Softwareparadigmen SS 2018, Übungsblatt P

Allgemeines

Abgabe: 13. Juni 2018, bis 18:00 Uhr

1. Die Abgabe erfolgt über das SVN Ihrer Gruppe, im Ordner `SWPInterpreter`.
2. Beurteilt wird der letzte Commit vor der Deadline. Spätabgaben werden nicht akzeptiert!

Aufbau des Repositories

Auf der Lehrveranstaltungsseite finden Sie eine Framework-Vorlage `Framework.zip`; entpacken Sie diese in den Root-Ordner.

Ihre Lösungen werden automatisiert getestet, halten Sie sich daher bitte an folgende Ordnerstruktur, und stellen Sie sicher, dass ihr Projekt mit SBT compiliert, ausgeführt und getestet werden kann.

Vorlage für die Ordnerstruktur:

```
<Repository-URL>/SWPInterpreter/build.sbt
<Repository-URL>/SWPInterpreter/src/main/scala/AST.scala
<Repository-URL>/SWPInterpreter/src/main/scala/Value.scala
<Repository-URL>/SWPInterpreter/src/main/scala/Interpreter.scala
<Repository-URL>/SWPInterpreter/src/main/scala/Parser.scala
<Repository-URL>/SWPInterpreter/src/main/scala/SWPInterpreter.scala
<Repository-URL>/SWPInterpreter/src/test/scala/SWPInterpreterTests.scala
```

Die Binärdaten in `SWPInterpreter/target` und `SWPInterpreter/project/target` sollen nicht mit eingecheckt werden.

Framework

In der Framework-Vorlage befindet sich ein SBT-Project "SWPInterpreter", definiert durch `build.sbt`, sowie unter `src/main/scala` ein Grundgerüst für die Aufgabe.

Um das Projekt zu starten, müssen Sie zuerst SBT installieren; dann kann die SBT-Konsole gestartet werden, indem sie ein Terminal im Projektverzeichnis (in dem sich `build.sbt` befindet) öffnen und `sbt` eingeben.

Zum Kompilieren geben Sie `compile` ein. Das erste Kompilieren kann etwas länger dauern, da zuerst Scala und die notwendigen Libraries heruntergeladen und installiert werden müssen (das heißt, es ist nicht notwendig, einen Scala-Compiler herunterzuladen – alles wird von SBT lokal verwaltet).

Um ein Programm in der Datei `test.exp` zu interpretieren, geben sie `run test.exp` ein. Diese Datei können sie modifizieren, um mit ihrer Implementierung zu experimentieren.

Tests

In der Datei `src/test/scala/SWPInterpreterTests` finden Sie einige einfache Testcases. Diese decken nicht alle Funktionalitäten ab, sondern sollen zur Orientierung dienen.

Sie können weitere Tests hinzufügen, um ihre Implementierung zu testen. Diese können auch gerne über die Newsgroup oder auf anderem Weg mit anderen Studierenden geteilt werden.

Die Tests können wieder über die SBT-Konsole mit dem Kommando `test` ausgeführt werden. Im leeren Framework:

```
> test
[info] SWPInterpreterTests:
[info] - Parser minimal example *** FAILED ***
[info] scala.NotImplementedError: an implementation is missing
...
```

Vollständig Implementiert:

```
> test
[info] SWPInterpreterTests:
[info] - Parser minimal example
...
[info] Tests: succeeded 7, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[success] Total time: 7 s, completed Apr 20, 2016 3:50:47 PM
```

Aufbau der Aufgaben

Dieses Aufgabenblatt beinhaltet zwei Kategorien von Aufgaben: Die Grundfunktionalität und die erweiterte Funktionalität. Zuerst sind alle Aufgaben aus der ersten Kategorie zu implementieren. Diese bauen direkt aufeinander auf und sollten daher auch in der angegebene Reihenfolge erledigt werden.

Wenn Sie die Grundfunktionalität implementiert haben, können Sie entscheiden, welche Aufgaben der erweiterten Funktionalität Sie implementieren möchten. Diese Aufgaben sind unabhängig voneinander und können alle auf der Grundfunktionalität aufbauend implementiert werden.

Insgesamt können für dieses Aufgabenblatt bis zu 55 Punkte erreicht werden. Sie müssen also nicht alle Aufgaben implementieren, um die maximale Punktezahl zu erhalten.

Grundfunktionalität

1 Parser (15 P.)

Implementieren Sie einen Parser der folgende an EXP angelehnte (aber modifizierte) Sprache parsen kann. Definieren Sie dazu in `AST.scala` passende Case-Klassen, um die Syntax zu repräsentieren, und schreiben sie den dazu passenden Parser mit der Scala Parser Combinator Library in `Parser.scala`.

