Übungsblatt 3

Aufgabe 3.1: Lexer und Parser

Was sind die Aufgaben des *Lexers* und des *Parsers*?

Welche Art von Grammatik wird jeweils verwendet? Was versteht man unter $LL(k)$ und $LR(k)$?

Warum wird die syntaktische Analyse nicht ausschließlich mit einem Lexer durchgeführt? Warum nicht ausschließlich mit einem Parser?

Aufgabe 3.2: Shift-Reduce-Parser; konkreter und abstrakter Syntaxbaum

Gegeben sei die folgende Grammatik für Zuweisungen mit einfachen arithmetischen Ausdrücken:

```
assgn
  : IDENTIFIER '=' add_expr ';' ;

add_expr
  : add_expr '+' mul_expr
  | mul_expr ;

mul_expr
  : mul_expr '*' factor
  | factor ;

factor : NUMBER | IDENTIFIER ;
```

Parsen Sie mit Hilfe des zugehörigen *Shift-Reduce-Parsers* die folgende Eingabe: `x = 13 + y * 3;`

Konstruieren Sie neben dem *konkreten Syntaxbaum* auch einen geeigneten *abstrakten Syntaxbaum* (AST).

Aufgabe 3.3: Besucher-Entwurfsmuster

Wie funktioniert das *Besucher-Entwurfsmuster* und wofür wird dieses verwendet?

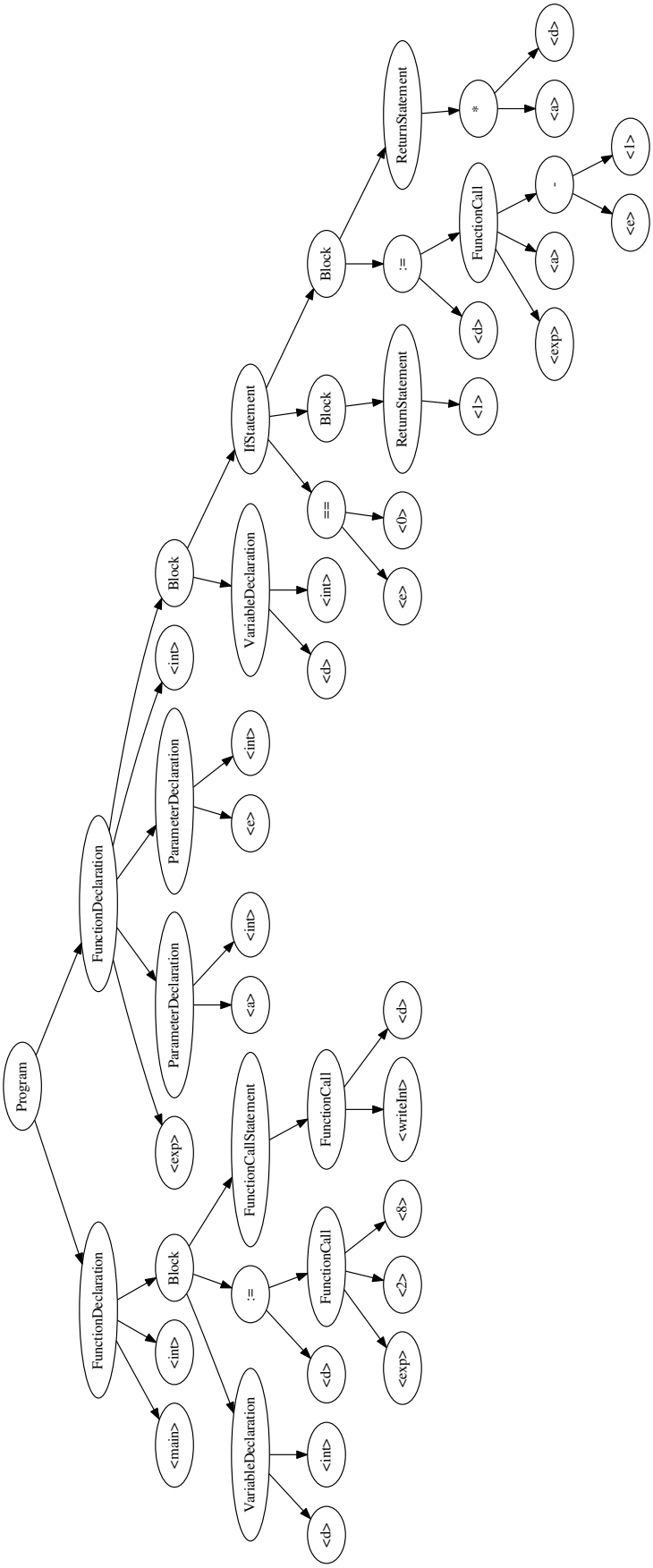
Aufgabe 3.4: Namensanalyse

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

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

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

Führen Sie die Namensanalyse für das gegebene Programm mit Hilfe einer „Symboltabelle auf Papier“ durch.


