Ausarbeitung

Ausarbeitung

Compilerbau

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

von
Alexander Weidemann
Bijan Riesenberg
Johanna Kraken
Matr.-Nr. 7206374

Betreuer: Prof. Dr. Robert Rettinger

Dortmund, October 8, 2020

Contents

1 Introduction

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

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

2 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, die als erstes in Größer- und Kleinerzeichen definiert werden. Danach wird mit der Häuftigkeit der eckigen Klammern die Dimension des Arrays angezeigt.

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

Danach 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 Operatoren ~~von~~ UND (doppelten Und-Zeichen ”) und ODER (doppelten wargerechtem Stricht ’|’).

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

```
    "!="> 
|   <BIN_GATE_OPERATOR: "||" | "&&">
 }
```

Die nächsten Token, die definiert werden, sind die Keywords. Die Gruppe beinhaltet das return-Wort, das Komma, hier als ENUM_SEPERATOR bezeichnet, und das Semikolong, hier als EXPRESSION_TERMINATOR 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 Add-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: "}">
 }
```

Die Selector classes Tokengruppe werden als nächstes definiert. Diese sind punct, graph, lower, alpha, alnum, print, cntrl, space, blank und digit, die ebenfals jeweils so aber in Großbuchzeichen bezeichnet werden.

```
// Selector classes
TOKEN : {
    < SMART_SWITCH_SELECTOR: "punct"
|   "graph" // Letters, numbers and punctuation
```

```
|   "lower" // Lowercase Letters
|   "alpha" // Letters
|   "alnum" // Alphanumerics
|   "print" // Letters, numb ers, punctuation and whitespace
|   "cntrl" // Control characters
|   "space" // Space characters
|   "blank" // Space and tab
|   "digit" >// Digits
 }
```

Als nächstes werden die Literale definiert. Hier werden Integer, Dezimalzahlen, Floats, Charakter, Strings und Booleans definiert und dargestellt wie diese aufgebaut werden. Beispielsweise ist ein BOOLEAN_LITERAL nur "true" oder "false", ein INTEGER_LITERAL ist DECIMAL_LITERAL und ein DECIMAL_LITERAL besteht immer aus einem oder keinem Plus- oder Minuszeichen, einer Zahl von 1 bis 9 und danach folgt eine unbestimmte Anzahl von Zahlen im Bereich von 0 bis 9. Der FLOATING_POINT_LITERAL bestehlt aus einem oder keinem Plus- oder Minuszeichen, einem oder mehreren Zahlen zwischen 0 und 9, dem Punkt und keinem oder mehreren Zahlen zwischen 0 und 9. 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">
 }
```

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

```
// 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" )>
 }
```

Als letztes werden die Identifier definiert. In der Gruppe befinden sich #LETTER, der ein Groß- oder Kleinbuchstaben ist, und #DIGIT, das eine Zahl von 0 bis 9 ist. Der IDENTIFIER beginnt immer mit einem LETTER und hat dann eine unbestimmte Kombination und Anzahl von #LETTER und #DIGIT.

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

```
}
```

3 Grammatik

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

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

Die verwendetetn Funtionen in der Grammatik sind folgende:
N = { compilationUnit, Statement, VariableDeclaration, Block, FunctionBlock, Expression, LogicalExpression, CompareExpression, AdditiveExpression, MultiplicativeExpression, UnaryExpression, Atom, FunctionCall, Identifier, ReturnStatement, IsType, Convert, Length, red, write, Datatype, ReturnDatatype }
(Funktionennamen in NewAwk.jjt)

Die verwendeten Token werden in T definiert und sind folgende:
int, double, float, char, bool, string, type, void, null, @, return, ',', ';', if, else, while, foreach, in, LETTER, DIGIT, am, pm, ':', IsType, Convert Length, (,), +, -, *, /, =, >, <, |, &, !, %, punct, graph, lower, alpha, alnum, print, cntrl, space, blank, digit, EOF, read, write

Die Startfunktion S ist compilationUnit.

