Hochschule Merseburg

Fachbereich Ingenieur- und Naturwissenschaften

Ein Löser für teilkorrekte Speicherbelegungsprotokolle in Programmieraufgaben

Bachelorarbeit

im Studiengang Angewandte Informatik

Vorgelegt von

Lukas Reinicke

Halle (Saale), 06.10.2023

Erstprüfer/in: Prof. Dr. Sven Karol

Zweitprüfer/in: Dipl.-Wirtsch.-Inform. Petra Schwerin

Aufgabenstellung für die Bachelorarbeit

Titel: Ein Löser für teilkorrekte Speicherbelegungsprotokolle in Programmieraufgaben

Eine typische Aufgabenstellung in der Lehrveranstaltung „Grundlagen der Programmierung“ ist die Erstellung tabellarischer Speicherbelegungsprotokolle für einfache C-Programme. Dabei sind die Variablenbelegungen während der Programmausführung an durch Kommentare gekennzeichneten Stellen anzugeben. Die Programme, die dabei verwendet werden, bestehen in der Regel aus wenigen Funktionen mit mehreren Anweisungsblöcken. Die Anweisungsblöcke enthalten wiederum einfache Anweisungen wie Variablenzuweisungen, arithmetische Berechnungen sowie implizite und explizite Typumwandlungen. Außerdem lassen sich Anweisungsblöcke verschachteln. Insgesamt wird für die Speicherbelegungsprotokolle nur eine Teilmenge der Programmiersprache C in Betracht gezogen.

Die Korrektheit eines Speicherprotokolls lässt sich maschinell sehr einfach überprüfen. Ein Problem besteht jedoch in der Bewertung bzw. Auswertung von Speicherprotokollen, die nur teilweise korrekt sind. Da auch in solchen Fällen Punkte vergeben werden sollen, reicht die Prüfung auf Korrektheit nicht aus. Ähnlich zu mathematischen Aufgaben, wo mit falschen Zwischenergebnissen weitergerechnet werden kann, können Speicherbelegungsprotokolle ebenfalls mit falschen Zwischenergebnissen weitergeführt werden.

Das wesentliche Ziel dieser Arbeit besteht daher darin, einen Löser für einfache Speicherprotokollierungsaufgaben zu entwickeln, der ein gegebenes Belegungsprotokoll mit der berechneten, korrekten Lösung abgleicht und die korrekten Teile des gegebenen Protokolls identifiziert. Dabei sind Folgefehler mit einzubeziehen und die zu vergebende Punktzahl automatisch zu berechnen und Fehler nachvollziehbar zu kennzeichnen. Dafür ist ein geeigneter Algorithmus zu entwerfen und zu implementieren. Als Eingabeparameter sollen das zu protokollierende Programm, das zu überprüfende Belegungsprotokoll und ein Punkteschema genutzt werden. Die Nutzerschnittstelle soll möglichst einfach und effizient benutzbar sein.

Zusammenfassend ergeben sich folgende Zielstellungen der Arbeit:

- Einarbeitung in die Themenfelder Parsen und kontextfreie Grammatiken
- Entwurf und Implementierung eines Parsers für das infrage kommende C-Fragment
- Gegebenenfalls Verwendung eines Parsergenerators (z.B. ANTLR)
- Einlesen von Speicherbelegungsprotokollen mit einem textbasierten Format (z.B. CSV, XML, DSL, JSON)
- Entwicklung eines Algorithmus, zum
    - Bestimmen der korrekten Lösung von Speicherbelegungsprototollen
    - Abgleich mit einer gegebenen Lösung
    - Finden von Teilkorrekten Lösungen unter Berücksichtigung fehlerhafter Zwischenlösungen
- Prototypische Umsetzung als Anwendung

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbstständig und ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe. Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken (inkl. Internetinhalte) als solche kenntlich gemacht habe.

Halle (Saale), 06.10.2023 — Lukas Reinicke

(Ort, Datum) — (Unterschrift)

Abstract

Speicherbelegungsprotokolle dienen als Überprüfung, damit das Wissen von Programmiersprachenlernenden fachgerecht überprüft werden kann. Um die faire und effiziente Bewertung von Speicherbelegungsprotokollen zu ermöglichen, bedarf es einer modernen Form der Überprüfung.

Das Ziel der vorliegenden Arbeit ist es, einen Löser für einfache Speicherbelegungsprotokollierungsaufgaben zu entwickeln und prototypisch umzusetzen. Dabei sollen insbesondere Folgefehler, eine nachvollziehbare Berechnung der Punkte sowie eine Nachvollziehbarkeit von aufgetretenen Fehlern miteinbezogen werden. Der implementierte Löser soll in der Lage sein, zu einfachen C-Programmen gehörige Speicherbelegungsprotokolle auszuwerten und zu bepunkten.

Hierfür wurde in der Arbeit ein funktionierender Algorithmus entworfen, welcher die Lösungsberechnung für Speicherbelegungsprotokollaufgaben unter Einbeziehung eines potenziell fehlerhaften Protokolls ermöglicht. Dafür wurde ein Interpreter für einen begrenzten Ausschnitt aus der Programmiersprache C entworfen, welcher die Grundlage für die nachfolgende Implementierung des Algorithmus darstellte. Für die nötige Interpretation von C-Code wurden die Grundlagen der Sprachdefinition und -verarbeitung betrachtet. Auf Grundlage dessen wurde eine Grammatik für den betrachteten C-Ausschnitt entworfen und ein Parser für die Verwendung in dem C-Interpreter generiert.

Mit dem implementierten Löser ist es möglich, teilkorrekte Speicherbelegungsprotokolle nicht nur auf deren Richtigkeit zu überprüfen, sondern auch etwaige Folgefehler korrekt zu bewerten. Dadurch ist eine Überprüfung von komplexeren Protokollen umsetzbar, welche mit manueller Kontrolle kaum zu handhaben wäre.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listingverzeichnis

Abkürzungsverzeichnis

ABNF	**a**ugmented **B**ackus–**N**aur-**f**orm
ANTLR	**AN**other **T**ool for **L**anguage **R**ecognition
BNF	**B**ackus-**N**aur-**F**orm
EBNF	**e**xtended **B**ackus-**N**aur-**f**orm
IEC	**I**nternational **E**lectrotechnical **C**ommission
ISO	**I**nternational **O**rganization for **S**tandardization
RFC	**R**equests **f**or **C**omments
WPF	**W**indows **P**resentation **F**oundation
XML	E**x**tensible **M**arkup **L**anguage

1 Einleitung

Motivation

Eine Schwierigkeit beim Erlernen einer Programmiersprache ist die schlechte Überprüfbarkeit des Erlernten. Sogar in der Selbstbeurteilung gehen viele Lernende davon aus, etwas begriffen zu haben, bis es dann zur Anwendung kommt. In Programmiersprachen kommt das besonders zum Tragen, weil es so viele Kombinationen gibt, Elemente einer Programmiersprache zu kombinieren. Der funktionale Unterschied zwischen einer Post- und einer Präfix-Inkrementierung hat schon so manchen das richtige Ergebnis gekostet. Auch Operatorprioritäten oder das Missverstehen von arithmetischen Operationen auf bestimmte Datentypen können schnell zu falschen Ergebnissen führen. Hier wird allerdings auch der Lehrende vor eine große Aufgabe gestellt, da das Überprüfen auf Verständnis, gerade für eine ganze Klasse oder gar für einen ganzen Hörsaal, keine einfache Aufgabe ist. Hier bedarf es Formen der Verständniskontrolle, die es der Lehrkraft ermöglichen, Abgaben schnell und effizient zu prüfen. Viele klassische Überprüfungsformen sind dafür nicht geeignet, weil sie zu aufwendig zu kontrollieren sind. Den Abgabecode von dutzenden Schülern oder Studenten zu kompilieren und zu testen, ohne bei der ersten Fehlermeldung die gesamte Abgabe zu verwerfen, klingt vermutlich nicht zu Unrecht wie eine zeitraubende Angelegenheit. Als alternatives Mittel zur Leistungsüberprüfung eignen sich daher sogenannte Speicherbelegungsprotokolle. Aufgaben für Speicherbelegungsprotokolle bestehen nur aus einem speziell annotierten Code und nach Bedarf noch einer tabellarischen Vorlage für die Eintragung der Ergebnisse. Aufgabe ist es dann, an den im Code markierten Stellen alle sichtbaren Variablen und deren Belegungen anzugeben. Das Ergebnis ist ein Speicherbelegungsprotokoll, welches für jede Markierung eine Zeile aufweist und in deren Spalten die Belegung der Variablen steht. Solche Tabellen gegen eine einmal ermittelte Lösung zu kontrollieren, ist eine effiziente und schnelle Angelegenheit. Leider stellt sich bei dieser Art der Kontrolle schnell ein neues Problem ein, nämlich Folgefehler. Sobald Variablen innerhalb des Quellcodes voneinander abhängen, was schon in sehr einfachen Aufgaben der Fall sein kann, können sich frühzeitige Fehler schnell durch das ganze Protokoll ziehen. Das verkompliziert die Bewertung extrem, weil nach dem fehlerhaften Eintrag wohlmöglich alle folgenden Einträge abweichen. Für eine faire Bewertung muss jetzt jede Belegung neu überprüft werden, und zwar auf Basis des erkannten Fehlers. Bei komplexeren Aufgaben können solche Fehler schnell vermehrt auftreten und den ganzen Bewertungsprozess deutlich verkomplizieren. Eine automatisierte Überprüfung solcher Protokolle kann den Bewertungsprozess enorm vereinfachen und ist Gegenstand dieser Arbeit. Eine solche Automatisierung der Kontrolle von teilkorrekten Speicherbelegungsprotokollen mit der Berücksichtigung von Folgefehlern erfordert allerdings einen anderen Lösungsansatz als eine Kontrolle, die nur die Richtigkeit gegenüber der korrekten Lösung feststellen soll. Eine so geartete Fehlerverfolgung kann nicht ohne eine Verarbeitung des Quellcodes auskommen. In der folgenden Arbeit sollen die zugrundliegenden Theorien und Techniken für eine solche Verarbeitung aufgezeigt und dann in Form einer prototypischen Anwendung implementiert werden.

Aufbau der Arbeit

In Kapitel 2 geht es um die Grundlagen formaler Sprachen und deren Definition. Es werden die Grundbestandteile von formalen Sprachen erklärt und aufgezeigt, welche Bedeutung sie innerhalb der Sprachedefinition haben. Außerdem werden Grammatiken als Beschreibungsform von Sprachen vorgestellt und definiert, sowie eine Notationsform für diese Grammatiken eingeführt. Es wird aufgezeigt, wie diese Grammatiken klassifiziert werden können und welche Bedeutung diese Klassifizierung für die Sprachen der Grammatiken haben.

Kapitel 3 behandelt die für diese Arbeit relevanten Phasen beim Bau von Compilern und Interpretern. Es wird die Bedeutung von kontextfreien Grammatiken im Compilerbau erläutert sowie die Funktion der einzelnen Phasen. Abschließend werden Parsergeneratoren als ein Werkzeug im Compiler- und Interpreterbau vorgestellt.

In Kapitel 4 findet die Implementierung des angestrebten Lösers statt. Dafür wird anfangs eine Definition von Speicherbelegungsprotokollen vorgenommen und im Weiteren ein theoretischer Algorithmus für die Lösung solcher Speicherbelegungsprotokolle aufgezeigt. Dabei wird auch ein Ansatz zur Implementierung beschrieben. Mithilfe eines Parsergenerators wird dann ein Parser für einen vorher definierten Ausschnitt der Programmiersprache C entworfen und generiert. Auf Basis dieses Parsers wird ein Interpreter für das gewählte Sprachfragment umgesetzt. Abschließend wird der Lösungsalgorithmus zusammen mit dem Interpreter in einer prototypischen Anwendung implementiert.

Kapitel 5 fasst die Ergebnisse der Arbeit kurz zusammen und gibt einen Ausblick für mögliche Erweiterungen und Verbesserungen des entworfenen Lösers.

2 Grundlagen formaler Sprachen und Grammatiken

2.1 Motivation Formale Sprachen

Sowohl in der Kommunikation zwischen Menschen als auch in der Mensch-Maschine-Kommunikation stellen Sprachen die Grundlage des Informations- und Aufgabenaustauschs dar. Es spielt dabei im Grunde keine Rolle, ob diese Sprache gesprochen (und gehört) oder geschrieben (und gelesen) wird. Die zugrunde liegende Intention ist immer, Kommunikation und damit Informationsaustausch zu ermöglichen. Bei der Kommunikation mit Computern geht es hierbei um verschiedenste Formen der Auftragserteilung. Der Mensch will in der Regel mit dem Computer kommunizieren, um dem Computer eine Aufgabe erledigen zu lassen. Damit der Computer die Anweisungen ausführen kann, müssen diese in für ihn verständlicher Sprache formuliert werden. Dazu ist es nötig, zu definieren, was genau in der vereinbarten Sprache erlaubt ist und somit durch den Computer verstanden werden kann und was nicht. Die Definition und Formalisierung von Sprachen stellen den Grundgedanken für die Theorie der formalen Sprachen dar. Die folgenden Kapitel widmen sich der grundsätzlichen Definition formaler Sprache und ihrer Bestandteile.

2.2 Alphabet

Ein Alphabet ist eine endliche, nichtleere Menge von Symbolen. Diese Menge wird häufig mit dem Zeichen $\sum$ (Sigma) bezeichnet. Alphabete können grundsätzlich aus beliebigen Zeichen bestehen, oftmals werden Buchstaben oder Zahlen verwendet, es gibt aber keine Einschränkung, die besagt, dass ein Alphabet nicht auch aus Smileys oder geometrischen Formen bestehen kann. Grundsätzlich kann jede nichtleere endliche Menge als Alphabet betrachtet werden. Einige Beispiele für mögliche Alphabete sind in Listing 1 aufgeführt.

$\sum_{binär} = \{0,1\}$ Das binäre Alphabet, das ausschließlich die Zeichen „0“ und „1“ enthält.

$\sum_{ziffern} = \{0,1,2,3,4,5,6,7,8,9\}$ Das Alphabet der Ziffern 0-9.

