Projektbericht: Erweiterung von SL um ein Modulsystem

Benjamin Bisping, Rico Jasper, Sebastian Lohmeier, Friedrich Psiorz

Compilerbauprojekt SoSe 2013
Technische Universität Berlin

Zusammenfassung. Worum geht es hier?

Inhaltsverzeichnis

1 Einleitung

Das World Wide Web hat heutzutage eine größere Bedeutung denn je und ist aus dem Alltag nicht mehr wegzudenken. Browser bieten die Möglichkeit eine breite Palette an Anwendungen auszuführen. Sogenannte „Rich Web Applications“ basieren zum Großteil auf dem Zusammenspiel von HTML, CSS und JavaScript. Praktisch alle modernen Browser unterstützen die Ausführung von JavaScript, welche hier als Programmiersprache dominiert. Somit sind Programmierer für Webapplikation de-facto dazu gezwungen JavaScript einzusetzen.

Einige Programmierer würden jedoch eine funktionale Sprache mit strikter Typüberprüfung vorziehen. Solch eine Sprache kann eine höhere Robustheit und Sicherheit bieten. „Simple Language“ (SL) bietet die Möglichkeit Programme im funktionalen Stil für Plattformen zu entwickeln, die JavaScript ausführen. Denn in SL entwickelte Software kann in JavaScript übersetzt werden.

Im Rahmen unseres Projekts haben wir die Sprache SL und den dazugehörigen Compiler weiterentwickelt. Das nachfolgende Kapitel bietet einen Überblick über die Änderungen und Ergänzungen. Die weitere Gliederung folgt den Verarbeitungsschritten eines Übersetzungsvorgangs des Compilers.

2 Überblick

Ziel des Projektes war es, SL um ein Modulsystem zu erweitern. Dazu reicherten 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 mittels `IMPORT` importiert 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 Kompilat 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 3.4)

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 vormals 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

3.1 Grammatik

Die Grammatik von SL ist in der Abbildung 1, die lexikalische Struktur ist in Abbildung 2 aufgelistet. Nicht erwähnt darin sind die Kommentare und Operator-Präzedenzen. SL folgt bei den Kommentaren der Tradition von Haskell; es gibt also einerseits Zeilenkommentare, die mit `--` eingeleitet werden, andererseits auch Blockkommentare, die von `{-` und `-}` eingeschlossen werden. Die Päzedenzen sind in der folgenden Tabelle aufgelistet:

Operatoren	Beschreibung	Präzedenz
kein	Funktionsapplikation	4
`*`, `/`	Multiplikation und Division	3
`+`, `-`	Addition und Subtraktion	2
`>`, `<`, `>=`, `<=`, `==`, `/=`	Vergleichsoperatoren	1
`&`, `&=`	Bind-Operatoren	1
sonstige	Andere Operatoren	0

Eine höher Präzedenz bedeutet, dass der Operator stärker bindet. Alle Operatoren sind linksassoziativ.

3.2 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.

