehungsweise größere Dateien zu rechnen. Die durchschnittliche Zeitdauer für die Überprüfung einer einzelnen Datei liegt damit bei 0.62 Sekunden, was ein durchaus kleiner Zeitaufwand für den gebotenen Komfort der automatischen Stilüberprüfung ist.

Die geforderte leichte Erweiterbarkeit von `casc` ist durch die Modularisierung von der ausführenden Logik und den Validierungsfunktionen gegeben. Im Ausblick (Kapitel 9.1) befinden sich Hinweise dazu.

8. Zusammenfassung

Im Rahmen dieser Thesis wurde `casc` entwickelt – ein Werkzeug zur automatischen Stilüberprüfung von Curry-Quelltext. Motiviert wurde diese Entwicklung durch Stilrichtlinien aus dem Curry Style Guide [14] und dem Wunsch, sich Stilfehler in einem Dokument auf Knopfdruck anzeigen lassen zu können, wie es beispielsweise in Haskell oder Ruby möglich ist.

Weil gute Lesbarkeit eine wichtige Eigenschaft von Programmen ist und oft viele Entwickler gemeinsam an einem Projekt arbeiten, ist es wichtig, sich auf bestimmte Stilrichtlinien zur besseren Lesbarkeit zu einigen. Ein Werkzeug wie `casc` kann durch Überprüfung oder automatischer Korrektur von Quelltext dabei helfen, diese Richtlinien umzusetzen.

`casc` wurde komplett in der logisch-funktionalen Programmiersprache Curry geschrieben. Es bietet die Möglichkeit, Curry-Quelltext auf Stilfehler zu überprüfen, die die in Kapitel 6.4.4 aufgelisteten bisher implementierten Richtlinien verletzen. Außerdem sind Ansätze für die automatische Korrektur von Stilfehlern vorhanden. Das Verhalten von `casc` lässt sich durch bestimmte Aufruf-Optionen steuern. Darunter befinden sich Steuerungsmöglichkeiten für die Verbosity, das heißt die Menge an Statusmeldungen, die während eines Durchlaufes ausgegeben werden sollen, sowie ein Parameter, der die rekursive Anwendung von `casc` auf Unterordner und deren Inhalte ermöglicht. Alle möglichen Aufruf-Optionen kann man sich durch den Befehl `casc --help` anzeigen lassen. Zusätzlich dazu existiert eine Konfigurationsdatei, in der die Überprüfung und die automatische Korrektur für einzelne Stilrichtlinien ein- und ausschaltbar sind.

Grundlage für die Stilüberprüfung von `casc` sind auf der einen Seite die Tokenfolge, eine Liste von Token mit ihren Positionen im Quelltext, und auf der anderen Seite der abstrakte Syntaxbaum des zu überprüfenden Quelltextes.

Um `casc` zu realisieren, war daher zunächst eine Anpassung im Curry-Frontend nötig. Es musste eine Möglichkeit geschaffen werden, das Ergebnis der lexikalischen Analyse einer Curry-Datei ausgeben zu lassen. Die Tokenfolge war zwar natürlich vorher schon als Zwischenprodukt im Frontend enthalten (siehe Abbildung 3.3), es wurde jedoch ein Befehl ergänzt, mit dem man sie gezielt abrufen kann: `cymake file --token`.

Ein dazu analoger Befehl für den Abruf des AST war im Frontend schon vorhanden: `cymake file --parse-only`. Beide Dateien werden wie alle anderen Zwischenprodukte des Kompilierungsvorgangs im `.curry`-Ordner abgelegt, von wo aus `casc` auf sie zugreifen kann.

Weil in der Tokenfolge keine Informationen über semantische Strukturen des analysierten Quelltextes und im AST nicht alle benötigten Positionsangaben vorhanden sind, war es notwendig, eine neue Datenstruktur zu schaffen, die all diese Informationen in sich vereint. Der sogenannte PosAST ist also ein abstrakter Syntaxbaum, der um Positionsangaben aus dem Tokenstream erweitert wurde.

Mithilfe des PosAST kann `casc` Stilrichtlinien überprüfen. Dazu wird der gesamte PosAST traversiert und an allen relevanten Knoten passende Check-Funktionen aufgerufen, sofern der entsprechende Check in der Konfigurationsdatei eingeschaltet ist. Eine Check-Funktion enthält den Aufruf einer Validierungsfunktion sowie eine Nachricht, die zurückgegeben wird, wenn der Knoten nicht valide ist. Eine Validierungsfunktion ist eine Funktion mit prädikatenlogischem Charakter, die festlegt, wie bestimmte Sprachkonstrukte auszusehen haben, damit sie der zugehörigen Stilrichtlinie entsprechen. Ein Beispiel dafür ist folgende Validierungsfunktion, die die Einrückung der Schlüsselwörter `if`, `then` und `else` eines `IfThenElse`-Konstruktes validiert:

```
-- |Check indentation of keywords
validITEkw :: Pos -> Pos -> Pos -> Bool
validITEkw pi pt pe =
    (line pi == line pt && line pt == line pe)
 || (col pt == ((col pi) + 2) && col pt == col pe)
```

Weiterhin bietet `casc` Ansätze für die automatische Korrektur von Stilfehlern. Auch hierfür wird der PosAST traversiert und solche Knoten, die zu Stilrichtlinien gehören, für die die automatische Korrektur eingeschaltet ist, den Stilrichtlinien entsprechend neu zusammengebaut. Hier besteht aber noch Entwicklungsbedarf, damit zum einen nur solche Knoten neu zusammengesetzt werden, die die Stilrichtlinien auch verletzen, und zum anderen alle Positionen zugehöriger Unterausdrücke mitverschoben werden.