Die Grammatik ist gegeben als EBNF (Extended Backus–Naur Form). Es werden folgende Konstrukte verwendet:

`[ ]` : 0 oder einmal

`{ }` : 0 bis n-mal

`|` : Alternative

`" "` : Token

`/ /` : Interpretation als Regex

Hinweise:

- Implementieren Sie die Tokens `id` und `int` auch tatsächlich als Scala-Regex.
- Sie können alle in der Parser-Library vorhandenen Hilfsfunktionen verwenden. Besonders hilfreich sind: `opt`, `rep`, `rep1`, `repsep` und `rep1sep`.

Grammatik

```
program = { function, ";" }, expr

function = "fun", id, "(", [ { id, "," }, id ], ")", "=", expr

braces = "(", expr, ")"

expr = block | recordDef | recordAccess | varDecl | varAssign |
       cond | call | list | variable | int | bool | braces

block = "{", { expr, ";" }, "}"

varDecl = "$", id, "=", expr;

varAssign = id, "=", expr;

cond = "if", expr, "then", expr, "else", expr

call = id, "(", [ { expr, ", "}, expr ], ")"
```

```
list = "[", [ { expr, ", "}, expr ], "]"

variable = id

bool = "True" | "False"

id = /(a-z)(A-Z|a-z|0-9|?|_)*/

int = /0 | (1-9)(0-9)* | -(1-9)(0-9)*/

recordDef = "object", "{", { recordValueDecl, ";" }, "}"

recordValueDecl = "$", id, "=", expr

recordRef = block | call | variable | recordDef | braces

recordAccess = recordRef, ".", id
```

Ein Beispiel:

```
fun getResult(list) = {
  $var = first(list);
  if eq?(var, 0) then {
    $var = build(1, list);
    list = if eq?([], rest(list)) then [1, 2, 3] else var;
  } else
    var = list = [];
};
getResult([0, 2, 4])
```

Hinweise: Erstens werden zur besseren Lesbarkeit im Folgenden Ausdrücke aus EXP in `Festbreitenschrift`, Ausdrücke aus dem Wertebereich der Interpretationsfunktion (also Scala-Expressions) in serifenloser Schrift, und metasprachliche Ausdrücke in normaler Schrift gesetzt, mit $\gamma\rho\iota\epsilon\chi\iota\sigma\chi\epsilon\nu$ Buchstaben für Metavariablen.

Zweitens: die Sprache verfügt über kein Typsystem. Es geht bei der Interpretation auch nicht darum, fehlerhafte Programme zu entdecken, sondern korrekte Programme korrekt zu evaluieren. Deshalb ist das Verhalten für "sinnlose" Programme (zB. die Addition einer Liste und einer Funktion) unspezifiziert und muss nicht besonders behandelt werden; insbesondere sind Exceptions OK. Wenn sie den Exceptions Task implementieren, müssen Sie jedoch die dort spezifizierten Fehler entsprechend behandeln.

2 Primitive Werte und Konditionale (5 P.)

Implementieren Sie zuerst einen Teil des Interpreters, der eine Untermenge der Sprache, die Konditionale und folgende primitive Funktionen beherrscht (korrenspondierend zu EXP mit Listen und Integers, bzw. die Grammatik ohne `fun_expr` und `let_expr`).

Verwenden Sie dazu den Typ `ExpValue` sowie die Funktion `interpret` in `Interpreter.scala`, unter Zuhilfename eines selbst definierten AST-Typs in `AST.scala`. Das Resultat der Interpretation soll ein Wert vom vordefinierten Typ `ExpValue` sein. Dieser `trait` und die davon abgeleiteten `case classes` dürfen nicht verändert werden, da wir sie zum Testen verwenden!

Builtin Functions sollen in einem vordefinierten Environment `builtinFunctions` definiert werden (es gibt dazu ein Beispiel im Code). Zum Debuggen können Sie den dazugehörigen Pretty-Printer in `AST.scala` verwenden.

