

# Projektbericht: Erweiterung von SL um ein Modulsystem

Benjamin Bisping, Rico Jasper, Sebastian Lohmeier, Friedrich Moritz Psiorz

Compilerbauprojekt SoSe 2013, Technische Universität Berlin

## 1 Einleitung

TODO (\* Hällü Wörlö \*)

## 2 Überblick

Ziel des Projektes war es, SL um ein Modulsystem zu erweitern. Dazu reichten wir den Sprachumfang von SL mit einer Reihe von Konstrukten an. Für diese besprechen wir die Syntax genauer in Abschnitt 3, die semantische Analyse in Abschnitt 4 und die Codegenerierung in Abschnitt 5. Grob umfassen die neuen Features:

**Module** Jede SL-Datei wird jetzt als eine Modul-Definition aufgefasst.

**Export von Bezeichnern** Durch das `PUBLIC`-Keyword können Funktionsdeklarationen für andere Module importierbar gemacht werden.

`PUBLIC FUN f : Int -> Int` fügt `f : Int -> Int` der öffentlichen Signatur eines Moduls hinzu.

`PUBLIC DATA T = C` exportiert den Konstruktor `C : T`. Während der Übersetzung eine Moduls `code.sl` wird nicht nur eine `code.js`, sondern auch eine `code.signature` mit den Signaturinformationen angelegt.

**Import von Modulen** Module können importiert mittels `IMPORT` werden.

`IMPORT "code" AS M` macht die exportierten Definitionen aus `code.sl` unter dem lokalen Modulbezeichner `M` verfügbar.

**Qualifizierte Bezeichner** In Ausdrücken, Pattern und Typausdrücken können importierte Bezeichner vorkommen, beispielsweise so:

```
FUN g : M.T -> Int
```

```
DEF g M.C = M.f 23
```

**Externe Definitionen** Mithilfe von `EXTERN` lassen sich Funktionsdefinitionen als JavaScript angeben.

```
FUN myCast : String -> Int
```

```
DEF EXTERN myCast = {| parseInt |}
```

In diesem Code wird `myCast` durch JavaScripts `parseInt` definiert. Allerdings: `EXTERN` ist nicht zur Verwendung in normalen Modulen vorgesehen, sondern soll nur in der Definition von Bibliotheken zum Einsatz kommen. Aus einer `EXTERN`-Definition sollte auch nicht auf andere Definitionen des Moduls zugegriffen werden.

**Externe Datentypen** Durch die externen Definitionen werden häufig auch „externe“ Datentypen notwendig.

`DATA EXTERN Node` definiert einen Typ ohne Konstruktoren. Die Programmiererin hat dafür Sorge zu tragen, dass dadurch nicht versehentlich leere Typen entstehen.

**Import von JavaScript-Code** Zum Zusammenspiel mit `DEF EXTERN` gibt es auch noch die Möglichkeit, JavaScript-Code in ein Modul einzubinden.

`IMPORT EXTERN "_code"` zum Beispiel bindet die Datei `_code.js` direkt mit in das Compilat mit ein.

Darüber hinaus widmeten wir uns auch noch weiteren Themen:

**Syntaxanpassungen** Einige Details wie Inkonsistenzen zwischen den bestehenden Parsern und die implizite Klammerung bei Typausdrücken Zum Beispiel `Dat a -> b` wird jetzt als `(Dat a) -> b` und nicht mehr als `Dat (a->b)` gelesen. (Siehe Abschnitt 3.3)

**Compilationdriver** Das Ergebnis des SL-Compilers wird nicht mehr nach Stdout geschrieben, sondern in Dateien abgelegt. Dabei werden auch transitive Abhängigkeiten aufgelöst und notwendige zusätzliche Dateien mit angelegt. (Siehe Abschnitt 5.2)

**Fehlermeldungen** Fehlermeldungen von syntaktischer und semantischer Analyse sowie Modulauflösung enthalten jetzt für gewöhnlich auch Angaben zum Ort des Fehlers. Viele Fehlermeldungen sind etwas präziser geworden, auch wenn sie immer noch mäßig hilfreich beim produktiven Einsatz ohne tiefere Compilerkenntnisse sein dürften. (Siehe Abschnitt 6)

**Prelude** Die vormalig hart in den SL-Übersetzer eingebauten grundlegenden Funktionen werden jetzt aus einem Modul importiert. (Siehe Abschnitt 7.1) Prelude wird durch jedes SL-Programm implizit unqualifiziert importiert. Unqualifizierte Imports, die nicht das Prelude betreffen, haben wir nicht vorgesehen und aktuell würden sie auch Probleme bei der Typprüfung verursachen.

**Bibliotheken** Wir haben einige simple Bibliotheken für `List`, `Option`, `Either`, `Real`, `Dict`, `println-Debugging` und Webentwicklung geschrieben. Diese stehen in mannigfaltiger Abhängigkeit voneinander und versuchen vielseitig Gebrauch von den neuen Features zu machen. (Siehe Abschnitt 7)

**Beispielprogramme** Zusätzlich zu den Bibliotheken programmierten wir einige ausführbare SL-Programme, die auf den Bibliotheken aufbauen. (Siehe Abschnitt 8)

**Tests** Für viele der neuen Features schrieben wir auch Unit-Tests. Der Großteil unserer Arbeit an dieser Front floss jedoch darein, die alten Unit-Tests an die neuen Features anzupassen. (Siehe Abschnitt 8)

### 3 Syntax und Parser

TODO (\* Fritz: Syntaxanpassungen, Schwierigkeiten, Designentscheidungen \*)

### 3.1 Syntaxanpassungen

**Operatoren** In der ursprünglichen Version von SL wurde unterschieden zwischen eingebauten und selbst definierten Operatoren. Einige der vorgegebenen definierten Operatoren hatten überdies Namen, die für selbst definierte Operatoren nicht erlaubt wären, nämlich `+s` für die String-Konkatenation sowie die Gleitkommaoperatoren `+r`, `-r`, `*r` und `/r`. Außerdem war ein unäres Minus sowohl auf Gleitkomma- als auch auf Ganzzahlen definiert, was in der Sprache dahingehend einzigartig war, dass es ansonsten weder unäre Operatoren gibt, noch überladene Funktionen/Operatoren, noch Bezeichner, die je nach Position (präfix oder infix) eine unterschiedliche Funktion bezeichnen, wie in diesem Fall das Minuszeichen einerseits die Negation auf Gleitkomma- und Ganzzahlen und andererseits die Subtraktion von Ganzzahlen bezeichnete.

### 3.2 Qualifizierte Bezeichner

### 3.3 Weitere Anpassungen der bestehenden Grammatik

### 3.4 Grammatik

## 4 Semantische Analyse

Aufgabe der semantischen Analyse ist es, den Kontext des vom Parser eingelesenen Syntaxbaums zu überprüfen. Durch die Erweiterung der Sprache SL um ein Modulsystem musste die Analyse angepasst und ausgebaut werden. Ohne das Modulsystem war der Kontext auf eine Quelldatei sowie fest einprogrammierte Konstrukte (z.B. Operatoren für ganze Zahlen) beschränkt.

