Pflichtenheft SL2

Spracherweiterung der funktionalen Sprache SL (Simple Language)

1 Zielbestimmung

SL (Simple Language) ist eine einfache funktionale Programmiersprache. Sie wurde an der TU Berlin zur Lehrzwecken entwickelt. Der Sprachumfang soll in diesem Projekt um die folgenden Kriterien erweitert werden.

1.1. Musskriterien

1.1.1 Modulsystem

1. Einzelne SL-Quelldateien können als eigenständige Module separat übersetzt werden.
2. Eine SL-Datei deklariert implizit ein Modul.
3. Die Übersetzung der Quelldatei eines Moduls erzeugt eine Signatur- und eine JavaScript-Datei.
4. Übersetzte Module können von anderen Modulen importiert werden. Dabei wird ein Modulname vergeben, der innerhalb des importierenden Moduls gültig ist.
5. Modifizierte und neue Produktionen in der Syntax für Import-Statements:
   Program `::=` *Import* Def+*
   Import `::= IMPORT string AS module`
   `module ::= uppercase alphanum*`
6. Ein Modul wird über einen Pfadstring im Import-Statement importiert. Der Pfad wird relativ zum Klassenpfad, der dem Compiler übergeben wird, interpretiert. Als Dateitrenner wird unabhängig vom Betriebssystem, auf dem der Compiler oder der JavaScript-Interpreter ausgeführt werden, ein Slash (`/`) – kein Backslash (\) – verwendet. Das letzte Pfadelement entspricht dem unqualifizierten Namen der Datei, die das Modul implizit deklariert, ohne die Endung `.sl` (klein geschrieben).
7. Zwei Module sind identisch, wenn sie mit Import-Statements mit identischen Pfadstrings importiert wurden.
8. Wird Modul A von Modul B importiert, wird bei der Compilierung der Quelldatei von Modul B die Signaturdatei von Modul A verwendet, um eine Typkontrolle durchzuführen. Alle in Modul A deklarierten Funktionen, Datentypen und Operatoren werden durch den Import in Modul B verfügbar.
9. Ein Bezeichner, mit dem auf eine Funktion, eine Datentyp oder einen Operator aus einem importierten Modul zugegriffen wird, muss durch den im Import-Statement vergebenen Modulnamen qualifiziert werden.
10. Mit einem Compileraufruf kann eine einzige SL-Datei kompiliert werden.
11. Zirkuläre Abhängigkeiten zwischen Modulen (Modul A importiert Modul B und B importiert A) werden nicht unterstützt.
12. In den JavaScript-Dateien werden Module mittels requireJS realisiert.

1.1.2 SL-Prelude

1. Das bestehende SL-Prelude wird als Modul umgesetzt.
2. Alle SL-Quelldateien importieren dieses Modul implizit.
3. Bezeichner dieses Moduls sind in SL unqualifiziert.

1.1.3 JavaScript

1. Alle vom Compiler generierten JavaScript-Dateien, können sowohl im Browser als auch in Node.js ausgeführt werden.
2. Zum Aufruf eines Moduls im Browser wird jeweils eine HTML-Datei generiert, die die JavaScript-Dateien lädt.
3. Es wird die Ausführung unter node.js, Firefox, Chrome und Internet Explorer unterstützt.

1.1.4 Operatoren

1. Das unäre Minus wird entfernt.
2. Negative Zahlen gelten als eigenständige Literale, d.h. sie sind nicht aus einer positiven Zahl und unärem Minus zusammengesetzt.
3. Die Schreibweisen `-(foo bar)`, `-x`, `-(1)` und `- 1` sind nicht gestattet.

1.1.5 Fehlermeldungen

1. Die verschiedenen Verarbeitungsschritte des Compilers erzeugen hilfreiche Fehlermeldungen.
2. Die implementierten Parser (ParboiledParser und CombinatorParser) erkennen möglichst alle Syntaxfehler.
3. Zu Fehlermeldungen wird, falls möglich, die betreffende Zeile und Spalte des SL-Codes angezeigt.

1.2. Wunschkriterien

1.2.1 Modulsystem

1. Noch nicht übersetzte Module können in anderen Modulen eingebunden werden.
2. Mehrere Module können in einem Kompilationsprozess übersetzt werden.

1.2.2 Operatoren

1. Operatoren können partiell angewandt werden. D.h. man kann Operatoren an Higher-Order-Functions übergeben.
2. Fehlermeldungen können einfach lokalisiert werden.

1.2.3 Fehlermeldungen

1. Die betreffende Codezeile wird in der Ausgabe angezeigt und markiert.

1.3. Abgrenzungskriterien

1.3.1 Modulsystem

1. Es wird keine weltweit eindeutige Bezeichnung von Modulen angestrebt. Sie kann jedoch individuell z.B. durch das bei Java gängige Schema der Präfigierung mit dem umgekehrten Domainnamen des Entwicklers zu Beginn der relativen Modulpfade umgesetzt werden.
2. Es wird kein Versionierungsschema für Module vorgegeben. Es werden keine Mechanismen implementiert, um Modulversionen automatisch zu vergleichen und auf ihre Binärkompatibilität zu schließen.