$\sum_{stimmung} = \{wütend, fröhlich, gelangweilt\}$

Listing 1 Beispiele für valide Alphabete

Das Alphabet $\sum_{stimmung}$ definiert zum Beispiel mögliche Stimmungen. Zu beachten ist hier, dass jedes Zeichen „atomar“ ist, also nicht kleinteiliger als sein Ganzes betrachtet werden kann. Das Element „wütend“ wird also formal, anders als in der natürlichen Sprache, nicht als Wort aus Buchstaben betrachtet, sondern nur als atomares und nicht teilbares Symbol (vgl. Böckenhauer u. Hromkovic 2013; Hedtstück 2012; Wagenknecht u. Hielscher 2022).

2.3 Wörter

Wörter sind endliche Folgen von Symbolen über einem gegebenen Alphabet und werden oftmals mit ω bezeichnet. Ein gegebenes Wort ω ist immer dann ein gültiges Wort über ein Alphabet Σ, wenn gilt, dass jedes Zeichen aus ω im gegebenen Alphabet Σ vorkommt, wie in Listing 2 definiert.

$$\omega = (\omega_1, \dots, \omega_k) \text{ mit } \omega_i \in \sum \ (i = 1, \dots, k)$$

Listing 2 Definition für ein Wort über einem Alphabet

Das Wort $\omega_1 = 011100$ ist ein gültiges Wort über dem Alphabet $\sum_{binär}$. Während $\omega_2 = 012010$ kein gültiges Wort über $\sum_{binär}$ darstellt. Beides sind aber gültige Worte über dem Alphabet $\sum_{ziffern}$.

Die Länge eines gegebenen Worts ω wird mit $|\omega|$ angegeben und gibt die Anzahl aller Zeichen an, aus denen das Wort besteht. Mehrfach vorkommende Zeichen werden hierbei auch mehrfach gezählt (Wagenknecht u. Hielscher 2022, S. 11). Das Wort $\omega = 001$ über dem binären Alphabet $\sum_{binär}$ ist ein gültiges Wort mit der Länge $|\omega| = 3$.

Das kürzeste Wort über jedem Alphabet ist das sogenannte „leere Wort“ ε. Das leere Wort ist ein Wort über jedem Alphabet und das einzige Wort mit der Länge 0.

2.4 Kleenesche Hülle und Wortmengen

Die Menge aller möglichen endlichen Wörter über einem gegebenen Alphabet Σ wird mit Σ^* bezeichnet. Diese Menge Σ^* wird als „Kleenesche Hülle“[1] bezeichnet. Die Kleenesche Hülle ist für jedes mögliche Alphabet eine abzählbar unendliche Menge (Böckenhauer u. Hromkovic 2013, S. 21; Wagenknecht u. Hielscher 2022, S. 12 ff.). Die Menge Σ^+ bezeichnet die Menge aller Wörter über einem gegebenen Alphabet ohne das leere Wort und wird manchmal auch als „Kleene-Plus“ oder „positive Kleene Hülle“ bezeichnet. Die Menge aller Wörter einer bestimmten Länge n über einem gegebenen Alphabet Σ wird mit Σ^n bezeichnet. Die Wortmenge Σ^1 entspricht also genau dem Alphabet Σ, während Σ^2 alle möglichen Wörter über dem Alphabet Σ mit der Länge 2 enthält. Das leere Wort ε ist das einzige Wort mit der Länge 0. Die formale Definition der Kleeneschen Hülle sowie der Wortmengen für Wörter fester Länge sind in Listing 3 abgebildet.

$$\Sigma^0 = \{\varepsilon\}$$
$$\Sigma^1 = \Sigma$$
$$\Sigma^{n+1} = \{xy \mid x \in \Sigma, y \in \Sigma^n\}$$
$$\Sigma^+ = \bigcup_{i=1}^{\infty} \Sigma^i$$
$$\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i$$

Listing 3 formale Definition Kleenesche Hülle und Wortmengen spezifischer Wortlänge

(Hoffmann 2009, S. 154; Hromkovič 2014, S. 19 f.)

Da die Kleenesche Hülle eine unendliche Menge abbildet, ist diese nur ausschnittsweise als Aufzählung definierbar, wohingegen die Wortmengen fester Länge immer endliche Mengen abbilden und somit auch aufgelistet werden können, wie in Listing 4 gezeigt.

$$\sum_{binär} = \{0,1\}$$
$$\sum^1_{binär} = \{0,1\}$$
$$\sum^2_{binär} = \{00,01,10,11\}$$
$$\sum^+_{binär} = \{0,1,01,10,00,11,001,\dots\}$$
$$\sum^*_{binär} = \{\varepsilon,0,1,01,10,00,11,001,\dots\}$$

Listing 4 Beispiele für Wortmengen und Ausschnitt Kleenesche Hülle

[1] Auch „endlicher Abschluss“, „Kleene-Abschluss“ oder „Sternhülle“

2.5 Formale Sprache

Eine formale Sprache, meist mit L bezeichnet[2], über ein Alphabet Σ ist, wie in Listing 5 gezeigt, eine beliebige Teilmenge über der zugehörigen Kleeneschen-Hülle des Alphabets.

$$L \subseteq \Sigma^*$$

Listing 5 Definition formale Sprache

Das bedeutet, dass sowohl die Sprachen $L_1 = \emptyset$ sprich die Sprache, die kein einziges Wort enthält, als auch die Sprache $L_2 = \Sigma^*$ also die Sprache, die jedes über dem Alphabet formbare Wort enthält, gültige Sprachen sind (Böckenhauer u. Hromkovic 2013, S. 24 f.; Hedtstück 2012, 6 ff.; Wagenknecht u. Hielscher 2022, S. 16). Für die formale Definition einer Sprache bedarf es geeigneter Mittel zur Definition, welche Worte Teil der Sprache sind. Für einfache Sprachen mit wenigen Worten ist es eine valide Lösung, alle möglichen Worte, die Teil der Sprache sind, aufzulisten und so die Sprache vollständig zu definieren. Für unendliche Sprachen oder endliche Sprachen mit sehr vielen möglichen Wörtern ist das keine verwendbare Notation mehr. Für die Definition einer unendlichen Sprache, wie zum Beispiel einer Programmiersprache, benötigt es eine andere Art der Definition. In den meisten Fällen folgen formale Sprachen, die in der Praxis eine Rolle spielen, einer bestimmten formalen Beschreibung für die Zulässigkeit ihrer Wörter. Als Beispiel für solche Sprachen seien Programmiersprachen wie z.B. C zu nennen oder auch die „Sprache“ der Palindrome[3] genannt. Hierbei spricht man von der grammatikalischen oder auch syntaktischen Struktur der Wörter. Die Wörter der Sprache sind also – in diesen Fällen - nicht willkürlich ausgewählt aus der Sternmenge des zugehörigen Alphabets, sondern folgen bestimmten Regeln. Die formalen Beschreibungen dieser Regeln heißen Grammatiken. Grammatiken beschreiben den Aufbau der zulässigen Wörter einer Sprache, ohne sich dabei mit deren Bedeutung oder Sinn zu befassen. Wie auch in der natürlichen Sprache sind also auch in formalen Sprachen Sätze möglich, die zwar syntaktisch zulässig, aber inhaltlich bedeutungslos sind. Ein Beispiel aus der natürlichen Sprache ist der Satz: „Das Maßband singt Briefe.“. Dieser ist nach den Regeln der deutschen Sprache absolut korrekt, ist aber für jeden Leser völlig ohne echten Sinn. Allerdings bedeutet das nicht, dass die syntaktische Struktur der Worte deshalb keinen Zusammenhang mit der Semantik hat. Für die Zuordnung einer Semantik zu den Worten einer Sprache ist es in der Regel nötig, dass sich die Bedeutung in der Syntax widerspiegelt. Den Vorgang, dass eine Semantik sich nach dem syntaktischen Aufbau des Wortes richtet, bezeichnet man als „syntaxgesteuerte Semantik“ (Hedtstück 2012, S. 23 f.; Louden 1994, S. 89).

[2] oder L_n mit $n \in \mathbb{N}$ wenn mehrere Sprachen bezeichnet werden

[3] Ein Palindrom ist ein Wort, das vorwärts und rückwärts gelesen exakt identisch ist.

2.6 Grammatiken

Eine formale Grammatik wird, wie in Listing 6 abgebildet, als 4-Tupel (oder Quadrupel) angegeben und häufig mit dem Buchstaben G bezeichnet.

$$G = \{N, T, P, S\}^4$$

Listing 6 formale Definition Grammatik

Die Menge N bezeichnet das Nichtterminalalphabet, die Symbole dieser Menge werden als Nichtterminale[5] oder auch Variablen bezeichnet. Der Bezeichner T oder Σ_T steht für das Terminalalphabet, welches die Terminale[6] enthält. Der Definition eines Alphabets folgend sind beide Mengen nicht leer und endlich. Im Kontext einer Grammatik kommt hinzu, dass beide Mengen disjunkt zueinander sein müssen, wie in Listing 7 dargestellt.

$$N \cap T = \emptyset$$

Listing 7 Definition Disjunktheit von Nichtterminal- und Terminal-Alphabeten

Die Menge P bezeichnet eine endliche Menge von Produktionen oder auch Ableitungsregeln[7]. Diese Regeln beschreiben bestimmte Ersetzungen auf den gegebenen Terminal- und Nichtterminalsymbolen. Die Ersetzungsregeln werden meistens in der Form $\alpha \rightarrow \beta$ angegeben, wobei gilt: $\alpha \in (N \cup T)^+$, $\beta \in (N \cup T)^*$ (Hedtstück 2012, 25; Hoffmann 2009, S. 156 f.). Oftmals wird statt eines Pfeils $\rightarrow$ auch ein Gleichheitszeichen $=$ oder $:=$ verwendet. Bei der Notation der Produktionen kommt häufig eine Kurzschreibweise zum Einsatz, um Regeln mit gleicher linker Seite, also gleichem α zusammenzufassen. Bei dieser verkürzten Schreibweise wird in der Regel ein senkrechter Strich „|“ zum Anzeigen von Alternativen genutzt. Für eine Grammatik $G = (\{S, A, B\}, \{a, b\}, P, S)$ ist also für die Produktionen in P statt der in Listing 8 abgebildeten Schreibweise auch die verkürzte Schreibweise in Listing 9 möglich.

$$S = AB$$
$$A = AA$$
$$A = aB$$
$$A = a$$
$$B = b$$

Listing 8 Produktionen Beispiel Grammatik

$$S = AB$$
$$A = AA \mid aB \mid a$$
$$B = b$$

Listing 9 verkürzte Produktionen Beispiel Grammatik

Der letzte Bestandteil S bezeichnet das sogenannte Startsymbol oder auch die Startvariable. S ist ein Symbol aus dem Nichtterminalalphabet und definiert den Ausgangspunkt für die Worterzeugung.

---

[4] Die genauen Bezeichner der einzelnen Bestandteile unterscheiden sich je nach Literatur sehr stark, werden aber in der Regel gleich definiert.
[5] auch nichtterminal-Symbole oder Variablenmenge
[6] Auch terminal-Symbole
[7] Auch Produktionsregeln, Grammatikregeln, Regeln

Für die formale Darstellung der Anwendung einer oder mehrerer Produktionen, welche ein Wort in ein anderes überführen, wird eine Ableitungsrelation " $\Rightarrow$ "[8] definiert. Wenn gilt $V = N \cup T$ und $x, y, a, b \in V^*$ mit der Form $x = aub$ und $y = avb$, dann gilt $x \Rightarrow y$ genau dann, wenn es eine Produktion für $u \rightarrow v$ gibt. Man sagt *„x wird unmittelbar überführt in y“* (Hedtstück 2012, S. 28). Die Ableitungsrelation stellt also die direkte Ableitung aus einem Wort in ein anderes dar, wenn genau eine Produktion angewendet wird. Zusätzlich wird $\Rightarrow^*$ definiert, als Ableitung in mehreren, aber endlich vielen Schritten (Hoffmann 2009, S. 156 f.). Mittels wiederholter Anwendung der Produktionen auf das Startsymbol der Grammatik lassen sich nun die Worte der zugehörigen Sprache erzeugen. Die Sprache L, die von einer Grammatik G erzeugt wird, wird mit $L(G)$ bezeichnet. Die Sprache $L(G)$ definiert sich als die Sprache der Wörter, welche vom Startsymbol S der Grammatik unter Anwendung der Regeln aus P gebildet werden können.

$$L(G) = \{y \in T^* \mid S \Rightarrow^* y\}$$

Listing 10 Definition formale Sprache als Menge ihrer Wörter (Hoffmann 2009, S. 157)

Die Definition in Listing 10 verdeutlicht nochmals, dass die Wörter der Sprache $L(G)$ nur aus Terminalsymbolen bestehen. Ein solches Wort wird auch als Terminalwort bezeichnet. Im Umkehrschluss bedeutet dies ebenfalls, dass die Sprache $L(G)$ sich als Menge der Terminalwörter definiert, welche sich aus S ableiten lassen. Somit kann man auch sagen, dass ein gegebenes Wort ω genau dann zur Sprache $L(G)$ gehört, wenn es eine Ableitung der Form $S \overset{*}{\Rightarrow} \omega$ gibt (Hedtstück 2012, S. 29).

[8] Je nach Literatur auch $\underset{G}{\Rightarrow}$ um verschieden Grammatiken zu unterscheiden (G als Bezeichner der Grammatik)

2.7 Chomsky-Hierarchie

Der amerikanische Sprachwissenschaftler Noam Chomsky hat im Jahre 1956[9] ein Regelwerk beschrieben, nach welchem sich Grammatiken in vier verschiedene Klassen einteilen lassen (Hoffmann 2009, S. 160). Diese Klassen sind hierarchisch aufgebaut und werden mit einer Nummerierung von 0 bis 3 benannt. Die Klassen lassen sich relativ einfach durch die Einschränkungen beschreiben, welche sie an die Formen der Produktionsregeln ihrer Grammatiken stellen.

Bei Typ-0-Grammatiken spricht man von sogenannten Phrasenstrukturgrammatiken[10]. Hier gibt es keine weiteren Einschränkungen, als solche, die schon durch die Definition der Produktionen gegeben sind. Per Definition ist jede Grammatik eine Typ-0-Grammatik.

