Hochschule Aalen

Projektbericht
Studiengang Medieninformatik

---

expressionviewer

von

Laurin Agostini

60526

Betreuender Professor:	Prof. Dr. Winfried Bantel
Einreichungsdatum:	20. Februar 2020

Eidesstattliche Erklärung

Hiermit erkläre ich, **Laurin Agostini**, dass ich die vorliegenden Angaben in dieser Arbeit wahrheitsgetreu und selbständig verfasst habe.

Weiterhin versichere ich, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben, dass alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Ort, Datum Unterschrift (Student)

Kurzfassung

In diesem Bericht geht es um die Entwicklung eines Werkzeugs zur Erstellung von Syntaxbäumen aus einem Ausdruck der Programmiersprache C. Dies soll vorallem Programmieranfängern helfen, sich schnell einen Überblick über komplexere Ausdrücke zu verschaffen.

Zur Erstellung dieses Werkzeugs wurde wiederum selbst hauptsächlich C als Programmiersprache benutzt, aber auch der typische Web-Stack (HTML, CSS, PHP, JavaScript) zur Darstellung einer Eingabemaske im Web. Ein großer Teil des Berichts besteht aber aus der Erläuterung der Implementierung des Kommandozeilenprogramms **c-nodes**, welches auch das primäre Ergebnis dieser Arbeit darstellt. Somit wurde das eigentliche Ziel der Arbeit erreicht, jedoch fehlt noch eine Evaluation durch die Programmieranfänger, welche zeitlich nicht mehr möglich war.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

z.B.	zum Beispiel	1
d.h.	das heißt	2
AST	Abstract syntax tree	8
JSON	JavaScript Object Notation	4

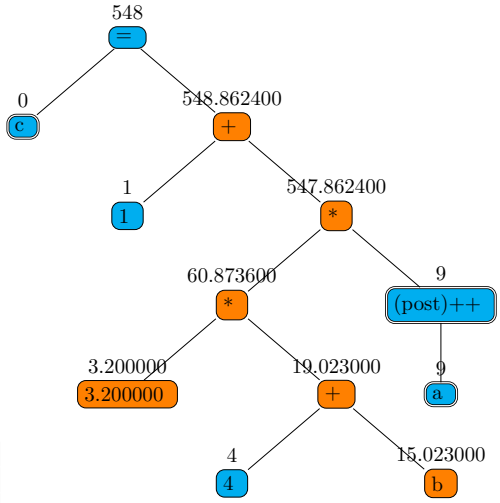
1 Einleitung

1.1 Motivation

Die Motivation für die Arbeit war es, ein Werkzeug für Programmieranfänger zu erstellen, damit diese sich Ausdrücke der C89 / ANSI C Programmiersprache grafisch darstellen können. Dies soll helfen das Verständnis für den Aufbau und die Abfolge der einzelnen Operationen innerhalb eines Ausdrucks zu verbessern. Die meisten der Anfänger dürften ein intuitives Verständnis für die Abhängigkeiten bei arithmetischen Operationen (+, -, *, /) und auch bei der Priorisierung von geklammerten Teilausdrücken mitbringen. Jedoch gibt es eine Vielzahl von spezifischen Operationen in Programmiersprachen, deren Auswirkung und Priorität erst gelernt werden muss. Auch die Unterscheidung zwischen Ganz- und Kommazahlen dürfte viele Anfänger erstmal verwirren, insbesondere bei den ersten (Ganzzahl-)Divisionen (zum Beispiel (z.B.) *6 / 4 == 1*) oder der Kombination von Ganz- und Kommazahlen (z.B. *6 / 4.0 == 1.5*). Relativ früh werden daher in der Vorlesung *Programmieren in C* Syntaxbäume eingeführt und den Studierenden beigebracht, einen Ausdruck in einen Syntaxbaum umzuwandeln. Um ein grundlegendes Verständnis für Aufbau eines Ausdrucks zu erlangen, mag das händische Umwandeln in einen Syntaxbaum sehr hilfreich sein, wird jedoch bei komplexeren Ausdrücken und insbesondere bei der expliziten Markierung von Datentypen und der Auswertung aller Zwischenwerte schnell zur Fleißarbeit. Hier soll das im Laufe der Projektarbeit geschriebene Werkzeug aushelfen und eine Unterstützung für das Lernen der Grundlagen des Programmierens sein. Das Werkzeug kann zudem auch zur Fehlersuche innerhalb von Ausdrücken benutzt werden, wenn diese sich anders verhalten als vom Programmierer vorgesehen.

1.2 Problemstellung und -abgrenzung

Als Aufgabenstellung wurde klar die Erstellung einer Webapplikation zur Eingabe eines C89 / ANSI C-Ausdrucks, welcher dann in einen durch Typinformationen angereicherten Syntaxbaum umgewandelt und dargestellt wird, festgelegt. Sonstige Ideen, wie die Bearbeitung des enstandenen Syntaxbaums und Konvertierung dieses zurück in einen Ausdruck, wurden angesprochen, sind aber nicht Teil der Problemstellung dieser Projektarbeit. Auch wurde festgelegt, dass nur die Datentypen *int*



Name	Value	Type	const?
a	9	int	0
b	15.023000	double	1
c	0	int	0

Name	Value	Type	const?
a	10	int	0
b	15.023000	double	1
c	548	int	0

Abbildung 1.1: Beispielausgabe in Tikz

(signed) und *double* berücksichtigt werden sollen. Auftretene Fehler bei der Erstellung oder Auswertung des Syntaxbaums sollen möglichst präzise dargestellt werden. So sollen Fehler bei der Erstellung einen Hinweis auf das fehlerhafte Zeichen oder Token geben. Bei Fehlern in der Auswertung (z.B. beim Versuch einer konstanten Variablen einen neuen Wert zuzuweisen) sollen diese im entsprechenden Knoten des Syntaxbaums mit einer aussagekräftigen Fehlermeldung markiert werden. Das Werkzeug soll aber ausdrücklich keine selbstständigen „Korrekturen“ am eingegeben Ausdruck vornehmen.

