Ausarbeitung

Ausarbeitung

Compilerbau

An der Fachhochschule Dortmund
im Fachbereich Informatik
Studiengang Informatik
erstellte Ausarbeitung
für das Modul
Formale Sprachen und Compilerbau

von
Alexander Weidemann
Matr.-Nr. 7206266

Bijan Riesenberg
Matr.-Nr. ???

Johanna Kraken
Matr.-Nr. 7206374

Betreuer: Prof. Dr. Robert Rettinger

Dortmund, 9. Oktober 2020

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

Dieses Dokument geht auf den Aufbau des Compilers ein, der für die Semesterbegleitleistung in der Vorlesung „Formale Sprachen und Compilerbau“ programmiert wurde.

Die Sprache, die der Compiler übersetzt, wird im 2 Kapitel beschrieben. Im Kapitel 3 werden die Token näher erläutert, die im Compiler verwendet werden. Im Nächsten Kapitel, dem 4 Kapitel, wird die Grammatik erklärt. Außerdem wird in diesem Kapitel anhand von Beispielen der Ablauf der Grammatik gezeigt. Im 5 Kapitel wird auf die Semantischen Regeln eingegangen. Im 6 und letzten Kapitel wird der Aufruf des Compilers erklärt und beschrieben.

2 Sprache

Die Sprache, die hier entwickelt wurde, ist angelehnt an die Programmiersprache Java. Es gibt aber die ein oder andere Abweichung. Es gibt wie in Java verschiedene Datentypen. Es gibt String, Integer, Double, Charakter und Boolean. Hier wird der Datentyp „boolean“ aber nicht so genannt, sondern „bool“. Variablen können so deklariert, definiert und mathematisch bearbeitet werden wie in Java.

Ebenfalls möglich ist der Umgang mit Arrays. Hierbei werden ein Get- und Set-Methode benutzt, um das Array zu füllen oder um etwas aus dem Array auszugeben. Die „get“-Funktion, die das Element an Position Index aus dem Array holt, hat zwei Parameter. Der erste ist das Array und der Zweite ist der Index als int-Datentyp, an dessen Position gesucht werden soll. Der Rückgabewert ist ein Element aus dem Array. Der Typ hängt vom Array ab und ist entweder ein BasisTyp oder ein Array mit um 1 verringerter Dimension. Die „set“-Funktion setzt ein Element an eine Position aus dem Array. Sie hat drei Parameter. Der Erste ist das Array und der Zweite ist die Position, an der der dritte Parameter, das Element, eingesetzt werden soll. Dieses Funtionen werden näher erläutert in dem Unterkapittel 4.2. Die Deklaration von Arrays wird wie in Java gehandhabt.

Außerdem ist es möglich sowohl in einen Anderen Datentypen zu casten und bei einer Variable die Länge abzufragen, als auch abzufragen ob eine Variable einen bestimmten Datentyp hat. Das Casten wird mit der Funktion „convert“ gemacht. Diese Funktion hat zwei Parameter. Der erste ist der Datentyp, in den gecastet wird, und

der Zweite ist die Variable oder Literal, dergecastet wird. Der Rückgabewert dier Funktion ist der zweite Parameter, aber als Datentype, der als erster Parameter übergeben wird. Für die Längenabfrage wird mit der Funktion „length“ gemacht. Diese Funktion hat einen Parameter, die Variable oder Literal, von der die Länge abgefragt wird. Der Rückgabewert ist ein Integer. Bei den Datentypen int, char, double und bool wird 1, bei einem String die Länge des Strings und bei einem Array wird die Anzahl der Elemente zurückgegeben. Beispiele zu dieser Funktion sind folgende: Die Funktion „isType“ wird dafür verwendet, um den Datentyp abzufragen. Die Funktion hat zwei Parameter. Der Erste ist der Datentyp, auf den geprüft wird. Der Zweite ist die Variable oder Literal, dessen Datentyp geprüft wird. Der Rückgabewert ist ein Boolean. „True“ wird zurückgegeben, wenn der Datentyp dem Datentyp des zweiten Parameters entspricht, und „False“, wenn nicht. Näher auch diese Funktionen wird im Kapitel 4.1 eingegangen. Beispiele zu diesen Funktionen sind folgende:

```
length("hallo") == 5          isType(string, "test") == true
length("a") == 1              isType(string, 1) == false
length(5) == 1                isType(int, "1") == false
length([a, b, c]) == 3        isType(int, 1) == true
convert(int, "1") == 1        convert(string, 1) == "1"
```

Listing 1: Beispielcode für die Funktionen length(), isType() und convert()