In `casc` sind bisher die Überprüfung von 17 Stilrichtlinien (darunter die Zeilenlänge, die Ausrichtung von `case`-Ausdrücken sowie die Ausrichtung von Record-Definitionen) und die automatische Korrektur von 2 Stilverletzungen (die Einrückung der Schlüsselwörter `if`, `then` und `else`, sowie `let` und `in`) implementiert.

Der Vergleich von einer mit PAKCS kompilierten mit einer mit KiCS2 kompilierten Version von `casc` zeigte, dass erstere zwar deutlich schneller kompiliert wird, letztere

aber drei- bis viermal kürzere Laufzeiten erzielt. Daher ist die Empfehlung, für die mögliche Weiterentwicklung von `casc` den Compiler PAKCS zu benutzen, eine zur tatsächlichen Verwendung gedachte Version aber mit KiCS2 zu kompilieren. Zum Abschluss sei das Beispiel von schlecht formatiertem Curry-Code aus der Einleitung aufgegriffen:

```
condAbs p x = if p  then  abs x
  else           (   x)

abs n
   | n <   0 = (( - n ) )
 | n    > 0 = n
     | otherwise   = 0
```

Dafür liefert `casc` mit aktivierter farbiger Ausgabe, Statusausgabe und automatischer Korrektur das in Abbildung 8.1 abgebildete Ergebnis, in dem der Nutzer über alle Stilverletzungen bis auf die redundanten Klammern informiert wird. Die AST-Datei `.curry/smellyCode.acy` wurde mit einer korrigierten Version des AST überschrieben.

```
krah @ krahbuntu  /Repositories/2015-krah-ma/src$
casc smellyCode.curry

Welcome to casc, the Curry Automatic Style Checker.
The following files will be checked:
smellyCode.curry

Processing file:  smellyCode.curry
Invoking:  ./cymake --tokens --parse-only -W none -q
           -i  /bin/pakcs/lib/ smellyCode.curry
Extending abstract syntax tree...  Checking...

    (1,15) IfThenElse:  Wrong indentation of subexpressions.
    (1,15) IfThenElse:  Wrong indentation of keywords.
    (5, 4) GuardedRhs:  Bars are not aligned.
    (5,14) GuardedRhs:  Equality signs are not aligned.

Done.
The corrections have been applied.

Done with all files.
```

Abbildung 8.1.: Ausgabe von `casc` für Beispiel aus Einleitung

9. Ausblick

Die vorliegende Implementierung bietet sowohl auf konzeptioneller als auch auf funktionaler Ebene noch Raum für Erweiterungen. Die funktionale Ebene betrifft die schon in der Einleitung angesprochene Erweiterbarkeit und Veränderbarkeit der zu überprüfenden Stilrichtlinien. Auf konzeptioneller Ebene sind vor allem Eigenschaften denkbar, die die Nutzung von `casc` komfortabler für den Benutzer machen.

9.1. Erweiterung der Stilrichtlinien

Die Erweiterung der zu überprüfenden Stilrichtlinien betrifft die Module im Ordner `scr/Check`. Hier können nach dem Muster der vorhandenen Check-Module beliebig vorhandene Checks erweitert und verändert, sowie weitere erstellt werden. Ein Check-Modul umfasst immer eine Check-Funktion, beispielsweise die in Kapitel 6.4 vorgestellte Funktion

```
expressionCheck :: CheckF Expression.
```

Wie in Kapitel 6.4 erwähnt, ist `CheckF` ein Typsynonym für Check-Funktionen für ein Element `a` mit einer `Message`-Liste als Rückgabe:

```
type CheckF a = a -> [Message].
```

Das zweite Element eines jeden Check-Moduls sind die entsprechend zur Check-Funktion gehörigen Validierungsfunktionen. Wie in Kapitel 6 beschrieben, erfolgt der Aufruf der Check-Funktionen im Modul `Check.CheckStyle`. Dort befinden sich die Funktionen, die den PosAST traversieren und an jedem Knoten die entsprechenden Check-Funktionen aufrufen. In diesem Modul müsste der Aufruf für einen neu hinzugefügten Check also an der passenden Stelle ergänzt werden. So lässt sich die Funktionsweise von `casc` individuell an den gewünschten Programmierstil der Benutzer anpassen.