1.3 Ziel der Arbeit

Es soll ein Werkzeug für Programmieranfänger erstellt werden, womit diese einen C89 / ANSI C Ausdrucke analysieren können. Das Werkzeug soll den Ausdruck in einen Syntaxbaum mit Typinformationen umwandeln, das heißt (d.h.) für jeden Knoten im Syntaxbaum soll der Datentyp und der aktuelle Wert angezeigt werden. Außerdem sollen Fehler und ihre Folgen im Syntaxbaum ausdrücklich markiert und beschrieben werden.

1.4 Vorgehen

Der Großteil der Arbeit wird voraussichtlich in die Entwicklung des Kommandozeilenprogramms zur Verwertung der Ausdrücke gehen. Das Kommandozeilenprogramm soll eigenständig benutzt werden können um einerseits von anderen Programmen und Interfaces angesprochen werden zu können. Dazu soll es eine relativ generische Ausgabe in die Datenformate JSON und TeX liefern, die dann von anderen Programmen weiterverarbeitet wird. Das Web-Interface soll die JSON-Ausgabe nutzen, welche jedoch nicht auf die Nutzung durch die verwendete D3.

2 Grundlagen

2.1 Programmiersprachen

2.1.1 C89 / ANSI C

Erster offizieller Standard der C-Programmiersprache, der 1989 veröffentlicht wurde. Wird als Grundlage für die zu verarbeitenden Ausdrücke genommen.

2.1.2 C99

Zweiter offizieller Standard der C-Programmiersprache, 1999 veröffentlicht. Mit diesem Standard wurde das c-nodes Kommandozeilenprogramm der Projektarbeit geschrieben.

2.2 Datenformate

2.2.1 JSON

JavaScript Object Notation (JSON) ist ein einfach lesbares Datenformat in Textform und stammt ursprünglich von der Programmiersprache JavaScript. Mittlerweile hat es sich aber als De-Facto-Standard zum Datenaustausch im Web und auch vielen anderen Applikationen etabliert.

```
1 {
2   "name": "abc",
3   "age": 10,
4   "child":
5    {
6     "name": "def",
7     "array": [5, 6]
8    }
9 }
```

Listing 2.1: Beispiel JSON

2.2.2 TeX

Quellcodedateien für das gleich heißende Textsatzsystem TeX, mit dem z.B. auch dieser Bericht verfasst wurde.

2.3 Benutzte Werkzeuge und Bibliotheken

Im Rahmen dieser Arbeit habe ich versucht so wenige externe Werkzeuge und Bibliotheken wie möglich zu nutzen um das Projekt so unabhängig wie möglich zu machen. Da es jedoch viel zu aufwendig und fehleranfällig wäre, Scanner und Parser von Grund auf neu zu schreiben, wurde hier zu bewährten Werkzeugen gegriffen.

2.3.1 flex

flex[1] ist ein Werkzeug zum Erstellen eines Scanners und die moderne Version des häufig verwendeten lex.

2.3.2 bison

Bison[2] ist analog zu flex ein Werkzeug zum Erstellen eines Parsers und die moderne Version des häufig mit lex verwendeten yacc.

2.3.3 D3.js

D3.js[3] ist eine Bibliothek zur Manipulation von Dokumenten für JavaScript. Im Zuge dieser Arbeit wurden vorallem die Schnittstellen zur Erstellung und Bearbeitung von SVG-Objekten benutzt.

2.3.4 CMake

CMake[4] ist ein plattformunabhängiges Werkzeug zum Erstellen von Build-Dateien für Softwareprojekte. Damit wurden die Build-Dateien für das c-nodes Kommandozeilenprogramm erstellt.

[1] https://github.com/westes/flex (Stand 19.02.2020)
[2] https://www.gnu.org/software/bison (Stand 19.02.2020)
[3] https://d3js.org/ (Stand 19.02.2020)
[4] https://cmake.org/ (Stand 19.02.2020)

2.4 Compilerbau

2.4.1 Bezeichner

Ein Bezeichner (oder auch selten Identifikator vom englischen *identifier*) ist eine eindeutige Benennung eines Objekts in einem Programm. Es ist eine Kombination von Namesraum und Namen. So kann es in unterschiedlichen Namensräumen einen Bezeichner mit dem gleichen Namen geben.

2.4.2 Symboltabelle

In der Symboltabelle wird jeder Bezeichner mit einer Reihe von Symbolattributen verknüpft. Dazu zählen Datentyp, Wert und Typqualifizierer. Die Symboltabelle dient dazu, Bezeichnerfehler (verwenden eines nicht deklarierten Bezeichners / erneute Deklaration eines Bezeichners) zu verhindern, in der semantischen Analyse den Datentyp im jeweiligen Kontext zu überprüfen und den Wert einer Variablen auszulesen bzw. zu verändern.

2.4.3 lvalues und rvalues

In dieser Implementation werden *rvalues* als konstante Werte (egal ob durch Konstanten im Eingabetext (z.B. *5* oder *3.2* als numerische Konstanten), konstante Variablen (`const int a = 2;`) oder als Ergebnis eines Funktionsaufruf (`sin(3.41)`)) angesehen. Im Gegenzug dazu stehen *lvalues* für Werte die verändert werden können (hier nur **nicht** konstante Variablen (`int b = 2;`)).