Eine Funktion, die nicht in Java vorhanden ist, ist Smart Switch („String @String{}“). Diese Funktion baut eine Zeichenkette auf Basis von regulären Ausdrücken zusammen. Dabei können mehrere Fälle abgedeckt werden. Bei einem einzelnen Fall ist das zurückgegebene Array eindimensional. Bei mehr als einem Fall ist das Array zweidimensional. Pro Fall wird somit ein Array erzeugt. Zu jedem Fall wird ein Funktionsblock definiert. Dieser muss eine Zeichenkette zurückgeben. Innerhalb des Blockes kann auf die Variable "thisßugegriffen werden, welche den übereinstimmenden Teil der ursprünglichen Zeichenkette enthält. In den Arrays gespeichert werden die Rückgabewerte der Funktionsblöcke. Bei den regulären Ausdrücken gibt es Besonderheiten: Das Ausrufezeichen "!"muss mittels des Backslashes escaped werden. Ein einfaches Ausrufezeichen "!äm Anfang eines regulären Ausdrucks kann genutzt

werden, um den gesamten Ausdruck zu negieren. Zudem werden die regulären Ausdrücke aus Java verwendet. Diese wurden allerdings um die POSIX Character Classes erweitert.en Ausdruck zu negieren. Der Rückgabewert ist hierbei ein String. Zwei Berispiele für diese Funktion werden unterhalb aufgeführt.

```
string[] p = @"test"{
    ":alnum:" { return this; }
};

// p == ["t", "e", "s", "t"]
string[][] p = @"hello world!"{
    ":alnum:" { return this; }
    ":space:" { return "space"; }
    ":punct:" { return "."; }
    "[l]" { return "L"; }
};

// p == ["h", "e", "l", "l", "o", "w", "o", "r", "l", "d"],
   ["space"], ["."], ["L", "L", "L"]
```

Listing 2: Beispielcode für Smart Switch Funktion

Eine weitere Funktion ist die Write-Funktion („write(str [, path])“). Sie schreibt den String in die Konsolenausgabe oder optional in die angegebene Datei. Bei der angabe eines Dateipfades wird keine Konsolenausgabe geschrieben. Der erste Parameter str ist der zu schreibene String. Der Parameter path ist optional. es ist der Pfad der Datei, in welche geschrieben werden soll. Die Funktion hat keinen Rückgabewert. Zwei Beispiel für die Funktion sind „write("test");“ und „write("test", "path/to/file");“.

Da es die Write-Funktion gibt, die in eine Datei schreibt, gibt es auch eine Read-Funktion („string read([path])“), die eine Dateien oder eine Zeile von der Kommandozeile ausließt und deren Inhalt ausgibt. Die Funktion hat einen Parameter, der den Pfad der Datei ist, aus welcher gelesen werden soll. Der Inhalt der Datei wird als String zurückgegeben. Passent dazu gibt es noch weitere Read-Funktionen, die nur einen bestimmten Datentypen wie bool, int oder double von der Komandozeile

einlesen. Diese haben keinen Parameter aber den gleichen Datentyp, der eingelesen wird, als Rückgabe. Beispiele für die Read-Funktion sond folgende:

```
string input = read();
input = input + "!";
write(input);

input = read("path/to/file");
write(input);

bool b = readBool();
write(b);

int i = readInteger();
i = i + 10;
write(i);

double j = readDouble();
j = j * 2.5;
write(j);
```

Listing 3: Beispielcode für die verschiedenen read-Funktionen

3 Festlegung der Token

In diesem Kapitel werden die Token, die beim Compiler verwendet werden, aufgezählt und erläutert.

Als erstes werden die Token der Typen definiert. Diese sind int (= Interger), double (= Double), char (= Character), bool (= Boolean), string (= String), void (= Void) und Array. Das Array kann die Typen int, double, char, bool oder string enthalten, welcher als erstes geschrieben wird. Danach wird mit der Häuftigkeit der eckigen Klammern die Anzahl der Dimension des Arrays angezeigt.

```
// Types
```

```
TOKEN : {
    < INT: "int" >
|   < DOUBLE : "double">
|   < CHAR : "char">
|   < BOOLEAN : "bool">
|   < STRING : "string" >
|   < ARRAY : (<INT> | <DOUBLE> | <CHAR> | <BOOLEAN> |
  <STRING>) ("[]")* >
|   < VOID : "void" >
 }
```

Listing 4: Tokendefinition der Types

Anschließend werden die Operatoren definiert. Diese bestehen unter anderem aus dem Plus- und Minus-Operator in der Gruppe der SUM_OPERATOR und aus dem Mal-, Geteilt- und Modulo-Operator in der Gruppe der PROD_OPERATOR. Außerdem gibt es noch die Gruppe der BIN_COMPARE_OPERATOR, die aus den Operatoren gleich, ungleich, gößer, größer gleich, kleiner und kleiner gleich bestehen. Die letzte Gruppe in diesem Bereich ist die BIN_GATE_OPERATOR. In dieser gibt es nur die zwei Binäroperatoren von UND und ODER.

```
// Operators
TOKEN :  {
    <SUM_OPERATOR: "+" | "-">
|   <PROD_OPERATOR: "*" | "/" | "\%">
|   <BIN_COMPARE_OPERATOR: ">=" | ">" | "<=" | "<" | "==" |
  "!=">
|   <BIN_GATE_OPERATOR: "||" | "&&">
 }
```