Da nun ein Modul auch andere Module importieren kann, erweitert sich der zu analysierende Kontext. Zum einen wurde die Grammatik um die `IMPORT`-Anweisung ergänzt. Einem Modul ist es beispielsweise nicht erlaubt ein anderes Modul mehrfach zu importieren. Zum anderen können Datentypen und Funktionen aus dem importierten Modul verwendet werden. Für das Type-Checking muss daher die sogenannte Signatur des Imports bekannt sein. Diese umfasst Datendefinitionen und Funktionssignaturen.

### 4.1 Auflösung von Importen

Nachdem die abstrakte Syntax vom Parser eingelesen wurde, müssen die Importe aufgelöst werden. Andernfalls ist Type-Checking nicht möglich, welches vor der Spracherweiterung direkt im Anschluss des Parsings stattfand. Die Auflösung von Importen bezeichnet das Suchen und Laden von Signaturen von externen Modulen. Zuvor müssen die Import-Anweisungen allerdings selbst auf Korrektheit überprüft werden.

**Import-Überprüfung** TODO (\* Modulbezeichner richtiger Begriff? \*)

Die Import-Anweisung ist im Grunde ein Paar aus Modulpfad und Modulbezeichner:

```
IMPORT "my/path/to/module-file" AS ModuleIde
```

Der Pfad gibt dabei an, wo das Modul zu finden ist. Der Modulbezeichner ermöglicht die Verwendung von Datentypen und Funktionen des Importierten Moduls.

Wir möchten verbieten, dass ein Modul mehrfach vom lokalen Modul importiert wird. Wir gehen dabei davon aus, dass ein Modul eindeutig einem Pfad zugeordnet ist. Also überprüfen wir, ob jeder Pfad nur genau einmal vorkommt. Ebenso möchten wir nicht zulassen, dass ein Modulbezeichner für mehrere verschiedene Module verwendet wird.

Zum Beispiel ist folgende Importliste nicht erlaubt, da hier zwei Mal derselbe Modulbezeichner „Duplicate“ verwendet wird:

```
IMPORT "my/path/module-a"    AS Duplicate
IMPORT "other/path/module-b" AS Duplicate
IMPORT "my/path/module-v"    AS Innocent
```

Unsere Annahme, dass Module eindeutig über den Pfad identifiziert werden, kann in einigen Fällen jedoch unzureichend sein. Dies betrifft die Groß- und Kleinschreibung auf Windowssystemen sowie die Verwendung von „.“ und „..“. Daher verbieten wir die Verwendung von Punkten und Großbuchstaben.

Neben dem regulären Import existiert auch noch der unqualifizierte Import. Dieser ist für den Programmierer unzugänglich. Er dient zum Einbinden des Preludes, sodass Funktionen wie Addition auch ohne Modulbezeichner verwendet werden können. Der unqualifizierte Import ist von der Prüfung von doppelten Modulbezeichnern ausgeschlossen, jedoch nicht von Tests auf Pfade

Laut unserer Grammatik TODO (\* tolle grammatik oder verweis zur grammatik \*)

wird der Pfad zu einem externen Modul als String definiert. Der Parser akzeptiert daher jede Art von gültigen Strings. Deshalb muss an dieser Stelle die Pfadsyntax überprüft werden. Grammatik 1 zeigt die Produktionsregeln für Pfade. Erlaubt sind relative Pfade bestehend aus beliebig vielen Verzeichnissen und dem Moduldateinamen am Ende. Als Trennungssymbol dient das Schrägstrichsymbol „/“. Verzeichnisse und Module dürfen Kleinbuchstaben, Zahlen, Minussymbole und Unterstriche enthalten. Der Modulname entspricht dem Dateinamen der Quelldatei ohne Endung „.sl“.

Dies sind Beispiele für korrekte Pfade:

```
IMPORT "module-a"    AS A
IMPORT "dir/module-b" AS B
IMPORT "123/module-c" AS C
IMPORT "module_4"    AS D
```

$$\begin{aligned}
Path &::= (Dir \ /)^* Module \\
Dir &::= char^+ \\
Module &::= char^+ \\
char &::= \begin{array}{c|c|c|c} a & \cdots & & z \\ \hline & 0 & \cdots & 9 \\ \hline & - & & - \end{array}
\end{aligned}$$

Grammar 1: Gültige Importpfade

Inkorrekt sind dagegen die folgenden:

```

IMPORT "MoDuLe"           AS A
IMPORT "module-b.sl"      AS B
IMPORT "dir/."            AS C
IMPORT "/absolute/path/module-d" AS D
IMPORT "m.o.d.u.l.e"      AS E
IMPORT "./module-f"       AS F
IMPORT "d.i.r/module-g"   AS G

```

**Laden der Signatur** Falls die Importe korrekt sind, werden die dazugehörigen Moduldateien geladen. Ein Modul liegt dabei immer in zwei Dateien vor: Die Signatur („\*.signatur“) und die in JavaScript übersetzte Implementierung („\*.js“). Können nicht alle benötigten Dateien gefunden werden, so kann der Kompilervorgang nicht fortgesetzt werden. Die Suche der Dateien erfolgt relativ zum Klassenpfad, Zielverzeichnis, **TODO (\* mainUnit? \*)** und aktuellen Verzeichnis.

Die Signatur-Datei enthält einen Teil des abstrakten Syntaxbaums des zu importierenden Moduls. Datentypdefinitionen und Funktionssignaturen sind hier in einer serialisierten Form abgespeichert. Das Format dieser Datei wird in Abschnitt 5.3 beschrieben.

## 4.2 Type-Checking

Durch das Auflösen besteht nun Zugriff auf die Signaturen der Importe. Diese können in ihrer ursprünglichen Form aber noch nicht dem Type-Checker übergeben werden. Vorher erfolgt eine Normalisierungsphase.

**Modulnormalisierung** Jedes Modul kann wiederum Module importieren. Demzufolge ist es auch möglich, dass zwei importierte Module A und B von einem dritten Modul „List“ Gebrauch machen. Im folgenden Beispiel importiert das Modul C diese Module.

```
-- Module A --
```

```

IMPORT "std/list" AS List
PUBLIC FUN foo : List.List -> List.List

-- Module B --
IMPORT "std/list" AS L
PUBLIC DATA MyType = Ctor L.List

-- Modul C --
IMPORT "a" AS A
IMPORT "b" AS B
IMPORT "std/list" AS StdList

FUN bar : B.MyType -> StdList.List
DEF bar x = CASE x OF B.Ctor l THEN A.foo l

```

Hier würde der Type-Checker nicht wissen, dass der Typ `List.List` aus Modul A derselbe ist wie `L.List` in Modul C. Deshalb müssen die aufgelösten Importe normalisiert werden. Die Idee dahinter ist, dass jedem Modul genau ein Modulbezeichner zugeordnet wird. Dieser Bezeichner wird vom lokalen Modul, hier C, bestimmt. Das bedeutet, dass der Modulbezeichner für das Modul List durch `StdList` ersetzt wird.

Nach der Normalisierung sieht die abstrakte Syntax der Importierten Module also so aus:

```

-- Module A --
IMPORT "std/list" AS StdList

PUBLIC FUN foo : StdList.List -> StdList.List

-- Module B --
IMPORT "std/list" AS StdList

PUBLIC DATA MyType = Ctor StdList.List

```

Diese Ersetzung ist möglich, da auch C das Modul List importiert. Anhand des Pfades `"std/list"` kann dieses Modul auch in den anderen Modulen erkannt und der Bezeichner substituiert werden.