2.4.4 Ausdruck

Eine Reihe von Konstanten, Variablen und Funktionsaufrufen, welche durch Operationen kombiniert werden und genau einen Wert liefern.

2.4.5 Syntaxbaum

Hierarchische Darstellung der Zergliederung des zu verarbeitenden Quellcodes.

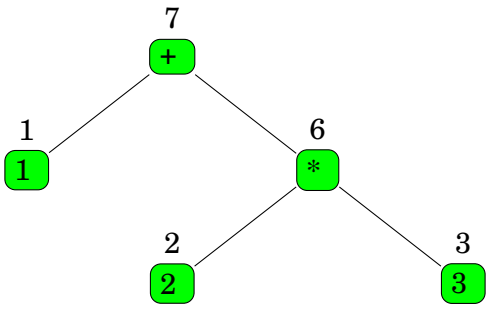


Abbildung 2.1: Syntaxbaum für den Ausdruck *1 + 2 * 3*

2.4.6 Token

Ein Token ist eine Zeichenkette, der innerhalb der lexikalischen Analyse ein Typ (z.B. Zahl, Bezeichner oder Schlüsselwort(*int*, *if*, etc.)) zugewiesen wird.

2.4.7 Scanner

Ein Werkzeug, das den Eingabetext in Token umwandelt. Kann Fehler in der Zeichenabfolge innerhalb eines Token finden, aber nicht in der Abfolge mehrerer Token.

2.4.8 Parser

Ein Werkzeug, das die vom Scanner generierte Abfolge von Token einliest und validiert. Kann für jede Kombination von Token eine Aktion auslösen, wie zum Beispiel das Hinzufügen eines Knoten im Syntaxbaum.

3 Problemanalyse

Das Ziel soll es sein, einerseits ein alleinstehendes Programm zur Konvertierung von Ausdrücken zu Syntaxbäumen (erstmal nur in Datenform), zu erstellen. Und anderseits ein Web-Interface zum Eingeben der Ausdrücke und Betrachten der erzeugten Syntaxbäume. Somit ist die eigentliche Logik des Programms getrennt von der Benutzerschnittstelle.

Das Programm zur Konvertierung der Ausdrücke besteht dann aus den typischen Komponenten eines Compilers: eine Komponente zur Speicherung des Abstract syntax tree (AST), eine Komponente zur Verwaltung der Symbole (Symboltabelle), generierte Scanner und Parser und schließendlich eine Menge von Knotentypen für die einzelnen Operationen.

4 Implementierung

Die Grundarchitektur (siehe Abbildung 4.1) des Lösungskonzeptes besteht aus einer klassischen Unterteilung in Front- und Backend. Das Frontend ist für die Datenabfrage und die Darstellung zuständig. Im Backend wiederum steckt die eigentliche Logik des Programms. Da das eigentliche Backend auch als eigenständiges Kommandozeilenprogramm funktionieren soll, wurde die Kommunikation mit dem Frontend auf ein Backend-Interface ausgelagert, die das Kommandozeilenprogramm auf dem Server aufruft und die Resultate zurück an das Frontend schickt.

4.1 Statistiken

4.1.1 Quellcodestatistiken

Die Quellcodestatistiken wurden mit dem Tool **cloc**[1] erstellt und sind vom aktuellen Projektstand am 19.02.2020.

[1] https://github.com/AlDanial/cloc (Stand 19.02.2020)

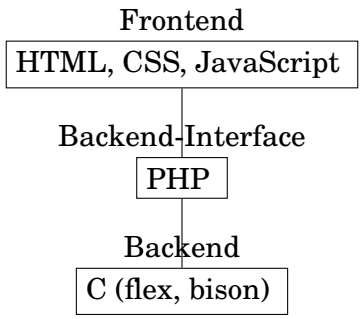


Abbildung 4.1: Grundarchitektur

Language	files	blank	comment	code
JavaScript	2	25	3	225
CSS	1	10	0	37
HTML	1	2	0	34
SUM	5	37	3	296

Tabelle 4.1: Quellcodestatistiken zum Web-Interface

Language	files	blank	comment	code
PHP	1	1	0	19
SUM	1	1	0	19

Tabelle 4.2: Quellcodestatistiken zum Backend-Interface

Language	files	blank	comment	code
C	18	778	833	4848
C/C++ Header	23	139	28	599
yacc	1	45	0	330
lex	1	22	0	99
SUM	43	984	861	5876

Tabelle 4.3: Quellcodestatistiken zum c-nodes Kommandozeilenprogramm

4.2 Web-Interface

Das gesamte Frontend besteht aus einer HTML-, einer CSS- und zwei JavaScript-Dateien (eine davon ist die zur Darstellung des Syntaxbaum verwendete D3.js Bibliothek). Die HTML und CSS Dateien sind je unter 50 Quellcodezeilen lang und somit sehr trivial gehalten. In der eigenen JavaScript-Datei werden die Eingaben des Benutzers ausgelesen, formatiert und dann an das Backend geschickt. Die Antwort wird wiederum verwendet um die resultierenden Syntaxbäume zu zeichnen.