$$
\begin{array}{lll}
\textit{Program} & ::= & \textit{Toplevel}^{+} \\
\textit{Toplevel} & ::= & \textit{Import} \mid \textit{Signature} \mid \textit{FunDef} \mid \textit{DataDef} \\
\textit{Import} & ::= & \texttt{IMPORT EXTERN}\ \textit{string} \\
 & \mid & \texttt{IMPORT}\ \textit{string}\ \texttt{AS}\ \textit{module} \\
\textit{Signature} & ::= & [\texttt{PUBLIC}]\ \texttt{FUN}\ \textit{var}\ \texttt{:}\ \textit{Type} \\
\textit{Type} & ::= & \textit{BaseType}\ (\texttt{->}\ \textit{BaseType})^{*} \\
\textit{BaseType} & ::= & \textit{typevar} \\
 & \mid & \textit{TypeExpr} \\
 & \mid & \texttt{(}\ \textit{Type}\ \texttt{)} \\
\textit{TypeExpr} & ::= & \textit{TypeCon}\ \textit{BaseType}^{*} \\
\textit{DataDef} & ::= & [\texttt{PUBLIC}]\ \texttt{DATA EXTERN}\ \textit{string} \\
 & \mid & [\texttt{PUBLIC}]\ \texttt{DATA}\ \textit{typecon}\ \textit{typevar}^{*}\ \texttt{=}\ \textit{DataBody} \\
\textit{DataBody} & ::= & \textit{con}\ \textit{TypeArg}^{*}\ \left(\texttt{|}\ \textit{con}\ \textit{TypeArg}^{*}\right)^{*} \\
\textit{TypeArg} & ::= & \textit{typevar} \\
 & \mid & \textit{TypeCon} \\
 & \mid & \texttt{(}\ \textit{Type}\ \texttt{)} \\
\textit{FunDef} & ::= & \texttt{DEF EXTERN}\ \textit{var}\ \texttt{=}\ \textit{JsQuote} \\
 & \mid & \texttt{DEF EXTERN}\ \textit{op}\ \texttt{=}\ \textit{JsQuote} \\
 & \mid & \texttt{DEF}\ \textit{var}\ \textit{Pat}^{*}\ \texttt{=}\ \textit{Expr} \\
 & \mid & \texttt{DEF}\ \textit{Pat}\ \textit{op}\ \textit{Pat}\ \texttt{=}\ \textit{Expr} \\
\textit{Pat} & ::= & \textit{var} \\
 & \mid & \textit{Con} \\
 & \mid & \texttt{(}\ \textit{Con}\ \textit{Pat}^{*}\ \texttt{)} \\
\textit{Expr} & ::= & \texttt{IF}\ \textit{Expr}\ \texttt{THEN}\ \textit{Expr}\ \texttt{ELSE}\ \textit{Expr} \\
 & \mid & \texttt{\textbackslash}\ \textit{Pat}^{+}\ \texttt{.}\ \textit{Expr} \\
 & \mid & \texttt{CASE}\ \textit{Expr}\ \textit{Alt}^{+} \\
 & \mid & \texttt{LET}\ \textit{LocalDef}^{+}\ \texttt{IN}\ \textit{Expr} \\
 & \mid & \textit{Expr}\ \textit{Expr} \\
 & \mid & \texttt{(}\ \textit{Expr}\ \texttt{)} \\
 & \mid & \textit{Expr}\ \textit{op}\ \textit{Expr} \\
 & \mid & \textit{JsQuote}\ [\texttt{:}\ \textit{Type}] \\
 & \mid & \textit{Var} \mid \textit{Con} \mid \textit{int} \mid \textit{real} \mid \textit{char} \mid \textit{string} \\
\textit{Alt} & ::= & \texttt{OF}\ \textit{PPat}\ \texttt{THEN}\ \textit{Expr} \\
\textit{PPat} & ::= & \textit{var} \mid \textit{con}\ \textit{Pat}^{*} \\
\textit{LocalDef} & ::= & \textit{var}\ \texttt{=}\ \textit{Expr} \\
\textit{JsQuote} & ::= & \texttt{\{|}\ \textit{string}\ \texttt{|\}} \\
\textit{Var} & ::= & [\textit{module}\ .]\ \textit{var} \\
\textit{Con} & ::= & [\textit{module}\ .]\ \textit{con} \\
\textit{TypeCon} & ::= & [\textit{module}\ .]\ \textit{typecon}
\end{array}
$$

Grammar 1: Grammatik von SL

$$
\begin{array}{lll}
\textit{var} & ::= & \textit{lowercase}\ \textit{alphanum}^* \\
\textit{con} & ::= & \textit{uppercase}\ \textit{alphanum}^* \\
\textit{typevar} & ::= & \textit{lowercase}\ \textit{alphanum}^* \\
\textit{typecon} & ::= & \textit{uppercase}\ \textit{alphanum}^* \\
\textit{module} & ::= & \textit{uppercase}\ \textit{alphanum}^* \\
\textit{op} & ::= & \textit{singleop} \mid \textit{multiop}^+ \\
\textit{singleop} & ::= & \texttt{!} \mid \texttt{§} \mid \texttt{\%} \mid \texttt{\&} \mid \texttt{/} \mid \texttt{?} \\
 & & \mid \texttt{+} \mid \texttt{*} \mid \texttt{\#} \mid \texttt{-} \mid \texttt{<} \mid \texttt{>} \\
\textit{multiop} & ::= & \textit{singleop} \mid \texttt{=} \\
\textit{lowercase} & ::= & \texttt{a} \mid \cdots \mid \texttt{z} \\
\textit{uppercase} & ::= & \texttt{A} \mid \cdots \mid \texttt{Z} \\
\textit{digit} & ::= & \texttt{0} \mid \cdots \mid \texttt{9} \\
\textit{alphanum} & ::= & \textit{lowercase} \mid \textit{uppercase} \mid \textit{digit} \\
\textit{int} & ::= & [\texttt{-}]\ \textit{digit}^+ \\
\textit{real} & ::= & [\texttt{-}]\ \textit{realbody}\ [\textit{exp}] \\
\textit{realbody} & ::= & \textit{digit}^+\ \texttt{.}\ \textit{digit}^* \mid \texttt{.}\ \textit{digit}^+ \\
\textit{exp} & ::= & \texttt{e}\ \textit{int} \mid \texttt{E}\ \textit{int} \\
\textit{char} & ::= & \texttt{'}\ \textit{anychar}\ \texttt{'} \\
\textit{string} & ::= & \texttt{"}\ \textit{anychar}^*\ \texttt{"}
\end{array}
$$