Listing 5: Tokendefinition der Operatoren

Die nächsten Token, die definiert werden, sind die verschiedenen Keywords. Die Gruppe beinhaltet das return-Wort, das Komma, hier als ENUM_SEPERATOR

bezeichnet, und das Semikolon, hier als SEMICOLON benannt. Darunter sind die Keywords aufgelistet, die für Funktionen und Schleifen benötigt werden. Das sind if, else, while, foreach und in, die jeweils so aber in Großbuchzeichen bezeichnet werden wie alle Token. Außerdem beinhaltet diese Gruppe auch das Ad-Zeichen unter dem Namen SMARTSWITCH und die geschweiften Klammern mit der Öffnenen als OPEN_BLOCK und der Schließenden CLOSE_BLOCK.

```
// Keywords
TOKEN : {
    < SMARTSWITCH : "@">
|   < RETURN: "return">
|   < ENUM_SEPERATOR: ",">
|   < SEMICOLON: ";">
|   < IF: "if">
|   < ELSE: "else">
|   < WHILE: "while">
|   < FOREACH: "foreach">
|   < IN: "in">
|   < OPEN_BLOCK: "{">
|   < CLOSE_BLOCK: "}">
 }
```

Listing 6: Tokendefinition der Keywords

Als nächstes werden die Literale definiert. Hier werden Ganzahlen, Dezimalzahlen, Kommazahlen, Buchstaben, Wörter und Booleans definiert und dargestellt wie diese aufgebaut werden. Beispielsweise ist ein BOOLEAN_LITERAL nur "trueöder "false", ein INTEGER_LITERAL ist #DECIMAL_LITERAL und ein DECIMAL_LITERAL besteht immer aus einem oder keinem Plus- oder Minuszeichen, einer Ziffer von 1 bis 9 und danach folgt eine unbestimmte Anzahl von Ziffern im Bereich von 0 bis 9. Ein richtiges Beispiel dafür ist -24. Der FLOATING_POINT_LITERAL bestehlt aus einem oder keinem Plus- oder Minuszeichen, einem oder mehreren Ziffern zwischen 0 und 9, dem Punkt und keinem oder mehreren Ziffern zwischen 0 und 9. 2.859 ist zum Beispiel eine richtiges FLOATING_POINT_LITERAL. Zudem werden Token

wie runde und eckige Klammern, das Ausrufe-, das einfache und doppelte Minus- und Plus-, das Mal-, das Geteiltzeichen und der Doppelpunkt definiert. Diese werden dafür benutzt, um die Literale zu bearbeiten.

```
// Literals
TOKEN : {
    "(" | ")" | "!" | "--" | "++" | "+" | "-" | "*" | "/" |
       "%" | ":" | "[" | "]"
|   <INTEGER_LITERAL: <DECIMAL\_LITERAL>>
|   <#DECIMAL_LITERAL: ("+" | "-")? ["1"-"9"] (["0"-"9"])*>
|   <FLOATING_POINT_LITERAL: ("+" | "-")? (["0"-"9"])+ "."
   (["0"-"9"])*>
|   <CHARACTER_LITERAL: "\'" (\~["\'", "\\", "\n","\r"] |
   "\" (["n","t","b","r","f", "\\", "\'", "\""] | ["0"-"7"]
   (["0"-"7"])? | ["0"-"3"] ["0"-"7"] ["0"-"7"])) "\'">
|   <STRING_LITERAL: "\"" ( \~["\"", "\\", "\n","\r"] |
   "\\" ( ["n", "t", "b", "r", "f", "\\", "\'", "\""] |
   ["0"-"7"] (["0"-"7"])? | ["0"-"3"] ["0"-"7"] ["0"-"7"]
   | ( ["\n","\r"] | "\r \n")))* "\"">
|   <TYPE_LITERAL: ">" ( \~["<",">","\\","\ n", "\r"] |
   "\\" ( [">", "<", "n", "t", "b", "r", "f", "\\"] |
   ["0"-"7"] (["0"-"7"])? | ["0"-"3"] ["0"-"7"] ["0"-"7"]
   | ( ["\n", "\r"] | "\r \n")))* "<">
|   <BOOLEAN_LITERAL: "true" | "false">
 }
```

Listing 7: Tokendefinition der Literale

Dann werden Built-in-Funktionen-Token definiert. Darunter befinden sich CONTERT_FUNCTION ("convert"), CHECKTYPE_FUNCTION (ïsType"), LENGTH_FUNCTION ("length"), READ_FUNCTION ("read"), WRITE_FUNCTION ("write"), GET_FUMCTION ("get") und SET_FUMCTION (ßet").

```
// Built-in Functions
TOKEN : {
    < CONTERT_FUNCTION: ( "convert" )>
|   < CHECKTYPE_FUNCTION: ( "isType" )>
|   < LENGTH_FUNCTION: ( "length" )>
|   < READ_FUNCTION: ( "read" )>
|   < WRITE_FUNCTION: ( "write" )>
|   < GET_FUMCTION: ( "get" )>
|   < SET_FUMCTION: ( "set" )>
 }
```

Listing 8: Tokendefinition der Buil-in Funktionen

Als letztes werden die Identifier definiert. In der Gruppe befinden sich #LETTER, der ein Groß- oder Kleinbuchstaben ist (bsp: a oder C), und #DIGIT, das eine Ziffer von 0 bis 9 ist (bsp: 5). Der IDENTIFIER beginnt immer mit einem LETTER und hat dann eine unbestimmte Kombination und Anzahl von #LETTER und #DIGIT. Ein gültiger IDENTIFIER ist G5 oder bla24bla.

```
// Identifier
TOKEN : {
    < #LETTER: ["A"-"Z","_","a"-"z"]>
|   < #DIGIT: ["0"-"9"]>
|   < IDENTIFIER: <LETTER> (<LETTER> | <DIGIT>)*>
 }
```

Listing 9: Tokendefinition der Identifier

4 Grammatik

Diese Kapitel geht näher auf die Grammatik ein, die Für den Compiler benutzt wird.