9.2. Automatische Korrektur

Die wichtigste Erweiterung, die die Nutzung von `casc` komfortabler machen würde, ist die Möglichkeit der automatischen Korrektur von gefundenen Stilverletzungen. Ansätze für die automatische Korrektur sind, wie in Kapitel 6.6 beschrieben, schon vorhanden: Es existieren Funktionen, die den Baum traversieren, sowie Korrektur-Funktionen, die falsch positionierte Schlüsselwörter bei den Ausdrücken `IfThenElse` und `Let` korrigieren:

```
repairExpression :: PosAST.Expression -> PosAST.Expression
```

Dies funktioniert ähnlich wie bei den Check-Funktionen im Modul `Check.CheckStyle`: Wie dort wird im Modul `AutoCorr.AutoCorrPosAST` ausgehend vom Wurzelknoten `Module` mittels der Funktion `correctModule` der Baum durchlaufen und an allen Knoten die Teilbäume mit entsprechend aufgerufenen Reparatur-Funktionen neu zusammengesetzt. Hier ist es allerdings bisher so gelöst, dass eine Reparatur-Funktion nicht nur „falsche“ Knoten neu aufbaut, sondern alle. Es fehlt also zusätzlich zu dem Prädikat `shallCorrect` noch eine Unterscheidung, ob der behandelte Knoten tatsächlich eine Stilverletzung aufweist. Dazu müsste die Stilüberprüfung mit ihren Validierungsfunktionen mit der Korrektur verschmolzen werden.

Dies ist jedoch nur die halbe Lösung – es reicht nicht aus, nur die Schlüsselwörter neu zu positionieren. Auch die dahinter folgenden Teilausdrücke oder Deklarationen müssen umgesetzt werden, damit keine Lücken entstehen oder gar wegen der Layoutsensitivität der Sprache Curry der Quellcode unkompilierbar wird. Wird beispielsweise ein `else`-Fall eine Zeile tiefer gesetzt, damit er sich unter dem `then`-Fall anstatt in derselben Zeile befindet, muss jede darauf folgende Programmzeile ebenfalls verschoben werden.

Realisierbar wäre diese ganzheitliche Verschiebung aller Elemente durch die Übergabe eines weiteren Parameters beim Aufruf aller Reparatur-Funktionen. Dieser Parameter könnte ein Tupel `(Int, Int)` sein, das ein immer aktuelles δ angibt, um wie viele Zeilen und Spalten Elemente und Unterelemente aktuell verschoben werden müssen. Dieses Tupel `(`δ `Line, `δ `Column)` müsste bei jeder vorgenommenen Korrektur aktualisiert werden. Dabei ist darauf zu achten, dass sich zeilenweise Verschiebungen durch den gesamten Quelltext ziehen und spaltenweise Verschiebungen im Allgemeinen höchstens die aktuelle Funktion, meistens aber nur durch den aktuellen Teilausdruck betreffen. Dazu wird das Beispiel aus Listing 9.1 betrachtet, für das `casc` die in Abbildung 9.1 abgebildeten Nachrichten produziert.

Die automatische Korrektur würde beide Stilverletzungen korrigieren, indem zunächst das erste Schlüsselwort `in` in Zeile 4 direkt unter das erste Schlüsselwort `let` (Zeile 3)

```
1 greater4 :: Int -> Int
2 greater4 x = if x > 4
3                then let y = 10
4                       in x+y
5                  else let y = 5
6                       in x*y
```

Listing 9.1.: Beispiel für Positionskorrektur

```
(4,14) IfThenElse:  Wrong indentation of keywords.
(5,21) Let:  Keywords 'let' and 'in' are not aligned.
```

Abbildung 9.1.: Ergebnis der Überprüfung für das Beispiel aus Listing 9.1

verschoben werden würde. Dazu müssten alle drei Zeichen vom `x+y` in Zeile 4 ebenfalls verschoben werden, indem das Tupel `(0,-2)` – welches für die Verschiebung um zwei Spalten nach links steht – durch den ganzen `InfixApply`-Ausdruck geschleift und auf alle Positionen angewandt wird:

```
(InfixApply (Variable (QualIdent Nothing (Ident (6,26) "x" 1)))
            (InfixOp  (QualIdent Nothing (Ident (6,27) "+" 0)))
            (Variable (QualIdent Nothing (Ident (6,28) "y" 4)))))
```

Bei der automatischen Korrektur von `IfThenElse` wird das Schlüsselwort `else` in Zeile 5, um es unter dem Schlüsselwort `then` aus Zeile 3 zu positionieren, ebenfalls um zwei Zeichen nach links verschoben. Auch hier sind alle Unterausdrücke betroffen: Alle Zeichen des Ausdrucks `let y = 5 in x*y` in den Zeilen 5 und 6 müssen ebenfalls um zwei Zeichen nach links verschoben werden. Die Linksverschiebung von der vorigen Korrektur wirkt sich hier jedoch nicht mehr aus.

