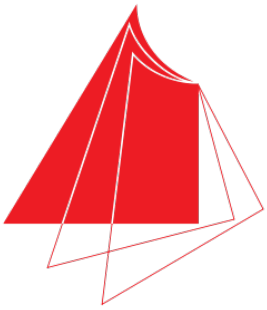

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

SYSTEMNAHES PROGRAMMIEREN

Compiler

Timo Blust, 48594

Gennadi Eirich, 50629

Tim Essig, 49683

Gruppe 17 (WS 15/16)

20. Juli 2017

Inhaltsverzeichnis

1 Scanner

1.1 State Machine

Da im zweiten Teil des Kompiler-Projekts neue Schlüsselworter hinzukommen, muss zunächst der Token Scanner (der Verwalter der Automaten) angepasst werden. Durch die modulare Struktur des Token Scanners ist dies jedoch mit geringem Aufwand möglich, da jedes Token (also auch jedes Schlüsselwort) einen eigenen Automaten besitzt.

Die Automaten für die neuen Schlüsselwörter

```
int
else
read
write
```

können mithilfe der statischen Funktion `StateMachine::createString` generiert werden. Diese Funktion erzeugt einen Zustandsautomaten, welcher genau die übergebenen Strings akzeptiert. Der Automat für das Schlüsselwort `else` wird zum Beispiel so erzeugt:

```
StateMachine::createString(Token::KW_ELSE,
                           new const char*[2]{ "ELSE", "else" },
                           2);
```

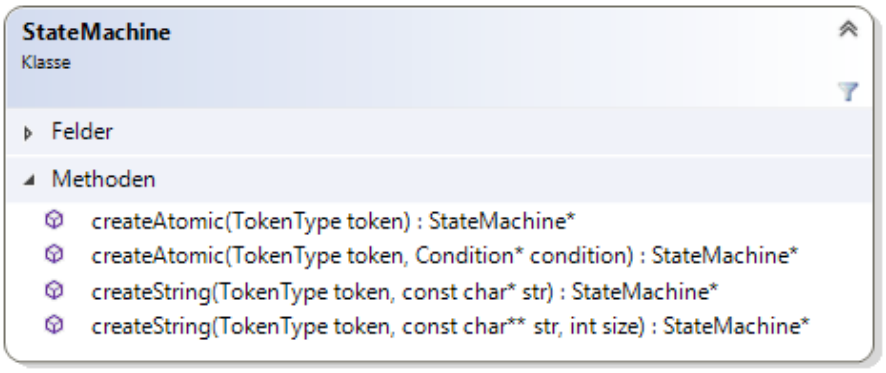

Abbildung 1.1: Vereinfachtes Klassendiagramm von StateMachine

2 Parser

Der Parser hat die Aufgabe, aus der Tokensequenz die der Scanner liefert, einen Strukturbaum zu erstellen, hierbei prüft er auch die syntaktische Korrektheit. Der Datenfluss ist wie folgt: Der Parser fordert die Tokens vom Scanner an. Der Parser führt nach der Erstellung des Strukturbauems ebenfalls einen Typenprüfung durch und generiert den Code für die VM.

2.1 Parse Tree

Der Parser hat die Aufgabe aus den Tokens, welche der Scanner liefert, einen Parse-Tree auf zu spannen. Er erkennt hierbei syntaktische Fehler, allerdings keine semantischen(z.B. ob einer Variable eine Wert vom richtigen Typ zugewiesen wird). Die Aufgabe dies zu überprüfen hat der Typechecker, dieser wird aufgerufen, sobald der Parser erfolgreich beendet ist. Der Parser startet mit dem Startsymbol *PROG* und erzeugt von da ausgehen für jede erkannte Regel einen neuen Teilbaum.

Er erkennt folgende Grammatik:

```
PROG        ::= DECLS STATEMENTS
DECLS       ::= DECL ; DECLS | eps
DECL        ::= int ARRAY identifier
ARRAY       ::= [ integer ] | eps


STATEMENTS ::= STATEMENT; STATEMENTS | eps
STATEMENT  ::= identifier INDEX := EXP  | write( EXP ) |
               read(identifier INDEX) | {STATEMENTS} |
               if ( EXP ) STATEMENT else STATEMENT      |
               while ( EXP ) STATEMENT
EXP                ::= EXP 2 OP_EXP
EXP2        ::= ( EXP ) | identifier INDEX | integer | -EXP2 | !EXP2
INDEX       ::= [ EXP ] | eps
OP_EXP      ::= OP EXP | eps
OP                 ::= + | - | * | : | < | > | = | =:= | &&
```

eps = ϵ

In dieser Grammatik sind bereits alle Linksrekursionen und Mehrdeutigkeiten aufgelöst wurden, somit ist eine Ableitung immer eindeutig, was die Arbeit des Parsers erheblich vereinfacht. Beim Rekursiven-Abstieg schaut sich der Parser zunächst den nächsten Token an, bevor er einen Schritt tiefer geht. Er akzeptiert diesen allerdings nur, wenn er gültig ist. Ist der nächste Token an dieser Stelle des Baumes nicht gültig, beendet er mit einem Fehler. In diesem Fall enthält die Fehlermeldung die Informationen, welcher Token erwartet wurde(ggf. mehrere) und welcher gefunden wurde.

