Aufgabe A4.5

AST und wichtige Informationen

Für die weitere Auswertung der Programme wird ein abstrakter Syntaxbaum (AST) benutzt. Im Unterschied zum Parse-Tree lässt der AST alle rein syntaktischen Details weg (z.B. Klammern, Hilfsknoten wie `ListExpr`) und enthält nur noch die Informationen, die für die Bedeutung des Programms wichtig sind.

Im AST müssen unter anderem folgende Dinge aus dem Eingabeprogramm repräsentiert werden:

- die Folge der Top-Level-Ausdrücke (ein `Program`-Knoten),
- Literalwerte: Integer, Strings und Booleans,
- Variablen- und Funktionsnamen,
- Funktionsdefinitionen mit Parameterliste und Funktionskörper,
- Funktions- und Operatoraufrufe mit ihren Argumenten,
- `if`-Ausdrücke mit Bedingung, then- und optionalem else-Zweig,
- `let`-Ausdrücke mit ihren lokalen Bindungen,
- `do`-Blöcke als einfache Folge von Ausdrücken.

Java-Datenstrukturen für den AST

Der AST wurde in Java als kleine Klassenhierarchie umgesetzt. (um es zu starten, einfach: **"java AstTest.java"** im cmd. Bei IntelliJ wird die main methode nicht direkt erkannt) Es gibt eine Basisklasse `AstNode` und für alle Ausdrücke eine Klasse `Expr`. Darauf aufbauend wurden konkrete Klassen für die verschiedenen Sprachkonstrukte definiert. Ein Ausschnitt:

```
abstract class AstNode { }
abstract class Expr extends AstNode { }

class Program extends AstNode {
    private final List<Expr> body;
    ...
}

class IntLiteral extends Expr {
    private final int value;
    ...
}
```

```
class Var extends Expr {
    private final String name;
    ...
}

class Call extends Expr {
    private final String func;
    private final List<Expr> args;
    ...
}

class Defn extends Expr {
    private final String name;
    private final List<String> params;
    private final Expr body;
    ...
}

class IfExpr extends Expr {
    private final Expr cond;
    private final Expr thenBranch;
    private final Expr elseBranch; // kann null sein
    ...
}
```

Ähnlich wurden weitere Klassen wie `Def`, `LetExpr`, `LetBinding` und `DoExpr` definiert. Damit sind alle wichtigen Konstrukte der Zielsprache im AST abbildbar.

Traversierung des Parse-Trees

Als einfache Parse-Tree-Struktur dient eine Klasse `ParseNode` mit einem `kind`-Feld (z.B. `"Program"`, `ÏntLiteral"`, `"DefnExpr"`), einem optionalen `value` (z.B. Name oder Literalwert) und einer Liste von Kindknoten:

```
class ParseNode {
    public final String kind;
    public final String value;
    public final List<ParseNode> children;

    public ParseNode(String kind, String value, List<ParseNode> children) { ... }
    ...
}
```

Auf Basis dieses Parse-Trees baut die Klasse `AstBuilder` den AST auf. Sie traversiert den Parse-Tree rekursiv und erzeugt die entsprechenden AST-Knoten:

```
class AstBuilder {

    public Program buildProgram(ParseNode root) {
        if (!"Program".equals(root.kind)) {
            List<Expr> single = new ArrayList<>();
            single.add(buildExpr(root));
            return new Program(single);
        }
        List<Expr> body = new ArrayList<>();
        for (ParseNode child : root.children) {
            body.add(buildExpr(child));
        }
        return new Program(body);
    }

    private Expr buildExpr(ParseNode node) {
        switch (node.kind) {
            case "IntLiteral":
                return new IntLiteral(Integer.parseInt(node.value));
            case "Identifier":
                return new Var(node.value);
            case "AppExpr":
                String func = node.value;
                List<Expr> args = ...
                return new Call(func, args);
            case "DefnExpr":
                // Name, Parameterliste und Body auslesen
                ...
            case "IfExpr":
                // cond, then und optional else rekursiv bauen
                ...
            // weitere Fälle: DefExpr, LetExpr, DoExpr, ...
        }
    }
}
```

Test mit Beispielprogrammen

Zum Testen wurde zunächst ein kleiner Parse-Tree von Hand aufgebaut, der dem Ausdruck `(hello 5)` entspricht. Die Traversierung durch den `AstBuilder`

erzeugt daraus den AST

```
Program[Call(hello, args=[Int(5)])]
```

was genau dem erwarteten Aufbau entspricht: ein Programm mit einem Top-Level-Aufruf der Funktion `hello` mit dem Argument `5`.

Analog können die in Aufgabe A4.2 verwendeten Beispielprogramme (z.B. Funktionsdefinition mit `defn` und `if`) in einen `ParseNode`-Baum überführt und mit `AstBuilder` in einen AST übersetzt werden. Die Tests zeigen, dass alle wichtigen Informationen des Eingabeprogramms (Namen, Literale, Funktionskörper, Bedingungen, Bindungen und Aufrufe) im AST korrekt repräsentiert sind.