Die Positionsanpassung könnte dann durch eine Funktion `relocate` umgesetzt werden. Sie nimmt als Eingabe eine Position `p = (l, c)` und ein Tupel mit Verschiebungsanweisungen für `p`, nämlich `(`δ `Line, `δ `Column)`, entgegen. Die Ausgabe ist die entsprechend verschobene Position `p`.

```
relocate :: Pos -> (Int, Int) -> Pos
relocate (l, c) (dl, dc) = (l + dl, c + dc)
```

Bisher ist die Rückgabe der Korrektur so gelöst, dass der korrigierte PosAST mittels der Funktionen aus dem Modul `AST.RemovePositions` zurück in einen einfachen

AST konvertiert wird, indem alle hinzugefügten Positionen entfernt und alle dabei vorgenommenen Änderungen wieder rückgängig gemacht werden. So umgewandelt ersetzt der korrigierte AST die Datei `.curry/file.cy`. Daraus könnte mithilfe des in `cymake` eingebauten Pretty Printers wieder Curry-Quelltext erstellt werden.

Das hätte allerdings noch nicht ganz das gewünschte Ergebnis: Die automatische Korrektur verändert Positionen von Schlüsselwörtern und anderen Elementen, deren Positionen im nicht-erweiterten AST gar nicht vorhanden sind. So kann der eingebaute Pretty Printer von `cymake` den genauen Abdruck der korrigierten Positionen nicht leisten. Hierfür müsste ein exakter Pretty Printer geschrieben werden. Er sollte einen PosAST als Eingabe nehmen und diesen unter Beachtung aller angegebenen Positionen exakt als Curry-Quelltext abdrucken.

Ein anderes Problem bei der automatischen Korrektur ist bisher, dass schon durch die im ersten Schritt erfolgte Behandlung des Quelltextes durch das Tool alle Kommentare herausgefiltert werden. Dies geschieht, weil bei der Erstellung des PosAST der von `cymake` erstellte AST synchron mit der Tokenfolge laufen muss. In der Tokenfolge befinden sich ursprünglich Kommentare, im AST nicht. So erschien es zunächst als einfachste Lösung für die Synchronisation, die Kommentare aus der Tokenfolge herauszufiltern, da sie für die Überprüfung der Stilrichtlinien nicht benötigt werden.

Eine mögliche Lösung für dieses Problem wäre, die Kommentare aus der Tokenfolge nicht wegzuwerfen, sondern in einer separaten Liste zu speichern. So würden die Listen `token` und `commentToken` zu späteren Verarbeitung gespeichert. Mit der Liste `token`, in der sich keine Kommentare befinden, ließe sich wie bisher der PosAST durch Synchronisation mit dem AST erstellen.

Nach dem Durchlauf von Überprüfung und automatischer Korrektur müssten die Kommentare wieder in den Quelltext eingefügt werden. Dazu muss allerdings auch die Korrektur des Quelltextes berücksichtigt werden und die Positionen der Kommentar-Token ebenfalls entsprechend korrigiert werden. Wurde beispielsweise in Zeile 7 ein `else`-Schlüsselwort um eine Zeile verschoben, müssen alle Kommentare, die im Quellcode *nach* Zeile 7 standen, ebenfalls um eine Zeile verschoben werden. Auch, wenn beispielsweise in Zeile 12 ein Schlüsselwort `let` um zwei Spalten nach links geschoben wurde, muss ein eventuell in der Zeile stehender Zeilenkommentar mit verschoben werden. Wurde diese Positionsanpassung für alle Kommentar-Token durchgeführt, können die Kommentare in den vom exakten Pretty Printer abgedruckten korrigierten Quelltext einsortiert werden.

9.3. Functional Patterns

In Kapitel 3.1.2 wurde das Konzept des Functional Pattern Matching in Curry vorgestellt. Weil `casc` den AST des zu prüfenden Programms mehrfach durchlaufen muss, würde es sich anbieten, anstelle eines Baumdurchlaufes Functional Pattern Matching zu benutzen, um bestimmte Knoten zu erkennen. Dies könnte etwa folgendermaßen aussehen:

```
checkExpr ("IfThenElse" ++ pi ++ _ ++ pt _ ++ pe)
    | (validITE pi pt pe) = []
    | otherwise           = [Message pi "Keywords are not aligned."]
  where validITE pi pt pe =
      ((line pi == line pt) && (line p2 == line p3))
    || ((col pi == (col pt)-2) && (col pt == col pe))
```

Auch Vereinfachungen wie die Benutzung von `concatMap` anstelle von `concat $ map` lassen sich mit Functional Patterns leicht umsetzen, beispielsweise auf diese Art, bei der der AST direkt korrigiert werden würde:

```
simp (Paren (Paren e))       = Paren e
simp ('concat' ('map' f xs)) = 'concatMap f xs
```

Es wäre interessant zu untersuchen, ob das Werkzeug zur Stilüberprüfung und -korrektur mit einem Ansatz aus Functional Patterns und Default Regeln performanter wäre als der bisherige rein funktionale Ansatz.