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

```
R = {
    compilationUnit ->                  ( 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()
    ")"
  WriteStatement ->                "write" "("
    Expression() ( "," Expression() )* ")" ";"
  Get ->                           "get" "(" Expression()
    "," Expression() ")"
  Set ->                           "set" "(" Expression()
    "," Expression() "," Expression() ")" ";"
  // Datatype
  Datatype ->                      "double" | "int" |
    "char" | "bool" | "string" | <ARRAY>
}
```

3.1 Bsp IsType

Ein Unterschied zu dem Vorgaben ~~des Professors~~ sind die Funktionen "IsType(Datatype, Expression)", "Convert(Datatype, Expression)" und "Length(Expression)". Um den Vorgang bei diesen Funtionen zu erläutern ,wird der Syntaxbaum für die Beispielfuntion "IsType(Datatype, Expression)", die in Figur 1 dargestellt wird, durchlaufen.

Um zu der Funktion "IsType(Datatype, Expression)" zu kommen, muss der Syntaxbaum sehr tief durchlaufen werden. Als Beispielcode wurde folgender ausgesucht:

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

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

Die Anfangsfunktion ist "compilationUnit". Danach wird die Funktion "VariableDeclaration" aufgerufen, mit der, ~~wie der Name vermuten lässt~~, eine Variable deklarieren kann. Dafür werden nacheinander folgende Funktionen ausgeführt oder Token ausgegeben???: "Datatype" "Identifier" "=" "Expression" ";"

Als erstes wird die Funktion "Datatype" ausgeführt, die zu dem Datentyp "bool" führt. Darauf wird die Funktion "Identifier" ~~aufgerufen, die zu dem Token <IDENTIFIER> führt~~. In diesem Beispiel ist es der Identifier "t3". Danach kommt das Token "=". Die nächste Funktion, die aufgeführt wird, ist "Expression", die eine Reihe von weiteren Funktionen nach sich zieht. Die sind der Reihe nach "LogicalExpression", "LogicalExpression", "AdditiveExpression", "MultiplicativeExpression", "UnaryExpression", "Atom" und "IsType". Die letzte Funktion in der Reihe führt zu folgenden Token und Funktionen: "IsType" "(" "Datatype" "," "Expression" ")" ";"

Die Funktion "Datatype", die schon einmal ausgeführt wurde, ist in diesem Fall "int". Die zweite Funktion die von der Funktion "IsType" ausgeführt wird, ist "Expression". Wie vorher führt sie Funktion auch dieses Mal fast die gleiche Reihe von Funtionen. Der einzige Unterschied dabei ist, das die Funktion "Atom" nicht die Funktion "IsType" ausführt sondern zu dem Token <INTEGERLITERAL> führt. In diesem Fall ist das "y". Zum Schluss kommt noch das Token ";".