4.3 Backend-Interface

Das Backend-Interface besteht aus einer einzigen PHP-Datei, die im Grunde nichts anderes macht, als das eigentliche Backend mit den Parametern aus der Anfrage des Frontends aufzurufen und die Ergebnisse formatiert wieder ans Frontend zurückzuschicken. Da das Skript sehr kurz ist, habe ich es direkt in diese Dokumentation eingefügt.

```
1  <?php
2    header("Content-Type: application/json");
3    $json_output = array();
4    $post = file_get_contents('php://input');
5    $data = JSON_decode($post, true);
6    if( !isset($data['expr'])) {
7      # 'leere' Antwort, falls kein Ausdruck übergeben wurde
8      echo("{\"expr\":\"\", \"trees\":{}}");
9    }
10   else {
11     # End of line ('\n') Symbole entfernen
12     $expr = str_replace(PHP_EOL, '', $data['expr']);
13     # Das eigentliche Kommandozeilenprogramm aufrufen und die Ausgabe speichern
14     exec("./c-nodes.out -expr \"".$expr."\" -symbols \"".$data['symbols']."\"
           -json", $json_output);
15
16     echo "{ \"expr\": \"".$expr."\",";
17     echo "  \"list\": ";
18     # Die Ausgabe Zeile üfr Zeile üzurckgeben
19     foreach($json_output as $line) {
20       echo $line;
21     }
22     echo "}";
23   }
24 ?>
```

Listing 4.1: Das gesamte Backend-Interface-Skript

4.4 c-nodes Kommandozeilenprogramm

Der Großteil der Arbeit ist in das Kommandozeilenprogramm *c-nodes* geflossen. Dieses nimmt grundlegend einen Text, bestehend aus Deklarationen und Ausdrücken, und eine optionale Reihe von Symbolen entgegen. Als Resultat gibt das Programm eine Liste der Syntaxbäume in einem der Formate (JSON oder tex) auf der Standardausgabe aus. Alternativ kann die Ausgabe auch direkt in eine angegebene Datei geschrieben werden. Für die Generierung des Scanners wurde hierbei flex und für die Generierung des Parsers Bison verwendet.

4.4.1 Codekonventionen

struct / union

Um in C nicht bei jedem Benutzen einer *struct* (im folgenden analog auch *union*), das Schlüsselwort *struct* vor den definierten Typs zu schreiben, ist es verbreitet, ein anonymes *struct* mit *typedef* auf den gewünschten Typnamen zu definieren.

```
1 typedef struct {
2     int a, b, c;
3 } MyType;
4 MyType my_type; // statt 'struct MyType my_type;'
```

Listing 4.2: Benennen eines anonymen Typs mit typedef

Die hat allerdings zur Folge, dass das *struct* nun nicht mehr voraus deklariert werden kann.

```
1 // Vorausdeklaration von MyType und eine Funktion, die MyType benutzt
2 struct MyType;
3 void func(struct MyType my_type);
4
5 // Definition des struct mit typedef
6 typedef struct {
7     int a, b, c;
8 } MyType;
9
10 void func(MyType my_type) { // <= Warnung 'parameter different from declaration'
11     ...
12 }
```

Listing 4.3: Problem beim Vorausdeklarieren eines anonymen Typs

Um das zu umgehen, brauchen wir sowohl die *struct*-Definition, als auch den *typedef*, mit dem Namen des *struct*.

```
1 typedef struct MyType {
2     int a, b, c;
3 } MyType;
```

Listing 4.4: Lösung zur Definition von structs

Zwar wird in diesem Projekt keine Vorausdeklaration von *struct*s benutzt, jedoch wird diese Konvention benutzt um bei einer eventuellen Erweiterung dies zu ermöglichen. **Dieses Vorgehen gilt auch analog für die Definition von *union*s.**

4.4.2 Implementierung des Scanners

Als Grundlage für den Scanner wurde die **Lex-Spezifikation**[2] von Jutta Degener genutzt, aber alle nicht für die Implementierung benötigten Tokens (z.B. *volatile*, *auto* oder *static*) entfernt. Allerdings wurde statt lex das modernere flex zur Generierung des Scanners benutzt.

4.4.3 Implementierung des Parsers

Analog zur Implementierung des Scanner wurde die **Yacc-Spezifikation**[3] von Jutta Degener als Basis genutzt, und auch an den Umfang dieses Projektes angepasst. Auch hier wurde der Generator yacc durch das modernere Bison ersetzt.

4.4.4 Repräsentation von Datentypen und Werten

Jeder Knoten hat genau einen Datentyp, der entweder beim Erschaffen des Knoten (z.B. Konstanten) festgelegt oder bei der semantischen Analyse von den Kindknoten abgeleitet wird. Bei letzter Art von Knoten wird der Datentyp beim Erschaffen auf den Datentyp *VT_ERROR* gesetzt. Gibt es nach der semantischen Analyse noch Knoten mit dem Datentyp *VT_ERROR* wird die Interpretation des Syntaxbaums nicht gestartet. Die verfügbaren Datentypen sind wie folgt definiert:

```
1 typedef enum ValueType {
2     VT_ERROR = 0,
3     VT_INT = 1,
4     VT_DOUBLE = 2,
5 } ValueType;
```

Listing 4.5: ValueType

Die einzelnen Werte repräsentieren hierbei Bits, d.h. würde ein weiterer Datentyp hinzukommen (z.B. *char*) bekäme dieser den Wert $2^2 = 4$.Kein gesetztes Bit repräsentiert keinen „richtigen“ Datentyp bzw. den Datentyp *VT_ERROR*. Dies wurde so gewählt damit Datentypen mit Bitoperationen einfach kombiniert werden können. Darauf wird im Abschnitt Knoten genauer eingegangen.

[2] https://www.lysator.liu.se/c/ANSI-C-grammar-l.html (Stand 19.02.2020)
[3] https://www.lysator.liu.se/c/ANSI-C-grammar-y.html (Stand 19.02.2020)

Um nicht für jeden Datentyp eine Variable für den jeweiligen Wert zu definieren (jeweils eine Variable für die Datentypen *int* und *double*), wurden die Werte in einer *union* zusammengefasst:

```
1 typedef union Value {
2     int i_value;
3     double d_value;
4 } Value;
```

Listing 4.6: Value

Somit ist immer nur ein Wert „aktiv“. Der Speicherbedarf für alle Werte ist hierbei immer der Speicherbedarf des größten Datentyps (hier *double* mit 8 Bytes). Nachteil bei diesem Vorgehen ist, dass z.B. *i_value* gesetzt, aber *d_value* ausgelesen wird. Um das zu verhindern muss immer ein ValueType mit einem Value assoziiert werden, der den zu benutzenden Wert bestimmt.