9.4. Einbinden in Kompilierungsvorgang

Eine weitere mögliche Erweiterungen nimmt sich die in Kapitel 2 vorgestellten Werkzeuge zur Stilüberprüfung in andereren Programmiersprachen zum Vorbild. So wäre es denkbar, `casc` genau wie `style_check` für Prolog in den Kompilierungsvorgang von `cymake` einzubinden. Auf diese Weise könnten gefundene Stilfehler direkt beim Kompilieren als Warnungen ausgegeben werden. Eine Unterteilung in „Vorschläge“, „Warnungen“ und „Fehler“ wäre wie in HLint für Haskell und `rubocop` für Ruby ebenfalls möglich. Die Schwere eines Verstoßes könnte in der Konfigurationsdatei eingestellt werden. Hier würde sich auch eine farbliche Unterteilung der einzelnen Kategorien anbieten, beispielsweise könnten Hinweise in blau, Warnungen in orange und Fehler in rot angezeigt werden.

A. Anleitung zur Inbetriebnahme von `casc`

Wie die Ergebnisse aus Kapitel 7 gezeigt haben, erzielt `casc` mit KiCS2 kompiliert etwa viermal schnellere Laufzeiten als mit PAKCS kompiliert. Daher ist die Empfehlung, `casc` zum Ausprobieren mit KiCS2 zu kompilieren. Die folgende Anleitung konzentriert sich daher auf KiCS2; das Kompilieren mit PAKCS funktioniert aber analog.

1. Wechseln Sie in das Verzeichnis `2015-krah-ma/src`
2. Starten Sie KiCS2
3. Laden Sie das Hauptprogramm: `:load casc`
4. Speichern Sie das Hauptprogramm als ausführbare Datei: `:save`
5. Fügen Sie das Verzeichnis, in dem sich `casc` und die Verknüpfung zu `cymake` aus dem aktuellen Curry-Frontend befinden, zum Pfad hinzu

Die Verknüpfung zum Build-Tool `cymake` des aktuellen Curry-Frontends ist deshalb nötig, weil die Compiler PAKCS und KiCS2 möglicherweise noch nicht die aktuell benötigte Version benutzen. Im Laufe dieser Arbeit waren einige Anpassungen am Curry-Frontend nötig, daher funktioniert `casc` erst mit `cymake` ab Version 0.4.1.

Die einfachste Möglichkeit, das Curry-Frontend einzeln zu installieren, ist die Nutzung von `cabal sandboxes`. Sandboxes sind eine Möglichkeit, Haskell-Pakete isoliert aber trotzdem mit allen benötigten Abhängigkeiten zu installieren[14].

1. Klonen Sie das Curry-Frontend: `git clone ssh://git@git.ps.informatik.uni-kiel.de:55055/curry/curry-frontend.git`
2. Klonen Sie Curry-Base: `git clone ssh://git@git.ps.informatik.uni-kiel.de:55055/curry/curry-base.git`

[14] http://coldwa.st/e/blog/2013-08-20-Cabal-sandbox.html

3. Initialisieren Sie in beiden Verzeichnissen eine Sandbox: `cabal sandbox init`
4. Fügen Sie dem Frontend das Base-Paket als Quelle hinzu:
   `cabal sandbox add-source path/to/curry-frontend`
5. Installieren Sie alle Abhängigkeiten in den Sandboxes:
   `cabal install --only-dependencies`
6. Konfigurieren Sie die Abhängigkeiten: `configure`
7. Bauen Sie das Frontend: `cabal build`
8. Erstellen Sie einen symbolischen Link zu der ausführbaren Datei `cymake` des aktuellen Frontends aus der Sandbox und nennen Sie ihn `mycymake`.
9. Legen Sie die Verknüpfung `mycymake` am Besten im selben Ordner ab wie `casc`.

Der Name `mycymake` für die Verknüpfung ist nötig, weil er fest einprogrammiert ist. Sobald PAKCS und KiCS2 die aktuelle Version des Frontends benutzen, kann folgende Funktion im Modul `Utils` auf `"cymake"` verändert werden:

```
cymakeCmd :: String
cymakeCmd = "mycymake"
```

Wurden alle Schritte dieser Anleitung befolgt, können Sie `casc` nun in jedem beliebigen Ordner mit Curry-Dateien benutzen.