Typ-1-Grammatiken werden auch als kontextsensitive Grammatiken bezeichnet. Eine Grammatik wird dann als kontextsensitiv bezeichnet, wenn alle Produktionsregeln die Form $a \rightarrow b$ mit $\alpha \in (N \cup T)^+$, $\beta \in (N \cup T)^*$ aufweisen. Weiterhin muss für alle Produktionen $|a| \leq |b|$ gelten. Diese Einschränkung wird auch als Längenmonotonie bezeichnet und hat zur Folge, dass das Ableiten eines Wortes, egal welche Produktion der Grammatik verwendet wird, nicht zu einer Verkürzung eben dieses führt. Hiervon kann es allerdings eine Ausnahme geben, nämlich kann die Produktion $S \rightarrow \varepsilon$ (S = Startsymbol) eine gültige Produktion sein, wenn S in keiner Produktion auf der rechten Seite vorkommt (Wagenknecht u. Hielscher 2022, S. 41).

Um eine Typ-2-Grammatik oder auch kontextfreie Grammatik handelt es sich, wenn jede Produktion dem Format $\alpha \rightarrow \beta$ mit $a \in N$ entspricht. Die linke Seite jeder Produktionsregel darf also nur aus einem einzigen Nichtterminal bestehen. Die rechte Seite der Ersetzung darf wie bei den Typ-1-Grammatiken aus einer beliebigen nichtleeren Folge von Terminalen und Nichtterminalen bestehen. Es gelten die gleichen Ausnahmeregeln für die Ableitung des Startsymbols zum leeren Wort.

Für die Definition der Typ-3-Grammatiken, der sogenannten regulären Grammatiken, wird zusätzlich zu den Einschränkungen einer kontextfreien Grammatik gefordert, dass für alle Produktionen der Grammatik die rechte Seite der Ersetzung entweder aus einem einzelnen Terminalsymbol oder einem Terminalsymbol gefolgt von einem Nichtterminalsymbol besteht. Formal bedeutet das, dass jede Produktion die Form $\alpha \rightarrow \beta$ oder $\alpha \rightarrow \beta\gamma$ mit $\alpha, \gamma \in N$ und $\beta \in T$ haben muss. Da bei dieser Art von Definition das Wort rekursiv auf der rechten Seite erweitert wird, nennt man diese auch rechtslineare Grammatiken. Auch die umgedrehte Definition, die die Produktionsform $\alpha \rightarrow \gamma\beta$ zulässt, ist möglich und definiert die sogenannten linkslinearen Grammatiken. Beide Definitionen beschreiben die gleiche Sprachklasse (Hedtstück 2012, S. 32 f.; Hoffmann 2009, S. 160).

Durch die hierarchische Definition und die Regel der Längenmonotonie für kontextsensitive Sprachen sind theoretisch auch in Typ-2- und Typ-3-Sprachen keine Ableitungen von Nichtterminalen zu Epsilon möglich. Durch die Ausnahmeregel, dass das Startsymbol zu S abgeleitet werden darf, wenn es auf keiner rechten Seite vorkommt, bleibt es weiterhin möglich, das leere Wort abzuleiten. Weitere Regeln der Form $A \rightarrow \varepsilon \; mit \; A \in N \backslash \{S\}$, könnte man theoretisch zulassen, ohne dass sich dadurch die Menge der erzeugbaren Sprachen verändern würde (Hedtstück 2012, S. 33).

[9] manchmal wird auch 1957 angegeben

[10] Auch Chomsky-Grammatiken oder unbeschränkte Grammatiken

Die so definierten Mengen von Grammatiken bestimmen über ihre erzeugten Sprachen sogenannte Sprachklassen. Diese werden passend zu ihrer erzeugenden Grammatik Typ-n-Sprache bezeichnet. Eine Typ-2-Grammatik erzeugt also auch eine Typ-2-Sprache. Im Umkehrschluss gilt, dass es zu jeder Typ-n-Sprache eine erzeugende Typ-n-Grammatik gibt. Die Sprachklassen werden formal oftmals mit einem stilisierten großen L mit tiefgestellter n-Typisierung bezeichnet. Die Klasse der Typ-0-Sprachen wäre also formal $\mathcal{L}_0$. Diese Sprachklassen werden äquivalent zu ihren Grammatiken als unbeschränkte, kontextsensitive, kontextfreie oder reguläre Sprachen bezeichnet. Diese Sprachklassen bilden echte Teilmengen voneinander ab, wie in Abbildung 1 dargestellt.

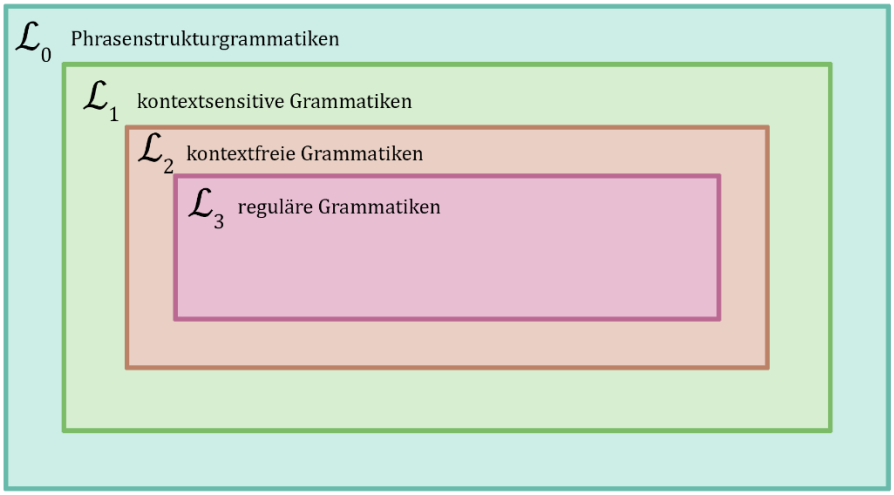


Abbildung 1 Visualisierung der Teilmengenbeziehung zwischen Sprachklassen nach Chomsky[11]

Es gilt also: $\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0$. Hierbei ist zu beachten, dass $\mathcal{L}_0$ nicht die Menge aller Sprachen abbildet, sondern nur die Mengen der Sprachen, welche überhaupt mit einer Grammatik abbildbar sind. Allerdings ist auch die Sprachklasse $\mathcal{L}_0$ eine echte Teilmenge über der Menge aller Sprachen für ein festes Alphabet (Socher 2008, S. 98).

[11] Die Größe der Felder steht nicht im Zusammenhang mit der Mächtigkeit der Sprachklasse.

2.8 Kontextfreie Grammatiken und Sprachen

Die Bezeichnung kontextfrei für die Typ-2-Grammatiken der Chomsky-Hierarchie leitet sich aus der Form ihrer Produktionen ab. Da bei diesen Produktionen auf der linken Seite der Produktion immer nur ein einzelnes Nichtterminal steht, kann dieses unabhängig von seinen Nachbarsymbolen (egal ob Terminal oder Nichtterminal) ersetzt werden. Die Ersetzungen können also ohne Betrachtung des Kontextes erfolgen (Louden 1994, S. 85; Wirth 2008, S. 13). Die Kontext Für die Darstellung von Ableitungssequenzen in kontextfreien Grammatiken hat sich die Darstellung mittels sogenannter „Ableitungsbäume“[12] als geeignet erwiesen. Diese Ableitungsbäume eignen sich gut, um die syntaktische Struktur eines Wortes zu visualisieren und zu verarbeiten (Hopcroft et al. 2011, S. 219). Für eine Grammatik $G = (\{S\}, \{a, b\}, P, S)$ mit den Produktionen $P = \{S \rightarrow SS \mid aSa \mid bSb \mid aa \mid bb\}$ lässt sich für das Wort $aabaabaa$ die in Listing 11 dargestellte Ableitungssequenz herleiten.

$$S \Rightarrow SS \Rightarrow aaS \Rightarrow aaSS \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aabaabaa$$

Listing 11 Ableitungssequenz für das Wort „aabaabaa“

Die Ableitungssequenz aus Listing 11 lässt sich grafisch als Baum visualisieren, wie in Abbildung 2 gezeigt.

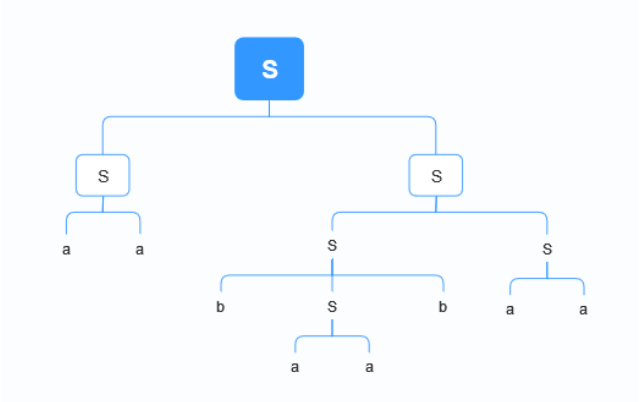


Abbildung 2 Ableitungsbaum für Sequenz „aabaabaa“

Ableitungsbäume sind einfache Strukturen, die die gewählten Ersetzungen dokumentieren und die Beziehungen der Symbole zueinander optisch darstellen. Es handelt sich um gerichtete Bäume, welche zu einer spezifischen Worterzeugung auf einer Grammatik gehören. Die Wurzel ist hierbei immer mit dem Startknoten S beschriftet, da nur damit eine valide Worterzeugung starten kann. Alle inneren Knoten des Baums stellen Nichtterminale dar und sind auch entsprechend beschriftet. Alle Blätter eines solchen Baumes sind mit Terminalen versehen. Liest man alle Blätter von links nach rechts, ergibt sich das ableitete Wort. Für jeden inneren Knoten $B \in N$ mit den direkten Nachfolgern $C_1, \dots, C_n$ gibt es eine Produktion $B \rightarrow C_1, \dots, C_n$ in P (Böckenhauer u. Hromkovic 2013, S. 194; Hedtstück 2012, S. 98). Mithilfe solcher Ableitungsbäume lässt sich auch eine Schwierigkeit beim Arbeiten mit kontextfreien Grammatiken aufzeigen, nämlich deren mögliche Mehrdeutigkeit. Die Worte einer Sprache über einer gegebenen Grammatik wurden definiert als alle Terminalworte, welche sich in beliebig vielen Schritten aus dem Startsymbol ableiten lassen. Diese Definition hat aber nicht genauer definiert, wie und wann

[12] auch: „Parsebaum, Strukturbaum oder Syntaxbaum“ Hedtstück (2012, S. 98).

welche Produktion angewendet wird, für den Fall, dass mehrere anwendbar sind. Für das gegebene Wort wäre auch eine alternative Ableitungssequenz wie in Listing 12 möglich.

$$S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aabaabaa$$

Listing 12 alternative Ableitungssequenz für "aabaabaa"

Die bloße Möglichkeit, einen Begriff auf verschiedene Arten abzuleiten, ist noch kein Problem, da auch mehrere Ableitungen für den gleichen Ableitungsbaum existieren können. Dies beruht darauf, dass Syntaxbäume die Reihenfolge der angewendeten Regeln abstrahieren. Solange die Ableitungssequenzen auf den gleichen Ableitungsbaum hinauslaufen, stellt dies noch kein Problem dar, da der resultierende strukturelle Aufbau der gleiche ist (Hoffmann 2009, S. 159; Louden 1994, S. 88 ff.). Im Fall der in Listing 11 gezeigten Ableitungssequenz lässt sich durch Betrachtung des zugehörigen Ableitungsbaums in Abbildung 3 zeigen, dass der Ableitungsbaum der zweiten Sequenz sich vom Ableitungsbaum der ersten Sequenz strukturell unterscheidet.

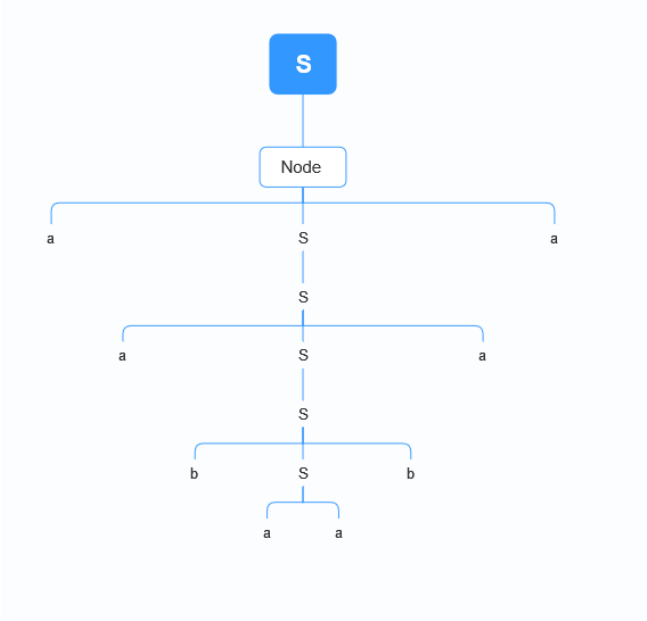


Abbildung 3 Ableitungsbaum für alternative Sequenz für „aabaabaa“

Diese Mehrdeutigkeit ist oftmals unerwünscht, da sie es schwierig macht, bei der Verarbeitung eines Ausdrucks zu entscheiden, welche Strukturdarstellung genutzt werden soll. Deswegen fordert man in der Regel, dass nur sogenannte Links- oder Rechtsableitungen betrachtet werden sollen. Bei diesen Ableitungsformen wird immer das am weitesten links- oder rechtsstehende Nichtterminal ersetzt. Durch diese Einschränkung wird die Anzahl der mehrdeutigen Strukturbäume für valide Ausdrücke stark reduziert. Die in Listing 11 dargestellte Ableitung für das Wort „aabaabaa“ entspricht nämlich keiner Links- oder Rechtsableitung. Die zweite Sequenz stellt eine korrekte Links- und auch Rechtsableitung dar. Dass Links- und Rechtsableitungen die gleichen Strukturbäume erzeugen, ist nicht allgemeingültig (Hoffmann 2009, S. 158 f.). Allerdings lässt sich zeigen, dass auch mit der Einschränkung auf linksseitige Ableitungen es noch mindestens eine weitere Möglichkeit zur Ableitung des Wortes gibt, wie in Listing 13 dargestellt.

$$S \Rightarrow SS \Rightarrow aaS \Rightarrow aaSS \Rightarrow aabSbS \Rightarrow aabaabS \Rightarrow aabaabaa$$

Listing 13 Alternative linksseitige Ableitung für "aabaabaa"

Der grafische Ableitungsbaum für die in Listing 13 dargestellte Ableitungssequenz entspricht dem in Abbildung 2 gezeigten, was verdeutlicht, dass ein solcher Baum die Reihenfolge der angewandten Produktionen abstrahiert. Die Existenz von mehreren möglichen linksseitigen Ableitungsbäumen beweist auch die Mehrdeutigkeit der gegebenen Grammatik.