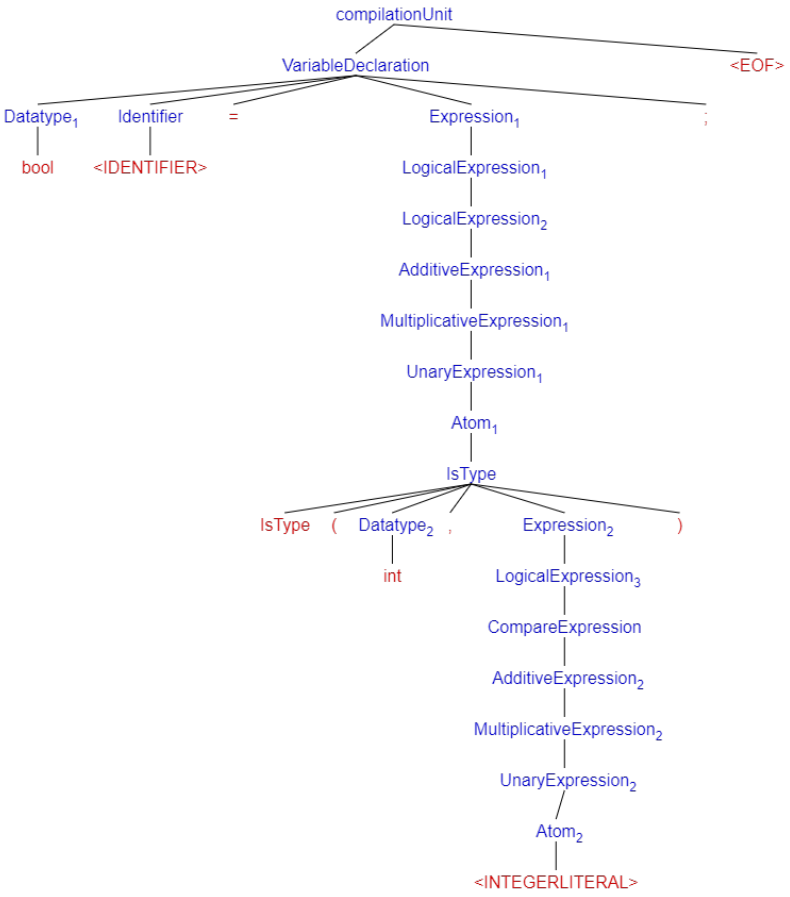

Figure 1: Syntaxbaum von der Funktion IsType()

Ähnlich wird auch bei der Beispielcode Funtion “Convert” vorgegangen. Der einzige Unterschied besteht darin, dass nicht die Funktion “IsType” ausgeführt wird sondern die Funktion “Convert”. Diese sind aber ähnlich aufgebaut. Der Unterschied ist, dass nicht das nicht der Token “IsType” sondern “Convert” verwendet wird.

Ebenfalls ähnlich aufgebaut ist die Funktion “Length”. Diese hat aber nicht nur einen anderes Token (“Length” statt “IsType”) sondern es wird auch der Datentype und somit das Komma nicht benötigt.

3.2 Bsp Array

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

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

In der Figur 2 ist der Anfang des Syntaxbaums abgebildet, der für die Darstellung des Beispielcodes für Arrays benötigt wird. Es wurde sich dafür endschieden diesen Syntaxbaum auzuteilen, da sonst dieser sehr groß und unübersichtlich geworden wäre. Die Anfangsfunktion ist hierbei ebenfalls “compilationUnit”. Diese Ruft aber nicht nur eine Funktion auf sondern drei. Die erste ist “VariableDeclaration”. Danach folgen “Statement” und “VariableDeclaration”.

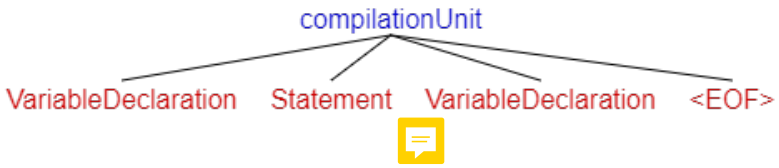

Figure 2: Syntaxbaum für den Umgang mit Arrays - Uberblick

Die Ableitung der ersten Funktion “VariableDeclaration” wird in der Figure 3 dargestellt. Die Funktion wird so abgeleitet, dass die erste Zeile des Beispielcodes die Arraydeklaration am Ende herauskommt. Diese Funktion wird wie folgd abgeleitet: “Datatype” “Identifier” “=” “Expression” “;”

Als erstes wird die Funktion “Datatype” ausgeführt, die zu dem Datentyp “<ARRAY>” führt. Nachdem die Funktion “Identifier” aufgerufen wird, die zu dem Token <IDENTIFIER> (hier “arr”) führt, kommt der Token “=”. Darauf wird die gleiche Reihe von Funktionen ausgeführt wie schon im vorherigem Beispiel (von “Expression” bis “Atom”). Die Funktion “Atom” führt in diesem Fall zu der Funktion “ArrayExpression”, welche zu folgenden zwei Token und einer Funktion abgeleitet wird: “[” “Expression” “]” Auf die zweite Funktion “Expression” folgt wieder die gleiche Reihe von Funktionen, aber “Atom” wird dieses Mal zu “Identifier” abgeleitet, was wiederum zum Token <IDENTIFIER> wird.