B. Konfigurationsdatei für `casc`

Hier ist eine vollständig ausgefüllte Konfigurationsdatei für `casc` zu sehen. Sie ist auch im Repository `2015-krah-ma` im Ordner `src` zu finden. Ihre Komponenten wurden in Kapitel 6.3 erläutert. Um sie zu benutzen, muss diese Datei unter dem Namen `.cascrc` entweder in dem Ordner liegen, von dem aus `casc` aufgerufen wurde, oder sich aber im Home-Verzeichnis des Nutzers befinden. Hierbei wird eine Konfigurationsdatei am erstgenannten Ort bevorzugt behandelt. Ist an beiden Orten keine Konfigurationsdatei vorhanden, werden die in Kapitel 6.3.2 vorgestellten Default-Werte benutzt.

```
-- Config file for StyleChecker.
-- Every line starting with '--' is a comment and will be ignored.
-- The config file contains:
--   * a list of all available checks
--   * an Integer for maximum line length
--   * the path to pakcs or kics2 libraries
--   * a parameter for the activation of autocorrect

-- You can tell the StyleChecker your desired options by using the numbers 0-2.
--  0:  Don't check this.
--  1:  Check this and show warning messages for violations.
--  2:  Check this and try to correct it automatically.
--     (Available for Expression.IfThenElseKW and Expression.Let.)

Data.Bars                    = 1
Data.Components              = 1
Data.Constructors            = 1
Export.List                  = 1
Expression.Case              = 1
Expression.IfThenElseKW      = 2
Expression.IfThenElseSubExpr = 1
Expression.Let               = 2
Import.Explicit              = 1
LineLength                   = 1
Record.Commas                = 1
Record.DoubleColons          = 1
Record.FieldNames            = 1
```

```
Record.Types                = 1
Rhs.Bars                    = 1
Rhs.Eq                      = 1
Type.Arrow                  = 1

-- Parameter for LineLength-Check
-- How long do you want to allow your lines of code to be?
maxLength = 80

-- This is the path to where the libraries of your pakcs- or kics2-installation are.
-- It will be needed for invocations of cymake during the checking process.
libPath = ~/bin/pakcs/lib/

-- Shall casc try to correct your source file if there are
-- style errors from checks that are marked with "2"?
-- Write "yes" or "y" if you want this feature.
autoCorrect = no
```

C. Definition der Curry-Token

Im Folgenden ist als Ergänzung zu Kapitel 3.3 eine vollständige Liste der Curry-Token, wie sie im Modul `AST.Token` in Curry definiert wurden, abgedruckt.

```
type Ident = String

-- |Curry tokens
data Token
  -- literals
  = CharTok   Char
  | IntTok    Int
  | FloatTok  Float
  | StringTok String

  -- identifiers
  | Id        Ident      -- identifier
  | QId       Ident      -- qualified identifier
  | Sym       String     -- symbol
  | QSym      String     -- qualified symbol

  -- punctuation symbols
  | LeftParen            -- (
  | RightParen           -- )
  | Semicolon            -- ;
  | LeftBrace            -- {
  | RightBrace           -- }
  | LeftBracket          -- [
  | RightBracket         -- ]
  | Comma                -- ,
  | Underscore           -- _
  | Backquote            -- `

  -- layout
  | LeftBraceSemicolon   -- {; (turn off layout)
  | VSemicolon           -- virtual ;
  | VRightBrace          -- virtual }
```

```
  -- reserved keywords
  | KW_case
  | KW_data
  | KW_do
  | KW_else
  | KW_external
  | KW_fcase
  | KW_foreign
  | KW_free
  | KW_if
  | KW_import
  | KW_in
  | KW_infix
  | KW_infixl
  | KW_infixr
  | KW_let
  | KW_module
  | KW_newtype
  | KW_of
  | KW_then
  | KW_type
  | KW_where

  -- reserved operators
  | At                    -- @
  | Colon                 -- :
  | DotDot                -- ..
  | DoubleColon           -- ::
  | Equals                -- =
  | Backslash             -- \
  | Bar                   -- |
  | LeftArrow             -- <-
  | RightArrow            -- ->
  | Tilde                 -- ~
  | Bind                  -- :=
  | Select                -- :>

  -- special identifiers
  | Id_as
  | Id_ccall
  | Id_forall
  | Id_hiding
  | Id_interface
  | Id_primitive
```

```
  | Id_qualified

  -- special operators
  | SymDot                -- .
  | SymMinus              -- -
  | SymMinusDot           -- -.

  -- pragmas
  | PragmaLanguage        -- {-# LANGUAGE
  | PragmaOptions String  -- {-# OPTIONS
  | PragmaHiding          -- {-# HIDING
  | PragmaEnd             -- #-}

  -- comments
  | LineComment String
  | NestedComment String

  -- end-of-file token
  | EOF
```