3. Zur Laufzeit wird nicht überprüft, ob die geladene JavaScript-Datei eines Moduls binärkompatibel zu der bei der Kompilierung verwendeten Signaturdatei des Moduls ist. Das Verhalten des Systems in diesem Fall wird nicht definiert.

1.3.2 Fehlermeldungen

1. Sofern zur Laufzeit ein Fehler auftritt, der darauf zurückzuführen ist, dass die JavaScript-Datei eines Moduls nicht binärkompatibel zur bei der Kompilierung verwendeten Signaturdatei ist, wird nicht versucht, eine hilfreiche Fehlermeldung auszugeben, die auf diesen Umstand als Problemursache hinweist.

2 Produkteinsatz

Die Spracherweiterung soll prinzipiell den Produkteinsatz nicht verändert. Dennoch wird damit ein verbesserter Einsatz der Sprache bezweckt.

2.1 Anwendungsbereich

Trotz der des übergeordneten Lehrzweckes der Sprache SL, liegt der Anwendungsbereich innerhalb der Webentwicklung. Es soll für die etablierten Browser eine Möglichkeit geschaffen werden, Programme mit funktionalem Ursprung ausführen zu können. JavaScript erfüllt diesen Zweck nicht, da Typfehler erst zur Laufzeit erkannt werden können. Da jedoch de facto alle Browser JS unterstützen, wird die hier entwickelte funktionale Sprache SL in JS übersetzt. Somit wird Programmierern eine funktionale Sprache mit statischer Typsicherheit für die Webentwicklung zur Verfügung gestellt.

2.2 Zielgruppen

Nutzer sind Programmierer, die in der Webentwicklung tätig sind. Dies kann allgemein auch auf Personen ausgeweitet werden, die bislang Programme für JS-Plattformen entwickelt haben.
Dies betrifft vor allem solche, die auf eine funktionale Sprache mit statischer Typsicherheit zurückgreifen wollen, um zuverlässigere Programme entwickeln zu können.
Die Spracherweiterung soll demnach eine modulare Entwicklung begünstigen. Daher kommen für den Entwickler ein einfacherer Kompilierungsprozess und die Möglichkeit Module getrennt zu entwickeln, sowie auszutauschen.

2.3 Betriebsbedingungen

Die Sprache SL und der Compiler kommen selbst nur beim Entwickler zum Einsatz. Endnutzer erhalten lediglich das übersetzte JS-Kompilat. Eingesetzt wird der in JS übersetzte SL-Code vor allem in Webseiten und damit Browsern. Dennoch sollen Programme auch prinzipiell in anderen JS-Umgebungen wie node.js ausführbar sein.

3 Produktumgebung

Das Produkt im Sinne der Spracherweiterung ist die Sprache SL, die durch den Compiler umgesetzt wird.

- Compiler
    - Der Compiler wird erweitert, um die spezifizierte Spracherweiterung umzusetzen
    - Die verwendete Sprache ist Scala. Damit wird eine JVM zur Ausführung vorausgesetzt.
- SL-Programm

    - Zur Ausführung von übersetzten SL-Programmen wird eine JS-Laufzeitumgebung benötigt.

4 Produktfunktionen

Die Produktfunktion im Sinne des Compilers als Produkt enthält nur eine Funktion:

- Das Kompilieren einer SL-Datei zu einem Modul bestehend aus Signatur und JS-Kompilat.

5 Produktdaten

Im Sinne des SL-Programmierers als Nutzer, werden folgende Daten gespeichert.

- Die SL-Quelldatei, auch Modul genannt.
- Das Kompilat eines SL-Moduls bestehend aus:
    - Signatur-Datei: Funktionssignaturen, Datendefinitionen, Importe
    - JavaScript-Dateien: enthält die Semantik des SL-Codes in JavaScript
    - HTML-Datei: ermöglicht die Ausführung der JavaScript-Datei im Browser

6 Globale Testszenarien

- Ein zu importierendes Modul fehlt.
- Importierte Bezeichner werden falsch verwendet (i.S.d. Typüberprüfung)
- Ein Modul wird doppelt importiert.
- Zwei Module verwenden denselben qualifizierten Bezeichner.
- Es wird eine importierte Funktion verwendet, die unbekannte Datentypen verwendet.
- Es wird ein importierter Datentyp mit unbekannten Datentypen verwendet.
- Falls mehrere Module kompiliert werden: Erkennung von zyklischen Abhängigkeiten.
- Problematische Modulpfade ((in)sensitive Pfade unter Linux/Windows)
- Eine Browser-Anwendung importiert mehrere unabhängig voneinander kompilierte Module.