Übungsblatt 4

Wichtig: Abgabe des ersten Meilensteins

Bitte denken Sie daran, Ihre Implementierung des ersten Meilensteins bis spätestens **19.11.** fertigzustellen und die entsprechend markierte Revision auf den `git`-Server des Lehrstuhls zu pushen.

Aufgabe 4.1: Typanalyse

Was ist die Aufgabe der *Typanalyse*? Aus welchen Schritten besteht diese?

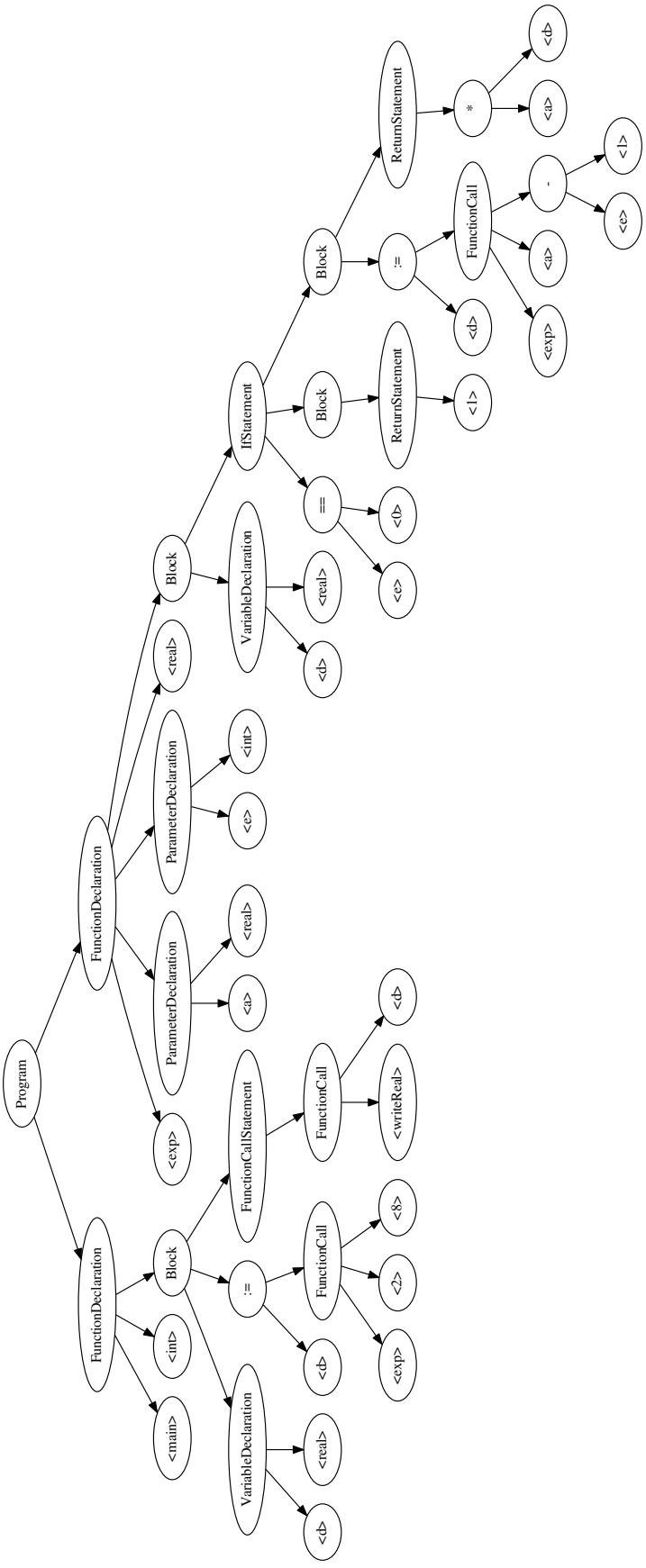
Aufgabe 4.2: Typanalyse am Beispiel

Gegeben sei das folgende `e2`-Programm[1] (der zugehörige AST befindet sich auf der nächsten Seite):

```
func main(): int
  var d : real;
  d := exp(2, 8);
  writeReal(d);
end

func exp(a : real, e : int): real
  var d : real;
  if e == 0 then
    return 1;
  else
    d := exp(a, e-1);
    return a * d;
  end
end
```

Führen Sie die Typanalyse (*ohne* Verwendung von Prototypen) für das gegebene Programm durch.

[1] **Hinweis**: Es handelt sich im Wesentlichen um dasselbe Programm wie in Aufgabe 3.4, allerdings wird hier an manchen Stellen der Datentyp `real` statt `int` verwendet.

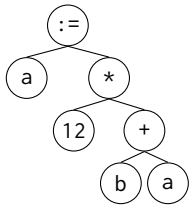

Program
FunctionDeclaration
<main>
<int>
Block
:=
<d>
FunctionCall
<8>
<2>
<exp>
FunctionCallStatement
FunctionCall
<writeReal>
<d>
VariableDeclaration
<real>
<d>
ParameterDeclaration
<int>
<e>
ParameterDeclaration
<real>
<a>
<exp>
Block
<real>
VariableDeclaration
<real>
<d>
IfStatement
==
<d>
<e>
Block
ReturnStatement
<1>
Block
:=
<d>
FunctionCall
-
<1>
<e>
<a>
<exp>
ReturnStatement
*
<d>
<a>

Aufgabe 4.3: Prototypen

Gegeben seien die folgenden e2-Fragmente sowie die dazugehörigen AST-Ausschnitte:

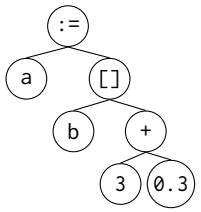
a)

```
var a : int;
var b : real;

a := 12 * (b + a);
```



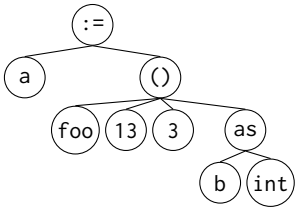
b)

```
var a : real;
var b : int[13];

a := b[3 + 0.3];
```



c)

```
var a : real;
var b : real;

// foo(real, int, int): int
a := foo(13, 3, (b as int));
```



Führen Sie jeweils die Typanalyse einmal *ohne* und einmal *mit* Prototypen durch.