Es kann jedoch auch der Fall eintreten, dass C das List-Modul unbekannt ist. In diesem Fall wird ein neuer Bezeichner generiert. Dieser hat die Form `#<Nummer>` wobei `<Nummer>` durch eine fortlaufende Nummer ersetzt wird. Durch dieses Vorgehen kann auch das folgende Beispielprogramm typgeprüft werden:

```

-- Module A --
IMPORT "std/list" AS L
IMPORT "std/option" AS O

PUBLIC FUN foo : O.Option -> Int

```

```

-- Module B --
IMPORT "std/list" AS L
IMPORT "std/option" AS O

PUBLIC FUN bar : List.List -> O.Option

-- Modul C --
IMPORT "a" AS A
IMPORT "b" AS B
IMPORT "std/list" AS L

FUN baz : L.List -> Int
DEF baz l = B.foo(A.bar(l))

```

Das Modul Option ist C unbekannt. Es kann also keine Funktionen oder Typen aus diesem Modul direkt verwenden. Dennoch ist es möglich die Funktionen `foo` und `bar` wie in Modul C aufzurufen. Während der Normalisierung wird der Modulbezeichner `O` in A und B durch `#1` ersetzt.

**Modulkontext** Vor dem eigentlichen Type-Checking werden die Datentyp- und Funktionsdefinitionen überprüft. Einige der Tests mussten erweitert werden, um auch importierte Definitionen berücksichtigen zu können.

Im Falle der Datentypdefinitionen gibt es die zwei Tests, `checkNoUndefinedTypeCons` und `checkTypeConsApp`, welche auch den Kontext von Importen beachten müssen. `checkNoUndefinedTypeCons` überprüft ob die im Programm verwendeten Konstruktoren existieren. Bisher war dies auf den lokalen Kontext beschränkt. Da jedoch auch Konstruktoren aus anderen Modulen verwendet werden können sollen, werden jene zu der Menge der bekannten Konstruktoren hinzugefügt. Aus demselben Grund musste auch `checkTypeConsApp` erweitert werden. Diese Funktion testet, ob genügend Parameter für einen Konstruktor angegeben wurden.

Da Datendefinitionen von unqualifizierten Importen Probleme verursachen können, müssen diese in weiteren Tests betrachtet werden. Typbezeichner und Konstruktoren dürfen nicht mit lokalen Definitionen in Konflikt geraten. Deshalb werden jene importierten Datendefinitionen in den Tests `checkTypeConsDisjoint` und `unqualifiedImportedDataDefs` ebenfalls betrachtet

Für den Type-Check ist es notwendig den initialen Kontext um den Modulkontext zu erweitern. Dazu muss der Modulkontext zunächst gebildet werden. Dies geschieht auf ähnliche Weise, wie auch der Kontext des lokalen Moduls aufgebaut wird. Alle Konstruktoren der Module erhalten ein Typschema und werden zusammen mit den Funktionen dem Kontext hinzugefügt.

Die lokalen Funktionssignaturen und -definitionen sowie die Signaturen aus den externen Modulen werden genutzt um das Programm in  $\text{ELC}^1$  zu überset-

---

<sup>1</sup> Enriched Lambda Calculus

zen. Die Übersetzung wird dann zusammen mit dem initialen Kontext inklusive Modulkontext an den Type-Checker übergeben, der weitestgehend unangerührt blieb.

## 5 Codegenerierung

Der SL2-Compiler generiert Code, der unter `node.js`<sup>2</sup> und im Browser<sup>3</sup> ausgeführt werden kann. Bei einem Compileraufruf können jeweils mehrere Dateien angegeben werden (siehe Abschnitt 5.1). Während der Compilierung werden aus dem `<classpathDirectory>` die Signatur-Dateien (siehe Abschnitt 5.3) bereits kompilierter Module geladen. Anschließend werden die angegebenen `<module files>` sowie von diesem verwendete Module, kompiliert (Details siehe Abschnitt 5.2). Während der Codegenerierung werden Signaturen, sowie JavaScript-Dateien für alle kompilierten Module erstellt, wobei `require.js` (siehe Abschnitt 5.4) verwendet wird, um die JavaScript-Dateien der Module zur Laufzeit zu laden.

Sofern das beim Aufruf des Compilers angegebene `<module files>` eine Funktion namens `main` deklariert, werden für dieses noch eine `main.js`-Datei, eine Kopie von `require.js` und eine `index.html`-Datei erstellt, die den Aufruf der `main`-Funktion in `node.js` und im Browser erlauben (siehe Abschnitt 5.2).

### 5.1 Compileraufruf und Pfadangaben

Der Aufruf des Compilers erfolgt im Scala Build Tool nach dem folgenden Schema – mit eckigen Klammern `[ ]` umschlossene Parameter sind optional.

```
run-main de.tuberlin.uebb.sl2.impl.Main [-d <output directory>]
[-cp <classpath directory>] -sourcepath <source directory>
<module files>
```

Die Parameter haben folgende Verwendung:

- `-d <output directory>` Das Verzeichnis, in das die generierten Dateien gespeichert werden. Fehlt der Parameter, werden die Dateien im `<source directory>` gespeichert.
- `-cp <classpath directory>` Ein Verzeichnis, aus dem bereits kompilierte Module geladen werden. Fehlt der Parameter, wird versucht, bereits kompilierte Module aus dem `<source directory>` zu laden.
- `-sourcepath <source directory>` Das Verzeichnis, relativ zu dem die `<module files>` geladen werden. Der Sourcepath wird auch verwendet, um von den `<module files>` importierte Dateien zu finden, die noch kompiliert werden müssen.
- `<module files>` Die Pfade der zu kompilierenden SL-Dateien, relativ zum `<source directory>`. Da der Compiler Importe automatisch auflöst, müssen zu kompilierende importierte Dateien nicht angegeben werden, sofern ein Datei angegeben ist, die sie (indirekt) importiert.

<sup>2</sup> <http://nodejs.org/> - getestet mit Version 0.10.10

<sup>3</sup> getestet mit Firefox, Chrome und Internet Explorer



Durch die Unterscheidung zwischen `<classpath directory>` und `<source directory>` kann dem Compiler sowohl eine Quelldatei als auch eine Kompilation eines Moduls oder zweier gleichnamiger Module verfügbar sein. Abschnitt 5.2 definiert, wie in einer solchen Situation vorgegangen wird.

Der folgende Konsolen-Mitschnitt zeigt die Compilierung des boxsort-Beispiels aus Abschnitt 8.1 mit Compileraufruf und den Ausgaben des Compilers, die über die erzeugten Dateien informieren.

```
> run-main de.tuberlin.uebb.sl2.impl.Main -sourcepath
src/main/sl/examples/ boxsort.sl
[info] Running de.tuberlin.uebb.sl2.impl.Main -sourcepath
src/main/sl/examples/ boxsort.sl
compiling boxsort to src\main\sl\examples\boxsort.sl.js
writing signature of boxsort to src\main\sl\examples\
boxsort.sl.signature
copied C:\Users\monochromata\git\sl2\target\scala-2.10\
classes\js\index.html to src\main\sl\examples\index.html
copied C:\Users\monochromata\git\sl2\target\scala-2.10\
classes\js\require.js to src\main\sl\examples\require.js
compilation successful
[success] Total time: 1 s, completed 21.07.2013 12:38:04
```

Die grundlegenden Funktionen und Datentypen im Prelude und in der Standardbibliothek (siehe Abschnitt 7) sind bereits fertig kompiliert im Compiler enthalten. Sofern sie angepasst werden können sie mit folgenden Compileraufrufen neu kompiliert werden. Zu beachten ist dabei, dass Prelude erzeugt worden sein muss, bevor die Standardbibliotheken erzeugt werden.

```
> run-main de.tuberlin.uebb.sl2.impl.Main -sourcepath
src/main/resources/lib/ prelude.sl

> run-main de.tuberlin.uebb.sl2.impl.Main -sourcepath
src/main/resources/lib/ buildstd.sl
```

## 5.2 Build-Prozess

Nach dem Aufruf von `Main` wird die Kompilierung intern durch den `MultiDriver` gesteuert, der folgende Phasen durchläuft.