Grammatiken, die solche mehrdeutigen Ableitungen zulassen, werden als mehrdeutige Grammatiken bezeichnet und sind in der praktischen Anwendung oftmals unerwünscht. Häufig lassen sich mehrdeutige spracherzeugende Grammatiken in eindeutige umformen. Hierbei werden die Produktionen angepasst und zusätzliche Nichtterminale eingeführt. Die Umformung von einer mehrdeutigen zu einer eindeutigen Grammatik ist oftmals kein einfacher Prozess und kann zu sehr komplexen Grammatiken führen. Allerdings gibt es auch Sprachen, die ausschließlich durch mehrdeutige Grammatiken beschreibbar sind. Solche Sprachen werden dann als inhärent mehrdeutig bezeichnet (Hoffmann 2009, S. 159; Louden 1994, S. 94 ff.).

2.9 Backus-Naur-Form

Zur Notation kontextfreier Grammatiken wird oftmals die von J. Backus und P. Naur eingeführte und nach ihnen benannte „Backus-Naur-Form“ BNF verwendet. Genauer gesagt, wird heute häufig eine erweiterte Backus-Naur-Form EBNF verwendet, welche die BNF um Notationen zum Ausdrücken von rekursiven und optionalen Elementen erweitert. Die genaue Notation der BNF bzw. auch der EBNF wird in der Literatur nicht immer exakt befolgt, die Grundstruktur der Produktionen wird dabei aber beibehalten. Produktionen werden in der Form einer linken und rechten Seite angegeben, wobei auf der linken Seite ein einzelnes Nichtterminal steht. Nichtterminale werden dabei meistens kenntlich gemacht, indem sie in spitze Klammern gesetzt werden. Als Operator zwischen linker und rechter Seite wird in der BNF eine Kombination aus zwei Doppelpunkten gefolgt von einem Gleichheitszeichen verwendet, wobei die Doppelpunkte in der EBNF meist weggelassen werden. Auf der rechten Seite der Produktion stehen beliebige Folgen von Terminalen und Nichtterminalen. Auf der rechten Seite der Produktion können Alternativen auch in der BNF mittels eines senkrechten Strichs | getrennt und somit in einer einzigen Produktion notiert werden. Leerzeichen oder Platzhalter können benutzt werden, um die Lesbarkeit zu erhöhen (Backus et al. 1960, S. 108 f.). Die größte Schwierigkeit, um komplexere Sprachen mittels BNF zu beschreiben, ist die Darstellung von optionalen oder repetitiven Elementen. In der klassischen BNF können Grammatiken sehr schnell sehr komplex werden, sobald Symbole optional oder wiederholend zugelassen werden sollen, da hierfür immer neue rekursive Produktionen eingeführt werden müssen (Wirth 2008, S. 7 f.). Besonders bei bestimmten Anzahlen von Wiederholungen kann die Anzahl der Regeln und der unter Umständen nötigen zusätzlichen Nichtterminale schnell sehr groß werden. Um der Wichtigkeit einer einheitlichen Notation für die formale Dokumentation von Grammatiken Rechnung zu tragen, wurde die erweiterte BNF im Jahre 1996 unter dem Name „Extended BNF“ (ISO/IEC 1996) im Standard ISO/IEC 14977 formal standardisiert. Wie auch die vielen nicht-standard-Versionen einer EBNF enthält sie vereinfachte Notationen für optionale und repetitive Elemente. Im Gegensatz zu vielen alternativ Erweiterungen definiert sich aber noch einiges mehr. So werden neben den eckigen Klammern „[ ]“, welche ein oder mehrere optionale Elemente einschließen, und den geschweiften Klammern „{ }“, welche sich beliebig häufig vorkommende Elemente anzeigt, auch runde Klammern „( )“ zur Gruppierung von Elementen und ein Stern „*“ für die explizite Angabe einer Wiederholungsanzahl eingeführt. Auch werden Kommentare und eine Notation zum Ausschluss gewisser Symbole definiert. Neben diesen zusätzlichen

Symboliken und Notationsmöglichkeiten wurde die zugrundeliegende Notationsweise der Regeln angepasst. So wird nur noch ein klassisches Gleichheitszeichen $=$ statt der ursprünglichen BNF-Notation mit den 2 Doppelpunkten und einem Gleichheitszeichen $::=$ genutzt. Auch wird mit dem Komma , ein expliziter Konkatenationsoperator eingeführt und ein explizites Terminalsymbol für das Ende einer Produktion. Diese Änderungen sollen zum einen eine kürzere und einfachere Notation vieler Regeln ermöglichen und zum anderen eine einfachere Lesbarkeit und Verarbeitbarkeit ermöglichen (ISO/IEC 1996, S. 1 ff.). Abseits des ISO/IEC-Standards gibt es unter anderem auch einen RFC-Standard der IETF unter der Bezeichnung „ABNF“, was für „Augmented BNF“ steht und in Deutsch ebenfalls mit erweiterte BNF übersetzt wird.[13] Die aktuelle Version des ABNF-Standards ist dabei deutlich neuer als der bis heute nicht aktualisierte Standard der EBNF. Die Definition der ABNF geht noch ein Stück weiter als die der EBNF und definiert einen festen Zeichensatz, mit dem sich zum Beispiel in Form von Hex - oder Dezimalangabe auch nicht darstellbare Zeichen oder Byte-Werte abbilden lassen. Auch werden Zeichenreihen unterstützt, während man also in der EBNF jedes Zeichen einzeln aufzählen müsste: $hexchar = 'A' \mid 'B' \mid 'C' \mid 'D' \mid 'E' \mid 'F'$; könnte man in der ABNF schreiben: $hexchar = \%\mathrm{x}41 - 46;$[14] . Die ABNF definiert gegenüber der BNF und der EBNF eine deutlich andere Symbolik für viele Regeldarstellung (IETF, S. 3 ff.). In der praktischen Nutzung von Grammatiken ist am häufigsten die EBNF zu finden, wobei auch dabei die Implementierungen teilweise vom standardisierten Format abweicht.

[13] Manchmal wird auch „angereicherte BNF“ als Übersetzung verwendet. Im Folgenden wir aber der einfachhalt halber mit dem Kürzel ABNF gearbeitet.

[14] 41-46 sind die Buchstaben A-F in HEX Darstellung

3 Kontextfreie Grammatiken im Compilerbau

3.1 Einleitung

Auch wenn kontextfreie Grammatiken und viele Konzepte formaler Sprache ursprünglich für die Beschreibung von natürlicher Sprache konzipiert und erdacht wurden, erwiesen sie sich auch als geeignetes Mittel, um Konzepte der praktischen Informatik formal zu definieren (Hopcroft et al. 2011, S. 579). Gerade für die Definition der Syntax von Programmiersprachen erwiesen sich kontextfreie Grammatiken als hilfreich und lösten in Folge viele informelle und kaum automatisch verarbeitbare Formen der Sprachbeschreibungen ab (Louden 1994, S. 20 f.). Kontextfreie Grammatiken sind deutlich ausdrucksstärker als reguläre Grammatiken und ermöglichen dadurch auch das Ausdrücken komplexerer Sprachkonstrukte. Als Konsequenz aus der Möglichkeit, die syntaktische Struktur einer Programmiersprache mittels kontextfreier Grammatiken beschreiben zu können, folgt auch die Möglichkeit, mittels dieser Grammatiken gegebene Programme einer Programmiersprache analytisch betrachten zu können. Der Quellcode eines Programms – im Folgenden meist einfach als Programm bezeichnet – tritt dabei an die Stelle des Wortes einer formalen Sprache und die Menge der Schlüsselwörter mit allen weiteren zulässigen Zeichen an die Position der Symbole. Durch diese Betrachtung können Programmiersprachen als formale Sprachen definiert und entsprechend verarbeitet werden. Durch diese Anwendung von Erkenntnissen der theoretischen Informatik und der Sprachwissenschaften in der praktischen Informatik wird es nun möglich, analytische Erkenntnisse für Programme einer Programmiersprache zu gewinnen. Als eine der frühsten Anwendung einer solchen Analyse gilt es für ein gegebenes Programm, unter Zuhilfenahme einer kontextfreien Grammatik für die zugehörige Programmiersprache, die Struktur des Programms zu analysieren und mithilfe eines Parsebaumes darzustellen (Hopcroft et al. 2011, S. 231). Ein Parsebaum ist dabei äquivalent zu einem Ableitungsbaum. Der Unterschied in der Verwendung der Begrifflichkeiten liegt meistens darin, dass bei der Erzeugung eines Wortes meistens von einem Ableitungsbaum gesprochen wird, und bei einem für ein gegebenes Wort erzeugten Baum von einem Parsebaum.

Diese Analysen von Programmen ist ein wichtiger Schritt beim Bau von Compilern und Interpretern und spielt eine wichtige Rolle bei der Entwicklung und Verwendung von höheren Programmiersprachen[15]. Programmiersprachen, welche heute am meisten Verwendung finden, wie zum Beispiel Python, Java, C# oder C, sind keine Maschinensprachen und werden vom Computer selbst nicht verstanden. Um Programme einer solchen Sprache vom Computer ausführen lassen zu können, müssen diese erst übersetzt werden. Die Übersetzung einer Programmiersprache in Maschinensprache erfolgt in der Regel durch ein weiteres Programm, den sogenannten Compiler. Ein Compiler ist ein Übersetzer, der ein Programm aus einer Quellsprache in eine Zielsprache übersetzt. Häufig ist die Zielsprache eine Maschinensprache, welche von einem Computer verarbeitet werden kann. Ein Compiler ist eine Form eines Sprachprozessors, der ein Zielprogramm ausgibt. Eine andere Form eines Sprachprozessors ist ein Interpreter, welcher im Gegensatz zum Compiler kein Zielprogramm ausgibt, sondern das Quellprogramm direkt ausführt. Da ein Interpreter das Quellprogramm immer nur Schritt-für-Schritt ausführt und dadurch weniger Optimierung vornehmen kann als ein Compiler, ist eine interpretierte Ausführung meistens langsamer.[16] Dafür entfällt bei Interpretern die Übersetzungszeit bis zum Start

[15] Da die meisten gängigen Programmiersprachen als höhere Programmiersprachen gelten, wird im Folgenden auf den Zusatz „höhere“ verzichtet.

[16] Bei ähnlicher oder gleicher Code-Basis.

der Ausführung. Beide Konzepte haben in der Praxis beide ihre Bedeutung und finden in vielen Sprachen verbreitete Anwendung. Programmiersprachen, die in der Regel interpretiert werden, sind zum Beispiel PHP[17] und PERL. Sprachen, deren Ergebnis normalerweise mit einem Compiler vor der Ausführung in Maschinencode übersetzt werden, sind zum Beispiel C (auch C++ und C#) und Pascal. Es gibt auch Sprachen, die beide Ansätze vermischen, wie z.B. Java mit der Generierung von Bytecode (Kompilierung), welcher dann erst zur Laufzeit in Maschinensprache übersetzt wird (Interpretation). Java optimiert diesen Interpretationsvorgang zusätzlich mit Kompilierungen zur Laufzeit durch einen sogenannten Just-in-time-Compiler (JIT-Compiler) (Aho et al. 2008, S. 2 ff.; Louden 1994, S. 24 ff.).

Beim Compilerbau wird der Ablauf des Übersetzungsvorganges oftmals in mehrere Phasen eingeteilt, da diese Übersetzung nicht in einem Schritt vonstattengeht. Diese Phasen wiederum werden in zwei Bereiche eingeteilt: das Frontend und das Backend. Das Frontend bezeichnet dabei die analytischen Phasen der Verarbeitung. Die Frontend-Phasen sind in der Regel sowohl bei Compilern als auch bei Interpretern anzufinden. Während im Frontend der Programmcode analysiert, strukturiert und auf syntaktische und semantische Korrektheit geprüft wird, folgen im Compiler anschließend die Synthesephasen, welche die Codegenerierung mit ihren verschiedenen Zwischenschritten beschreibt. Im Interpreter wird kein Code generiert, da der analysierte Code direkt ausgeführt werden soll. Somit erfolgt eine direkte Ausführung der in der Analyse erkannten Code-Bestandteile und keine zusätzliche Synthesephase. Die analytischen Frontend-Phasen bestehen dabei aus der lexikalischen Analyse, welche den Source-Code in verarbeitbare Tokens mappt, die syntaktische Analyse, die die Struktur des Programms bestimmt, und der semantischen Analyse, die semantische Regeln prüft. Diese Phasen laufen dabei nicht zwangsläufig strikt nacheinander ab, sondern sind häufig miteinander verbunden, um unnötigen Overhead zu vermeiden (Aho et al. 2008, S. 6 ff.; Louden 1994, S. 24 ff.). Die Backend-Phasen können im weiten Sinne als Optimierung und Codegenerierung bezeichnet werden und sind stark Compilerabhängig. Im Folgenden sollen nur die Frontend-Phasen genauer betrachtet werden und ihre Funktion und Arbeitsweise, sowie ihre Abhängigkeiten voneinander.

[17] Zumindest bis vor Version 7. Ab Version 7 wird auch PHP-Code teilweise kompiliert.