Konstanten

- `True`, `False`
- Integer
- Listen
- (Strings)

Eingebaute Funktionen

Folgende Funktionen, mit der üblichen (call-by-value) Semantik:

Generisch:

- `eq?(a, b)` $\rightarrow$ a = b

Arithmetik:

- `add(a,b)` $\rightarrow$ a + b
- `sub(a,b)` $\rightarrow$ b − b
- `lt?(a, b)` $\rightarrow$ a < b

Listen:

- `first(xs)` $\rightarrow$ xs.head
- `rest(xs)` $\rightarrow$ xs.tail
- `build(x, xs)` $\rightarrow$ x :: xs

Logik:

- `and(a, b)` $\rightarrow$ a ∧ b
- `or(a, b)` $\rightarrow$ a ∨ b
- `not(a)` $\rightarrow$ ¬a

Konditionale

Verwenden Sie folgende Semantikdefinition (im Unterschied zum Skriptum sind Booleans hier first-class, und die Condition daher eine normale Expression):

$$I(\omega, \texttt{if } \pi \texttt{ them } \eta_1 \texttt{ else } \eta_2) = \begin{cases} I(\omega, \eta_1) & \text{wenn } I(\omega, \pi) = \mathsf{true} \\ I(\omega, \eta_2) & \text{wenn } I(\omega, \pi) = \mathsf{false} \end{cases}$$

3 Funktionen (5 P.)

Im Laufe dieses Tasks erweitern Sie den Interpreter um benutzerdefinierte Funktionen. Die Funktionen sollen wie im Skriptum auf Seite 59 beschrieben mit einer call-by-value Semantik interpretiert werden. Das Verhalten von Funktionen mit gleichem Namen ist undefiniert.

4 Blöcke und Variablen (5 P.)

Die zu implementierende Sprache enthält einen imperativen Teil, welcher durch Blöcke und Variablen realisiert wird. Ein Block beginnt mit `{` und endet mit `}`, dazwischen befinden sich mehrere Expressions, die mit `;` abgeschlossen werden.

In einem Block können Variablen deklariert (`$var = 10`), verändert (`var = 11`) sowie gelesen (`add(var, 1)`) werden. Variablen aus äußeren Blöcken können redeklariert, also "geshadowed"werden, wobei die Variable im äußeren Block ihren Wert behält. Nach Verlassen eines Blocks werden die darin deklarierten Variablen verworfen. Beim Aufruf einer Funktion wird für diesen Funktionsaufruf ein eigenes Environment erstellt, welches nur die übergebenen Argumente als Werte enthält.

Ein Block evaluiert zum Wert seiner letzten Expression. Ein leerer Block evaluiert zu 0. Zuweisung und Deklaration von Variablen evaluieren zum zugewiesenen Wert. Das Verhalten ist undefiniert, wenn eine Variable im selben Block redeklariert wird, auf eine undefinierte Variable zugegriffen wird und wenn eine Variable wie eine Funktion heißt.

5 Records (5 P.)

Ein Record ist eine Datenstruktur, die benannte Werte speichert, auf die über ihren Namen zugegriffen werden kann. Records können mit `object` gefolgt von der Zuweisung der einzelnen Felder instanziiert werden. Der Zugriff auf gespeicherte erfolgt mit dem Punkt Operator gefolgt von dem Namen des Wertes. Wenn der Record ein Feld mit diesem Namen besitzt, wird der dort gespeicherte Wert zurückgegeben, ansonsten ist das Verhalten undefiniert.

Beispiel:

```
{ $record = object {$a = 1; $b = 2;}; record.a; }
```

Erweiterte Funktionalität

6 Strings und I/O (5 P.)

Implementierern Sie Strings sowie eine `print` und `read` Funktion. Erweitern Sie dazu ihren Parser sowie Ihren Interpreter. Ein Stringliteral wird von `"` eingeschlossen und muss die druckbaren ASCII-Zeichen (`0x20 - 0x7e`), ausgenommen `"` (`0x22`), `\` (`0x5c`), `^` (`0x5e`) und `‘` (`0x60`) enthalten können. Zum Beispiel: `"1: Hello World!"`

Um interaktive Programme zu ermöglichen, sollen die Funktionen `print` und `read` implementiert werden. `print` besitzt einen String Parameter und evaluiert zu diesem, als Seiteneffekt wird der Text auf der Konsole ausgeben. `read` besitzt eine leere Parameterliste. Beim Aufruf blockiert das Programm, bis eine Zeile über die Konsole eingelesen wird, die von der Funktion zurückgegeben wird.

Beispiel:

```
{print("Enter your message: "); $in = read(); print("You entered: "); print(in);}
```

Verwenden Sie unbedingt die Mock-Funktionen `reader` und `writer` aus `Interpreter.scala`, da wir Ihre Implementierung sonst nicht testen können.

7 Kommentare (5 P.)

Es sollen einzeilige Kommentare und mehrzeilige Blockkommentare implementiert werden. Ein einzeiliger Kommentar beginnt mit einer `#` und geht bis ans Ende der Zeile. Ein Blockkommentar beginnt mit `#*` und geht bis zum nächsten `*#`, er kann sich auch über mehrere Zeilen erstrecken.