Grammar 2: Lexikalische Struktur von SL

Was die ungewöhnlich benannten Operatoren für Zeichenketten- und Gleitkommaoperationen betrifft, so haben wir uns dafür entschieden, sie aus der Sprache zu entfernen. Dies verhindert auch Parserprobleme, wenn Bezeichner direkt hinter Operatoren geschrieben werden; so wurde etwa `1+sum` bisher als `1 +s um` geparst, was zu Verwirrung führen kann. Der Operator für die String-Konkatenation heißt nun `++`, die Gleitkommaoperationen sind nun in einem eigenen Modul untergebracht und müssen in Programmen entsprechend qualifiziert verwendet werden, z.B. schreibt man nun `1.0 R.+ 2.5` statt `1.0 +r 2.5`, falls das Modul *std/real* als `R` importiert wurde.

Das unäre Minus haben wir komplett aus der Sprache entfernt. Stattdessen gibt es jetzt in der Prelude eine Funktion `neg` für die Negation von Ganzzahlen; außerdem können nun Gleitkomma- und Ganzzahlliterale ein Minus als Vorzeichen enthalten, wenn dieses nicht durch Leerschritte oder Klammern von der ersten Ziffer bzw. dem Dezimalpunkt getrennt ist. So sind beispielsweise die Schreibweisen `5 * (-1)` und `5 * -1` immer noch zulässig, nicht aber `5 * -(1)`, `5 * - 1` oder auch `5 * (-x)`. Problematisch bei dieser Lösung ist allerdings, dass sie beim Parsen auch zu unintuitiven Ergebnissen führt. Beispielsweise würde man erwarten, dass der Ausdruck `x-2` (ohne Leerschritte) als Subtraktion geparst wird, also gleichbedeutend mit `x - 2`. Tatsächlich wird der Ausdruck aber als Applikation von x mit dem Argument -2 geparst, gleichbedeutend mit als `x (-2)`.

Import-Statements Um das Modulsystem nutzen zu können, muss es eine Möglichkeit geben, andere Module zu importieren. Dazu haben wir ein *Import*-Statement eingebaut, mit der folgenden Syntax:

`IMPORT` *Modulpfad* `AS` *Modulbezeichner*

Dabei ist der *Modulbezeichner* ein großgeschriebener SL-Bezeichner, der zur Qualifikation von importierten Bezeichnern verwendet wird. Der *Modulpfad* ist der Pfad der Moduldateien, ohne Dateiendung. Wenn das Modul etwa aus der Datei *testmodule.sl* im aktuellen Verzeichnis kompiliert wurde, so heißt der *Modulpfad* einfach `"testmodule"`. Beginnt der Modulpfad jedoch mit dem Präfix *std/*, so wird das Modul in der Standardbibliothek gesucht. Der *Modulpfad* für das Listenmodul in der Standardbibliothek ist also `"std/list"`.

Die Sprache stellt nur Syntax für solche qualifizierten Importe zur Verfügung. Der einzige unqualifizierte Import eines Moduls ist der implizite Import von *std/prelude*, der automatisch bei jeder Kompilierung vorgenommen wird.

Wir haben uns für das Design mit den qualifizierten Imports entschieden, da es bei komplexeren Projekten beim Verständnis des Codes stark hilft, wenn leicht nachvollzogen werden kann, aus welchem Modul eine bestimmte Funktion oder ein Typ stammt. Eine alternative Möglichkeit, dieses Ziel zu erreichen, wäre beim Import alle zu importierenden Bezeichner einzeln aufzulisten. Dadurch wären zwar Ausdrücke sauberer lesbar, allerdings müsste dann bei jeder Verwendung einer neuen Funktion diese mühsam in der Importzeile hinzugefügt werden. Außerdem lässt sich auch mit unserer Lösung ein ähnliches Ergebnis erreichen, wie das folgende Beispiel zeigt:

```
IMPORT "std/list" AS List
DEF length = List.length
```

Dies funktioniert allerdings nur für Funktionen, nicht für Typen oder Konstruktoren. Eine dahingehende Erweiterung, vielleicht auch eine entsprechende Kurzschreibweise, wäre für die Zukunft sinnvoll. Ebenfalls wünschenswert wäre die Möglichkeit, das Modul *std/prelude* qualifiziert zu importieren und dadurch den unqualifizierten Import zu überschreiben. Dies wäre etwa sinnvoll für Anwendungen, die viel mit Gleitkommazahlen arbeiten und darum lieber die Operatoren `+`, `-`, `*`, `/` aus *std/real* unqualifiziert verwenden würden. Es ist zwar jetzt schon möglich, so etwas zu schreiben wie `DEF a + b = a Real.+ b`, aber dadurch wird es unmöglich, die Addition aus der Prelude im gleichen Modul zu verwenden.