3.2 Lexikalische Analyse

Die lexikalische Analyse ist in der Regel der erste Schritt beim automatisierten Verarbeiten von Sprachen. Der Vorgang wird auch als lexen, scannen oder tokenizen bezeichnet und wird vom sogenannten Lexer, Scanner oder auch Tokenizer ausgeführt. Die Notwendigkeit einer lexikalischen Analyse (oder zumindest eines ähnlichen Prozesses) ergibt sich daraus, dass ein Quellprogramm in der Regel als Fließtext oder Zeichenstream vorliegt (Aho et al. 2008, S. 135). Das heißt, das Quellprogramm besteht in seiner ursprünglichen Form aus einer Sequenz von einzelnen Zeichen (z.B. Buchstaben, Leer- und Sonderzeichen), welche in der Regel nicht den Symbolen des Alphabets der abzubildenden Sprache gleichen (Wirth 2008, S. 1).

Der Lexer hat nun die Aufgabe, diese Eingabezeichenkette in „bedeutungsvolle Sequenzen“ (Aho et al. 2008, S. 7) sogenannte Lexeme zu zergliedern und damit für die weitere Analyse vorzubereiten. Aus den erkannten Lexemen generiert der Lexer sogenannte Tokens, welche eine abstrakte Repräsentation der Symbole der entsprechenden Sprache darstellen. Der genaue Aufbau von Token wird häufig unterschiedlich beschrieben, da er auch abhängig von der genauen Implementierung ist. Im Allgemeinen besteht ein Token aber aus einem Namen und, wenn nötig, einem Verweis auf die Symboltabelle. Klassen von Token können zum Beispiel – aber nicht ausschließlich – reservierte Wörter der Programmiersprache, Literale und Konstanten, Operatoren und Bezeichner sein. Oftmals wird zu einem Token auch ein Eintrag in eine Symboltabelle angelegt, wo dann weitere Informationen wie das erkannte Lexem oder die Position des Lexems hinterlegt werden. Die Symboltabelle enthält Informationen, die in der weiteren Verarbeitung unterschiedlich genutzt werden können (Aho et al. 2008, S. 135 ff.; Louden 1994, S. 80 ff.). Während bei einem Token für ein einzelnes reserviertes Wort wie „while“ keine weiteren Inhaltsspezifischen Attribute erforderlich sind, weil nur genau dieses eine Lexem auf diese Tokenklasse passt, ist bei einem Token für Vergleichsoperatoren zwangsläufig auch das erkannte Lexem von Bedeutung, um bei der weiteren Analyse die genaue Operation erkennen zu können. Im Falle einer eindeutigen Zuordnung von Token zu Lexem kann auch der Eintrag in die Symboltabelle entfallen (Aho et al. 2008, S. 138 f.).

Die Beschreibung der Tokenklassen, genauer gesagt der zu einer Tokenklasse gehörigen Lexeme, erfolgt häufig in Form von regulären Ausdrücken oder einer Notation, die der Beschreibung von regulären Ausdrücken sehr ähnlich ist. Durch die Nutzung regulärer Ausdrücke - welche die Beschreibung einer regulären Sprache darstellen - für die Beschreibung der zu den Tokenklassen gehörenden Lexeme wird für die Implementierung der Erkennung oftmals ein endlicher Automat genutzt. Die genaue Form und Zulässigkeit der Notation für die Lexer-Regeln ist aber abhängig von der Implementierung des spezifischen Lexers. Manchmal wird dabei von einer erweiterten Form der regulären Ausdrücke gesprochen, da die meisten gängigen Formen zusätzliche formale Mechanismen wie z.B. Zeichenklassen (a-z, A-Z, 0-9 etc.) und Operatoren für optionale und widerholende Symbole unterstützen. Diese Erweiterungen dienen primär dem Komfort und erhöhen in der Regel nicht die Mächtigkeit der beschriebenen Sprachen (Wilhelm et al. 2012, S. 31 ff.). [18]

Die so definierten Lexer-Regeln können oftmals auch bereits Aktionen enthalten, welche der Lexer bei der Erkennung des Tokens ausführt. Diese Aktionen können zum Beispiel das Befüllen der

[18] In vielen Mustererkennungen werden die Musterangaben auch reguläre Ausdrücke genannt, obwohl sie tatsächlich eine größere Sprachklasse beschreiben.

Symboltabelle sowie das Verwerfen von Tokens sein. Auch wenn das Verwerfen im ersten Moment nicht wie eine sinnvolle Aktion klingt, ist es doch ein elementarer Bestandteil bei der Vorverarbeitung für den eigentlichen Parser. In fast allen gängigen Sprachen werden zum Beispiel Kommentare in einem solchen vorgelagerten Schritt verworfen, da diese keinen funktionellen Einfluss auf die Struktur des Programmcodes haben. In formatfreien Sprachen werden zum Beispiel in der Regel auch Tokens für Leerzeichen oder Zeilenumbrüche verworfen, wenn diese nicht explizit Teil eines anderen Tokens sind. Manchmal wird die Aufgabe des Erkennens und Filterns auch als zwei getrennte Vorgänge innerhalb der lexikalischen Analyse beschrieben, wobei auch dann beide Komponenten sehr eng gekoppelt sind (Aho et al. 2008, S. 133 ff.; Wilhelm et al. 2012, 3-5, 40-44). Auch wenn die Erkennung der Lexeme durch endliche Automaten, beziehungsweise die Beschreibung der Lexeme durch reguläre Ausdrücke eine verhältnismäßig einfache Implementierung ermöglicht, gibt es im Kontext von Programmiersprachen verschiedene Hürden bei der Erkennung der Tokens. Ein häufig genutztes Beispiel für eine solche Hürde ist die Unterscheidung von reservierten Wörtern und Bezeichnern (zum Beispiel bei der Definition von Variablen). Zwar sind in den meisten modernen Programmiersprachen reservierte Wörter nicht als Bezeichner zulässig, sie können aber sehr wohl Teil eines solchen sein. So darf zum Beispiel ein Bezeichner in C# nicht if lauten, $ifNextIsValid$ wäre aber valide und muss als entsprechender Bezeichner erkannt werden. Es muss also sichergestellt werden, dass überlappende Lexeme korrekt erkannt und der richtigen Tokenklasse zugeordnet wird. Hierfür gibt es verschiedene Möglichkeiten zur Auflösung solcher Probleme. Eine der geläufigsten Lösungen ist das Akzeptieren der längsten matchenden Zeichenfolge. Diese Regel wird meistens als „longest match“ oder auch „maximal munch“ bezeichnet. Zwar löst sie das Problem von überlappenden Bezeichnern mit Schlüsselwörtern in der Regel ausreichend, sorgt aber auch an anderen Stellen dafür, dass eine längere Übereinstimmung, möglichweise unbeabsichtigt, gematcht wird. Eine Alternative stellt die „First-wins-Strategie“ (Wagenknecht u. Hielscher 2022, S. 130 ff.) dar, welche sich immer für den ersten Match entscheidet. Keine der Strategien kann das Problem vollständig lösen. Ein häufig gezeigtes Beispiel ist der Rechts-Shift-Operator $>>$ in C++ (oder auch anderen Sprachen). Bei einem per Definition validen Typisierungsausdruck wie $T < T < T >>$ würden die letzten zwei spitzen Klammern als Rechts-Shift-Operator erkannt werden, statt als zwei einzelne schließende Klammern. Als Lösung dafür wäre es zum Beispiel möglich, das Erkennen des Rechts-Shift-Operators komplett zu entfernen und rechte-spitze-Klammern immer nur als genau solche zu erkennen. Damit tritt zumindest keine fehlerhafte Erkennung mehr auf, stattdessen muss dann der Parser damit umgehen und auf ein solches Vorkommen spezifisch prüfen (Parr 2012, S. 211 ff.).

Je nach Implementierung des Lexers und je nach Anforderung der nächsten Verarbeitungsstufe kann sich die genaue Ausgabeform des Lexers unterscheiden. Eine Möglichkeit ist es, dass der Lexer den gesamten Code verarbeitet und einfach nur eine geordnete Liste von Tokens ausgibt. Häufiger wird aber eine Verzahnung von Lexer und Parser vorgenommen, bei der der Parser den nächsten Token aktiv abfragt.

3.3 Syntaktische Analyse

Die syntaktische Analyse oder auch Syntaxanalyse generiert für das Programm, genauer für den Tokenstream, der in der Regel mittels lexikalischer Analyse aus dem Quellprogramm erzeugt wird, eine syntaktische Struktur. Die syntaktische Struktur oder auch grammatische Struktur repräsentiert den Zusammenhang, beziehungsweise die zwischen den einzelnen Elementen der Sprache bestehenden Zusammenhänge innerhalb des Programms. Auch wenn diese syntaktische Struktur nicht die Semantik des Programms in direkter Art und Weise widerspiegelt, so besteht doch – zumindest bei allen Programmiersprachen – ein Zusammenhang zwischen Syntax und Semantik. Dabei wird die Analyse der syntaktischen Struktur häufig als Aufbauen eines Syntaxbaumes beschrieben, wobei es dafür keine Rolle spielt, ob die Struktur tatsächlich erzeugt wird. Für einfache Fälle kann es auch ausreichend sein, direkt bei der Analyse und bei Erkennen bestimmter Muster bestimmte Aktionen direkt auszuführen, so dass es nicht zwangsläufig folgende Verarbeitungsschritte geben muss (Aho et al. 2008, S. 233 f.).

Das Aufbauen eines Syntaxbaumes ist in dieser Form gleichzusetzen mit dem Überprüfen der syntaktischen Korrektheit. Man spricht von syntaktischer Korrektheit beziehungsweise einem syntaktisch korrekten Programm, wenn für das Programm ein valider Syntaxbaum aufgebaut werden kann. Im Allgemeinen gibt es zwei verschiedene Ansätze zum Aufbau eines Syntaxbaumes: Top-Down-Parsing oder Bottom-Up-Parsing. Die Ansätze unterscheiden sich ihren Namen folgend darin, dass beim Top-Down-Parsing der Baum von oben her, also von der Wurzel aus, aufgebaut wird und beim Botton-Up-Parsing von den Blättern aus. In Bezug auf das zu verarbeitende Programm verarbeiten beide Methoden die Eingabe von links nach rechts. Für die Praxis am relevantesten sind dabei die sogenannten LL- und LR-Parser, beziehungsweise darauf beruhende Typen. LL-Parser sind Top-Down-Parser, die mittels rekursiven Abstiegs oder mittels tabellengesteuerter Mechanismen arbeiten. Damit diese Parser deterministisch arbeiten können, verwenden sie oftmals einen sogenannten „Lookahead“ (dt. „vorausschauen“), um nicht nur den aktuellen Token zu betrachten, sondern bereits mehrere Tokens vorauszuschauen. Das hilft bei der Entscheidung der Produktionen und ermöglicht es, tabellengesteuertes Backtracking im Falle einer falschen Wahl zu vermeiden. Einen LL-Parser der k-Tokens vorausschaut, nennt man entsprechend $LL(k)$-Parser. Die damit analysierbaren Sprachen sind eine Teilmenge der kontextfreien Sprachen, deren Grammatiken auch als LL-Grammatiken bezeichnet werden. Das Äquivalent der Bottom-Up-Parser sind die LR-Parser. Sie werden oftmals als mächtigere Parser angesehen, da die zugehörigen LR-Grammatiken eine noch größere Menge an Sprachen abbilden können. Dafür sind LR-Parser deutlich schwieriger von Hand zu erstellen und werden häufig mittels automatisierter Verfahren generiert. Als Unterkategorien von LR-Parsern gibt es unter anderem SLR, $LALR$ und CLR, die zwar alle nach dem Bottom-Up-Prinzip parsen, sich in Verwendung von Lookahead, Parser-Tabellen und Mächtigkeit ihrer abbildbaren Grammatiken unterscheiden. Allgemein werden Parser, die vorrausschauend arbeiten, auch prädikative Parser genannt. Beide Parsingstrategien finden in der Praxis Anwendung, da je nach Anwendung verschiedene Vor- und Nachteile der erzeugten Parser eine Rolle spielen (Aho et al. 2008, 76-82,263-266,291-292; Louden 1994, S. 103 ff.; Wilhelm et al. 2012, S. 86 ff.).

Wie auch im Kontext von Grammatiken bereits definiert, bestimmt syntaktische Korrektheit aber nur die strukturelle Akzeptanz in Bezug auf die zugrundeliegende Sprache, nicht die Richtigkeit im Kontext einer bestimmten Funktion oder einem genauen funktionalen Ziel. Auch eine korrekte und fehlerfreie Ausführbarkeit ist durch syntaktische Richtigkeit noch nicht gegeben, da zum Beispiel eine

Typenprüfung für Variablen und zugehörige Zuweisungen nicht Teil dieser ist. Grammatikalische Fehler wiederum, wie zum Beispiel das Vergessen eines Semikolons (wenn in der Sprache erforderlich) oder das inkorrekte Klammern von Ausdrücken, sind Probleme, welche bei der syntaktischen Analyse für Fehler sorgen (da kein valider Syntaxbaum existiert) und somit angezeigt werden können. Je nach Ausführung und Qualität eines Parsers sind diese in der Lage, Fehler genau zu lokalisieren und zum Teil auch zu diagnostizieren, was bei der Entwicklung von Programmen hilfreich ist.