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

Projektübung 3 (Abgabe bis 03.12.2017)

Im Rahmen dieser Projektübung erweitern Sie Ihren e2-Übersetzer um die Namensanalyse. Ihr Übersetzer ist dann in der Lage, jede Verwendung eines Bezeichners der jeweiligen Deklaration zuzuordnen bzw. einen Fehler zu melden, falls das Eingabeprogramm fehlerhaft ist.

a) Machen Sie sich mit den bereits vorgegebenen Klassen vertraut, die im Rahmen der Namensanalyse verwendet werden sollen. Objekte der Klasse `e2c.frontend.semantics.symbols.Symbol` repräsentieren die für die im Eingabeprogramm deklarierten Variablen (Unterklasse `VariableSymbol`) bzw. Funktionen (Unterklasse `FunctionSymbol`) eingeführten Symbole. Die Klassen in `e2c.errors.frontend.symbols` sollen für das Werfen von Ausnahmen im Fehlerfall verwendet werden.

Jedes Symbol speichert eine Referenz auf den zugehörigen Deklarationsknoten im AST. Des Weiteren soll nach der Namensanalyse auch jeder Deklarationsknoten und jede Verwendung eines Bezeichners (`Identifier`-Knoten) eine Referenz auf das zugehörige Symbol speichern.

Hinweis: Jedem Symbol wird später auch ein Typ (`e2c.frontend.semantics.types.Type`) zugeordnet. Dies passiert jedoch erst während der Typanalyse (spätere Übung) und spielt daher zunächst keine Rolle.

b) Für die eigentliche Namensanalyse wird eine Datenstruktur zur Speicherung der jeweils aktuellen Symboltabelle benötigt. Diese muss mindestens die folgende Funktionalität bieten:

- Betreten eines neuen Sichtbarkeitsblocks.
- Verlassen des aktuellen Sichtbarkeitsblocks.
- Hinzufügen eines Symbols im aktuellen Sichtbarkeitsblock. Falls dieser bereits ein Symbol mit dem gleichen Namen beinhaltet, muss ein Fehler (`SymbolAlreadyDefinedException`) gemeldet werden.
- Nachschlagen eines Symbols anhand eines Bezeichners. Falls kein Symbol mit diesem Bezeichner exisitert, muss ein Fehler (`SymbolNotDefinedException`) gemeldet werden.

In der Vorlesung haben Sie einige Varianten für die Implementierung einer Symboltabelle kennengelernt. Implementieren Sie eine davon in Ihrem e2-Übersetzer. Sie *können* sich an dem von uns zur Verfügung gestellten Interface `e2c.frontend.semantics.symbols.SymbolTable` orientieren und dieses implementieren.

c) Die implizit deklarierten Funktionen der e2-Standardbibliothek (`readInt()`, ...) müssen dem Übersetzer bekannt gemacht werden, damit ihre Verwendung nicht zu einem Fehler führt. Definieren Sie in der Klasse `e2c.frontend.semantics.symbols.SymbolTableFactory` Symbole für diese Funktionen und fügen Sie diese zu dem dort definierten Array `predefinedFunctions` hinzu (wichtig für eine spätere Aufgabe!). Sie *können* dieser Klasse auch Methoden zum Erzeugen und Befüllen einer Symboltabelle hinzufügen.

d) Implementieren Sie die eigentliche Namensanalyse in einem AST-Besucher `NameAnalysis`. Setzen Sie dabei die Angaben bezüglich der Sichtbarkeitsschachteln aus der e2-Spezifikation um.

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.

Hinweis: Beachten Sie, dass während unserer Namensanalyse noch keinerlei Typ-Überprüfungen stattfinden. Dies bedeutet, dass beispielsweise folgendes Programm die Namensanalyse bestehen soll, da das in der Zuweisung verwendete Symbol `foo` im Eingabeprogramm deklariert wurde:

```
func foo()
  foo := 13;
end
```

Die Ausgabe einer entsprechenden Fehlermeldung soll erst während der Typprüfung in einer späteren Projektübung erfolgen.

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

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

Meilenstein 2 (Abgabe bis 03.12.2017)

Stellen Sie sicher, dass Ihr Übersetzer alle zum zweiten Meilenstein gehörenden Testfälle korrekt behandelt, indem Sie den Gradle-Task „`milestone2`“ ausführen und kontrollieren, dass am Ende die Ausgabe „`BUILD SUCCESSFUL`“ erscheint. Versehen Sie den finalen Stand mittels „`git tag`“ mit der Markierung „`milestone2`“ und pushen Sie Ihre Änderungen auf den `git`-Server des Lehrstuhls.