D. Definition des AST

Im Folgenden ist als Ergänzung zu Kapitel 3.4 die Definition des AST mit allen zugehörigen Datentypen aus dem Modul `AST.AST` in Curry abgedruckt. Auf die Auflistung der Definitionen für den PosAST wird hier aus Redundanz-Gründen verzichtet. Seine Definition ist im Modul `AST.PosAST` nachzulesen und erweitert den hier abgedruckten AST an allen notwendigen Stellen um Positionsangaben. Notwendige Stellen sind solche, an denen zusätzliche Positionsinformationen benötigt werden um die Position jedes einzelnen Elementes aus dem Quelltext bestimmen zu können.

```
-- ---------------------------------------------------------------------------
-- General Types
-- ---------------------------------------------------------------------------

-- |Position
type Pos = (Int, Int)

-- |Simple identifier
data Ident = Ident
  { idPosition :: Pos      -- ^ Source code 'Position'
  , idName     :: String   -- ^ Name of the identifier
  , idUnique   :: Int      -- ^ Unique number of the identifier
  }

-- |Qualified identifier
data QualIdent = QualIdent
  { qidModule :: Maybe ModuleIdent -- ^ optional module identifier
  , qidIdent  :: Ident             -- ^ identifier itself
  }

-- | Module identifier
data ModuleIdent = ModuleIdent
  { midPosition   :: Pos      -- ^ source code 'Position'
  , midQualifiers :: [String] -- ^ hierarchical idenfiers
  }

-- |Specified language extensions, either known or unknown.
```

```
data Extension
  = KnownExtension   Pos KnownExtension -- ^ a known extension
  | UnknownExtension Pos String         -- ^ an unknown extension

data KnownExtension
  = AnonFreeVars       -- ^ anonymous free variables
  | FunctionalPatterns -- ^ functional patterns
  | NegativeLiterals   -- ^ negative literals
  | NoImplicitPrelude  -- ^ no implicit import of the prelude

-- |Different Curry tools which may accept compiler options.
data Tool = KICS2 | PAKCS | CYMAKE | UnknownTool String

-- ---------------------------------------------------------------------------
-- Definition of AST: Module
-- ---------------------------------------------------------------------------

-- |Curry module
data Module = Module [ModulePragma] ModuleIdent (Maybe ExportSpec)
                     [ImportDecl] [Decl]

-- |Module pragma
data ModulePragma
  = LanguagePragma Pos [Extension]
  | OptionsPragma  Pos (Maybe Tool) String


-- |Export specification
data ExportSpec = Exporting Pos [Export]

-- |Single exported entity
data Export
  = Export         QualIdent
  | ExportTypeWith QualIdent [Ident]
  | ExportTypeAll  QualIdent
  | ExportModule   ModuleIdent

-- |Import declaration
data ImportDecl = ImportDecl Pos ModuleIdent Qualified
                             (Maybe ModuleIdent) (Maybe ImportSpec)

-- |Flag to signal qualified import
type Qualified = Bool
```

```
-- |Import specification
data ImportSpec
  = Importing Pos [Import]
  | Hiding    Pos [Import]

-- |Single imported entity
data Import
  = Import         Ident
  | ImportTypeWith Ident [Ident]
  | ImportTypeAll  Ident

-- ---------------------------------------------------------------------------
-- Declarations (local or top-level)
-- ---------------------------------------------------------------------------

-- |Declaration in a module
data Decl
  = InfixDecl    Pos Infix (Maybe Precedence) [Ident]
  | DataDecl     Pos Ident [Ident] [ConstrDecl]
  | NewtypeDecl  Pos Ident [Ident] NewConstrDecl
  | TypeDecl     Pos Ident [Ident] TypeExpr
  | TypeSig      Pos [Ident] TypeExpr
  | FunctionDecl Pos Ident [Equation]
  | ForeignDecl  Pos CallConv (Maybe String) Ident TypeExpr
  | ExternalDecl Pos [Ident]
  | PatternDecl  Pos Pattern Rhs
  | FreeDecl     Pos [Ident]

-- |Operator precedence
type Precedence = Int

-- |Fixity of operators
data Infix
  = InfixL -- ^ left-associative
  | InfixR -- ^ right-associative
  | Infix  -- ^ no associativity

-- |Constructor declaration for algebraic data types
data ConstrDecl
  = ConstrDecl Pos [Ident] Ident [TypeExpr]
  | ConOpDecl  Pos [Ident] TypeExpr Ident TypeExpr
  | RecordDecl Pos [Ident] Ident [FieldDecl]

-- |Constructor declaration for renaming types (newtypes)
```

```
data NewConstrDecl
  = NewConstrDecl Pos [Ident] Ident TypeExpr
  | NewRecordDecl Pos [Ident] Ident (Ident, TypeExpr)

-- |Declaration for labelled fields
data FieldDecl = FieldDecl Pos [Ident] TypeExpr

-- |Calling convention for C code
data CallConv
  = CallConvPrimitive
  | CallConvCCall

-- |Type expressions
data TypeExpr
  = ConstructorType QualIdent [TypeExpr]
  | VariableType    Ident
  | TupleType       [TypeExpr]
  | ListType        TypeExpr
  | ArrowType       TypeExpr TypeExpr
  | ParenType       TypeExpr

-- ---------------------------------------------------------------------------
-- Functions
-- ---------------------------------------------------------------------------

-- |Equation
data Equation = Equation Pos Lhs Rhs

-- |Left-hand-side of an `Equation` (function identifier and patterns)
data Lhs
  = FunLhs Ident [Pattern]
  | OpLhs  Pattern Ident Pattern
  | ApLhs  Lhs [Pattern]

-- |Right-hand-side of an `Equation`
data Rhs
  = SimpleRhs  Pos Expression [Decl]
  | GuardedRhs [CondExpr] [Decl]

-- |Conditional expression (expression conditioned by a guard)
data CondExpr = CondExpr Pos Expression Expression

-- |Literal
data Literal
```

```
  = Char   Char
  | Int    Ident  Int
  | Float  Float
  | String String

-- |Constructor term (used for patterns)
data Pattern
  = LiteralPattern     Literal
  | NegativePattern    Ident Literal
  | VariablePattern    Ident
  | ConstructorPattern QualIdent [Pattern]
  | InfixPattern       Pattern QualIdent Pattern
  | ParenPattern       Pattern
  | RecordPattern      QualIdent [Field Pattern]
  | TuplePattern       [Pattern]
  | ListPattern        [Pattern]
  | AsPattern          Ident Pattern
  | LazyPattern        Pattern
  | FunctionPattern    QualIdent [Pattern]
  | InfixFuncPattern   Pattern QualIdent Pattern

-- |Expression
data Expression
  = Literal           Literal
  | Variable          QualIdent
  | Constructor       QualIdent
  | Paren             Expression
  | Typed             Expression TypeExpr
  | Record            QualIdent [Field Expression]
  | RecordUpdate      Expression [Field Expression]
  | Tuple             [Expression]
  | List              [Expression]
  | ListCompr         Expression [Statement]
  | EnumFrom          Expression
  | EnumFromThen      Expression Expression
  | EnumFromTo        Expression Expression
  | EnumFromThenTo    Expression Expression Expression
  | UnaryMinus        Ident Expression
  | Apply             Expression Expression
  | InfixApply        Expression InfixOp Expression
  | LeftSection       Expression InfixOp
  | RightSection      InfixOp Expression
  | Lambda            [Pattern] Expression
  | Let               [Decl] Expression
```

```
  | Do               [Statement] Expression
  | IfThenElse       Expression Expression Expression
  | Case             CaseType Expression [Alt]

-- |Infix operation
data InfixOp
  = InfixOp     QualIdent
  | InfixConstr QualIdent

-- |Statement (used for do-sequence and list comprehensions)
data Statement
  = StmtExpr Expression
  | StmtDecl [Decl]
  | StmtBind Pattern Expression

-- |Type of case expressions
data CaseType
  = Rigid
  | Flex

-- |Single case alternative
data Alt = Alt Pos Pattern Rhs

-- |Record field
data Field a = Field Pos QualIdent a
```