Zum Schluss kommt noch das Token “;”.

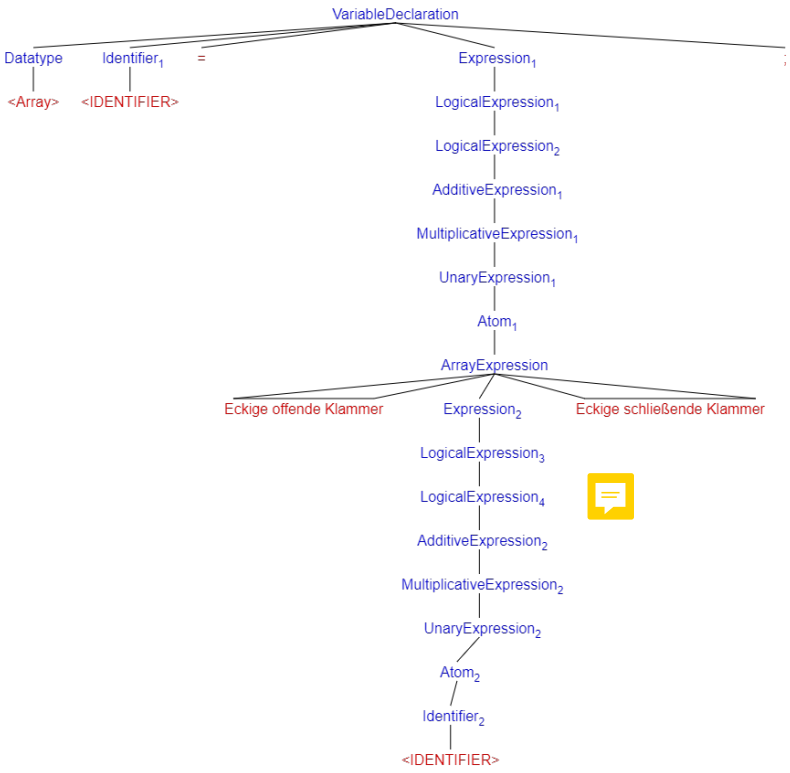

Figure 3: Syntaxbaum für die Deklaration von Arrays

Die Ableitung der zweiten Funktion “Statement” wird in der Figure 4 dargestellt. Die Funktion wird nun so abgeleitet, dass die “Set”-Funktion für Arrays herauskommt.

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

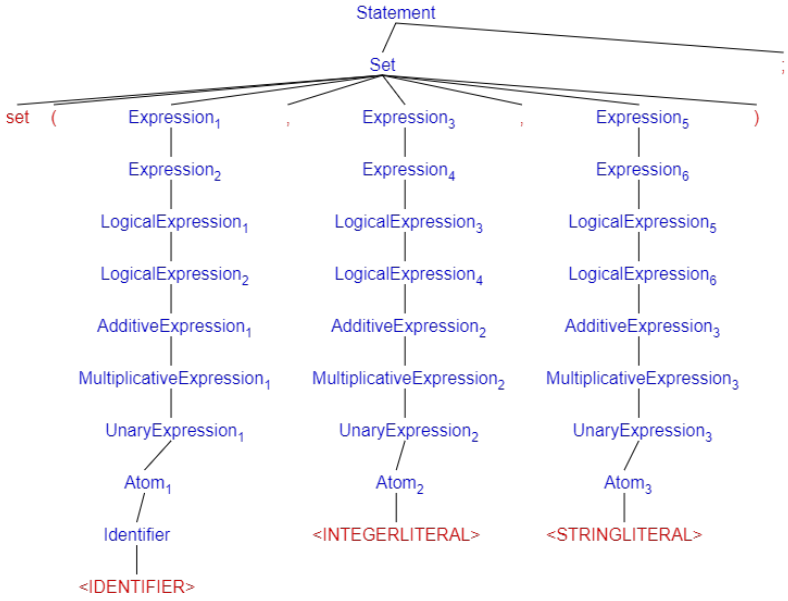

Figure 4: Syntaxbaum für die Funktion set

Die Figure 5 zeigt den Syntaxbaum für die letzte Zeile des Beispielcodes, die "get"-Funktion.