Die Grammatik wird wie folgt definiert: G = (N, T, R, S). Es wird die Linksableitung benutzt.

Die verwendeten Funktionen in der Grammatik sind folgende: N = { Program, Statement, VariableDeclaration, Block, FunctionBlock, ArrayExpression, Expression, LogicalExpression, CompareExpression, AdditiveExpression, MultiplicativeExpression, UnaryExpression, Atom, FunctionCall, Identifier, IfStatement, WhileStatement, ForeachStatement, FunctionDeclaration, ReturnStatement, SmartSwitch, IsType, Convert, Length, Read, WriteStatement, Datatype, Get, Set } (Funktionennamen in NewAwk.jjt)

Die verwendeten Terminale werden in T definiert und sind folgende: int, double, float, char, bool, string, type, void, null, @, return, ',', ';', if, else, while, foreach, in, LETTER, DIGIT, IDENTIFIER, ':',isType, convert length, (,), +, -, *, /, =, >, <, |, &, !, %, EOF, read, write

Die Startfunktion S ist Program.

In R sind die Regeln, auf denen die Grammatik basiert, aufgeführt.

```
R = {
    Program -> ( FunctionDeclaration() )* ( Statement() |
        VariableDeclaration() )+ <EOF>
    Statement -> Block() | WhileStatement() |
        ForeachStatement() | IfStatement() | WriteStatement()
        | Set() | Expression() ";"
    VariableDeclaration -> Datatype() Identifier() ["="
        Expression()] ";" | Datatype() Identifier() ["="
        Expression() SmartSwitch()] ";"
    Block -> "{" ( Statement() | VariableDeclaration() )* "}"
    FunctionBlock -> "{" ( Statement() |
        VariableDeclaration() )* ( ReturnStatement() )? "}"
    ArrayExpression -> "[" ( Expression() ( "," Expression()
        )* )? "]"
    Expression -> Identifier() "=" Expression() |
        LogicalExpression()
    LogicalExpression -> CompareExpression() (
        <BIN_GATE_OPERATOR> CompareExpression() )*
    CompareExpression -> AdditiveExpression()
```

```
    [<BIN_COMPARE_OPERATOR> AdditiveExpression()]
AdditiveExpression -> MultiplicativeExpression() (
    <SUM_OPERATOR> AdditiveExpression() )*
MultiplicativeExpression -> UnaryExpression() (
    <PROD_OPERATOR> MultiplicativeExpression() )*
UnaryExpression -> ( <SUM_OPERATOR> | "!" )
    UnaryExpression() | Atom()
Atom -> FunctionCall() | Identifier() ["++" | "--"] | (
    "++" | "--" ) Identifier() | <INTEGER_LITERAL> |
    <FLOATING_POINT_LITERAL> | <CHARACTER_LITERAL> |
    <STRING_LITERAL> | <BOOLEAN_LITERAL> | "("
    Expression() ")" | ArrayExpression() | IsType() |
    Convert() | Length() | Read() | Get() | SmartSwitch()
FunctionCall -> Identifier() "(" [LogicalExpression() (
    "," LogicalExpression() )*] ")"
Identifier -> <IDENTIFIER>
// Function
ReturnStatement -> "return" [LogicalExpression()] ";"
FunctionDeclaration -> Datatype() Identifier() "(" (
    Datatype() Identifier() )? ( "," Datatype()
    Identifier() )* ")" FunctionBlock()
ForeachStatement -> "foreach" "(" Datatype()
    Identifier() "in" Expression() ")" Statement()
WhileStatement -> "while" "(" LogicalExpression() ")"
    Statement()
IfStatement -> "if" "(" LogicalExpression() ")"
    Statement() ( "else" ( IfStatement() | Statement() )
    )?
SmartSwitch -> "@" Expression() "{" ( Expression()
    FunctionBlock() )+ "}"
// Integrated Funtions
IsType -> "isType" "(" Datatype() "," Expression() ")"
Convert -> "convert" "(" Datatype() "," Expression() ")"
```

```
    Length -> "length" "(" Expression() ")"
    Read -> "read" "(" Expression() ")" | "read" "(" ")" |
        "readBool" "(" ")" | "readInteger" "(" ")" |
        "readDouble" "(" ")"
    WriteStatement -> "write" "(" Expression() ( ","
        Expression() )* ")" ";"
    Get -> "get" "(" Expression() "," Expression() ")"
    Set -> "set" "(" Expression() "," Expression() ","
        Expression() ")" ";"
    // Datatype
    Datatype -> "double" | "int" | "char" | "bool" |
        "string" | <ARRAY>
}
```

Listing 10: Definition der Grammatikregeln

4.1 Bsp IsType

Ein Unterschied zu dem Vorgaben sind die Funktionen „boolean isType(x, DATATYPE)“, „DATATYPE convert(DATATYPE, VALUE)“ und „int length(x)“. Um den Vorgang bei diesen Funtionen zu erläutern, wird der Syntaxbaum für die Beispielfuntion „isType()“, die in Abbildung 1 dargestellt wird, durchlaufen.

