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 7, 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.

Als erstes werden die Token der Types definiert. Diese sind int, double, char, bool (boolean), string, void und Array. Das Array kann Integer, Double, Charakter, Boolean oder String enthalten.

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

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. Die letzte Gruppe in diesem Bereich ist die BIN_GATE_OPERATOR. In dieser gibt es nur zwei Operatoren von UND und ODER.

```
\(
// 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 und das Semikolong. Darunter sind die Keywords die für Funktionen und Schleifen benötigt werden. Das sind if, else, while, foreach und in. Außerdem beinhaltet diese Gruppe auch das Add-Zeichen unter dem Namen SMARTSWITCH.

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

Die Selector classes werden als nächstes definiert. Diese sind punct, graph, lower, alpha, alnum, print, cntrl, space, blank und digit.

```
\(
// 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. Außerdem werden die Zeichen definert um die Literale zu bearbeiten, indem diese z.B. mit anderen addiert oder subtrahiert werden.

```
\(
// Literals\\
TOKEN :  \{\\
    "(" | ")" | "!" | " \{" | " \}" | "--" | "++" | "+" |
       "-" | "*" | "/" | "\%" | ":"\\
|   <INTEGER\_LITERAL: <DECIMAL\_LITERAL>>\\
|   <\#DECIMAL\_LITERAL: ("+" | "-")? ["1"-"9"]
   (["0"-"9"])*>\\
|   <FLOATING\_POINT_LITERAL: ("+" | "-")? (["0"-"9"])+ "."
   (["0"-"9"])*>\\
|   <CHARACTER\_LITERAL: "\textbackslash'"
   (\~["\textbackslash'", "\textbackslash\textbackslash",
   "\textbackslash n","\textbackslashr"] | "\textbackslash"
   (["n","t","b","r","f", "\textbackslash\textbackslash",
   "\textbackslash'", "\textbackslash""] | ["0"-"7"]
```

```
   (["0"-"7"])? | ["0"-"3"] ["0"-"7"] ["0"-"7"]))
   "\textbackslash'">\\
|   <STRING\_LITERAL: "\textbackslash"" (
   \~["\textbackslash"", "\textbackslash\textbackslash",
   "\textbackslash n","\textbackslash r"] |
   "\textbackslash\textbackslash" ( ["n", "t", "b", "r",
   "f", "\textbackslash\textbackslash", "\textbackslash'",
   "\textbackslash""] | ["0"-"7"] (["0"-"7"])?  | ["0"-"3"]
   ["0"-"7"] ["0"-"7"] | ( ["\textbackslash
   n","\textbackslash r"] | "\textbackslash r
   \textbackslash n")))* "\textbackslash"">\\
|   <TYPE\_LITERAL: ">" (
   \~["<",">","\textbackslash\textbackslash","\textbackslash
   n", "\textbackslash r"] | "\textbackslash\textbackslash"
   ( [">", "<", "n", "t", "b", "r", "f",
   "\textbackslash\textbackslash"] | ["0"-"7"] (["0"-"7"])?
    | ["0"-"3"] ["0"-"7"] ["0"-"7"] | ( ["\textbackslash
   n", "\textbackslash r"] | "\textbackslash r
   \textbackslash n")))* "<">\\
|   <BOOLEAN\_LITERAL: "true" | "false">\\
 \}\\
\)
```

Dann werden Built-in Funktionen definiert. Darunter befinden sich CONTERT_FUNCTION ("convert"), CHECKTYPE_FUNCTION ("isType"), LENGTH_FUNCTION ("length"), READ_FUNCTION ("read") und WRITE_FUNCTION ("write").

```
\(
// Built-in Functions\\
TOKEN :  \{\\
    < CONTERT\_FUNCTION: ( "convert" )>\\
|   < CHECKTYPE\_FUNCTION: ( "isType" )>\\
|   < LENGTH\_FUNCTION: ( "length" )>\\
|   < READ\_FUNCTION: ( "read" )>\\
```

```
|    < WRITE\_FUNCTION: ( "write" )>\\
 \}\\
\)
```

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 )