Die einzelnen Knoten werden über ein *Node-Objekt* abgebildet. Dieses repräsentiert alle Knotentypen, über die Methode *Node::getType()* kann der Typ abgefragt werden. Für jeden Knotentyp existiert eine Facotry-Methode von folgendem Typ:

```
static Node* makeProg(Node* decls, Node* statements, Token* token);
```

Hierdurch werden die einzelnen Kind-Knoten für jeden Typ bestimmt und es wird unterbunden, ein PROG-Knoten mit zu wenigen oder zu vielen Kindern zu erstellen. Umgekehrt hat Node verschiedene Funktionen um die Kindknoten abzufragen, diese sind abhängig vom Knoten-Typ, z.B.:

```
Node* getStatements() const;
```

Dies macht das Verarbeiten des Baumes einfacher. Wird eine Funktion aufgerufen, die für den aktuellen Knoten-Typ unzulässig ist, terminiert das Programm mit einem Fehler.

Ein Klassendiagramm(ohne alle Funktionen) ist in Abbildung 2.1 zu sehen.

2.2 Type Checking

Nach der erfolgreichen Generierung des Parse Trees erolgt eine Typ-Prüfung. Hierbei wird geprüft, ob sämtliche Ausdrückte semantisch korrekt sind. Wird zum Beispiel ein Identifier mehrfach definiert (egal ob vom selben Typ oder verschiedenen Typen), ist es Aufgabe des Type Checks diesen Fehler zu ermittlen und zu melden.

Folgende Punkte werden während des Type Checks geprüft:

- Mehrfache Definition des gleichen Identifiers
- Inkompatible Datentypen (z.B. einer Variable vom Typ Integer-Array wird ein Wert vom Typ Integer zugewiesen)
- Verwendung von undefinierten Identifiern
- Ungültige Länge in Arraydefinitionen (z.B. `int[0] toSmall;`)