Externe Definitionen, externe Imports und externe Datentypen Bei der Implementation von Modulen, insbesondere des Moduls *std/prelude*, das Funktionen definiert, die vormals direkt in den Sprachkern eingebaut waren, wurden auch bestimmte Spracherweiterungen nötig. Für all diese Erweiterungen verwenden wir das gleiche Keyword `EXTERN`. Dies soll signalisieren, dass es sich dabei um Features handelt, die nur nötig sind, wenn man mit SL-„externem“, d.h. JavaScript-Code arbeitet.

Externe Definitionen erlauben den Aufruf von JavaScript-Code ohne Verwendung der DOM-Monade. Dadurch können Funktionen aus JavaScript verwendet werden, um die Primitiven von SL zu implementieren. Der Programmierer hat bei Verwendung von externen Definitionen selbst darauf zu achten, dass die Funktion keine Nebeneffekte hat und dass die Funktion wirklich den in der Signatur angegebenen Typ hat, ohne dass der Compiler dies prüfen kann. Die Syntax hierfür ist:

`DEF EXTERN` *Funtkions-/Operatorname* `= {|` *JavaScript-Code* `|}`

Dabei ist zu beachten, dass im Gegensatz zu normalen Funktions- bzw. Operatordefinitionen keine Argumente angegeben werden dürfen.

Externe Imports ermöglichen es, dem Compiler mitzuteilen, dass er den Inhalt einer beliebigen JavaScript-Datei bei der Codegenerierung vor die Ausgabe kopieren soll. Dadurch kann auf die in dieser Datei definierten Funktionen von JavaScript-Blöcken aus zugegriffen werden. Dies ist besonders hilfreich in Kombination mit externen Definitionen, wobei der tatsächliche Inhalt der JavaScript-Funktionen in der importierten Datei liegt und in der externen Definition lediglich deren Name angegeben ist. Das Modul *std/prelude* ist etwa auf diese Weise geschrieben. Die Syntax für externe Importe ist:

`IMPORT EXTERN` *Pfad*

Dabei ist *Pfad* wieder ein Zeichenkettenliteral; die Endung *.js* entfällt dabei.

Externe Datentypen sind Datentypen ohne Konstruktoren. Dadurch können etwa verschiedene Typen von JavaScript-Objekten dargestellt werden. Die Basistypen `Int` und `String` gehören etwa in diese Kategorie. Die Syntax lautet:

`DATA EXTERN` *Typname*

Sichtbarkeit Funktionssignaturen und Datentypendefinitionen kann jetzt das Keyword `PUBLIC` vorangestellt werden. Dadurch werden die entsprechenden Funktionen bzw. Konstruktoren exportiert, sind also in anderen Modulen sichtbar. Nicht als `PUBLIC` markierte Datentypen werden ebenfalls exportiert, allerdings ohne ihre Konstruktoren.

Eine alternative Notation für die Markierung von Exporten wäre ein explizites Export-Statement analog zum Import-Statement gewesen, in dem dann alle exportierten Bezeichner des Moduls aufgeführt werden. Uns hat die Annotation der Funktionssignaturen allerdings mehr zugesagt, da sie für exportierte, d.h. nach außen sichtbare Funktionen die Angabe einer Signatur erzwingt. Gerade bei nicht selbst geschriebenen Funktionen ist die Signatur elementar für das Verständnis, was der Zweck einer Funktion ist und wie sie verwendet werden kann; die Signatur anzugeben wäre in jedem Fall guter Stil. Das Keyword `PUBLIC` zu verwenden lag für uns nahe, da es in ähnlicher Funktion auch in anderen bekannten Sprachen wie Java oder C++ verwendet wird und seine Bedeutung dadurch eventuell intuitiver verständlich ist.

3.3 Qualifizierte Bezeichner

Da wir Module im Allgemeinen qualifiziert importieren, muss auf importierte Bezeichner auch qualifiziert zugegriffen werden. Die entsprechende Syntax lautet:

Modulbezeichner.Bezeichner

Der *Modulbezeichner* ist ein großgeschriebener Bezeichner, wie er auch für Typen und Konstruktoren verwendet wird. Der *Bezeichner* ist ein gewöhnlicher SL-Typ-, Konstruktor- oder Funktionsbezeichner. Namen sind immer nur einfach qualifiziert, da es in SL weder verschachtelte Modulnamen gibt, noch Datenstrukturen mit Feldnamen analog etwa zu Strukturen in C. Dadurch ist bei qualifizierten Bezeichnern immer auf den ersten ersten Blick ersichtlich, welcher Teil wofür steht.