4.4.5 Knoten

Jeder Knoten hat genau einen **Ausgang** und bis zu drei **Eingänge**, aus denen der Ausgangswert und -typ berechnet wird. Häufig wird auch **Kindknoten** und **Elternknoten** als Begriffe bei Bäumen verwendet. Bei dem hier verwendeten Modell entsprechen die **Kindknoten** den **Eingängen**, und die Knoten kennen ihren Elternknoten nicht. Hier wurde mit Absicht von der üblichen Benennung Abstand genommen, um deutlich zu machen, dass ein Knoten keinerlei Einfluss auf die Funktionsweise oder die Werte seiner **Eingänge** hat und nur Daten bereitstellt, die von einem eventuellen **Elternknoten** benutzt werden können. Der Knoten an sich hat aber keine Verbindung zu seinem **Elternknoten** und funktioniert unabhängig von dessen Funktion. Die genaue Funktionsweise wird in den Sektionen Knotenausgang und Knoteneingänge erläutert.

```
1 typedef struct Node {
2     NodeIn in;
3     NodeOut out;
4     bool (*processNode)(Node* node, const ProcessMode process_mode);
5     char text[20];
6     char error[200];
7     void* additional_info;
8     SymbolHandle symbol_handle;
9 } Node;
```

Listing 4.7: Node

processNode ist ein Funktionszeiger auf eine Funktion, die den jeweiligen Knoten verarbeiten kann (siehe als Beispiel processNode_GetSymbol). In *text* wird der Namen des Knoten gespeichert, welcher in den allermeisten Fällen dem vom Scanner erzeugten Token entspricht (z.B. **+** für einen Additionsknoten). In *error* wird eine eventuelle Fehlernachricht zu einem Fehler, der bei diesem Knoten aufgetreten ist, gespeichert. *additional_info* ist ein Zeiger auf einen nicht näher spezifierten

Datenblock und kann für verschiedene Dinge benutzt werden (z.B. um eine Referenz zur Symboltabelle zu speichern). In *symbol_handle* kann ein Verweis zu einem Symbol gespeichert werden (standardmäßig auf *0 / kein Symbol* gesetzt).

Knotenausgang

Im Ausgang eines Knotens werden primär der Datentyp und der Wert des Knoten gespeichert. Dazu gibt es noch die Metadaten *is_lvalue* (gibt an, dass der Knoten auf ein manipulierbares Symbol verweist) und *is_processed* (gibt an, ob der Knoten schon verarbeitet wurde, um z.B. Mehrfachverarbeitung zu verhindern). All diese Daten werden einerseits vom Elternknoten als Eingangsdaten, und anderseits zur Darstellung des Knoten im Syntaxbaum, benutzt.

```
1 typedef struct NodeOut {
2     ValueType   type;
3     Value       value;
4     bool        is_lvalue;
5     bool        is_processed;
6 } NodeOut;
```

Listing 4.8: NodeOut

Knoteneingänge

In einem Eingangsslot wird ein Verweis auf einen Knoten und dazugehörige Metadaten gespeichert. In *allowed_value_types* wird eine Bitmaske für die ValueTypes gespeichert, mit der abgeglichen wird, ob ein Eingangsknoten den richtigen Ausganstyp hat. Dazu werden die jeweiligen ValueTypes einfach mit einem binären Oder-Operator verknüpft.

```
1 typedef struct InSlot {
2     Node* node;
3     uint32_t allowed_value_types;
4     bool allow_rvalues;
5 } InSlot;
```

Listing 4.9: InSlot

```
1 slot_0.allowed_value_types = VT_INT; // nur 'int' erlaubt
2 slot_1.allowed_value_types = VT_INT | VT_DOUBLE; // 'int' und 'double' erlaubt
```

Listing 4.10: Verknüpfen von ValueTypes

allow_rvalues gibt an, ob der Eingang rvalues als Eingang akzeptiert oder nicht. Meistens ist dies auf *true* gesetzt, Ausnahmen wären der Zuweisungsoperator(**=**) oder die Inkrement-/Dekrementoperatoren(**++**, **–**), die zwingend einen lvalue brauchen, den sie manipulieren können.

Jeder Knoten kann bis zu 3 Eingangsslots haben, die konkrete Anzahl der Eingänge wird in *slot_count* gespeichert. Auf die jeweiligen Slots kann man entweder einzeln (*slot_0*, *slot_1* und *slot_2*) oder iterativ (*slot[i]*) zugreifen.

```
typedef struct NodeIn {
    union {
        struct {
            InSlot slot_0;
            InSlot slot_1;
            InSlot slot_2;
        };
        InSlot slot[3];
    };
    uint8_t slot_count;
} NodeIn;
```

Listing 4.11: NodeIn

Definition eines Knotentyps

Für jeden Knotentyp (d.h. für jede Operation und noch ein paar zusätzliche) werden zwei Funktionen benötigt. Einerseits eine Funktion, die den Knoten verarbeitet (*processNode_<Knotentyp>*) und anderseits eine Funktion die den Knoten erschafft (*createNode_<Knotentyp>*, also alle benötigten Parameter eines Nodes festlegt.