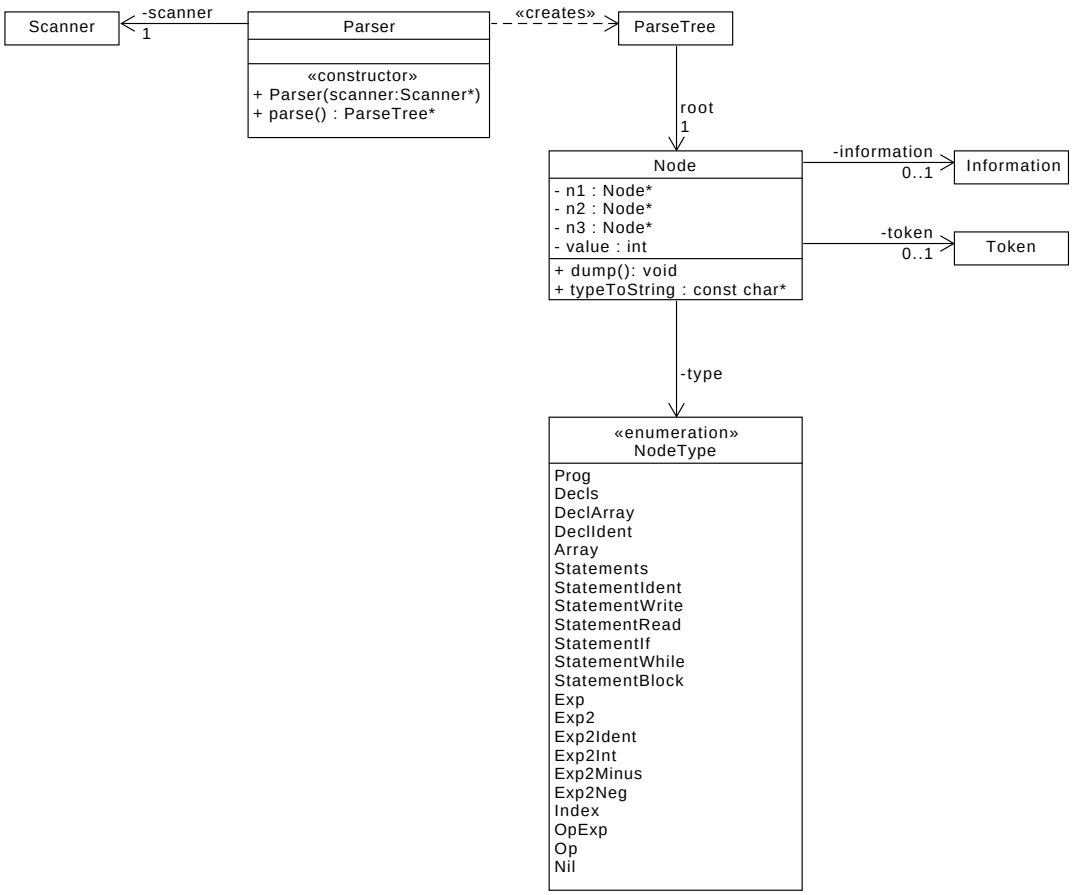

Abbildung 2.1: Klassendiagramm des ParseTree

Das Ablaufdiagramm in Abbildung 2.2 und 2.3 zeigt die Funktionsweise der Typ-Prüfung.

2.3 Code Generation

//TODO: Gena

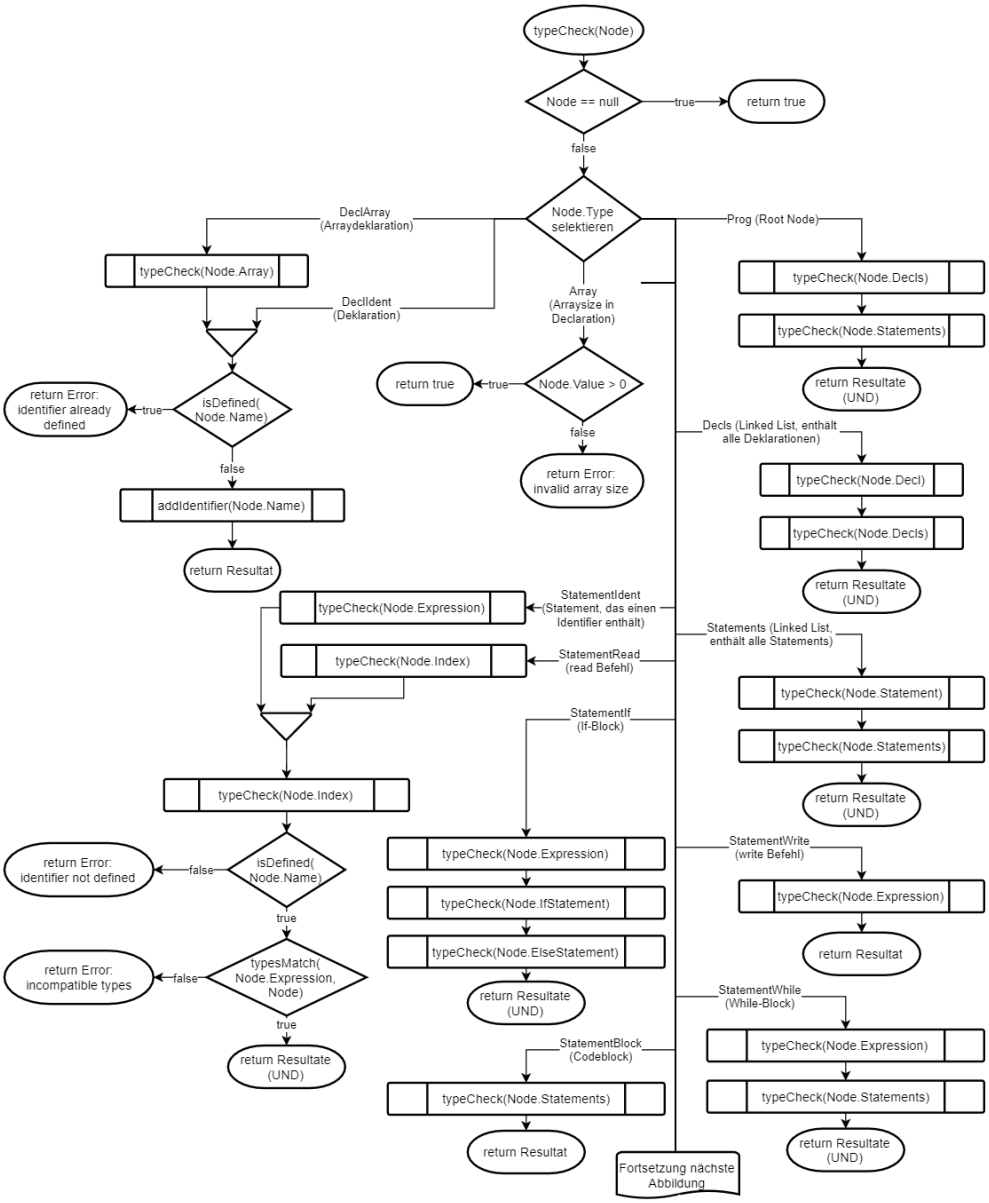

Abbildung 2.2: Ablaufdiagramm des Type Checks (Teil 1)