**Initiale Modulobjekte erstellen** Zuerst werden Modulobjekte zur Verwaltung des Compilierungsprozesses für alle beim Aufruf des Compilers explizit angegebenen `<module files>` erstellt.

**Abhängigkeiten analysieren** Die Module werden ein erstes Mal geparsed, um anhand der Import-Deklarationen (siehe Abschnitt 4.1) weitere potentiell zu kompilierende Module zu ermitteln. Dabei werden bis auf `std/Prelude.sl` alle Module überprüft, die direkt von einem zu kompilierenden Modul importiert werden.

Ein Modul ist zu kompilieren, wenn

1. die Quell-Datei des Moduls zu den an den Compiler übergebenen `<module files>` gehört, oder
2. eine Quell-Datei des Moduls im `<source directory>` vorhanden ist, jedoch keine Signatur-Datei des Moduls im `<classpath directory>` oder im `<output directory>` vorhanden ist, oder
3. eine Quell-Datei des Moduls im `<source directory>` vorhanden ist, ebenso wie eine Signatur-Datei des Moduls im `<classpath directory>` oder im `<output directory>` und die Quelldatei jünger als oder genauso alt wie die Signatur-Datei ist.

Dabei ist zu beachten, dass die Identität eines Moduls durch seinen Modulpfad gegeben ist (siehe Abschnitt 4.1) und durch ein nicht durch den Compiler erzwungenes Benennungsschema sichergestellt werden muss, dass Quell- und Signatur-Dateien mit dem selben Modulpfad zum gleichen Modul gehören.

Für das Prelude und Bibliotheken unterhalb `std/` werden abweichend von obiger Darstellung aus dem Ressourcenverzeichnis des Compilers geladen, das sich für Scala Version 2.10 relativ zum SL2-Projektverzeichnis unter `target/scala-2.10/classes/lib/` befindet. Das `std/` wird dabei durch `lib/` ersetzt.

**Module linearisieren** Nachdem alle zu kompilierenden Module und die Import-Abhängigkeiten zwischen ihnen ermittelt wurden, werden die Module anhand ihrer Abhängigkeiten topologisch sortiert. Dabei werden auch zyklische Abhängigkeiten erkannt und die Compilierung abgebrochen, sofern sie vorliegen (siehe `cycleA` und `cycleB` in Abschnitt 8.1).

**Modul übersetzen** Nach der Linearisierung werden die zu kompilierenden Module in der gefundenen Reihenfolge übersetzt. Für jedes Module werden dabei folgenden Schritte durchgeführt.

**Parsing** Der Syntaxbaum des Moduls wird erstellt (siehe Abschnitt 3).

**Normalisierung der Modulbezeichner** Die Modulbezeichner werden normalisiert (siehe Abschnitt 4.2).

**Typprüfung** Es wird eine Typprüfung durchgeführt (siehe Abschnitt 4.2).

**Qualifizierung unqualifizierter Module** Die Funktionen und Datentypen aus allen unqualifiziert importierten Modulen werden qualifiziert, damit die von den Module deklarierten Funktionen und Konstruktoren aus den JavaScript-Objekt gelesen werden können, das zur Laufzeit das Module repräsentiert.

Da unqualifizierte Imports kein Sprachbestandteil von SL2 sind und nur `std/Prelude` standardmäßig implizit unqualifiziert importiert wird, wird dieser Schritt effektiv nur auf dieses Modul angewandt.

**Code-Generierung** Die Code-Generierung ist Template-basiert. Alle generierten Dateien wird relativ zum `<output directory>` erzeugt.

Für jedes zu kompilierende Modul wird eine Signatur-Datei (siehe Abschnitt 5.3) und mindestens eine JavaScript-Datei (siehe Abschnitt 5.4) erstellt.

Für das JavaScript werden Templates werden aus dem Ressourcenverzeichnis des Compilers geladen, die sich für Scala Version 2.10 relativ

zum SL2-Projektverzeichnis unter `target/scala-2.10/classes/js/` befinden (die Originale befinden sich unter `src/main/js/`). Folgende Templates sind verfügbar:

**module\_template.js** Wird für jedes zu kompilierende Modul verwendet, um den per PrettyPrinter formatierten vom Compiler generierten JavaScript-Quellcode so zu speichern, dass das Modul zur Laufzeit als `require.js`-Modul geladen werden kann. Der Name der aus dem Template generierten Datei entspricht dem Namen der Modul-Quelldatei mit dem Suffix `.js`, bspw. `std/prelude.sl.js` für das Modul `std/prelude`.

**main\_template.js** Wird zusätzlich für jedes Modul verwendet, dass eine `main`-Funktion `PUBLIC` deklariert. Mit dem Template wird eine Datei `main.js` erstellt, die das Modul lädt und seine `main`-Funktion aufruft.

Abschnitt 5.4 spezifiziert die Erstellung der JavaScript-Dateien aus den Templates.

Zusätzlich werden noch folgende Dateien aus dem Ressourcenverzeichnis ins `<output directory>` kopiert, sofern eines der kompilierten Module eine `main`-Funktion `PUBLIC` deklariert:

**index.html** Ruft die `main`-Funktion des kompilierten Moduls auf, indem `main.js` mittels `require.js` geladen wird.

**require.js** Der JavaScript-Code von `require.js`.

Beim Aufruf von

```
> run-main de.tuberlin.uebb.sl2.impl.Main -sourcepath
src/main/sl/examples/ boxsort.sl
```

werden bspw. folgende Dateien erzeugt, da `boxsort` eine `main`-Funktion deklariert:

```
boxsort.sl.signature
boxsort.sl.js
main.js
require.js
index.html
```

Da kein `<output directory>` angegeben wurden, befinden sich die Dateien unterhalb von `src/main/sl/examples`.

Zu beachten ist, dass, wenn mehr als ein Modul kompiliert werden, das eine `main`-Methode deklariert, die Dateien `main.js`, `index.html` und `require.js` für jedes Modul mit einer `main`-Methode erstellt werden. Dadurch werden die Dateien während der Compilierung überschrieben und nur das zuletzt kompilierte Modul verfügt nach Abschluss des Compiler-Laufes über die entsprechenden Dateien. Dieses Problem kann umgangen werden, indem Module mit `main`-Methoden separat kompiliert werden. Selbst wenn das nicht der Fall ist, kann die Linearisierung dafür sorgen, dass die `main.js`-Datei für das korrekte Modul erhalten bleibt, da das Modul zuletzt kompiliert wird, dass am Ende eines Abhängigkeitsbaumes steht.