Als Beispiel wird hier die Verarbeitungsfunktion des Knotens zum Abrufen eines Symbols gezeigt, da diese relativ kompakt in der Länge, aber doch relativ umfangreich bei den benutzten Parametern ist. *process_mode* gibt hierbei an, ob die Verarbeitung nur den Datentyp des Knoten bestimmen (**PM_TYPE_ONLY**), also im Prinzip die semantische Analyse, oder den Knoten komplett verarbeiten soll (**PM_FULL**), also auch die restlichen Werte verändern.

```
bool processNode_GetSymbol(Node* node, const ProcessMode process_mode) {
    SymbolTable* sym_tab = (SymbolTable*) node->additional_info;
    if(!sym_tab) {
        strcpy(node->error, "DEBUG: No reference to symbol table");
        return false;
    }
    SymbolHandle handle = getSymbolHandle(sym_tab, node->text);
    if(handle.value == 0) {
        sprintf(node->error, "There's no variable named '%s'", node->text);
        return false;
    }
    SymbolValue sym_value = getSymbolValue(sym_tab, handle);
    node->out.type = sym_value.type;
    if(process_mode == PM_FULL) {
        node->out.is_lvalue = !sym_value.is_const;
        node->symbol_handle = handle;
        node->out.value = sym_value.value;
    }
    return true;
}
```

Listing 4.12: processNode_GetSymbol

Und folgend die Erstellungsfunktion des Knoten, die die Verarbeitungsfunktion mit dem Knoten verknüpft und die restlichen Parameter des Nodes initial setzt.

```
Node createNode_GetSymbol(const char* identifier) {
    Node node = {
        .in = {
            .slot_count = 0,
        },
        .out = {
            .type = VT_ERROR,
            .value.i_value = 0,
            .is_lvalue = false,
            .is_processed = false,
        },
        .processNode = processNode_GetSymbol,
        .additional_info = NULL,
        .error = "",
        .symbol_handle = 0,
    };
    strcpy(node.text, identifier);
    return node;
}
```

Listing 4.13: createNode_GetSymbol

4.4.6 Abstrakter Syntaxbaum (AST)

In der Implementierung wird der AST nur als Speicherverwaltung für die Knoten benutzt.

```
#define AST_MAX_NODES 256

typedef struct AbstractSyntaxTree {
    Node nodes[AST_MAX_NODES];
    uint16_t node_count;
} AbstractSyntaxTree;
```

Listing 4.14: AbstractSyntaxTree

Dazu gibt es noch eine Reihe von „makeNode“-Funktionen, um die Knoten in den AST einzufügen. Diese folgen alle dem gleichen Schema und haben als Parameter mindestens einen Zeiger auf ein AbstractSyntaxTree-Objekt und eine Knotenerstellungsfunktion. Zusätzlich kann es noch mehr Parameter geben, die zur Erstellung eines Knoten benötigt werden.

```
Node* makeNode_0(AbstractSyntaxTree* ast, Node (*createNode)(void));
Node* makeNode_1(AbstractSyntaxTree* ast, Node (*createNode)(Node* node_0), Node*
    node_0);
Node* makeNode_0_INT(AbstractSyntaxTree* ast, Node (*createNode)(const int value),
    const int value);
Node* makeNode_2_STRING(AbstractSyntaxTree* ast, Node (*createNode)(Node* node_0,
    Node* node_1, const char* value), Node* node_0, Node* node_1, const char*
    value);
Node* makeNode_0_SymbolValue(AbstractSyntaxTree* ast, Node
    (*createNode)(SymbolValue value), SymbolValue value);
```

Listing 4.15: Auswahl verschiedener „makeNode“-Funktionen

Symboldefinition	Name	Datentyp	Wert	Typqualifizierer
const int a = 2	a	int	2	const
double b	b	double	? / 0.0	-

Tabelle 4.4: Beispiele von Symbolen

Die jeweiligen „makeNode“-Funktionen erzeugen dann mit der übergebenen Knotenerstellungsfunktion (siehe Listing 4.17), und den eventuellen zusätzlichen Parametern, ein Node-Objekt und fügen es der Liste des AbstractSyntaxTree-Objekt hinzu:

```
1 Node* makeNode_2_STRING(AbstractSyntaxTree* ast, Node (*createNode)(Node* node_0,
      Node* node_1, const char* value), Node* node_0, Node* node_1, const char*
      value) {
2     if(ast->node_count < AST_MAX_NODES) {
3         ast->nodes[ast->node_count++] = createNode(node_0, node_1, value);
4         PRINT_DEBUG("Inserted node from makeNode_2_STRING on #%d\n",
              ast->node_count-1);
5         return &(ast->nodes[ast->node_count-1]);
6     }
7     PRINT_DEBUG("Failed to insert node from makeNode_2_STRING");
8     return NULL;
9 }
```

Listing 4.16: Auswahl der „makeNode“-Funktionen

```
1 Node* node = makeNode_0_STRING(&ast, createNode_GetSymbol, $1);
```

Listing 4.17: Verknüpfung von „makeNode“- und „createNode“-Funktionen mit Parametern

4.4.7 Symboltabelle

Symbolattribute

Je nach Ansicht ist der Typqualifizierer (*const* oder *volatile*(nicht implementiert)) Teil des Datentyp oder ein extra Attribut. In der Implementierung wird der vereinfachte Typqualifizierer (*const* oder *nicht const*) als extra Symbolattribut behandelt. Beispiele:

Da es keine Namensräume in der Implementierung gibt, werden alle Symbole als global angesehen, d.h. ein nicht initialisiertes Symbol mit Datentyp *int* bekommt den Wert *0*, analog dazu *0.0* für ein *double*.