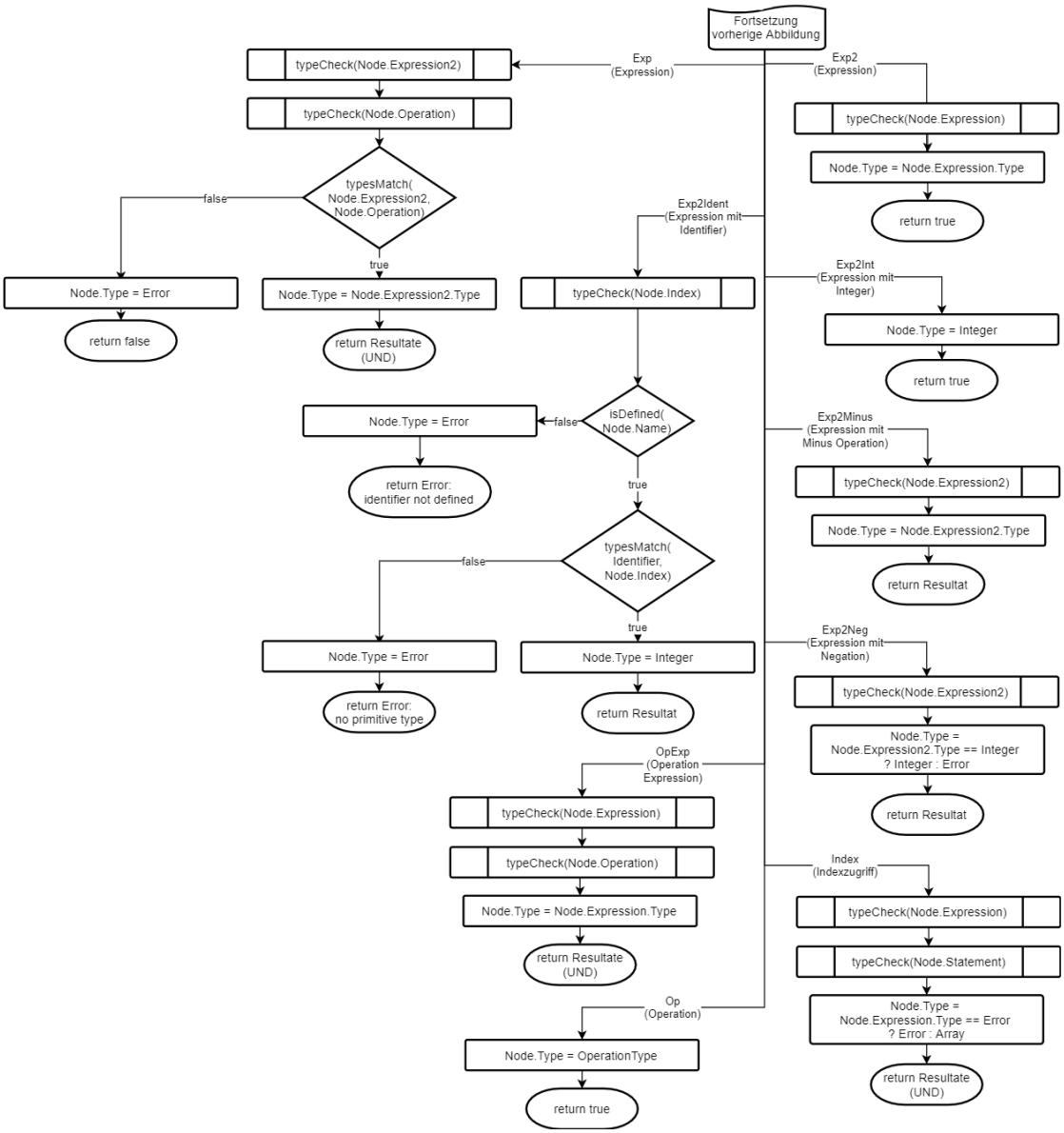

Abbildung 2.3: Ablaufdiagramm des Type Checks (Teil 2)