### 5.3 Signaturen

Wie bereits in Abschnitt 4 erläutert, ist es notwendig Signaturen für Module zu erzeugen. Dies ist ein neues Merkmal, welches vor Einführung des Modulsystems nicht erforderlich war. Die JavaScript-Datei, welche den übersetzten SL-Code enthält ist unzureichend um die Signatur eines Moduls auszulesen. Die Signatur umfasst die Funktionssignaturen, Datendefinitionen und eine Liste von Importen des Moduls. Mit diesen Informationen ist es später möglich eine Typüberprüfung durchzuführen und transitive Importe zu erfassen.

Die Signatur wird als Datei abgespeichert. Dazu muss sie zunächst jedoch serialisiert werden. Es gab mehrere Alternativen, wie diese Serialisierung umgesetzt werden kann. Wir haben drei Möglichkeiten betrachtet:

1. Die Signatur als SL-Code ausgeben und zum Deserialisieren erneut parsen.
2. Die in Java/Scala eingebaute Serialisierungsfunktion von Objekten verwenden.
3. Sie in ein JSON-Objekt umwandeln.

Die erste Möglichkeit hat den Vorteil einfach lesbar und editierbar zu sein. Allerdings sind wir dabei auf die Ausdrucksmöglichkeiten von SL beschränkt. Dies könnte spätere Erweiterungen erschweren, wenn zum Beispiel Metadaten abgespeichert werden sollen.

Die Möglichkeit in Java nahezu beliebige Objekte serialisieren und später wieder laden zu können wäre eine einfache und schnelle Möglichkeit gewesen einen Teil des Syntaxbaums als Datei abzuspeichern. Allerdings ist diese schwer zu lesen und von Hand zu editieren.

Aufgrund der beiden Schwächen der ersten beiden Möglichkeiten haben wir uns für die dritte entschieden. Sie ermöglicht die erwünschten Freiheiten und bleibt dennoch von Hand editierbar. Die Implementierung war ebenfalls unproblematisch. Prinzipiell hätte man auch ein anderes bekanntes Format wie z.B. XML verwenden können. Der Vorteil von JSON ist jedoch, dass es Teil der JavaScript-Syntax ist und damit nativ von JavaScript eingelesen werden kann. Zwar haben wir dies in unserem Projekt noch in keiner Form ausgenutzt, kann später jedoch von Vorteil sein.

Da die Signatur selbst ein Teil der abstrakten Syntax ist, haben wir uns deren Struktur als Beispiel für die JSON-Objektstruktur genommen. Diese Struktur ist am besten anhand eines Beispiels zu erläutern. Die Signatur des folgenden Moduls soll zu JSON serialisiert werden:

```
IMPORT "some/module" AS M

PUBLIC DATA MyType a = Ctor1 a | Ctor2

PUBLIC FUN foo : MyType Int -> Int
DEF foo x = ...
```

Das Wurzelobjekt ist ein JSON-Hash der stets die drei Elemente `imports`, `signatures` und `dataDefs` enthält. Dies sind auch die Bezeichner der korrespondierenden Felder aus der Klasse `Program`, welches den abstrakten Syntaxbaum speichert.

`imports` ist ein Array, welches alle Importfelder aus Modulpfad und -bezeichner beinhaltet:

```
"imports" : [
  {
    "name" : "M",
    "path" : "some/module"
  }
]
```

Dagegen ist `signatures` ein Hash, wie das Vorbild aus `Program` welches eine `Map[VarName, FunctionSig]` ist. In diesem Hash werden Bezeichner und Typ der Funktionen zugeordnet.

```
"signatures" : {
  "foo" : [
    {
      "type" : ".MyType",
      "params" : [{"type" : ".Int", "params" : []}]
    },
    {
      "type" : ".Int",
      "params" : []
    }
  ]
}
```

In diesem Beispiel sehen wir auch, wie der Typ der Funktion `foo` serialisiert wird. Es gibt drei Sorten von Typen:

**Typvariablen** die durch einen beliebigen Typen ersetzt werden können.

**Funktionstypen** welche prinzipiell eine Auflistung von Typen ist. Der Funktionstyp `A -> B -> C` wird als Liste der Typen `A`, `B` und `C` dargestellt.

**Typausdruck** ist ein konkreter singulärer Typ, welcher denselben Bezeichner wie in seiner `DATA`-Definition beinhaltet. Außerdem können diese Parameter-typen entgegennehmen, welche wiederum eine Liste von Typen ist.

Typvariablen werden in JSON als einfache Zeichenkette dargestellt. Funktionstypen sind Arrays, welche weitere Typen beinhalten. Der Typausdruck ist ein zwei-elementiger Hash mit den Werten `type` für den Bezeichner und `params` welcher eine Liste von Typen speichert. Da für jede Art von Typ ein anderer JSON-Objektyp verwendet wird, sind die Typarten auch in JSON einfach voneinander zu unterscheiden.

Zuletzt müssen noch die Datentypdefinitionen serialisiert werden. Diese werden wie Importe in einem Array aufgezählt. Eine einzelne Definition ist wiederum ein Hash, der die relevanten Felder der `DataDef`-Klasse enthält. Dies sind `ide` für den Typbezeichner, eine Liste `tvars` aus Typvariablen und eine Liste von Konstruktoren `constructors`. Ein Konstruktor besteht wiederum aus einem Bezeichner `constructor` und eine Liste aus Typvariablen `types`.

```
"dataDefs" : [
  {
    "ide" : "MyType",
    "tvars" : ["a"],
    "constructors" : [
      {
        "constructor" : "Ctor1",
        "types" : ["a"]
      },
      {
        "constructor" : "Ctor2",
        "types" : []
      }
    ]
  }
]
```

#### 5.4 require.js

Um die Module zur Laufzeit in JavaScript zu laden, wurde `require.js`<sup>4</sup> statt `CommonJS`<sup>5</sup> ausgewählt, da es im Gegensatz zum Modulsystem von `node.js` auch ohne größeren Aufwand sowohl im Browser als auch in `node.js` verfügbar ist.

Da der Compiler die JavaScript-Datei von `require.js` mit ins Ausgabeverzeichnis kopiert, wird `require.js` bei SL2-Module für den Browser gleich mitgeliefert. In `node.js` muss `require.js` allerdings noch installiert werden, damit kompilierte SL2-Programme in `node.js` geladen und ausgeführt werden können.

Mit dem über die Shell gestarteten `node package manager (npm)` kann `require.js` mit folgendem Befehl in `node.js` installiert werden:

```
> npm install requirejs
```

Dadurch wird die jeweils aktuellste Version von `require.js` heruntergeladen und installiert. Dabei ist zu beachten, dass die Installation auf bestimmten Betriebssystemen unter Umständen in einem lokalen Verzeichnis relativ zum aktuellen Verzeichnis während des `npm`-Aufrufs geschieht und in anderen Verzeichnissen wiederholt werden muss.

<sup>4</sup> <http://requirejs.org/> v. 2.1.6

<sup>5</sup> <http://www.commonjs.org/>

In `require.js` stehen zwei Wege zur Verfügung, um Abhängigkeiten zwischen Modulen zu deklarieren und zur Laufzeit aufzulösen. In beiden Fällen wird eine von `require.js` bereitgestellte `define`-Funktion aufgerufen, der eine Funktion übergeben wird die ein Modul-Objekt erstellt, für das Eigenschaften und Funktionen deklariert werden.