Um zu der Funktion „isType()“ zu kommen, muss der Syntaxbaum sehr tief durchlaufen werden. Als Beispielcode wurde folgender ausgesucht, wo die Variable y schon vorher einmal als 5 definiert wurde, was hier weggelassen wurde, um den Syntaxbaum nicht zu kompliziert zu machen:

```
    bool t3 = isType(int, y);
```

Listing 11: Beispielcode für isType()

In der Abbildung 1 ist der Syntaxbaum für den Beispielcode dargestellt. Der Baum zeigt welche Regeln wann abgeleitet werden.

Die Anfangsfunktion ist „ Program“. Danach wird die Funktion „VariableDeclaration“ aufgerufen, mit der eine Variable deklarieren kann. „VariableDeclaration“ besteht aus folgenden Terminalen und Nichtterminalen: „Datatype“ „Identifier“ „=“ „Expression“ „;“

Das Nichtterminal „Datatype“ wird aufgelöst zu dem Datentyp „bool“. Darauf wird das Nichtterminal „Identifier“ abgeleitet zu dem Terminal <IDENTIFIER>. In diesem Beispiel ist es der Identifier „t3“. Danach kommt das Terminal „=“. Das nächste Nichtterminal ist „Expression“, der eine Reihe von weiteren Nichtterminalen nach sich zieht. Die sind der Reihe nach „LogicalExpression“, „LogicalExpression“, „AdditiveExpression“, „MultiplicativeExpression“, „UnaryExpression“, „Atom“ und „IsType“. Der letzte Nichtterminal in der Reihe führt zu folgenden Terminalen und Nichtterminal: „isType“ „(“ „Datatype“ „,“ „Expression“ „)“ „;“

Der Nichtterminal „Datatype“, der schon einmal abgeleitet wurde, ist in diesem Fall „int“. Der zweite Nichtterminal der von dem Nichtterminal „IsType“ aufgelöst wird, ist „Expression“. Wie vorher leitet der Nichtterminal auch dieses Mal fast die gleiche Reihe von Nichtterminal ab. Der einzige Unterschied dabei ist, dass der Nichtterminal „Atom“ nicht den Nichtterminal „IsType“ auflöst sondern den Terminal <INTEGERLITERAL>. In diesem Fall ist das „y“. Zum Schluss kommt noch der Terminal „;“.

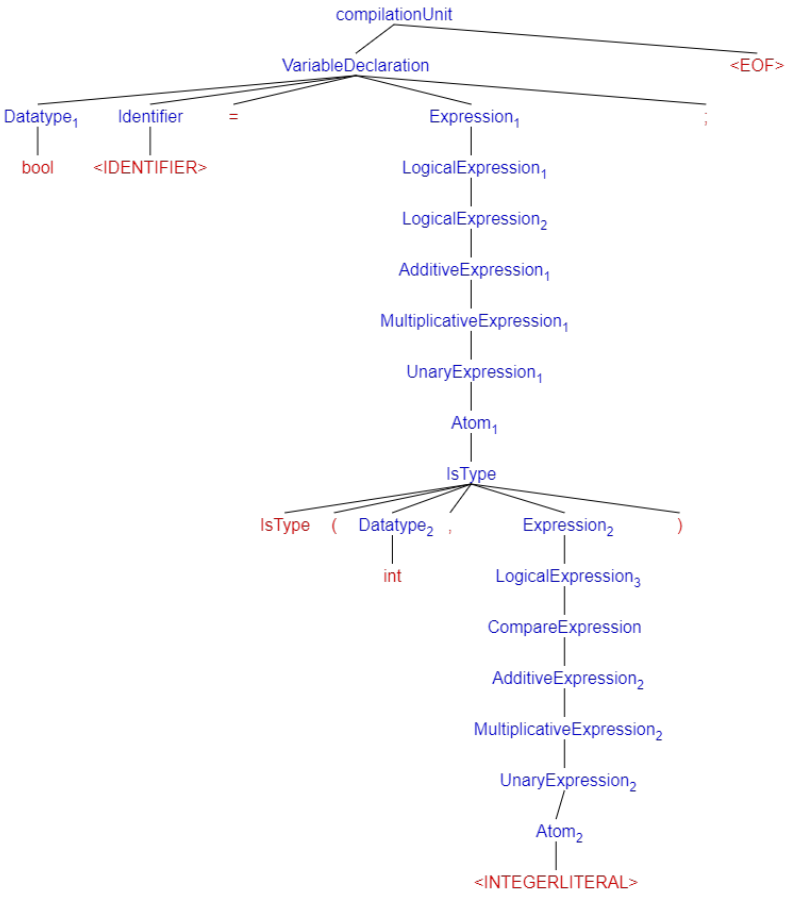

Abbildung 1: Syntaxbaum von der Funktion IsType()

Ähnlich wird auch bei der Funktion „Convert“ vorgegangen. Der einzige Unterschied besteht darin, dass nicht der Nichtterminal „IsType“ aufgelös wird, sondern der Nichtterminal „Convert“. Diese sind aber ähnlich aufgebaut. Der Unterschied ist, dass nicht das nicht das Terminal „isType“ sondern „convert“ verwendet wird.