Bei der Auswahl von Namen sollte bereits beim Schreiben eines Moduls darauf geachtet werden, dass diese später einmal qualifiziert verwendet werden. So könnte beispielsweise ein (imaginäres) Modul für die Verwendung von Dateien als `File` importiert werden und dann Typen wie `File.Name` oder `File.Handle` oder Funktionen wie `File.size` oder `File.owner` bereitstellen. Durch solch geschickte Auswahl von Namen kann der Nachteil des Overheads durch die obligatorische Qualifikation von Bezeichnern verringert oder gar ganz verhindert werden.

Nachdem die Erweiterung der Syntax um die Qualifikationen teilweise ein komplexeres Unterfangen war, können jetzt qualifizierte Bezeichner tatsächlich an allen Stellen stehen, an denen auch unqualifizierte Bezeichner stehen können, außer natürlich auf der linken Seite von Definitionen und als Typ- oder Patternvariablen. Während es zwischenzeitlich nötig war, qualifizierte Bezeichner etwa als Feldtypen bei der Definition von Datenstrukturen zu klammern, ist dies nun nicht mehr erforderlich.

3.4 Weitere Anpassungen der bestehenden Grammatik

Präzedenz in Funktionssignaturen Bisher wurde der Typ `List a -> b` als `List (a -> b)` interpretiert. Wir haben diese unintuitive Präzedenz geändert zu `(List a) -> b`. Dies entspricht nicht nur unserer Intuition, sondern auch etwa der Präzedenz in Haskell, von dem SL seine Typ-Syntax abgeleitet zu haben scheint.

Verbesserungen im Combinator-Parser Während bisher hauptsächlich der Parboiled-Parser verwendet wurde, war es unser Ziel, beide Parser gleichwertig immer auf dem aktuellen Stand der Entwicklung zu halten. Zu Beginn unserer Arbeiten fehlte dem Combinator-Parser noch die Unterstützung für einige Sprachfeatures, etwa die Definition von eigenen Operatoren oder das Parsen von Gleitkomma-Literalen. Auch unterschied sich das Verhalten der Parser in so manchem Detail, wie etwa der Behandlung von JavaScript-Blöcken mit Zeilenumbrüchen, oder dass er dem Operator `>` im gegensatz zu allen anderen Vergleichsoperatoren niedrigste Präzedenz gegeben hat. Wir haben nicht nur den

Combinator-Parser in an alle unsere Spracherweiterungen angepasst, sodass er gleichwertig mit dem Parboiled-Parser Programme korrekt parst, wir haben ihn sogar zu unserem primären Parser erhoben, da er beim Parsen jeder Produktion ein Attribut mit Dateiname sowie Zeilen- und Spaltennummer der Start- und Endposition zuweist. Dies hat sich bei der Ausgabe von sinnvollen Fehlermeldungen als unverzichtbar herausgestellt (siehe Abschnitt 6).

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 eineindeutig 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 (siehe Abb. 1 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 3 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“.

$$
\begin{array}{lll}
\mathit{Path} & ::= & (\mathit{Dir}\ /)^*\ \mathit{Module} \\
\mathit{Dir} & ::= & \mathit{char}^+ \\
\mathit{Module} & ::= & \mathit{char}^+ \\
\mathit{char} & ::= & \texttt{a} \mid \cdots \mid \texttt{z} \\
 & & \mid \texttt{0} \mid \cdots \mid \texttt{9} \\
 & & \mid - \mid \texttt{\_}
\end{array}
$$

Grammar 3: Gültige Importpfade

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
```

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 Kompiliervorgang nicht fortgesetzt werden. Die Suche der Dateien erfolgt relativ zum Klassenpfad, Zielverzeichnis, TODO (* was ist die 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
PUPBLIC 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 `0` 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 ELC[1] zu übersetzen. 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[2] und im Browser[3] ausgeführt werden kann. Bei einem Compileraufruf können jeweils mehrere Dateien angebenen 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

[1] Enriched Lambda Calculus
[2] http://nodejs.org/ - getestet mit Version 0.10.10
[3] getestet mit Firefox, Chrome und Internet Explorer

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.

Durch die Unterscheidung zwischen `<classpath directory>` und `<source directory>` kann dem Compiler sowohl eine Quelldatei als auch ein Kompilation eines Modules 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 8.2 mit Compileraufruf und den Ausgaben den 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 Signature-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 Bennennungschema 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 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 Anhängigkeiten erkannt und die Compilierung abgebrochen, sofern sie vorliegen (siehe `cycleA` und `cycleB` in Abschnitt 8.2).

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 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 enspricht 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 compiliert 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 compilierte Modul verfügt nach Abschluss des Compiler-Laufes über die entsprechenden Dateien. Dieses Problem kann umgangen werden, indem Module mit `main`-Methoden separat compiliert 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:

Typvariable die durch einen beliebigen Typen ersetzt werden kann.
Funktionstyp welcher 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 Parametertypen 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-Objekttyp 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[4] statt CommonJS[5] 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.

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"; },
```