Die erste Möglichkeit, bspw. in

```
define(["modules/B"], function(b) {
  return {
    "a" : function() { return "A.a"; },
    "b" : function() { return b.b(); }
  };
});
```

(Moduldefinitionen mit einem Array von Abhängigkeiten) erlaubt den Zugriff auf verwendete Module, kann jedoch keine zirkulären statischen Abhängigkeiten auflösen, da für die Erstellung gegenseitig abhängiger Module jeweils das andere Modul-Objekt als Parameter bei Erstellung des Moduls übergeben werden muss. Dieses Problem wird in `require.js` mittels Exports-Objekten gelöst, die beim Erstellen eines Moduls an die Funktion übergeben werden, die das Modul erstellt und später von anderen Modulen als Repräsentation des Moduls verwendet werden, wie im folgenden Beispiel

```
define(function(require, exports, module) {
  var b = require("modules/B");
  exports.a = function() { return "A.a"; };
  exports.b = function() { return b.b(); };
});
```

Die Moduldefinition mit Exports-Objekten wurde in SL2 gewählt, um später die Möglichkeit zu haben, statische zirkuläre Abhängigkeiten auflösen zu können. Dabei werden Module jeweils von der `require`-Funktion geladen, die das Export-Objekt zurückgibt, dem die Module bei ihrer Konstruktion (vor oder nach Ausführung der Funktion, die den `require`-Aufruf enthält) Funktionen und Parameter zuweisen. Die an `require` übergebenen Modulpfade werden absolut interpretiert (siehe auch das Beispiel `sub/relative.sl` in Abschnitt 8.1).

Die Template-basierte Modulgenerierung funktioniert folgender Maßen. Die JavaScript-Dateien der einzelnen Module werden mit dem folgenden `module_template.js` erstellt.

```
define(function(require, exports, module) {
  %%MODULE_BODY%%
});
```

Das Beispielprogramm `src/main/sl/examples/boxsort.sl` importiert u.a. `std/debuglog`, deklariert eine PUBLIC deklarierte `main`-Funktion, sowie eine nicht PUBLIC deklarierte Funktion `getNode`:

```

IMPORT "std/debuglog" AS Dbg
...

PUBLIC FUN main : DOM Void
DEF main =
Web.document &= \ doc .
    ...

DEF getNode (NodeWithNumber n1 i1) = n1
...

```

Folgender Ausschnitt aus der compilierten Datei `boxsort.sl.js` zeigt exemplarisch einige wichtige Aspekte der Codegenerierung für `require.js`

```

define(function(require, exports, module) {
    var $$std$prelude = require("std/prelude.sl");
    var Dbg = require("std/debuglog.sl");
    ...
    function $getNode(_arg0) { ... };
    ...
    var $main = function () { ... }();
    exports.$main = $main
});

```

1. `std/prelude.sl` wird implizit importiert und durch den Compiler mit dem konvertierten Modulpfad qualifiziert, da der eigentliche Import unqualifiziert war. Dazu werden der Modulpfad mit `$` präfigiert, sowie Vorkommen von `/` und `in` Modulpfad durch `$` ersetzt. ersetzt, wenn Variablenbezeichner erzeugt werden
2. `std/debuglog.sl` wurde explizit importiert, daher steht der Modulbezeichner als Variablenname zur Verfügung. Im Gegensatz zu Namen von Funktionen und Konstruktoren ist er nicht mit einem `$` präfigiert, sodass beide Namensräume getrennt sind.
3. Für die Funktion `getNode` wurde nicht `public` deklariert und wird daher nicht im `exports`-Objekt gespeichert. Sie ist daher nicht für andere Module erreichbar, denen gegenüber das `exports`-Objekt das Modul repräsentiert.
4. Die Funktion `main` wurde hingegen `PUBLIC` deklariert und wird daher auch `exports` hinzugefügt.

Damit `main` aus `boxsort.sl.js` aufgerufen werden kann, muss das Modul `boxsort.sl` geladen werden. Dies geschieht mit einer aus `main_template.js` erstellten Datei:

```

if (typeof window === 'undefined') {
    /* in node.js */
    var requirejs = require('requirejs');
}

```



```

requirejs.config({
    //Pass the top-level main.js/index.js require
    //function to requirejs so that node modules
    //are loaded relative to the top-level JS file.
    nodeRequire: require,
paths: %%STD_PATH%%
});

requirejs([%%MODULE_PATHS_LIST%%], function(%%MODULE_NAMES_LIST%%) {
    %%MAIN%%
});
} else {
    require.config({
paths: %%STD_URL%%
});

    /* in browsers*/
    require([%%MODULE_PATHS_LIST%%], function(%%MODULE_NAMES_LIST%%) {
        %%MAIN%%
    });
}

```

Das Template ist in zwei Teile untergliedert: im oberen Teil steht der Code, der ausgeführt wird, wenn die Datei in `node.js` aufgerufen wird. Der untere Teil wird im Browser ausgeführt.

Es werden ein Pfad bzw. eine URL zu den Standardbibliotheken, sowie Modulpfade, Modulbezeichner und der Aufruf der Main-Methode eingefügt, sodass folgende `main.js`-Datei für das `boxsort.sl`-Beispiel generiert wird.

```

if (typeof window === 'undefined') {
    /* in node.js */
    var requirejs = require('requirejs');

    requirejs.config({
        //Pass the top-level main.js/index.js require
        //function to requirejs so that node modules
        //are loaded relative to the top-level JS file.
        nodeRequire: require,
        paths: {std : "C:/Users/monochromata/git/sl2/target/scala-2.10/
            classes/lib" }
    });

    requirejs(["boxsort.sl"], function($$$boxsort) {
        $$$boxsort.$main()
    });
}

```

```

} else {
  require.config({
    paths: {std : "file:/C:/Users/monochromata/git/sl2/target/scala-2.10/
      classes/lib/" }
  });

  /* in browsers*/
  require(["boxsort.sl"], function($$$boxsort) {
    $$$boxsort.$main()
  });
}

```

Wichtig an dieser Datei sind vor allem die Pfad-Konfigurationen für den Präfix `std` des Prelude und der Standard-Bibliotheken. Die Konfiguration gilt für die gesamte Ausführung von `require.js`, also auch, wenn `boxsort.sl.js` ausgeführt wird, das u.a. `std/prelude.sl` und `std/debuglog.sl` lädt. Die Pfad-Konfiguration gibt einen Ort an, von dem `require.js` das Prelude und die Standardbibliotheken lädt. Durch diesen Mechanismus müssen Prelude und Standardbibliotheken nur 1x zentral auf dem Server abgelegt werden. Das bedingt jedoch, dass die Pfade auf jedem Rechner, auf dem die Datei `main.js` ausgeführt werden soll, angepasst wird. In homogenen Umgebungen kann man statt dessen auch lokal immer den selben Pfad für die Standardbibliothek verwenden. Zu beachten ist dabei noch, dass für `node.js` ein absoluter Pfad, im Browser jedoch eine `file:-URL` verwendet wird.

Die `index.html`-Datei, die `require.js` im Browser aufruft, wird lediglich kopiert und ist identisch für alle Module mit `main`-Methode. Sie kann jedoch im Nachhinein modifiziert werden, bspw. um eine Corporate Identity umzusetzen.

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>SL2</title>
<script data-main="main.js" src="require.js"></script>
</head>
<body>
</body>
</html>

```

SL2-Module mit `main`-Methode werden im Browser ausgeführt, indem die vom Compiler generierte `index.html`-Datei im Browser geöffnet wird. Die Module werden in `node.js` mit folgendem Shell-Aufruf gestartet.