Ebenfalls ähnlich aufgebaut ist die Funktion „Length“. Diese hat aber nicht nur einen anderen Terminal („length“ statt „isType“) sondern es wird auch der Datentype und somit das Komma nicht benötigt.

4.2 Bsp Array

Ein weiterer Unterschied zu den Vorgaben ist der Umgang mit Arrays. Der folgende Beispielcode zeigt diesen Unterschied:

```
string[] arr = ["a"];
set(a, 0, "b");
string str = get(a, 0); // == "b"
```

Listing 12: Beispielcode für den Umgang mit Arrays

In der Abbildung 2 ist der Anfang des Syntaxbaums abgebildet, der für die Darstellung des Beispielcodes für Arrays benötigt wird. Es wurde sich dafür entschieden diesen Syntaxbaum auzuteilen, da sonst dieser sehr groß und unübersichtlich geworden wäre. Die Anfangsfunktion ist hierbei ebenfalls „ Program“. Diese wird aber nicht nur einem Nichtterminal auf sondern drei aufgelöst. Dier erste ist „VariableDeclaration“. Danach folgen „Statement“ und „VariableDeclaration“.

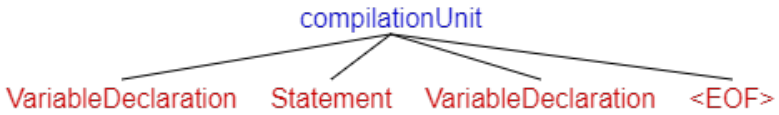

Abbildung 2: Syntaxbaum für den Umgang mit Arrays - Uberblick

Die Ableitung des ersten Nichtterminal „VariableDeclaration“ wird in der Abbildung 3 dargestellt. Der Nichtterminal wird so abgeleitet, dass die erste Zeile des Beispielcodes die Arraydeklaration am Ende herauskommt. Dieser Nichtterminal wird wie folgt abgeleitet: „Datatype“ „Identifier“ „=“ „Expression“ „;“

Als erstes wird der Nichtterminal „Datatype“ aufgelöst, der zu dem Datentyp „<ARRAY>“ wird. Nachdem der Nichtterminal „Identifier“ abgeleitet wird, die zu dem Terminal <IDENTIFIER> (hier „arr“) wird, kommt der Terminal „=“. Darauf wird die gleiche Reihe von Nichtterminal abgeleitet wie schon im vorherigem Beispiel (von „Expression“ bis „Atom“). Der Nichtterminal „Atom“ wird in diesem Fall zu dem Nichtterminal „ArrayExpression“, welche zu folgenden zwei Terminalen und einem Nichtterminal abgeleitet wird: „[“ „Expression“ „]“ Auf den zweiten Nichtterminal „Expression“ folgt wieder die gleiche Reihe von Nichtterminalen, aber „Atom“ wird dieses Mal zu „Identifier“ abgeleitet, was wiederum zum Terminal <IDENTIFIER> wird.

Zum Schluss kommt noch das Terminal „;“.

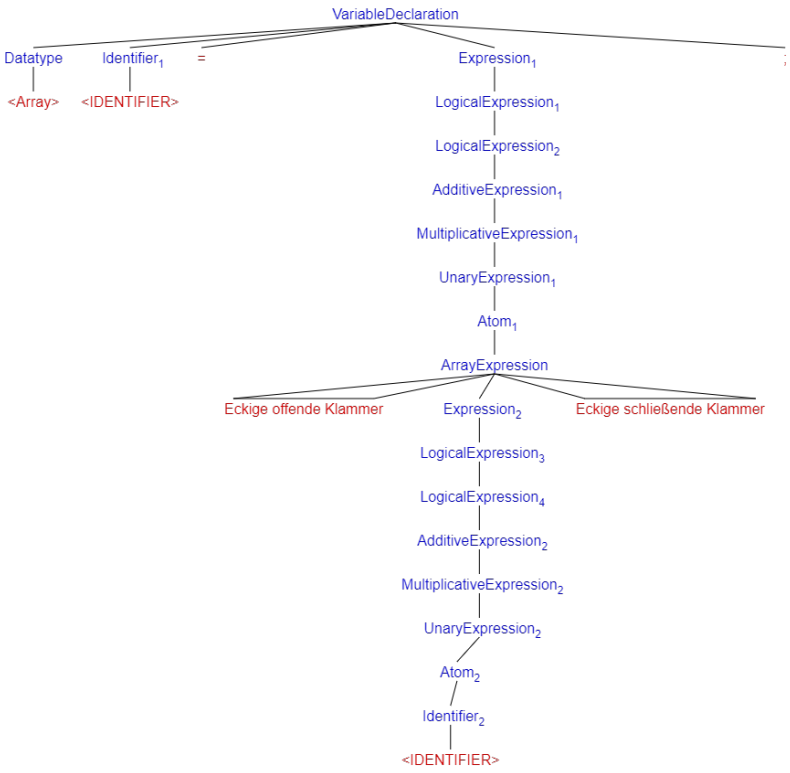

Abbildung 3: Syntaxbaum für die Deklaration von Arrays

Die Ableitung des zweiten Nichtterminal „Statement“ wird in der Abbildung 4 dargestellt. Der Nichtterminal wird nun so abgeleitet, dass die „Set“-Funktion für Arrays herauskommt.