3.4 Parsergeneratoren

Durch die Möglichkeit, die Syntax von Programmiersprachen mittels formaler Notation als kontextfreie Grammatik darzustellen, wurde es auch gangbar, diese automatisiert zu verarbeiten. Durch die Erkenntnisse aus der Theorie der formalen Sprache zusammen mit den Anwendungen eben dieser in der Informatik ist es möglich geworden, aus formalen Syntaxbeschreibungen in Form von Grammatiken automatisiert Parser zu generieren. Ein Programm, das diese Aufgabe erbringt, nennt sich Parsergenerator oder historisch manchmal Compiler-Compiler. Parsergeneratoren sind hilfreich bei der schnellen Entwicklung von Parsern, da nachdem einmal eine Grammatik erstellt wurde, diese nur angepasst und ein neuer Parser generiert werden muss. Die genaue Implementierung und auch die Qualität und Geschwindigkeit des generierten Parsers hängt fundamental von der Implementierung des Parser-Generators ab. Ebenso unterscheidet sich die Art und Notation der akzeptierten Grammatik. Auch der genaue Umfang dessen, was generiert werden kann, ist nicht pauschal definierbar. Einige Parsergeneratoren genieren selbst einen zugehörigen Lexer, andere benötigen bestimmte externe Lexer oder haben Schnittstellen zum Einsatz beliebiger Lexer. Aus diesen unterschiedlichen Herangehensweisen folgt auch, dass die genauen Notationen, die die Parsergeneratoren akzeptieren, sich unterscheiden. Verbreitet sind verschiedenste Formen von BNF, EBNF, ABNF oder proprietären Notationen. Ein weiterer bedeutender Unterschied liegt in der Ausgabeform des generierten Parsers. Parsergeneratoren unterstützen in der Regel nur bestimmte Programmiersprachen für den zu erzeugenden Parser (Aho et al. 2008, S. 343; Wagenknecht u. Hielscher 2022, S. 208).

Vertreter von Parsergeneratoren sind zum Beispiel GNU Bison, welcher ein Open-Source Parsergenerator ist, der verschiedene Arten von LR-Parsern erzeugen kann ($LALR(1)$, GLS, $IELR(1)$, $CLR(1)$[19]) (*Bison - GNU Project - Free Software Foundation* 2023). GNU Bison wurde als Nachfolger für einen der klassischsten Parsergeneratoren erdacht, nämlich für yacc („yet another compiler compiler“). Beide Tools benötigen einen separaten Lexer, wofür häufig Lex (älter, eher für yacc) oder Flex (moderner, eher zusammen mit GNU Bison) zum Einsatz kommen. Sowohl die mittels yacc als auch die mittels des neueren GNU Bison erzeugten Parser werden in der Programmiersprache C erzeugt. Ein Parsergenerator zur Generierung von Parsern in Python ist zum Beispiel Lark, welcher neben $LALR(1)$-Parsern auch $Earley$[20]- und CYK[21]-Parser erstellen kann (Erez Shinan 2023). Mit $Earley$- und CYK-Parsern können theoretisch alle kontextfreien Grammatiken (und den daraus resultierenden Sprachen) analysiert werden, allerdings sind diese Parser für die Praxis häufig nicht effizient genug (Aho et al. 2008, S. 358 f.; Wagenknecht u. Hielscher 2022, S. 154 f.). Ein weiterer Vertreter von modernen Parsergeneratoren ist ANTLR (ANother Tool for Language Recognition), welcher der aktuellen Version

[19] IELR(1)- und kanonische LR(1)-Parser sind experimentell *Bison - GNU Project - Free Software Foundation* (2023).
[20] Nach Jay Earley benannter Algorithmus (1970)
[21] Cock-Younger-Kasami-Algorithmus (1960er Jahre)

ANTLR4 statt LR-Parsern sogenannte $ALL(*)$-Parser erzeugt. $ALL(*)$ steht dabei für Adaptive LL, was für LL-Parser mit adaptivem Lookahead steht und im Falle von ANTLR4 sogar ohne Backtracking auskommt (Parr et al., S. 1 ff.). ANTLR kann Parser in einer Vielzahl von Programmiersprachen erzeugen, darunter: Java, C#, Python3, JavaScript, Go, C++, Swift, Dart, PHP.

4 Konzeption und Umsetzung des Lösers

4.1 Definition Speicherbelegungsprotokolle

Ein Speicherbelegungsprotokoll ist eine Form von Datenstruktur zur Verfolgung und Überprüfung von Variablendefinitionen, -belegungen und -sichtbarkeit und findet vor allem Verwendung in Programmierübungen. Speicherbelegung ist dabei in einem programmatischen Kontext zu verstehen und nicht im Kontext von physischem Speicher oder Ähnlichem. Die genaue Form von Speicherbelegungsprotokollen ist nicht formal standardisiert, aber in der Regel einfach genug, um diese nach kurzem Betrachten ohne Schwierigkeiten zu verstehen. Hierbei können je nach Aufgabenstellung verschiedene Schwierigkeitsstufen absolviert werden, indem zum Beispiel einzelne Zellen bereits vorgegeben werden, die Sichtbarkeit von Variablen bereits markiert oder Datentypen der Variablen gefordert werden, oder auch nicht.

Ein spezifisches Belegungsprotokoll gehört dabei auch immer zu einem spezifischen Quellcode. Speicherbelegungsprotokolle können grundsätzlich für jede Programmiersprache, welche die Möglichkeit zur Definition von Variablen hat, angelegt werden. Hierbei gibt der Code die Struktur und Größe des Protokolls vor, indem er definiert, welche Variablen wo sichtbar sind, und Stellen im Code definiert, an denen im Protokoll die Belegung aller sichtbaren Variablen angegeben werden soll. Diese Markierungen werden meistens mittels Kommentare deutlich gemacht. Die Definition lässt sich mit einem einfachen Beispiel verdeutlichen. Für das in Listing 14 angegebene C-Programm soll an den gekennzeichneten Kontrollstellen die Belegung der dort sichtbaren Variablen angegeben werden. Die Kontrollstellen sind dabei in der Form von Kommentaren mit der Form $/* \ Label \ x \ */$ bezeichnet, wobei x eine Zahl darstellt.

```c
int main(void) {
    int i = 1,
        o = 5;
    /* Label 1 */
    {
        char i = 'x';
        int x = ++o - 1;
        /* Label 2 */
    }
    o += i++;
    /* Label 3 */
    return 0;
}
```

Listing 14 C-Code für beispielhafte Speicherbelegungsprotokollaufgabe

Zusätzlich zu dem Code, für den das Speicherbelegungsprotokoll angegeben werden soll, kann eine tabellarische Vorlage wie in Abbildung 4 gegeben werden, aus der die Menge der zu bestimmenden Variablen bereits ersichtlich ist. Auch die Variablensichtbarkeit wurde bereits vorgegeben, in der Form, dass Variable x nur bei Label 2 anzugeben ist. Dies könnte für einen höheren Schwierigkeitsgrad entfallen.

Label	*i*	*o*	*x*
1			—
2			
3			—

Abbildung 4 Leeres Speicherbelegungsprotokoll als Teil der Aufgabenstellung

Zwar ist direkt aus der Tabelle ersichtlich, dass Variable x nur an Label 2 angegeben werden muss, aber es ergibt sich nur aus der Betrachtung des zugehörigen Codes, dass die Variable i nicht durchgängig dieselbe Variable ist. Das korrekt und vollständig ausgefüllte Speicherbelegungsprotokoll würde dann aussehen, wie in Abbildung 5 abgebildet ist.

Label	*i*	*o*	*x*
1	1	5	—
2	x	6	5
3	2	7	—

Abbildung 5 korrektes Speicherbelegungsprotokoll

Ein teilkorrektes Speicherbelegungsprotokoll ist ein Speicherbelegungsprotokoll, welches in mindestens einem Eintrag nicht mit der korrekten Belegung übereinstimmt. Hierbei kann es je nach Aufgabenstellung und Schwierigkeitsgrad primär zwei verschiedene Fehler geben. Zum einen kann ein angegebener Variablen-Wert nicht stimmen oder, wenn erforderlich, kann ein Variablen-Typ falsch angegeben sein. Zum Beispiel könnte die Variable o bei Label 2 mit dem Wert 5 angegeben sein, wenn der Ausfüllende den Post-Inkrement-Operator in Zeile 7 von Listing 14 übersehen oder falsch interpretiert hat. Die Auswertung solcher Fehler kann selbst für kurze Programmcodes schnell unübersichtlich werden, gerade dann, wenn Variablenveränderungen eng miteinander verbunden sind und Fehler frühzeitig passieren. Somit ist eine effiziente Bewertung von Folgefehlern schwierig und mitunter zu zeitaufwendig, um sie tatsächlich händisch durchzuführen. Auch die Verwendung von Zeigern – für Einsteiger oftmals eines der schwierigsten Konzepte – führt schnell zu Folgefehlern, welche händisch nachzuvollziehen sehr kompliziert sein kann. Da Zeiger keine vorhersagbaren Werte enthalten, sondern Speicheradressen, die für den Protokollanten nicht bekannt sind, können diese Werte nicht direkt abgefragt werden. Stattdessen wird nach dem Inhalt ihrer Zieladresse gefragt, beziehungsweise bei Zeigern auf Zeiger, nach dem Inhalt des letzten Zeigers. Dazu kommt, dass Zeiger in Speicherbelegungsprotokollen auch die Bewertung verkomplizieren, da es sein kann, dass korrekt erkannt wurde, auf welche Variable ein Zeiger zeigt, diese Variable aber falsch berechnet wurde. Bei einem einfachen Abgleich mit der Lösung würden hier unter Umständen zwei Fehler erkannt werden, obwohl das Ziel des Zeigers wohl möglich korrekt erkannt wurde. Das lässt einen zusätzlichen Spielraum für Fehler zu. Auch bei Zeigern auf Zeiger (oder beliebiger Vertiefung) wird nicht nach der Adresse gefragt, da die für den Protokollierenden nicht vorhersagbar ist.

Als Datenstruktur für Speicherbelegungsprotokolle kommen je nach Implementierung verschiedene Konstrukte in Frage. Eine einfache und dynamische Möglichkeit ist es, Speicherbelegungsprotokolle als

Liste von Labeln zu betrachten. Ein Label wiederum verfügt neben seiner Label-Nummer wiederum über eine Liste von Variablen und die Variablen können dann entsprechend den Namen, Typen und Wert der Variable enthalten. Das macht ein Speicherbelegungsprotokoll oberflächlich zu einer Liste von Listen von Elementen, wobei die einzelnen Listen mehr sind als die Summe ihrer Listeneinträge. Eine so flache Datenstruktur lässt sich sehr einfach verarbeiten und stellt in der Größe, wie es für einfache Speicherbelegungsprotokolle zu erwarten ist, auch eine sehr effiziente Möglichkeit zur Implementierung dar.

4.2 Herangehensweise zur Implementierung des Lösers für Speicherbelegungsprotokolle

Eine manuelle Überprüfung eines vorliegenden Speicherbelegungsprotokolls ist keine komplexe Aufgabe. Betrachtet man für den ersten Ansatz die korrekte Lösung als gegebenes Element und lässt sie in der gleichen Form wie das zu überprüfende Protokoll vorliegen, beschränkt sich die Kontrolle auf das Vergleichen eines jeden einzelnen Eintrags. Diese Überprüfung auf vollständige Korrektheit eines gegebenen Protokolls zu automatisieren, ist somit ebenfalls kein komplizierter Vorgang. Bleibt die Annahme bestehen, dass die korrekte Lösung und das zu überprüfende Protokoll in der gleichen Form vorliegen, so ist die Überprüfung einfach der eins-zu-eins-Vergleich eines jeden Eintrags zwischen Lösung und zu überprüfenden Protokoll. Auch unter bestehender Annahme, die korrekte Lösung sei ein gegebenes Element, ist die Überprüfung von teilkorrekten Protokollen nur dann trivial, wenn die Bewertung des Protokolls sich darauf beschränkt, richtige und falsche Angaben gegenüber der korrekten Lösung zu bewerten. Diese Bewertungsart vernachlässigt aber den Grundgedanken von Speicherbelegungsprotokollen, nämlich die Frage, ob der zum anzufertigenden Protokoll gegebene Code verstanden wurde oder nicht. Um bei der Bewertung solcher Protokolle tatsächlich ein Verständnis des gegebenen Codes zu berücksichtigen, müssen nicht nur die absoluten Werte überprüft werden, sondern auch, wie sich diese möglicherweise nach einem Fehler korrekt weiterverändert haben. Gerade in Code, in dem die Werte von Variablen stark von anderen Variablen abhängen, zum Beispiel im Falle von Rechnungen mit mehreren Variablen, ist ein früher Fehler ansonsten so schwerwiegend, dass ein Großteil des folgenden Protokolls zwangsläufig als falsch gewertet werden würde. Um eine Erkennung solcher Folgefehler zu ermöglichen, muss der Algorithmus sich von der absolut korrekten Lösung trennen und in der Lage sein, erkannte Fehler im weiteren Abgleich zu berücksichtigen.

Um eine Erkennung und Weiterverfolgung von teilkorrekten Speicherbelegungsprotokollen zu ermöglichen, ist es also erforderlich, erkannte Fehler in die weitere Korrektur einzubeziehen. Für diese Herangehensweise ist es allerdings nötig, nicht mehr nur mit der korrekten Lösung abzugleichen, sondern eine Lösung auf der Grundlage der erkannten Fehler herleiten zu können. Dies führt direkt zu der Notwendigkeit, den gegebenen Code zu verarbeiten und die erkannten Fehler als gegebene Prämissen innerhalb des Programmablaufs zu betrachten. Mittels dieser Möglichkeit zur Verarbeitung des Codes ist es möglich, eine neue Lösung zu erzeugen, welche den erkannten Fehler berücksichtigt und alle darauf beruhenden und nach dem Fehler folgenden Operationen entsprechend angepasst auszuführen. Durch diese Art des Eingriffs kann sich allerdings in direkter Folge auch der Programmablauf des gegebenen Codes verändern. Zum Beispiel ist es möglich, dass im Folgenden eine unvorhergesehene Integer-Division durch Null auftritt und somit ab dieser Stelle keine valide Lösung

mehr berechnet werden kann. Für viel wahrscheinlicher auftretende weniger schwerwiegende Fehler ist es damit aber möglich, Folgefehler korrekt zu bewerten. Die beschriebene separate und vollständige Berechnung einer kompletten Lösung ist nur zur besseren Verständlichkeit gewählt. Es besteht nicht die Notwendigkeit, die Lösung erst vollständig zu berechnen, um sie dann in weiteren Schritten abgleichen zu können. Da das Errechnen der Lösung auf der Interpretation des zugehörigen Quellcodes beruht, können das Erzeugen der Folgefehlerlösung und das Vergleichen mit dem gegebenen Protokoll miteinander verzahnt werden. Das vermeidet das unnötige Erzeugen von komplett neuen Lösungen, die möglicherweise bereits beim nächsten Kontrollpunkt wieder verworfen werden müssten. Der Grundgedanke des Algorithmus bleibt dabei aber bestehen, wodurch eine Lösung nicht ausschließlich anhand des Programmcodes berechnet wird, sondern zusätzlich anhand der erkannten Fehler im gegebenen Protokoll.