```
> node main.js
```

## 5.5 Externe Definitionen

Besonders beim Schreiben von Funktionsbibliotheken muss man häufig auf JavaScript-Funktionen zugreifen, ohne dass man dabei in eine DOM-Monade geraten will. Das ursprüngliche SL erlaubte nur JS-Code-Literale vom Typ `DOM x`. Da es kein `return` für `DOM` gibt und auch keins geben soll, ist es damit nicht möglich, neue Funktionen in SL zu schreiben, die einen anderen Rückgabetyt als `DOM` besitzen.

Um dennoch das Prelude von SL und weitere Bibliotheken (Abschnitt 7) in SL verfassen zu können, haben wir durch `DEF EXTERN` einen sehr definierten Platz geschaffen, an dem ein JavaScript-Literal `{|someJsCode()|}` ausgepackt werden darf. Die Übersetzung dazu ist vergleichsweise simpel:

```
DEF EXTERN bla = {| js |}  ⇔  var bla = js;
```

Um auf der rechten Seite dieser Definition bequem selbstdefinierte JavaScript-Funktionen anzugeben, bietet es sich an, diese in einem weiteren Modul zusammenzufassen und sie dann per `IMPORT EXTERN` einzubinden. Die Übersetzung zu `IMPORT EXTERN "<datei>"` ist dabei schlicht, dass an den Anfang des Kompilats der Inhalt der Datei `"datei.js"` gesetzt wird.

## 6 Fehlermeldungen

**TODO (\* Fritz \*)**

## 7 Prelude und Bibliotheken

Einerseits zur Erweiterung des ursprünglichen Funktionsumfangs, andererseits vor allem zum Testen des neuen Modulsystems, haben wir eine Reihe grundlegender Bibliotheken für SL entwickelt. Im Folgenden wollen wir Ausschnitte aus den Bibliothekssignaturen vorstellen, ihre Funktionen angerissen und Besonderheiten bei ihrer Verwendung des Modulsystems und neuer Sprachfeatures ansprechen. Die vollständigen Module inklusive Implementierung finden sich in `/src/main/resources/lib/`.

### 7.1 Prelude

Fast alle vormals fest in den Compiler eingebauten Funktionen und Konstruktoren werden jetzt durch ein eigenes, umfangreicheres Prelude-Modul definiert. Dieses wird implizit durch jedes SL-Programm unqualifiziert importiert.

Im Prelude werden unter anderem alle Basistypen deklariert. Zugleich sind diese allerdings noch in den Compiler integriert, damit die Literale einen Typ erhalten können unabhängig vom Prelude-Import. Die meisten dieser Datentypen kommen ohne Konstruktorendefinition daher, sind deshalb aber noch lange nicht leer, was wir durch `DATA EXTERN` anzeigen.

```
DATA EXTERN Int
DATA EXTERN Real
DATA EXTERN Char
DATA EXTERN String
```

```
PUBLIC DATA Void = Void
DATA EXTERN DOM a
```

Stärker als andere Module bildet das Prelude Funktionen auf handgeschriebenen JavaScript-Code ab. Diese Abbildung wurde bisher durch eine hardcodierte Umwandlung im SL-Compiler realisiert. Dank `IMPORT EXTERN` und `DEF EXTERN` **VERWEIS EINBAUEN** kann das Prelude selbst spezifizieren, dass `+` auf das JavaScript-Objekt `_add` aus `_prelude.js` abgebildet werden soll.

```
IMPORT EXTERN "_prelude"
[...]
PUBLIC FUN + : Int -> Int -> Int
DEF EXTERN + = {| _add |}
```

So sind weite Teile der Preludes umgesetzt. Andere grundlegende Aspekte sind hingegen völlig in SL definiert, zum Beispiel der Datentyp `BOOL`.

```
PUBLIC DATA Bool = True | False
```

```
PUBLIC FUN not : Bool -> Bool
DEF not True = False
DEF not False = True
```

Es sind auch einige neue Funktionen hinzugekommen, zum Beispiel `#` für Funktionskomposition<sup>6</sup> und `id` als Identitätsfunktion.

```
PUBLIC FUN # : (b -> c) -> (a -> b) -> (a -> c)
DEF f # g = \ x . f (g x)
```

```
PUBLIC FUN id : a -> a
DEF id a = a
```

Eine spannende neue Funktion im Prelude ist `error`. Diese hat einen beliebigen Rückgabebetyp, kann also an beliebigen Stellen in den Code geschrieben werden. Allerdings wird `error` niemals einen Wert zurückgeben, sondern schlicht das Programm mit einer Fehlermeldung enden lassen.<sup>7</sup> Man kann sich das `error` auch als eine Möglichkeit vorstellen, in der Abwesenheit von Subtyping, eine Art Bottom-Type einzuführen. Vor allem ist es aber praktisch: Häufig möchte man im

<sup>6</sup> Das ungewöhnliche Zeichen rührt daher, dass „o“ in SL kein Operator sein kann und „.“ für die Lambda-Abstraktion und Namensqualifizierung reserviert ist.

<sup>7</sup> Diese Funktion ist also keine echte, wohldefinierte Funktion, sondern hat dasselbe „Ergebnis“ wie eine Endlosrekursion.

Implementierungsprozess schon teile Testen, aber noch nicht überall sinnvollen Code eintragen. Manchmal lässt sich für einen Fall auch einfach kein sinnvolles Programmverhalten angeben.

```
-- The representation of the undefined.
PUBLIC FUN error : String -> a
DEF EXTERN error = {| function(msg){throw msg} |}
```

## 7.2 List, Option, Either

Unsere mitgelieferten Module enthalten die klassischen algebraischen, generischen Datentypen `List` (aka Sequence), `Option` (aka Maybe), `Either` (aka Union) und `Pair` (aka Product2). Bis auf `List.fromString` sind diese Module komplett in SL geschrieben ohne Rückgriff auf JavaScript. Wir haben auch ein paar der grundlegenden Funktionen wie `map` und `reduce` implementiert. Vorrangig ging es uns aber darum, komplexere importierte Konstruktoren beim Pattern Matching anhand dieser Typen auszuprobieren.

```
PUBLIC DATA List a      = Nil | Cons a (List a)
PUBLIC DATA Option a   = None | Some a
PUBLIC DATA Either a b = Left a | Right b
PUBLIC DATA Pair a b   = Pair a b
```

## 7.3 Reelle Zahlen — `real.sl`

Am Anfang des Projekts hatten wir reelle Zahlen in SL integriert. Diese und noch mehr Funktionen auf Reals werden jetzt in `real.sl` definiert durch Abbildung auf entsprechende Funktionen auf JavaScripts `num`. Bei der ursprünglichen Umsetzung erwies sich als ausgesprochen unhandlich, dass die Operatoren wie `+` und `/` schon durch ihre Verwendung für Integer belegt waren. `real.sl` überschreibt für sich die Operatoren. Zum Beispiel enthält es folgende Definitionen:

```
PUBLIC FUN + : Real -> Real -> Real
PUBLIC FUN / : Real -> Real -> Real
PUBLIC FUN == : Real -> Real -> Bool
PUBLIC FUN round : Real -> Int
PUBLIC FUN fromInt : Int -> Real
```

In einem anderen Modul kann somit also `(R.fromInt x) R.* 0.333` geschrieben werden. `real.sl` ist also für uns auch eine gute Möglichkeit, um das Zusammenspiel von aus dem Prelude importierten unqualifizierten Bezeichnern und Modulinternen deklarationen auszutesten.

## 7.4 Dictionaries — dict.sl

Anders als zum Beispiel `List` ist der abstrakte Datentyp `Dict` komplett ohne SLs algebraische Datentypen umgesetzt. Stattdessen arbeiten die Implementierungen der einzelnen Funktionen ausschließlich mit JavaScripts `Object`, also den in JavaScript grundlegenden Wörterbuchobjekten.