Der Anfang mach der Nichtterminal „Statement“. Diese wird zu dem Nichtterminal „Set“ und dem Terminal „;“. Der Nichtterminal wird wie folgt abgeleitet: „set“ „(“ „Expression“ „,“ „Expression“ „,“ „Expression“ „)“ Die drei Nichtterminalen „Expression“ führt wieder durch eine Reihe von Nichtterminalen zu „Atom“. Das erste „Atom“ wird abgeleitet zu „Identifier“, was wiederum zum Terminal <IDENTIFIER> wird, hier „a“. Das zweite wird zum Terminal <INTEGER_LITERAL>, hier 0, und der letzte zum <STRING_LITERAL>, hier „b“.

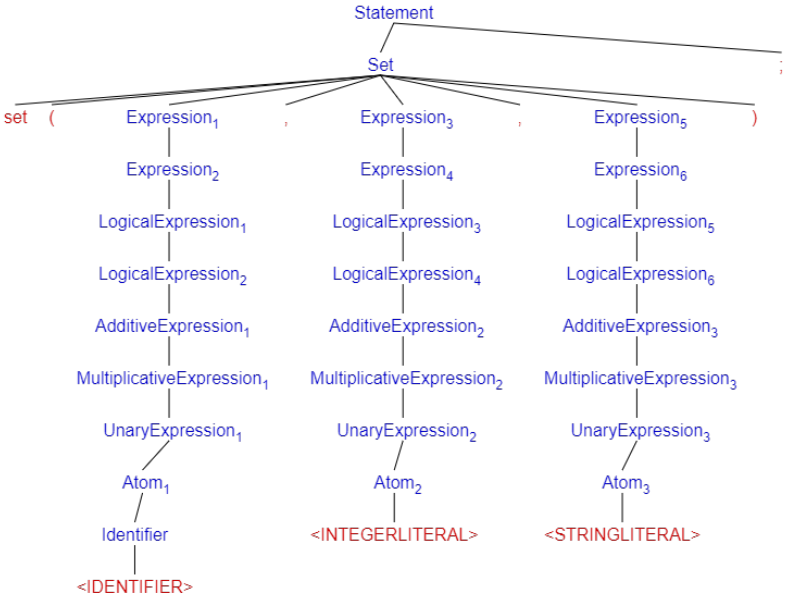

Abbildung 4: Syntaxbaum für die Funktion set

Die Abbildung 5 zeigt den Syntaxbaum für die letzte Zeile des Beispielcodes, die „get“-Funktion.

Der zweite Nichtterminal „VariableDeclaration“ wird ähnlich abgeleitet wie die erste. Es gibt nur zwei Unterschiede. Der erste ist, dass „Datatype“ zu „string“ abgeleitet wird. Der andere Unterschied ist, dass nach der Ableitungsreihe von „Expression“ bis „Atom“ das Nichtterminal „Get“ kommt, die folgendermaßen abgeleitet wird: „get“ „(“ „Expression“ „,“ „Expression“ „)“ Die erste „Expression“-Nichtterminalsreihe endet bei dem Nichtterminal „Identifier“, die zu dem Terminal „<IDENTIFIER>“ wird, hier „a“. Die zweite Reihe endet bei dem Nichtterminal „Atom“, welche wiederum zum Terminal „<INTEGERLITERAL>“, hier 0.