[4] http://requirejs.org/ v. 2.1.6
[5] http://www.commonjs.org/

```
    "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 Kontruktion (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 8.2).

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-Konfiguationen 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-Konfiguation 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 Rückgabetyp 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 |}` $\rightsquigarrow$ `var bla = js;`

Um auf auf der rechten Seite dieser Definition bequem selbstdefinierte JavaScript-Funktionen anzugeben, bietete 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

Die Generierung von aussagekräftigen Fehlermeldungen ist unverzichtbar für das Kompilieren eines jeden Programms, das über den Umfang kleiner Beispiele hinausgeht. Hilfreich ist eine Fehlermeldung dann, wenn aus ihr möglichst deutlich hervorgeht, um welche Art von Fehler es sich handelt und an welcher Stelle der Fehler am besten zu beheben ist. Leider waren die Fehlermeldungen des SL-Compilers bisher kaum hilfreich. Weder stellten sie Informationen über den Ort des Fehlers bereit, noch war die Ausgabe in irgend einer konsistenten Weise Formatiert. Zu sehen bekam man meist unformatierte Scala-Objekte, die kryptische Strings als Fehlermeldungen enthielten, oder gar direkt eine Exception mit Stacktrace.

6.1 Lokalisierung

Um Fehler finden und beheben zu können, stellen wir wenn möglich jeder Fehlermeldung eine kurze Positionsangabe voraus, aus der Dateiname und der Bereich der Datei, in dem der Fehler aufgetreten ist, ersichtlich werden. Wir verwenden dazu das knappe Format *Dateiname*:*Zeilennummer*[:*Spaltennummer*], wobei die *Zeilen-* und *Spaltennummer* einfache Zahlen oder auch Bereiche der Form *Nummer-Nummer* sein können. Bei mehrzeiligen Fehlerbereichen entfällt die Angabe der Spalten. Eine solche Lokalisierung erreichen wir, indem wir den Combinator- statt des Parboiled-Parsers verwenden, wodurch wir jedem Knoten in unserem abstrakten Syntaxbaum ein Attribut mit einer Position zuordnen können.

6.2 Syntaktische Fehler

Die für den Combinator-Parser verwendete Parsec-Bibliothek enthält bereits eine rudimentäre Fehlerbehandlung, die aber für unsere Zwecke nicht genügte. Fehlern konnte zwar eine Zeile und eine Spalte zugeordnet werden, nicht aber ein ganzer Bereich, wie von uns vorgesehen, außerdem waren die Fehlermeldungen sehr kryptisch; auch erschien es uns unpassend, die ja bereits erstellten Attribute im Combinator-Parser nicht auch für Parserfehler zu verwenden. Unsere Herangehensweise an das Problem war, zusätzliche Produktionen für häufige Fehler in den Parser einzubauen, die dann allerdings statt normal ein Ergebnis zurückzuliefern über Scalas Exception-Behandlung einen Fehler mit Attribut und genauer Beschreibung des Fehlers werfen. Teilweise entstehen dabei allerdings immer noch Parsec-Fehler, die dann einfach in unser internes Fehlerformat umgebaut und ebenfalls als Excptions geworfen werden.

Im Ergebnis haben wir nun deutlich hilfreichere Parser-Fehlermeldungen als zuvor, sehen aber auch noch einige Schwächen. Es ist leider immer noch nicht möglich, mehr als einen Fehler im Parser zu erkennen, was bei mehreren Fehlern mehrmaliges Kompilieren notwendig macht. Auch dringen teilweise immer noch die kryptischen Fehlermeldungen der Parsec-Bibliothek bis in die Ausgabe des Compilers vor. Nützlich wäre wohl ein Redesign des gesamten Parsers, bei dem Fehlerbehandlung von vorn herein mitgedacht wird. Oft hätten wir uns auch für das Bereitstellen von sinnvollen Fehlern eine separate lexikalische Analysephase gewünscht, die bisher leider weder im Parboiled- noch im Combinator-Parser existiert.

6.3 Semantische Fehler

Der Typchecker produziert Fehlermeldungen, wenn er einen unbekannten Bezeichner findet, oder wenn bekannte Bezeichner mit dem falschen Typen verwendet werden. Während Fehlermeldungen bei unbekannten bezeichnern aussagekräftig sind und auf den richtigen Bereich im Quelltext verweisen, sind Typfehler leider oft schwer nachvollziehbar. Gerade durch die Fähigkeit von SL, Typen inferieren zu können, werden Fehler oft an ganz anderen Stellen vom Typchecker entdeckt, als dort, wo sie von der Programmiererin behoben werden müssen. Erschwert wird das Problem zusätzlich noch dadurch, dass momentan lediglich die unterschiedlichen Typen angegben werden, nicht jedoch woher diese Typen hergeleitet wurden.

Weitere semantische Fehler sind

- die mehrfache Vergabe von Funktions-, Typ- und Konstruktor- und Modulnamen,
- die mehrfache Vergabe des gleichen lokalen Variablen- oder Typvariablennamen,
- der mehrfache Import ein- und desselben Moduls,
- unterschiedliche Aritäten bei verschiedenen Definitionen einer Funktion oder ihrer Signatur,
- Signaturen ohne dazugehörige Definitionen und
- externe Definitionen ohne dazugehörige Signaturen.

All diese Fehler werden erkannt und entsprechend mitsamt Position im Quelltext ausgegeben.

6.4 Importfehler

Beim Importieren von Modulen müssen auch Dateien geöffnet werden. Dabei können Fehler auftreten, wenn etwa die Datei nicht existiert oder nicht gelesen werden kann. Ursachen dafür sind meist entweder eine falsche Angabe des Modulpfads im Import-Statement, oder dass ein Modul noch nicht kompiliert wurde. Darum werden in der Fehlermeldung sowohl die Position des Import-Statements als auch der Pfad der Datei angegeben.

Weitere Fehler, die beim Importieren auftreten, sind zyklische Abhängigkeiten von Modulen oder der Qualifizierte Import von *std/prelude*. Da wir diese Fälle bisher ausschließen, werden sie als Fehler erkannt und die Position des entsprechenden Import-Statements mit ausgegeben. In Zukunft wäre es natürlich wünschenswert diese Fehler dadurch auszuschlißen, dass die entsprechenden Features von SL unterstützt werden.

6.5 Laufzeitfehler

Zur Laufzeit auftretende JavaScript-Fehler werden von uns in keiner Weise behandelt. Beim Schreiben von Code, insbesondere bei der Verwendung von JavaScript-Blöcken oder `EXTERN`-Definitionen ist also darauf zu achten, dass möglicherweise auftretende JavaScript-Fehler frühzeitig abgefangen werden und dass die Typen tatsächlich mit den in der Signatur angegebenen Typen übereinstimmen.

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` 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[6] 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 Rückgabetyp, 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.[7] 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 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} |}
```