Die Symbolattribute (ohne den Bezeichner) werden im *struct* SymbolValue zusammengefasst. Dazu gehören der Datentyp, der Wert und der (vereinfachte) Typqualifizierer.

```
1 typedef struct SymbolValue {
2     ValueType type;
3     Value value;
4     bool is_const;
5 } SymbolValue;
```

Listing 4.18: SymbolValue

Die Tabelle

Ein SymbolHandle ist im Prinzip nur ein einfacher Index in der Symboltabelle, der auf ein bestimmtes Symbol verweist, wobei der Wert **0**, ähnlich wie bei Zeigern, für kein Symbol bzw. ein ungültiges Symbol steht.

```
1 typedef struct SymbolHandle {
2     uint8_t value;
3 } SymbolHandle;
```

Listing 4.19: SymbolHandle

Man hätte hier auch einen einfachen *typedef* benutzen können: `typedef uint8_t SymbolHandle;`. Allerdings könnte es hier in Zukunft Probleme bei weiteren ähnlich definierten *Handles* geben:

```
1 typedef uint8_t SymbolHandle;
2 typedef uint8_t AnotherHandle;
3
4 void useSymbolHandle(SymbolHandle handle) {...}
5
6 AnotherHandle another_handle = 5;
7 useSymbolHandle(another_handle); // Der Compiler sieht hier keinen Typunterschied
      mehr
```

Listing 4.20: Problem beim Definieren von „Handle“-typen mit typedef

Mit der benutzten Methode ist ein unabsichtliches Verwechseln von verschiedenen *Handletypen* quasi nicht mehr möglich.

```
1 #define MAX_SYMBOL_LENGTH 20
2 #define MAX_SYMBOLS 64
3 typedef struct SymbolTable {
4     char identifiers[MAX_SYMBOL_LENGTH][MAX_SYMBOLS];
5     SymbolValue values[MAX_SYMBOLS];
6     uint8_t symbol_count;
7     SymbolValue currentConfig;
8 } SymbolTable;
```

Listing 4.21: SymbolTable

Die SymbolTable speichert primär zwei gleich große Arrays, mit einerseits den Bezeichnern und anderseits den Symbolattributen. Bezeichner und Symbolattribute werden einfach über den gleichen Index verbunden. So gehört der SymbolValue bei *values[2]* zu dem Bezeichner bei *identifiers[2]*. *symbol_count* gibt die Gesamtzahl

der gespeicherten Symbole an. *currentConfig* wird benutzt um neue Bezeichner mit einer bestimmten Konfiguration der Symbolattribute hinzuzufügen.

5 Evaluierung

Auf Seiten der Funktionalität des Werkzeugs wurden die Ziele der Arbeit vollständig erreicht oder sogar teilweise übertroffen. Benutzer können nun statt, wie geplant, nur einen einzelnen Ausdruck, regulären ANSI-C Quellcode (mit Einschränkungen bezüglich Programmsteuerungsanweisungen, Zeigern, Arrays, etc.) in das Werkzeug eintragen und sich die Syntaxbäume aller Ausdrücke anzeigen lassen.

Jedoch war leider keine Zeit mehr, das enstandene Werkzeug von den Programmieranfängern testen und evaluieren zu lassen. Dies wird allerdings im nächsten Semester nachgeholt und die Ergebnisse dieser Evaluation dokumentiert, so dass sie eventuell eine Basis für eine weitere Projektarbeit stellen, die sich mit der Erweiterung dieses Werkzeugs befasst.

6 Zusammenfassung und Ausblick

6.1 Erreichte Ergebnisse

Wie in Abbildung 6.1 zu sehen ist, wurde mit dem Werkzeug das Ziel der Arbeit erreicht und man kann sich auch für komplexere Ausdrücke, einen Syntaxbaum mit farbig kodierten Typinformationen und Werten für jeden einzelnen Knoten, darstellen lassen.

6.2 Ausblick

6.2.1 Erweiterbarkeit der Ergebnisse

Hinzufügen weiterer primitiver Datentypen (char, long, bool, float, etc.)

Prinzipiell dürfte das Hinzufügen weiterer primitiver Datentypen keine großen Probleme bereiten, jedoch müsste so gut wie jeder *Node* angepasst werden, um mit dem neuen Datentyp umgehen zu können.

- Schwierigkeit: Gering
- Aufwand: Mittel

Felder als Datentyp

Für die Implementierung von Feldern müsste auf jeden Fall die Symboltabelle überarbeitet werden. Ein einfacher Weg wäre hier zum Beispiel die Definition von 10 *int*-Symbolen mit automatisch generierten Namen für ein *int*-Feld der Größe 10. Wenn wir dann nur von einfachen Lese- und Schreiboperationen mithilfe des Index-Operators *[]* auf die einzelnen Datenfelder ausgehen, würde sich der Aufwand bei den *Nodes* hier wohl auf eine Anpassung des Lese-*Nodes* von Variablen und dem Zuweisungs-*Nodes* beschränken. Da die Speicherreservierung bisher jedoch nicht zwingend linear erfolgt, müsste wahrscheinlich extra Aufwand betrieben werden um zu garantieren, dass alle Elemente des Feldes in einem Block liegen und auch