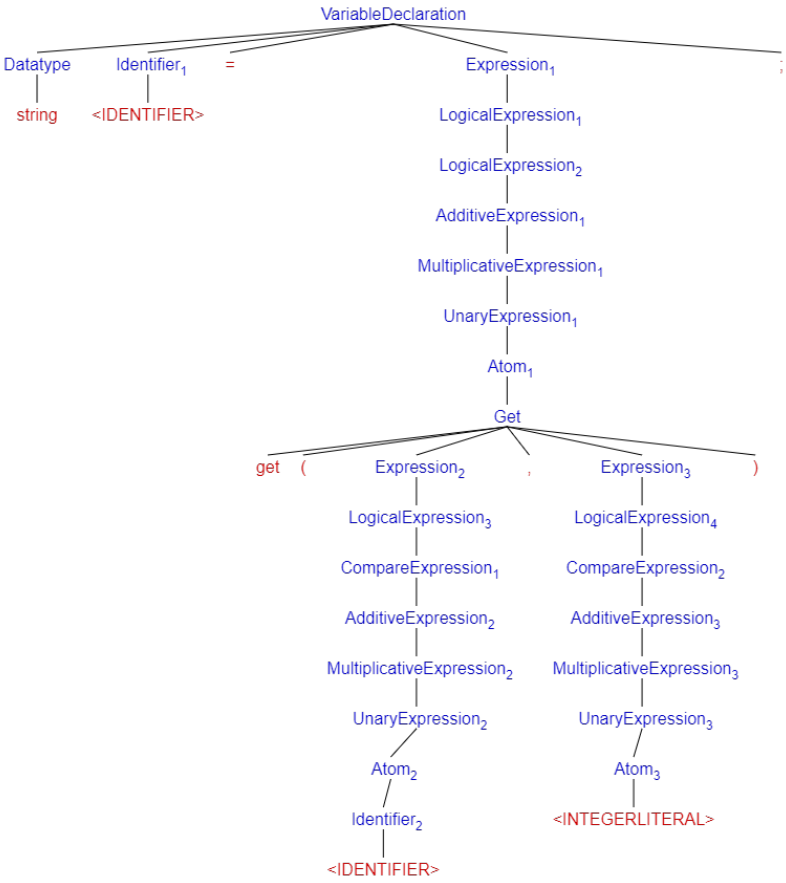

Abbildung 5: Syntaxbaum für die Funktion get

5 Semantische Regeln

Semantische Regeln sind notwendig, da nicht alle Programmierspracheneigenschaften in einer Grammatik ausdrückbar ist. So muss eine Variable vor der Verwendung deklariert, initialisiert und den richtigen Typ besitzen, z.B. für eine Multiplikation mit einer anderen variablen. Die Tests, die dafür benötigt werden, können mit einer (kontextfreien) Grammatik nicht unbedingt dargestellt werden. Die Datentypüberprüfung ist die hauptsächtliche Aufgabe der semantischen Analyse. Außerdem wird geprüft, ob eine Funktion die richtige Anzahl an Parametern und, wenn ein Rückgabewert vorhanden ist, ob diser den richtigen Rückgabetyp hat. Zudem werden weitere Informationen für die folgenden Phasen gesammelt. In diesem Kapitel werden die Regeln für die semantische Analyse beschrieben.

Die erste Regel ist dabei, dass bestimmte Datentypen in andere umgewandt werden können. Die Reihenfolge dafür ist folgende:

```
errorType -> boolType -> charType -> intType -> doubleType
```

```
-> stringType -> voidType
```

Somit kann jeder boolType zu einem String werden, aber nicht jeder stringType zu einem boolType. Ein Beispiel dafür ist das Wort 'true'. Es ist sowohl ein String, als auch die Zahl 1 als intType oder 1.0 als doubleType.

Außerdem können in alle Datentypen mathematisch bearbeitet werden. Die Addition kann zum Beispiel in genau der gleichen Reihenfolge durchgeführt werden wie das Casting, da die Datentypen dann zu dem Unteren Type gecastet und dann addiert werden. Zum Beispiel wenn ein Integer auf einem String addiert wird, kommt dabei ein String heraus, der erst den String und dahinter den Interger enthält (bsp: "Hallo"+ 5 == "Hallo5"). Das gleiche gilt für die Substraktion, Multiplikastion und Division von int- und doubleType. Die Addition von zwei Arrays wird so vorgenommen, dass die beiden zusammengenommen werden (bsp: [ä", "b", "c"] + ["c", "d", ë", "f"] == [ä", "b", "c", "d", ë", "f"]). Bei der Subtraktion von Arrays werden die überschneidenden Elemente aus dem ersten Array entfernt (bsp: [ä", "b", "c", "d", ë", "f"] - [ä", "b", "c"] == ["d", ë", "f"]).

Die Symboltabelle besteht aus zwei Bestandteilen. Zum Einen einer HashMap und zum Anderen einer "ParentSymboltabelle. In der HashMap werden alle Dekerationen gespeichert. Dazu gehören Variablendeklerationen, Funktionsdeklerationen und als besondere Variablendekleration die Parameterdekleration. Um Namensräume zu realisieren verfügt die Symboltabelle um die bereits genannte "ParentSymboltabelle. Jeder Namensraum verfügt über eine eigene Symboltabelle. Die "ParentSymboltabelle ist dabei die Symboltabelle des darüberliegenden Namensraumes. Die Symboltabelle des obersten und damit globale Namensraums verfügt über keine "ParentSymboltabelle. Hinzugefügt werden Deklerationen immer in der aktuellen Symboltabelle. Vor dem tatsächlichen Hinzufügen zur HashMap wird noch überprüft, ob der Identifier bereits vergeben ist. Ist dies der Fall, wird ein Fehler erzeugt. Anderenfalls wird die Dekleration zur Symboltabelle hinzugefügt. Beim Auslesen der Symboltabelle wird zuerst überprüft, ob das Element in der aktuellen Symboltabelle existiert. Ist dies der Fall, wird das Element zurückgegeben. Ist dies nicht der Fall, wird rekursiv mit der "ParentSymboltabelle fortgefahren. Falls keine "ParentSymboltabelle vorhanden ist, befinden wir uns im globalen Namensraum. Ist auch hier das Element nicht vorhanden wird null zurückgegeben.

6 Aufruf des Compilers

Beim Aufruf des Programms ist zu beachten, dass der erste Parameter einen gültigen Pfad zu einer NewAwk-Quellcodedatei enthält. Darauf können beliebig viele Parameter folgen, auf welche im Programm über get(args, i) zugegriffen werden kann.

Programmaufruf allgemein: java -jar newawk.jar QUELLTEXTDABEI (PARAMETER)*

Programmaufruf Beispiel:

java -jar newawk.jar source.na input1.txt input2.txt out.txt