Allerdings gibt es auch mit diesem Ansatz Folgefehler, welche nicht erkannt werden können, nämlich Folgefehler zwischen den Kontrollpunkten. Solche versteckte Folgefehler können zum Beispiel durch richtiges Anwenden einer Operation mit einer bereits falsch erkannten Variable entstehen. Dieses Problem kann beim Erstellen des Aufgaben-Programmcodes vermieden werden, indem nicht zu viele voneinander abhängige Operationen zwischen zwei Kontrollpunkten verwendet werden. In Listing 15 ist ein Code abgebildet, welcher dies in einem sehr kleinen und einfachen Maßstab zeigt. Wird eine der Variablen a oder b falsch erkannt, weil möglichweise eine der Variablen x oder y mit dem falschen Wert angenommen wurde, verursacht dies direkt auch eine falsche Angabe für die Variable c. Diese Art von Folgefehlern innerhalb eines Abschnittes ist nicht zu erkennen oder zu korrigieren, da dafür immer entscheidbar sein müsste, wo beim Erstellen des Belegungsprotokolls der Fehler gemacht wurde. Zwar wäre für ein so kleines Beispiel, wie das in Listing 15 gezeigte, eine solche Verfolgung noch durchführbar, doch schon bei wenig komplexeren Programmen ist eine weitere Nachverfolgung solcher Fehler zwischen den Kontrollpunkten nicht mehr realisierbar. Zur Verdeutlichung könnten zwischen der Zuweisung zu Variable c und dem Kontrollpunkt $Label\ 2$ noch weitere Zuweisungen für die Variablen a und b liegen, welche somit den Fehlerursprung für c völlig verschleiern würden. Dazu kommt, dass eine Korrektur dieser Fehler nicht möglich wäre, da ja zwischen den Kontrollpunkten ja keine Zustände bekannt sind, mit denen weiter gerechnet werden könnte.

```
int a = x + 1;
int b = y + 1;
int c = a + b;
/* Label 2 */
```

Listing 15 Anfälliger Code für versteckte Folgefehler

Eine weitere Schwierigkeit bei der Verarbeitung von Folgefehlern stellen Zeiger dar. Als Zeiger werden in C Variablen bezeichnet, welche als Wert die Speicheradresse von anderen Variablen enthalten. Bei Zeigern wird in Speicherbelegungsprotokollen in der Regel nicht nach deren direktem Inhalt gefragt, da dies dementsprechend eine Adresse im Speicher wäre, welche nicht voraussagbar ist. Stattdessen wird nach dem Wert der Variable gefragt, auf die verwiesen wird. Dies stellt noch kein Problem dar, da man dafür auch beim Abgleich mit dem Speicherbelegungsprotokoll den Verweis auflöst und somit den Wert, auf den gezeigt wird, abgleichen kann. Zum Problem werden diese Werte erst dann, wenn sie

falsch sind, und der Algorithmus versuchen würde, den falschen Wert zum Weiterrechnen anzunehmen, da dies ja nicht den Wert des Zeigers verändern würde, sondern den der Variable, auf die gezeigt wird. War diese aber möglicherweise richtig angegeben, würde es nun zur Berechnung einer falschen Lösung kommen, welche zwar durch den Zeiger so impliziert wurde, aber wohlmöglich nicht durch die Variable selbst. Wurde die Variable wiederum auch falsch angegeben, so wird diese auch korrigiert und somit auch das Ziel des Zeigers. Außerdem ist anhand des im Protokoll vermerkten Wertes nicht zwangsläufig die Zuordnung zu einer Variable herstellbar. Da die Veränderung eines Zeigerziels immer zu unerwünschten Ergebnissen führen kann, ist es die sinnvollste Lösung, die Zeiger nicht zu korrigieren. Ein Verändern des Wertes, auf den ein Zeiger verweist, könnte zu fehlerhafter Variablenbelegung führen, sogar dann, wenn die Variable selbst im Protokoll richtig angegeben war. Und die Zieladresse des Zeigers zu verändern, ist mitunter nicht möglich, da auch ein falscher Wert angegeben worden sein kann, der keiner anderen Variable zuzuordnen ist. Ein weiterer Nebeneffekt im Zusammenhang mit Zeigern ist, dass dadurch die Reihenfolge, in der die Variablen an einer Kontrollstelle überprüft werden, relevant werden kann. Wird ein Zeigerwert vor seiner zugehörigen Variable kontrolliert, so werden potenziell zwei Fehler erkannt, obwohl das Zeigerziel als solches korrekt erkannt wurde. Wird jedoch die Variable als erstes überprüft und im Fehlerfalle bereits korrigiert, dann wird der Zeiger als korrekt gegenüber den korrigierten Ausführungsdaten erkannt. Somit ist es eine sinnvolle Maßnahme die Reihenfolge bei der Überprüfung zu verändern, so dass einfache Variablen immer vor Zeigern überprüft werden. Damit ist eine korrekte Bewertung von Zeigerverweisen gewährleistet. Diese Herangehensweise ist also in der Lage, sowohl einfache Folgefehler aus früheren Falschangaben zu erkennen als auch korrekt erkannte Zeigerbeziehungen bei einer solchen falschen Angabe korrekt zu bewerten.

Um diese Herangehensweise zu implementieren, ist es also nötig, den Programmcode ausführen zu können und während der Ausführung in seinen Ablauf einzugreifen, um die entsprechend erkannten Fehler miteinbeziehen zu können. Das ist mit einer normalen Kompilierung von C kaum möglich, da der kompilierte Code bereits fertig übersetzt ist und die Ausführung aller weiteren Schritte somit fest definiert ist. Eine Möglichkeit wäre es, mit Debuggern wie dem GNU-Debugger in den Programmfluss Breakpoints zu setzen und dort dann in den Programmablauf eingreifen zu können. Zwar ist es damit auch möglich, in gewissem Maße Variablen auszulesen und Korrekturen durchzuführen, aber gerade im Bereich von Variablen-Typen bestehen dort Grenzen, da Variablen-Typen in C normalerweise unveränderlich sind. Dazu kommt, dass es sehr komplex sein kann, den Output eines solchen Debuggers automatisiert zu verarbeiten und entsprechend darauf zu reagieren, gerade wenn es darum geht, nicht nur eine einzelne Variable zu verfolgen. Da es bei Speicherbelegungsprotokollen aber nur um einen geringen Ausschnitt der Sprache geht, welcher zum Erlernen grundsätzlicher Abläufe und Konzepte der Programmierung ausgelegt ist, wird im Folgenden ein eigener Interpreter für diesen Ausschnitt der Sprache C implementiert. Dieser Ansatz, den Code selbst zu interpretieren, sorgt dafür, dass die Berechnung von Lösungen und auch das Abgleichen eines gegebenen Protokolls direkt bei der Interpretation des Codes durchgeführt werden kann. Für die Konzeption eines solchen Interpreters ist es als Erstes nötig, den Sprachausschnitt zu definieren, welcher abgebildet werden soll. Aus der Abgrenzung der abgebildeten Sprache gegenüber C kann dann ein Parser entwickelt werden, welcher für Programme innerhalb der Sprachdefinition das Parsen durchführt. Dieser Parser ermöglicht die Implementierung eines Interpreters für den definierten Sprachausschnitt. Dieser Interpreter kann dann mit entsprechendem Input teilkorrekte Lösungen berechnen und die Ergebnisse zum Abgleich

zurückgeben. Um den entsprechenden Input zu generieren und die Ergebnisse des Interpreters zu verwerten benötigt es eine weitere Anwendung, diese stellt dann den eigentlichen Vergleich an und kann darauf entsprechend reagieren. Die entstehende Anwendung muss in der Lage sein, Speicherbelegungsprotokolle einzulesen, Programmcode – mittels des Interpreters – auszuführen und dabei das gegebene Protokoll mit den Ergebnissen des Interpreters abzugleichen. Der betrachtete Ausschnitt aus der Programmiersprache C wird dabei im Folgenden „LimitC“ genannt. LimitC-Programme sind immer nur dann valide, wenn sie auch valide C-Programme sind und den Funktionsumfang von LimitC beachten.

4.3 Abgrenzung und Umfang von LimitC

LimitC soll ein beschränkter Ausschnitt der Programmiersprache C sein und muss somit auf den gleichen Regeln und Vorgaben wie C basieren. Das aktuelle standardisierte Regelwerk für C ist C17/C18, wobei beide Bezeichner den gleichen Standard ISO/IEC 9899:2018 bezeichnen. Im Folgenden wird die Bezeichnung C18 verwendet. Die doppelte Benamung kommt daher, dass der Standard 2017 fertig entwickelt wurde, aber erst 2018 als Standard veröffentlicht wurde. (C17 - cppreference.com 2023) Im Gegensatz zu dem vorherigen Standard C11 bzw. ISO/IEC 9899:2011 hat C18 keine neuen Sprachfunktionen gebracht, sondern nur bekannte Fehler von C11 eliminiert. Manchmal wird C18 daher auch als „Bug-Fix Version“ zu C11 bezeichnet.

Da LimitC ausschließlich für die Verarbeitung im Zusammenhang mit Speicherbelegungsprotokollen gedacht ist, spielt der genaue Standard nur eine untergeordnete Rolle, da die meisten der sprachlichen Änderungen seit C99 sich auf Standardbibliotheken beziehen, welche in LimitC nicht abgebildet werden. LimitC deckt nur einen begrenzten Teil des Sprachkerns von C ab, ohne die riesige Auswahl an Standard-Bibliotheken zu verwenden. Da tatsächlich nur ein sehr geringer Umfang der eigentlichen Programmiersprache C abgebildet wird, soll dieser im Folgenden genauer abgegrenzt werden. Weiterhin werden Abgrenzungen getätigt, welche zwar C-valide wären, aber eine Implementierung unnötig verkomplizieren und im Kontext von Speicherbelegungsprotokollen keinen Mehrwert bringen würden. Dazu gehören unter anderem auch Funktionen wie Klassen, Namespaces, Imports von Bibliotheken. LimitC wird im Wesentlichen auf die im Folgenden beschriebenen Funktionen beschränkt. Funktionen, die nicht ausdrücklich beschrieben sind, können unterstützt sein, müssen es aber nicht. Unabhängig davon sollte das Verhalten, welches in C als „Implementation-defined behavior“, „Undefined behavior“ oder „Unspecified behavior“ beschrieben wird, nicht verwendet werden, da die Ergebnisse maximal unter Kenntnissen der genauen Implementierung zu kennen sind und somit nicht verlässlich angegeben werden können.

Programmstruktur und Direktiven

In LimitC werden nur einfache Programme für Speicherbelegungsprotokolle abgebildet, daher gibt es keine Unterstützung von Imports in der Form von $\#include$-Anweisungen oder anderen Präprozessor-Direktiven. Alle für die Ausführung erforderlichen Definitionen befinden sich innerhalb einer einzigen Datei. Neben Includes betrifft das auch alle sonstigen Präprozessor-Direktiven. Um eine bessere Testbarkeit gegenüber C herzustellen, werden Direktiven zugelassen, aber ignoriert.

Programmeinstieg

In C ist der Standardeinstiegspunkt eine Funktion mit dem Namen $main$ und dem Rückgabetyp int. Diese Funktion darf entweder keine Parameter annehmen oder genau zwei (*ISO/IEC 9899:2017 C17 ballot,* S. 10 f.; Wolf u. Krooß 2020, S. 179 f.). Diesen Einstiegspunkt soll auch LimitC annehmen, mit der Einschränkung, dass nur $main$-Funktionen ohne Parameter akzeptiert werden bzw. die Parameter nicht verarbeitet werden.

Zulässige Datentypen

In Bezug auf Datentypen unterstützt LimitC die wichtigsten Basis-Datentypen, welche für einfache Speicherbelegungsprotokolle relevant sind. Für einfache Speicherbelegungsprotokolle spielen die genauen Grenzbereiche der Datentypen in der Regel keine Rolle, weshalb keine $signed$- oder $unsigned$-Datentypen unterstützt, sondern alle Datentypen grundsätzlich als $signed$ betrachtet werden. Die verfügbaren Datentypen in LimitC sind: $char, int, short, long, double, float$. Zusammensetzungen aus diesen Datentypen wie $long\ long\ int$ oder $long\ double$ werden nicht unterstützt. Obwohl seit C99 mit $\_Bool$ auch ein Datentyp für Wahrheitswert spezifiziert ist, wird dieser nicht in LimitC unterstützt, da er kein Arbeiten mit „echten“ Wahrheitswerten ermöglicht, sondern nur ein Integer-Typ mit dem Wertebereich 0 oder 1 darstellt. Als Funktionsrückgabetype ist zusätzlich $void$ möglich. Arrays werden aufgrund ihrer Komplexität, auch in Bezug auf die Implementierung, nicht unterstützt, da sie für einfache Speicherbelegungsprotokolle nicht notwendig sind. Da Strings in C nur als Char-Arrays definiert werden können, werden auch diese nicht unterstützt. Die unterstützten Datentypen sollen, soweit möglich, funktional identisch zu ihrem Äquivalent aus C sein, vor allem in Bezug auf Konvertierbarkeit. Explizite und implizite Typumwandlungen sollen also wie in C funktionieren. Das Verhalten in Bezug auf Speichergrößen oder Ähnliches wird dabei vernachlässigt, da dieses auch in C Compiler- bzw. Systemabhängig ist (*ISO/IEC 9899:2017 C17 ballot,* S. 31 ff.; Wolf u. Krooß 2020, S. 59 ff.).

Variablendefinition und Zuweisungen

Einfache Variablendefinitionen werden weitgehend wie in C unterstützt, abgesehen von vorgelagerten Schlüsselwörtern wie zum Beispiel $const, extern, unsigned\ oder\ static$. Variablen können wie in C sowohl nur deklariert als auch deklariert und direkt initialisiert werden. Auch das Definieren mehrerer Variablen eines Typs mittels kommagetrennter Deklaration bzw. Initialisierung ist möglich. LimitC unterstützt sowohl einfache Zuweisungen mittels $=$, als auch einfache Operator-Zuweisungen („compound statement“ (*ISO/IEC 9899:2017 C17 ballot,* S. 74)) für die unterstützten arithmetischen Operatoren.