Beispiel:

```
fun getResult(list) = {
  $var = first(list); # Dies ist ein Kommentar
  if eq?(var, 0) then {
    $var = build(1, list);
    list = if empty(rest(list)) then [1, 2, 3] else var;
  } else #* Ein weiterer Kommentar *# {
    var = list = [];
  };
  #*
    Ein mehrzeiliger
    Kommentar
  *#
};
getResult([0, 2, 4])
```

8 Record Assignment (10 P.)

Es soll auch möglich sein, Felder in einem Record zu verändern. Felder eines Records sollen sich wie Variablen verhalten. Mit `record.field = newvalue` kann der Wert in einem Feld geändert werden. Es soll auch möglich sein, neue Felder zu einem bestehenden Record hinzuzufügen: `record.$newfield = newvalue`. Beide neuen Assignments evaluieren zum zugewiesenen `newvalue` und ändern den Record als Seiteneffekt, wobei Records by-reference übergeben werden.

Die linke Seite der Zuweisung kann auch geschachtelt sein: `record.recordField.field = newvalue`.

9 Operatoren (5 P.)

Erweitern Sie Ihren Parser und Interpreter um folgende Infix-Operatoren:

- `a + b` → $a + b$
- `a - b` → $a - b$
- `a * b` → $a * b$
- `a / b` → a/b
- `a == b` → $a = b$
- `a != b` → $a \neq b$
- `a && b` → $a \wedge b$
- `a || b` → $a \vee b$

Dabei gelten folgende Regeln: `*` / `/` binden stärker als `+` / `-` und `&&` bindet stärker als `||`. Alle Operatoren sind links-assoziativ.

Die Boolschen Operatoren `&&` und `||` sollen Short-Circuit-Evaluation verwenden.

10 Exceptions (10 P.)

Erweitern Sie den Interpreter um Exceptions. Dazu wird die Grammatik um folgende Regeln ergänzt:

```
expr = ... | throw | tryCatch

throw = "throw", exceptionId;

tryCatch = "try", expr, "catch", "{", [ { handler, ";" }, handler ], "}";

handler = (exceptionId | "_"), ":", expr;

exceptionId = letterC, { letter | letterC }
```

Semantik

Wenn eine throw Expression evaluiert wird, soll die weitere Evaluierung unterbrochen werden und die entsprechende Expression wird zurückgeliefert. Eine Exception wird solange nach oben durchgereicht, bis sie eine try-catch Expression erreicht, oder man ganz oben ankommt.

Im Falle eines try-catch wird geprüft, ob die Exception in der Liste der Handler vorkommt, wenn ja wird die zugehörige Expression ausgewertet. Underscore stellt ein catch-all Statement da. Wenn keine spezifischen Exceptionhandler vorhanden sind, wird diese Expression ausgewertet. Sollte der richtige Exceptionhandler nicht vorhanden sein und auch kein catch-all, wird die Exception weiter geworfen, muss also auf einer höheren Ebene behandelt werden.

Wenn ein Programm mit einer unbehandelten Exception endet, soll der entsprechende Outputtext generiert werden "Uncaught exception ID!", wobei ID dem Exception Identifier entspricht.

Zusätzlich zu den User definierten Exceptions sollen auch die eingebauten Funktionen und Operationen um entsprechende Exceptions ergänzt werden:

- TypeMismatch : Diese Exception wird immer ausgelöst wenn Funktionen/Prädikate auf inkompatiblen Typen aufgerufen werden. z.B. `first(1)`, `eq?(0, [])`, `sub([1],[2])`
- EmptyList : Wenn first oder rest auf eine leere Liste angewandt wird. z.B. `fist([])`, `rest([])`
- FieldNotFound : Wenn ein Record Feld nicht existiert. z.B. `(object {$a = 1}).b`

11 Eigener Vorschlag (*bis zu* 10 P.)

Sie können eigene Vorschläge für Erweiterungen des Programms machen, diese müssen Sie im Vorhinein mit den Tutoren besprechen. Wir teilen Ihnen dann mit ob das Thema für die Lehrveranstaltung geeignet ist und wie viele Punkte Sie dafür erhalten. Erstellen Sie dazu eine Datei `bonus.txt` im Root-Verzeichnis des Repositories und beschreiben Sie dort kurz, was Sie implementiert haben. Schreiben Sie zusätzlich mindestens einen Testcase, der zeigt, dass Ihre Erweiterung auch funktioniert.

Viel Erfolg!