Literatur

[1] A. V. Aho u. a. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[2] S. Antoy und M. Hanus. *Curry: A Tutorial Introduction*. Available at `http://www.curry-language.org`. 2014.

[3] S. Antoy und M. Hanus. „Declarative Programming with Function Patterns“. In: *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*. Springer LNCS, 2005, S. 6–22.

[4] S. Antoy und M. Hanus. „Default Rules for Curry“. In: *Practical Aspects of Declarative Languages: 18th International Symposium, PADL 2016, St. Petersburg, FL, USA, January 18-19, 2016. Proceedings*. Hrsg. von M. Gavanelli und J. Reppy. Springer International Publishing, 2016, S. 65–82.

[5] B. Braßel u. a. „KiCS2: A New Compiler from Curry to Haskell“. In: *Functional and Constraint Logic Programming*. Hrsg. von H. Kuchen. Bd. 6816. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, S. 1–18.

[6] M. Hanus (ed.) *Curry: An Integrated Functional Logic Language (Vers. 0.9.0)*. Available at `http://www.curry-language.org`. 2015.

[7] M. Hanus (ed.) *PAKCS 1.14.0: User Manual*. Available at `https://www.informatik.uni-kiel.de/~pakcs`. 2015.

[8] Haskell. *Haskell Style Scanner*. Available at `http://projects.haskell.org/style-scanner/`. 2014-2016.

[9] C. A. R. Hoare. „Quicksort“. In: *The Computer Journal* 5.1 (1962), S. 10–16.

[10] S. Marlow (ed.) *Haskell 2010: Language Report*. Available at `https://www.haskell.org`. 2010.

[11] Y. Matsumoto. *Google Tech Talks: Ruby 1.9*. Available at `https://www.youtube.com/watch?v=oEkJvvGEtB4&pxtry`. 2008.

[12] N. Mitchell. *HLint Readme*. Available at `https://github.com/ndmitchell/hlint/blob/master/README.md`. 2016.

[13] E. Moggi. „Notions of computation and monads“. In: *Information and Computation* 93 (1991), S. 55–92.

[14] B. Peemöller. *Curry Style Guide*. Available at `https://www.informatik.uni-kiel.de/~mh/curry/curry-style-guide.html`. 2015.

[15] S. Peyton und J. P. Wadler. *Imperative Functional Programming*. 1993.

[16] SWI-Prolog. *SWI Prolog Dokumentation*. Available at `http://www.swi-prolog.org/pldoc/doc_for?object=manual`.

[17] J. Tibell. *Haskell Style Guide*. Available at `https://github.com/tibbe/haskell-style-guide`. 2015.

Index