[6] Das ungewöhnliche Zeichen rührt daher, dass „`o`“ in SL kein Operator sein kann und „`.`“ für die Lambda-Abstraktion und Namensqualifizierung reserviert ist.

[7] Diese Funktion ist also keine echte, wohldefinierte Funktion, sondern hat dasselbe „Ergebnis” wie eine Endlosrekursion.

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 Reele Zahlen — `real.sl`

Am Anfang des Projekts hatten wir reele 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 Verwendeung 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 Expolorer (ab IE8, Developer Tools) unauffällige Programmausgaben ermöglicht.

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

```
IO.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-Funkionen sind jetzt gleichberechtigte Funktionen innerhalb der Sprache und führen kein Eigenleben in Checks und Codegenerierung mehr.

8 Beispielprogramme und Tests

Im Laufe der Entwicklung und insbesondere am Ende haben wir uns um ausgiebige Tests der neuen und alten Features bemüht. Einige der dabei entdeckten Schwierigkeiten führten auch tatsächlich noch zu Designänderungen.

Zum Beispiel traten beim Testen mehrmals Inkompatibilitäten zwischen Windows und Linux auf, die uns zu sehr restriktive Vorgaben für Import-Pfade (vgl. Abschnitt 4.1) brachten.

Leider erlaubte es unsere Teamgröße nicht, eine Person exklusiv mit Qualitätssicherung zu betrauen. Dennoch stimmt uns der Umfang an stabil funktionierenden Tests und Beispielprogrammen zuversichtlich, dass unsere Implementierung eine gewisse Robustheit erreicht hat. Nachfolgend soll kurz ein Überblick über

8.1 Unittests

Die ursprüngliche SL-Implementierung kam mit vielen Unittests daher. Angesichts der weitreichenden Änderungen, die besonders die qualifizierten Bezeichner im bestehenden Code bedeuteten, waren diese Tests uns eine große Hilfe beim konsistenten Einpflegen neuer Features. Wir mussten im Laufe der Entwicklung nahezu alle Unittests anpassen und haben auch viele erweitert.

Eine 100%-ige Abdeckung der neuen Modulfeatures haben wir dabei nicht erreicht. Das Zusammenspiel von diamond-haften Modulimporten, bestimmte Namenskonflikte und Modulabhängigkeiten ließen sich besser durch den Einsatz des Modulsystems bei der Entwicklung kleinerer Programme überprüfen.

8.2 Beispielprogramme

Im Produktumfang sind auch eine Reihe von `.sl`-Dateien enthalten, die zur Dokumentation und zum Test der Funktionsweise von SL dienen. Die Programme liefen in aktuellen Versionen von Firefox und Chrome sowie unter Node.js (Die `basicweb`-Abhängigen Programme funktionieren natürlich nicht unter Node.js). Wir haben sowohl Kompilierung als auch Ausführung unter Linux und Windows getestet.

`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.

`sub/hello.sl` Gibt „Hello Moon!" aus und wird von `moonWorld.sl` verwendet, um das Einbinden verschieden qualifizierter Module mit identischem einfachen Namen zu testen.

`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` enhalten eine direkte zyklische Abhängigkeit, um zu testen, ob diese erkannt wird. Sie bilden damit ausnahmsweise kein Beispielprogramm, sondern eben ein Nicht-Programm.

`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. Setzt alle möglichen Features aus den `std`-Librarys ein.

`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` enhalten eine direkte zyklische Abhängigkeit, um zu testen, ob diese erkannt wird.

9 Zusammenfassung und Ausblick