Die zweite Funktion "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" die Funktion "Get" kommt, die folgendermaßen abgeleitet wird: "get" "(" "Expression" "," "Expression" ")" Die erste "Expression"-Funktionsreihe endet bei der Funktion "Identifier", die zu dem Token "<IDENTIFIER>" wird, hier "a". Die zweite Reihe endet bei der Funktion "Atom", welche wiederum zum Token "<INTEGERLITERAL>", hier 0.

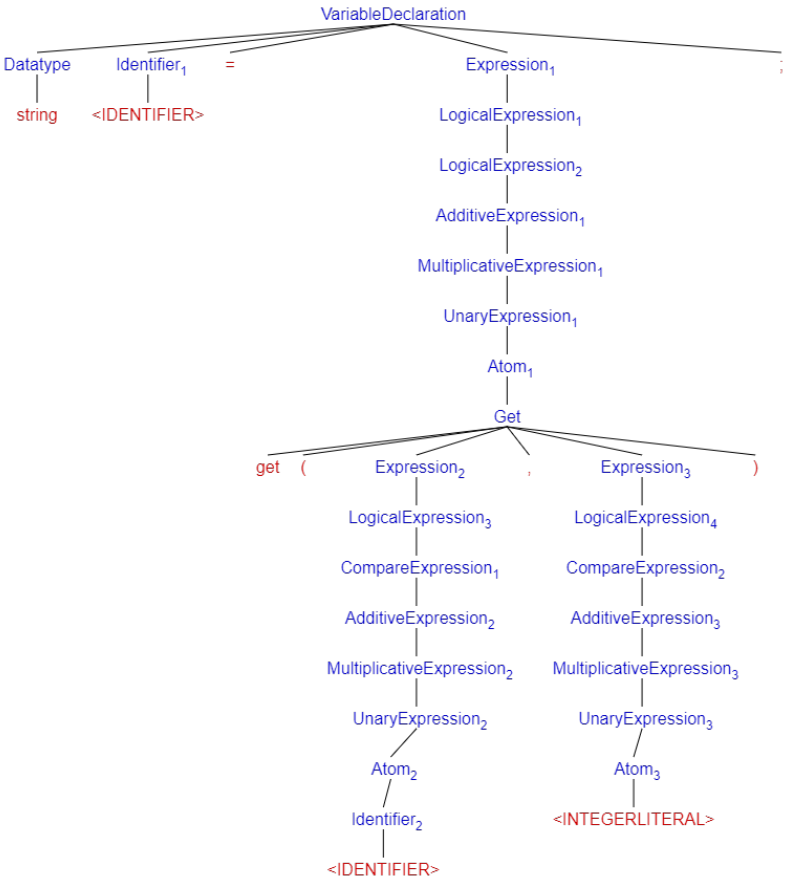

Figure 5: Syntaxbaum für die Funktion get

4 Semantische Regeln

// TODO Typprüfung (Semantische Analyse?)

Semantische Regeln sind notwendig, da nicht alle Programmierspracheneigenschaften in einer Grammatike 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 haubtsächtliche Aufgabe der semantischen Analyse. Außerdem wird geprüft, ob eine Funktion die richtige Anzahl an Parametern und, wenn ein Rückgabewert forthanden ist, ob diser den richtigen Rückgabetyp hat. Zudem werden weitere Informationen für die weiteren 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 nicht alle Datentypen mathematisch Bearbeitet werden. Die Addition ist zum Beispiel kann in genau der gleichen Reihenfolge durchgeführt werden wie das Casting, da die Datentypen dann zu dem unsteren type gecastet und dann addiert werden. Das gleiche gilt für die Substraktion, Multiplikastion und Division von int- und doubleType.

??? Symboltabelle ???

5 Aufruf des Compilers

//TODO