```
DATA EXTERN Dict a
PUBLIC FUN empty : Dict a
PUBLIC FUN put : Dict a -> String -> a -> Dict a
PUBLIC FUN has : Dict a -> String -> Bool
PUBLIC FUN get : Dict a -> String -> a
PUBLIC FUN getOpt : Dict a -> String -> Opt.Option a
PUBLIC FUN fromList : (String -> a) -> List.List String -> Dict a
```

`dict.sl` zeigt, wie man auch außerhalb des durch den SL-Compiler vorgesehenen besonderen Fleckchens `prelude.sl`, sinnvoll Strukturen durch Rückgriff auf JavaScript definieren kann, die auch mit rein SL-definierten Strukturen wie `List` und `Option` interagieren können.

## 7.5 println-Debugging — debuglog.sl

Das neue Modul `debuglog` erlaubt, normale Programme mit Konsolenausgaben zu versehen, die neben der Programmausführung ausgegeben werden.

```
PUBLIC FUN print : String -> DOM Void
PUBLIC FUN andPrint : a -> (a -> String) -> a
PUBLIC FUN andPrintMessage : a -> String -> a
```

Im Hintergrund bilden die Funktionen auf `console.log` ab, das unter `node.js` sowie neueren Versionen von Firefox (bzw. Firebug), Internet Explorer (ab IE8, Developer Tools) unauffällige Programmausgaben ermöglicht.

Allerdings bewegen sich `andPrint` sowie `andPrintMessage` und die Hilfsfunktion `logAvailable : Bool` am Rand des funktionalen Paradigmas.

```
I0.andPrint (L.Cons 1 (L.Cons 2 L.Nil)) (L.toString intToStr)
```

Dieser Ausdruck hat als Rückgabewert die Liste  $\langle 1, 2 \rangle$ , während als (fürs Programm hoffentlich unsichtbarer) Seiteneffekt, noch `"<1,2>"` auf die Konsole geschrieben wird. Semantisch sollten `andPrint` sowie `andPrintMessage` äquivalent zur Identitätsfunktion mit ein paar unnötigen Parametern sein. Solange man es wie `Debug.Trace.trace` in Haskell nur vorsichtig für Debugging-Zwecke einsetzt, sollte alles klar gehen.

## 7.6 Browseranbindung — `basicweb.sl`

Wir schrieben auch eine kleine Bibliothek `basicweb`, die einige der Input/Output-Möglichkeiten von Websites bereitstellt. Diese Bibliothek ergibt natürlich nur sinn, wenn das mit SL erzeugte JS-Script im Browser ausgeführt wird.

```
DATA EXTERN Node
DATA EXTERN Document

PUBLIC FUN document : DOM Document
PUBLIC FUN getBody : Document -> DOM Node

PUBLIC FUN appendChild : Node -> Node -> DOM Void
PUBLIC FUN removeChild : Node -> Node -> DOM Void
PUBLIC FUN getChildNodes : Node -> DOM (List.List Node)

PUBLIC FUN setOnClick : Node -> DOM Void -> DOM Void
PUBLIC FUN getValue : Node -> DOM String
PUBLIC FUN setValue : Node -> String -> DOM Void

PUBLIC FUN createElement : Document -> String -> DOM Node
PUBLIC FUN createButton : Document -> String -> DOM Void -> DOM Node
PUBLIC FUN createInput : Document -> String -> DOM Void -> DOM Node

PUBLIC FUN alert : String -> DOM Void
PUBLIC FUN prompt : String -> String -> DOM String
```

Wir haben nur einen sehr kleinen Teil der Standard-JavaScript-Befehle abgebildet. Mit diesem Teil lässt sich schon eine überschaubare Webanwendung wie in `boxsort.sl` gezeigt umsetzen, die in gängigen modernen Browsern läuft.

## 7.7 Zusammenfassung

Die entwickelten Bibliotheken sind weit davon entfernt, durchdacht und ausgewachsen zu sein. Sie zeigen jedoch schon gut, wie unsere neuen Features es erlauben, verschiedene Funktionen in Modulen zu sammeln und diese Module aufeinander aufbauen zu lassen.

Es wird deutlich, dass die vorgeschlagenen `EXTERN`-Konstrukte es erlauben, auch funktionale Bibliotheken wie `dict.sl` ohne eingriffe in den Compiler zu entwickeln. Die monadischen JavaScript-Literale sind mächtig genug, um Aspekte wie die Interaktion mit dem Browser in Modulen wie `basicweb.sl` zusammenzufassen.

Das Prelude als echtes Modul umzusetzen, gestaltet auch den Compiler übersichtlicher. Die Prelude-Funktionen sind jetzt gleichberechtigte Funktionen innerhalb der Sprache und führen kein Eigenleben in Checks und Codegenerierung mehr.

## 8 Beispielprogramme und Tests

### 8.1 Beispielprogramme

Wir haben eine Reihe kleinerer Testprogramme geschrieben.

`hello.sl` Das minimale „Hello World“-Programm. Verwendet nur `debuglog`.  
`helloworld.sl` Spielt mit diversen Grundlagen aus `list`, `option` und `dict`.  
`transitiveimports.sl` Verwendet `option`, ohne es direkt zu importieren. Stattdessen werden `dict` und `list` benutzt. (Das ist ein wichtiger Testfall!)  
`similarimports.sl` Importiert eine neudefinierte `Option` und zeigt, dass sie nicht mit `Option` aus `std/option` clasht.  
`librarytest.sl` Testet das Zusammenspiel einiger Funktionen aus `list` und `dict` sowie `real`. Macht außerdem vom lokalen Überschreiben von Prelude-Bezeichnern Gebrauch.  
`boxsort.sl` Größeres Beispiel, das mittels `basicweb` eine interaktive Website erzeugt. Macht starken Gebrauch von allen möglichen Features aus den `std`-Librarys.  
`koch.sl` Modifizierte Version des ursprünglichen Kochkurvenbeispiels. Verwendet Browseranzeige und `timing`, um eine Animation ausgehend von der Kochkurve zu zeichnen.  
`sub/hello.sl` Gibt "Hello Moon!äus und wird von `moonWorld.sl` verwendet, um das Einbinden verschieden qualifizierter Module mit identischem einfachen Namen zu testen.  
`sub/relative.sl` Wird verwendet, um zu zeigen, dass `require.js` Modulpfade absolut statt relativ zum aktuellen Modul interpretiert.  
`moonWorld.sl` Verwendet `hello.sl` und `sub/hello.sl`, um zu prüfen, ob die beiden Module getrennt kompiliert und eingebunden werden.  
`cycleA.sl` und `cycleB.sl` enthalten eine direkte zyklische Abhängigkeit, um zu testen, ob diese erkannt wird.

## 9 Zusammenfassung

TODO (\* ... \*)