Wir haben viel geschafft, aber ganz fertig ist SL wohl noch nicht.

9.1 Was erreicht wurde

In der aktuellen Version von SL existiert ein überschaubares und stabiles Modulsystem. Dazu erweiterten wir den Sprachumfang um qualifizierte Bezeichner, Modulimporte und Exportdeklarationen.

Mithilfe des Modulsystems und der neuen Einbindungsmöglichkeiten für externen Code, können auf JavaScript aufsetzende Bibliotheken geschrieben werden. Insbesondere konnten wir uns so der in den Compiler fest eingebauten Basisfunktionen entledigen und diese in einem gesonderten Prelude-Modul zusammenfassen.

Der Compiler gibt seine Ergebnisse nicht mehr bloß in der Konsole aus, sondern erzeugt separat kompilierte Module und Signaturen, die mithilfe von requirejs zur Laufzeit zusammengefasst werden können. Der Compiler übersetzt eigenständig auch benötigte Module und liefert das Erzeugte zusammen mit zur Ausführung benötigten Rahmendateien und einer `index.html` aus. Das Kompilat lässt sich mit gängigen modernen Browsern und node.js ausführen.

Bei einem Fehlschlag des Compile-Vorgangs geben die Fehlermeldungen jetzt, wenn möglich, auch genaue Stellen des Fehlers an.

Die neuen Features wurden getestet und auch in komplexer zusammenarbeitenden Modulen eingesetzt. Während der Entwicklung konnten wir auch viele kleine Fehler in SL fixen.

Insgesamt haben wir damit die am Anfang des Projekts im Pflichtenheft festgelegten Kriterien erfüllt und beispielsweise in Hinblick auf die Entwicklung von Bibliotheken auch übertroffen. Dennoch drängen sich einige Punkte auf, an denen die Entwicklung von SL fortgesetzt werden könnte.

9.2 Was noch erreicht werden kann

Aktuell müssen Importe qualifiziert sein. Auch wenn in unserem Team die Auffassung vorherrscht, dass das für wartbaren Sourcecode ohnehin der bessere Weg ist, könnten vielseitigere Import-Statements angestrebt werden zum Beispiel für unqualifizierte, selektive oder umbenennden Importe. Es könnte auch über ein Paketsystem mit paketweisem Import und gegebenenfalls hierarchischen qualifizierten Bezeichnern nachgedacht werden.

Die derzeitige Implementierung verbietet zyklische Abhängikeiten zwischen Modulen. SLs Design würde jedoch erlauben, solche Zyklen durch einen mehrschrittigen Kompilierungsprozess aufzulösen. Darüberhinaus wäre es auch möglich, Module nicht nur ihre exportierten, sondern auch ihre benötigten Schnittstellen angeben zu lassen und erst zur Ausführung oder Distribution die angebotenen und benötigten Schnittstellen zusammenzuführen.

Allgemein erwies sich die konsistente Ausgabe des Kompilats an einem geeigneten Ort als schwieriger als zuerst angenommen. Für eine sinnvoll automatisierbare Distribution müssten mehr Konfigurationen zur Einbettung der Ausgaben in HTML-Code und zur Konfiguration von requirejs am Ausgabeort angeboten werden.

Wir konnten zwar deutliche Fortschritte bei den Fehlermeldungen verbuchen. Wie die meisten Systeme mit Typinferenz, kann SL sich aber nicht immer so ausdrücken, dass Leute, die das Typsystem nicht selbst entwickelten, etwas mit den Fehlermeldungen anfangen könnten. Hier wäre noch viel Arbeit nötig.

Und zu guter letzt wären natürlich mehr und ausgereiftere Bibliotheken schön.

Insgesamt bleiben also auch an den von uns bearbeiteten Features von SL noch genug Baustellen übrig für das nächste Compilerbauprojekt oder die nächste Abschlussarbeit. Und das muss ja auch sein.