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()
       | Expression() ";"
    VariableDeclaration ->              Datatype() Identifier()
       ["=" Expression()] ";"
    Block ->                            "{" ( Statement() |
       VariableDeclaration() )* "}"
    FunctionBlock ->                    "{" ( Statement() |
       VariableDeclaration() )* ( ReturnStatement() )? "}"
    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() ")" | IsType() | Convert() | Length() |
   read() | write()
FunctionCall ->                  Identifier() "("
   [LogicalExpression() ( "," LogicalExpression() )*] ")"
Identifier ->                    <IDENTIFIER>
// Function
ReturnStatement ->               "return"
   [LogicalExpression()] ";"
FunctionDeclaration ->           ReturnDatatype()
   Identifier() "(" ( Datatype() Identifier() )? ( ","
   Datatype() Identifier() )* ")" FunctionBlock()
ForeachStatement ->              "foreach" "("
   Identifier() "in" Expression() ")" Statement()
WhileStatement ->                "while" "("
   LogicalExpression() ")" Statement()
IfStatement ->                   "if" "("
   LogicalExpression() ")" Statement() ( "else"
   IfStatement() | "else" Statement() )
// Integrated Funtions
IsType ->                        "isType" "(" Datatype()
   "," Expression() ")"
Convert ->                       "convert" "("
   Datatype() "," Expression() ")"
Length ->                        "length" "("
   Expression() ")"
read ->                          "read" "(" Expression()
   ")"
write ->                         "write" "("
```

```
        Expression() "," Expression() ")"
    // Datatype
    Datatype ->                          "double" | "int" |
        "char" | "bool" | "string" | <ARRAY>
    ReturnDatatype ->                    Datatype() | "void"
}
```

## 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 Regelbaum für die Beispielfuntion “IsType(Datatype, Expression)”, die in Figur 1 dargestellt wird, durchlaufen.

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

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

In der Figur 1 ist der Regelbaum für den Beispielcode dargestellt. Der Baum zeigt welche Regeln wann durchlaufen 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 “;”.

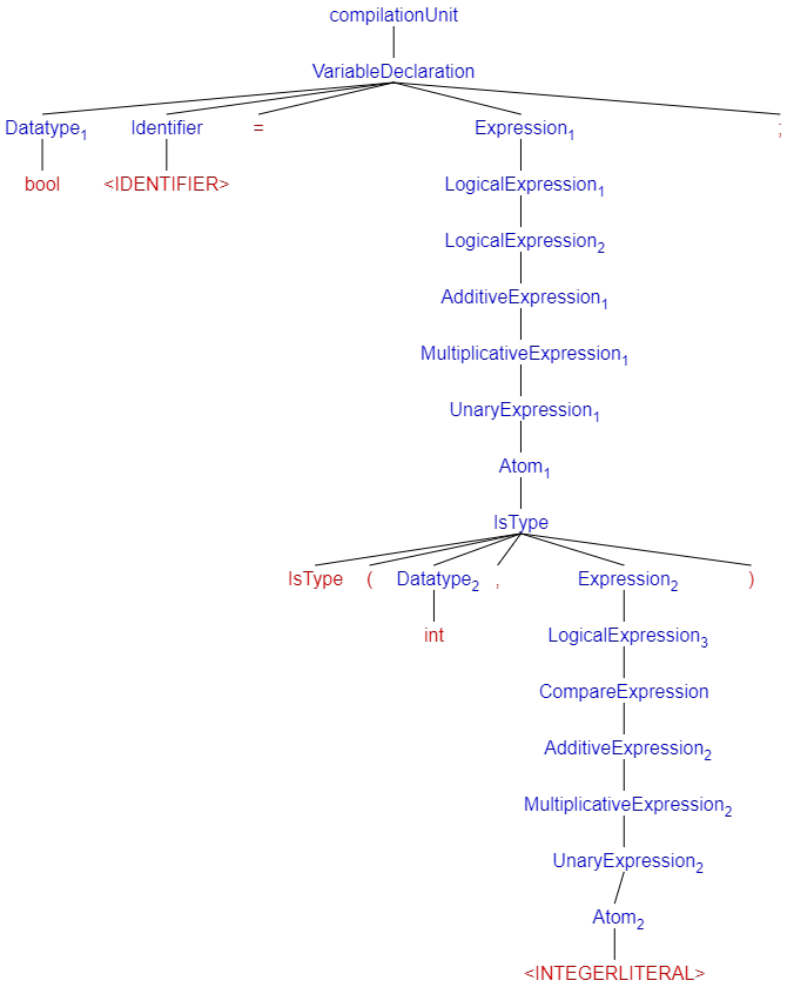


Figure 1: Regelbaum von der Funktion IsType()

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.

# 4 Semantische Regeln

// TODO Typprüfung (Semantische Analyse?)

# 5 Aufruf des Compilers

//TODO