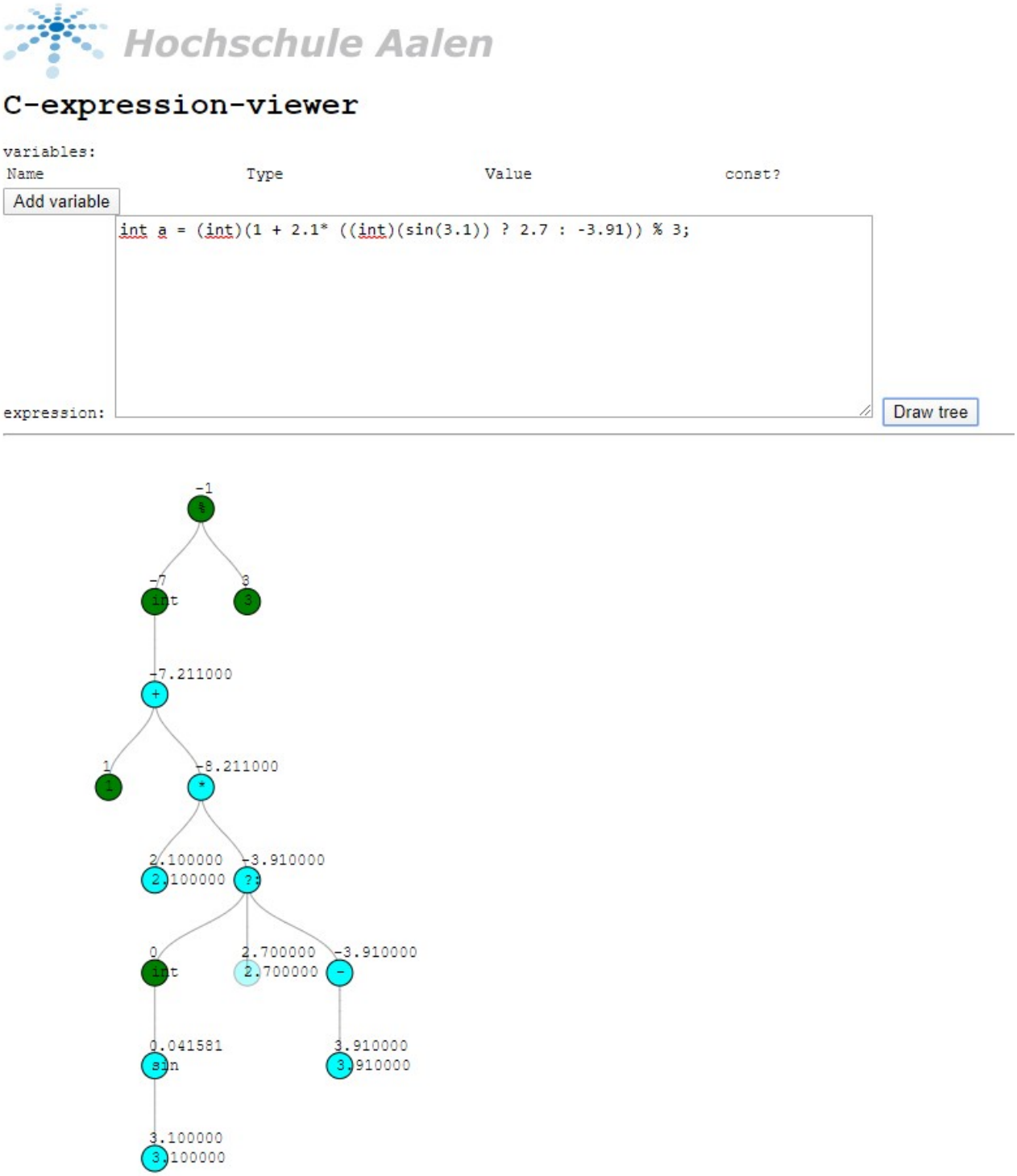


Abbildung 6.1: Syntaxbaum eines komplexeren Ausdrucks

der Zugriff auf das (n+1)-te Element eines Feldes, den „richtigen“ Effekt hat, also den Speicherbereich nach dem eigentlichen Feld ausliest bzw. manipuliert.

- Schwierigkeit: Mittel
- Aufwand: Mittel

Zeiger als Datentyp

Wahrscheinlich dürfte es reichen ein weiteres Attribut zu der Definition der Symboltabellenwerte hinzuzufügen. Zum Beispiel *is_pointer*, wobei der *ValueType* dann auf den referenzierten Datentyp verweist, als Wert jedoch die Adresse im *i_value* abgespeichert wird. Somit wären Adressen jedoch auf den Wertebereich eines *int* limitiert. Zusätzlich müssten natürlich noch die unären Operatoren & und * im Parser implementiert werden.

- Schwierigkeit: Mittel
- Aufwand: Mittel

Zeigerarithmetik

Zeigerarithmetik (Berechnung der Zeigeradresse inklusive Inkrement und Dekrement) benötigt eine Emulation des Speichers und somit grundlegende Änderungen an mehreren Basissystemen (zum Beispiel an der Symbol Tabelle). Der Aufwand hierfür überschneidet sich mit dem Aufwand zur (vollständigen) Implementierung von Feldern.

- Schwierigkeit: Hoch
- Aufwand: Hoch

Hinzufügen von Programmsteuerungsanweisungen (Verzweigungen, Schleifen)

Das Hinzufügen von Programmsteuerungsanweisungen übertrifft bei Weitem die Ursprungsfunktionalität eines Ausdruckauswerters und geht dann schon Richtung eines vollständigen Interpreters. Wahrscheinlich müssten dafür große Teile des bestehenden Projektes angepasst werden.

- Schwierigkeit: Hoch
- Aufwand: Sehr hoch

Bearbeiten des Syntaxbaums...

Das Bearbeiten des Syntaxbaums im Frontend müsste mit der D3.js-Bibliothek möglich sein, jedoch habe ich mich im Rahmen dieser Arbeit nicht weiter damit befasst.

- Schwierigkeit: Nicht einschätzbar
- Aufwand: Nicht einschätzbar

...und Generieren des entstandenen *C*-Ausdrucks

Das Generieren des *C*-Ausdrucks aus einem Syntaxbaum besteht größtenteils nur aus Textkonkatenation der Namen der einzelnen Knoten. Bei einzelnen Knoten (zum Beispiel der ternäre Operator oder Funktionen mit Parametern) ist etwas mehr Aufwand von Nöten.

- Schwierigkeit: Gering
- Aufwand: Gering