Projektübung 4 (Abgabe bis 17.12.2017)

Im Rahmen dieser Projektübung implementieren Sie die Typprüfung des Eingabeprogramms. Dabei werden die AST-Knoten mit Typinformationen attributiert bzw. die Übersetzung mit einer Fehlermeldung abgebrochen, falls das Programm nicht den Typregeln der Programmiersprache genügt.

a) Machen Sie sich mit den bereits vorgegebenen Klassen vertraut, die im Rahmen der Typanalyse verwendet werden sollen. Objekte der Klasse `e2c.frontend.semantics.types.Type` (und ihrer Unterklassen) repräsentieren die Typinstanzen. Die Klasse `e2c.errors.frontend.types.TypeMismatchException` soll für das Werfen von Ausnahmen im Fehlerfall verwendet werden.

Im Rahmen der Typanalyse sollen die bei der Namensanalyse erzeugten `Symbol`-Instanzen mit Typinformationen versehen werden. Des Weiteren sollen die Typen jedes Ausdrucksbaums (Unterklassen von `Expression`) und jedes Typnamens (Unterklassen von `TypeName`) bestimmt und die AST-Knoten entsprechend attributiert werden.

b) Implementieren Sie die eigentliche Typanalyse in einem AST-Besucher `TypeAnalysis`. Setzen Sie dabei alle Angaben bezüglich der Typregeln aus der e2-Spezifikation um. Die Typprüfung soll wie folgt geschehen:

- Blattknoten von Ausdrucksbäumen sind entweder Konstanten, deren Typ einfach bestimmt werden kann, oder Bezeichner, deren Typ sich aus der zugehörigen Deklaration ergibt.
- Bei der Traversierung des ASTs im Besucher werden die Typinformationen *bottom-up* von den Blättern zu den inneren Knoten propagiert. Die Typüberprüfung soll in diesen inneren Knoten geschehen, d.h. es sollen *keine* Prototypen verwendet werden.

- In manchen Fällen (beispielsweise bei einer Addition eines `int`- und eines `real`-Werts) müssen implizite Typkonvertierungen durchgeführt werden, die im AST durch Instanzen der AST-Klasse `TypeCast` repräsentiert werden sollen. Sorgen Sie dafür, dass Ihr Besucher die notwendigen Knoten einfügt. Halten Sie sich dabei an die Angaben aus der Sprachspezifikation – fügen Sie insbesondere keine „überflüssigen“ Typkonvertierungen ein! Müssen Konstanten konvertiert werden, so sollen ebenfalls Typkonvertierungsknoten eingefügt werden.
- Denken Sie daran, alle möglichen Fehler im Eingabeprogramm zu überprüfen (bei Funktionsaufrufen muss der Bezeichner zu einer Funktion gehören, Funktionen müssen mit der richtigen Anzahl an Argumenten aufgerufen werden, Array-Indizes müssen vom Typ `int` sein, ...).

Ihre Implementierung soll beim **ersten** gefundenen Fehler abbrechen und die entsprechende Ausnahme (siehe oben) samt passender Position im Quellprogramm (über den entsprechenden AST-Knoten zugreifbar) werfen. Die Fehlerposition wird bei der Ausführung der Testfälle (siehe unten) überprüft.

c) Rufen Sie den AST-Besucher in der Methode `e2c.E2Compiler.performTypeAnalysis()` geeignet auf (diese Methode wird von den Testfällen verwendet, siehe unten) und sorgen Sie dafür, dass die Typanalyse auch tatsächlich in Ihrem Übersetzer durchgeführt wird.

d) Falls die Kommandozeilenoption „`--dot`“ gesetzt ist, soll der AST (zusätzlich zu den bisherigen Ausgaben aus den früheren Projektübungen) *nach* der Typprüfung im Dot-Format ausgegeben werden, um die eingefügten Typkonvertierungen visuell überprüfen zu können. Ersetzen Sie für die Ausgabe nach der Typprüfung die Dateiendung durch „`.typing.dot`“.[2]

Ihr Übersetzer sollte nun in der Lage sein, die Typkonsistenz des Eingabeprogramms zu überprüfen. Testen Sie dies mit eigenen Testprogrammen bzw. mit den Testfällen zum dritten Meilenstein (siehe unten).

Meilenstein 3 (Abgabe bis 17.12.2017)

Stellen Sie sicher, dass Ihr Übersetzer alle zum dritten Meilenstein gehörenden Testfälle korrekt behandelt, indem Sie den Gradle-Task „`milestone3`“ ausführen und kontrollieren, dass am Ende die Ausgabe „`BUILD SUCCESSFUL`“ erscheint. Versehen Sie den finalen Stand mittels „`git tag`“ mit der Markierung „`milestone3`“ und pushen Sie Ihre Änderungen auf den `git`-Server des Lehrstuhls.

[2] **Tipp**: `e2c.util.FileUtil.changeFileExtension()`