Operatoren

Es werden nur die grundlegenden arithmetischen Operatoren unterstützt: $+, -, *, /$. Für die genannten Operatoren werden auch die Zuweisungsformen $+=, -=, *= und\ /=$ unterstützt. Auch die Inkrement- und Dekrement-Operatoren $++$ und $--$ werden als Post- und Präfix-Operatoren unterstützt. Bitoperatoren wie &, |, ^, ~, $\gg$, $\ll$ werden nicht unterstützt, ebenso keine Vergleichsoperatoren wie $==$, $<=$ oder $>$.

Funktionsdefinitionen und Aufrufe

LimitC unterstützt einfache Funktionsdefinitionen und -aufrufe und soll dabei Parameter als auch Rückgabetypen korrekt verarbeiten. Auch bei der Definition von Funktionen sind keine zusätzlichen Schlüsselwörter wie zum Beispiel $static$ zulässig. Erweiterte Konstrukte wie unter anderem das Definieren von Funktionen mit variabler Parameteranzahl mittels $f(int\ a, \dots)$ werden nicht unterstützt. Ebenso gibt es (wie auch in Standard-C) keine verschachtelten Funktionsdefinitionen. Funktionen können also nur auf der „Hauptebene“ definiert werden und haben demnach in LimitC immer File-Scope.

Zeiger

Für alle unterstützten Datentypen sollen auch entsprechende Zeiger definierbar sein, welche wie in C auf die Speicheradresse einer entsprechenden Variable zeigen. Auch Zeiger auf Zeiger werden unterstützt. Dafür werden die entsprechenden Operatoren $\&$ und $*$ zur Zuweisung von Speicheradressen und zum Auflösen eines Zeigers unterstützt. Void-Zeiger und das Konvertieren von Zeigern werden nicht unterstützt.

Zugriffsbereiche und Sichtbarkeit von Variablen

Der Scope, also der Bereich, in dem eine Variable sichtbar und zugreifbar ist, soll sich in LimitC verhalten, wie auch in C. In C gibt es vier unterschiedliche Scopes: „function, file, block, and function prototype“ (*ISO/IEC 9899:2017 C17 ballot,* S. 35), soweit anwendbar, funktionieren diese in LimitC identisch. Auch die Sichtbarkeit von Variablen ist identisch zu C, Variablen können mittels geschachtelter Blöcke also auch mehrfach definiert werden. Für den Zugriff auf eine Variable gilt dann immer die Variable, die im “innersten“ Scope definiert wurde.

Kommentare

Kommentare sind auch in LimitC zulässig und werden, wie auch in Standard-C, normalerweise ignoriert. Die Ausnahme stellen hierbei Label-Kommentare zur Definition von Kontrollpunkten in Speicherbelegungsprotokollen dar. Die genaue Form und Funktionsweise von Labeln wird in Kapitel 4.4 genauer erläutert.

4.4 Lexer- und Parser-Grammatik für LimitC in ANTLR

ANTLR steht für „ANother Tool for Language Recognition“ und ist ein Parsergenerator für verschiedenste Sprachanwendungen. ANTLR wurde von Terence Parr, Professor für Informatik an der „University of San Francisco“, entwickelt und ist derzeit in der Version 4[22] aktuell. Mit ANTLR können Parser für verschiedenste Anwendungen erstellt werden, unter anderem zur Auswertung von Suchanfragen, zum Entwickeln eigener Programmiersprachen oder als Analyse-Tools in IDEs oder anderen Tools, die mit formalen Sprachen arbeiten (Parr 2012, S. xi). ANTLR generiert Parser auf der Grundlage von Grammatiken. Diese Grammatiken sind keine EBNF- oder ABNF-Grammatiken, auch wenn das in ANTLR verwendete Format durchaus dem von EBNF durchaus ähnlich ist. ANTLR ist in der Lage Parser und Lexer aus einer einzigen gemischten Grammatik zu erstellen. Die Grammatik-Notation für Lexer unterscheidet sich dabei von der des Parsers, wobei sich die Notationen durchaus ähneln. (Parr 2012, S. 36 f.).

Die Angabe der Lexer-Regeln erfolgt dabei innerhalb einer sogenannten Lexer-Grammatik. Diese Lexer-Grammatik kann mit der Parser-Grammatik in einer Datei sein oder auch separat eingebunden werden. Eine Lexer-Grammatik besteht aus Lexer-Regeln, welche in verschiedene Lexer-Modes aufgeteilt werden können. Mit Lexer-Modes kann zum Beispiel beim Parsen von XML, die äußere Form der XML-Tags, anders geparst werden als die Attribute im Inneren der Tags (Parr 2012, S. 224 ff.). Lexer-Regeln beschreiben Tokens und bestehen in ihrer einfachsten Form aus einem Bezeichner für den Token, gefolgt von einem Doppelpunkt und einem Lexer-Ausdruck, welcher die zu matchenden Lexeme beschreibt. Zur Unterscheidung von Parser-Regeln beginnen Lexer-Bezeichner immer mit einem Großbuchstaben. Lexer-Ausdrücke sind in einer an reguläre Ausdrücke erinnernden Form notiert. Im Gegensatz zu klassischen regulären Ausdrücken können Lexer-Ausdrücke aber sowohl andere Lexer-Ausdrücke enthalten oder sogar rekursiv aufgebaut sein. Linksrekursion ist allerdings nicht möglich. Weiterhin können Lexer-Ausdrücke mit programmiersprachenspezifischen Ausdrücken versehen werden, sogenannten Aktionen[23]. Mit diesen Aktionen können bereits während der lexikalischen Analyse Anweisungen ausgeführt werden, wenn ein bestimmter Token erkannt wurde, wie zum Beispiel das Registrieren von verwendeten Variablennamen oder Ähnlichem. Eine weitere Möglichkeit für Lexer-Regeln ist die Verwendung von semantischen Prädikaten, welche es ermöglichen, Lexer-Regeln nach semantischen Abhängigkeiten zu deaktivieren. Ansonsten unterstützen Lexer-Ausdrücke die Angabe klassischer statischer Literale wie $'='$ oder $'int'$, Zeichenklassen wie $[a-z]$ oder $[1-9]$ (auch nur $a..z\ oder\ 1..9$), Negation, Wiederholung und Optionalität. Auch Darstellungsformen für Zeilenumbrüche oder Tabs stehen zur Verfügung. Es ist auch möglich, Lexer-Regeln nur als Hilfskonstrukt zu beschreiben, ohne dass diese als Token gematcht werden würden. Das geschieht mithilfe des $fragment$ Schlüsselworts. Eine Lexer-Regel kann aus verschiedenen alternativen Lexer-Ausdrücken bestehen, diese werden durch einen senkrechten Strick | voneinander getrennt (Parr 2012, S. 277 ff.).

Die Parser-Regeln sind vom Aufbau den Lexer-Regeln nicht völlig unähnlich, haben jedoch eine andere Syntax. Die Bezeichner für Parser-Regeln beginnen immer mit einem Kleinbuchstaben, gefolgt von einem Doppelpunkt und einem Parser-Ausdruck. Ein Parser-Ausdruck kann ähnlich wie ein Lexer-Ausdruck über Prädikate und Aktionen verfügen, aber es können auch Lexer-Bezeichner und andere

[22] 4.13.1 zum Zeitpunkt der Veröffentlichung, im Folgenden ist immer diese Version gemeint.
[23] Original: „$actions$“

Parser-Regeln (in Form von Bezeichnern für Parser-Regeln) darin vorkommen. Ähnlich wie bei Lexer-Regeln gibt es Operatoren für Optionalität und Wiederholungen, ähnlich denen von regulären Ausdrücken. Wie auch bei Lexer-Regeln können mehrere Alternativen durch einen senkrechten Strich getrennt werden. Hervorzuheben ist, dass ANTLR Parser-Regeln mittels einer erweiterten Syntax sogar mit Argumenten aufgerufen werden können und einen Rückgabetypen besitzen. Mit Attributen, Prädikaten und Aktionen lässt sich also mittels ANTLR sogar schon ohne weiteren Quellcode außerhalb der Grammatik ein begrenzt funktionaler Parser bauen (Parr 2012, S. 260 ff.). Aber auch ohne die Nutzung dieser Grammatik-Aktionen stellt ANTLR mächtige Möglichkeiten zur Verfügung, um mit dem Ergebnis des Parsers weiterzuarbeiten. So kann ANTLR für eine Reihe an Programmiersprachen Base-Klassen für die Verwendung mittels Visitor- und Listener-Pattern erzeugen, um damit den (standardmäßig) erzeugten Parsebaum zu traversieren (Parr 2012, S. 17 ff.). Die von ANTLR generierten Parser sind sogenannte ALL(*)-Parser und können durch ihren adaptiven Lookahead viele Grammatiken verarbeiten, welche mit klassischen LL(k)-Parsern teilweise nicht zu verarbeiten wären. Weiterhin sind die mit ANTLR generierten Parser im Gegensatz zu klassischen LL-Parsern in der Lage, direkte Linksrekursion zu verarbeiten. Beziehungsweise ist ANTLR dazu fähig, diese Regeln beim Generieren des Parsers so umzuschreiben, dass Links-Rekursion entfällt. Diese stellt normalerweise einen der größten Kritikpunkte an LL-Parsern dar, da es das Erstellen von Grammatiken deutlich verkomplizieren kann.

Die folgenden Grammatik-Ausschnitte beziehen sich alle auf ANTLRs Grammatikverarbeitung, gerade die Umsetzung von Operator-Prioritäten und direkte Linksrekursion würden diese Grammatik für die meisten anderen Parsergeneratoren nicht nutzbar machen. Im Folgenden sollen einige Ausschnitte der Grammatik gezeigt werden und ihr Nutzen in Bezug auf exemplarische C-Ausschnitte deutlich machen. Parser- und Lexer-Regeln werden immer anhand eines eindeutigen Bezeichners definiert. Auch wenn bei der Implementierung die von diesen Regeln erkannten Sequenzen nicht wirklich in individuelle Objekte gespeichert werden, wird im Folgenden zum Teil zur vereinfachten Betrachtung diese Ausdrucksweise verwendet. Das Auftreten oder Vorhandensein eines Elements meint dann das Zutreffen von mindestens einem für diese Regel definierten Ausdruck.

Einer der größten Unterschiede zu klassischem C ist, dass in LimitC eine bestimmte Form von Kommentar verarbeitet werden soll, um damit die Kontrollpunkte für die Verarbeitung der Speicherbelegungsprotokolle zu definieren. Dieser Unterschied ist tatsächlich nicht so trivial, wie er möglicherweise im ersten Moment klingt, da in C Kommentare nicht verarbeitet werden und daher normalerweise bereits bei der Vorverarbeitung des Codes entfernt (*ISO/IEC 9899:2017 C17 ballot,* S. 9). Um diese speziellen Kommentare nun anders zu matchen als andere mehrzeilige C-Kommentare, wurden zwei zusätzliche Lexer-Regeln eingeführt.

```
LABEL: COMSTART WS* LABELLIT WS* INTEGER WS* COMEND;
LABELLIT: [Ll] [Aa] [Bb] [Ee] [Ll];
MULTILINECOMMENT: COMSTART .*? COMEND -> skip;
```

Listing 16 Lexer-Regel für Label-Erkennung

Die in Listing 16 dargestellten Lexer-Regeln befinden sich noch vor der Regel für mehrzeilige Kommentare und werden daher noch vorher angewendet. $COMSTART$ und $COMEND$ stehen für die

Anfangs- und Endzeichen „/*“ und „*/“ eines mehrzeiligen Kommentars, WS matcht Leerzeichen und Tabs und $INTEGER$ matcht einer beliebigen nicht leeren Sequenz der Ziffern 0-9. Die Lexer-Regel „LABEL“ beschreibt dabei bereits das Format des Labels, was normalerweise nicht Aufgabe eines Labels ist, sondern in Parser-Regeln erfolgen sollte. Da aber die Erkennung von Kommentaren bereits in der lexikalischen Analyse erfolgt und diese bei Erkennen nicht mehr an den Parser übergeben werden würden, wird die Label-Erkennung ebenfalls in die lexikalische Analyse verschoben. In der Regel $LABELLIT$ ist zu erkennen, dass jede Form von Groß- und Kleinschreibung akzeptiert wird. Die Regel $LABEL$ passt also auf jede Sequenz aus Kommentaranfangszeichen, optional beliebig vielen Leerzeichen, dem Wort „Label“ in beliebiger Groß- und Kleinschreibung, beliebig vielen Leerzeichen, einer Zahl sowie gefolgt von beliebig vielen Leerzeichen und den Kommentarendzeichen. Zeilenumbrüche dürfen in einem Label also beispielsweise nicht vorkommen. In Listing 17 sind Beispiele für valide und invalide Label abgebildet.

```
/* Label 1 */                       -> valid Label
/*LABEL155*/                        -> valid Label
/*          LaBEl           5*/     -> valid Label
/* label
1 */                                -> invalid Label (-> multiline Comment)
/* Lebel 1 */                       -> invalid Label (-> multiline Comment)
```

Listing 17 Beispiele für valide und invalide Label

In Listing 18 ist der Beginn der LimitC-Grammatik zu sehen. Die Parser-Regel $prog$ definiert dabei die Top-Level-Ebene und lässt eine beliebige Anzahl von $gdecl$ Matches, gefolgt von einem EOF, also einem Endmarker der Datei, zu. Die Parser-Regel $gdecl$ wiederum definiert sich als genau eines von drei Elementen: eine Funktionsdefinition $funcDef$, eine Variablendefinition $varDef$ oder ein Label $label$.

```
grammar LimitC;

prog: gdecl* EOF;

gdecl: (funcDef | varDef | label);
```

Listing 18 Grammatikbeginn inkl. Einstiegspunkt

Die Parser-Regel $label$ akzeptiert dabei nur genau ein Element der in Listing 16 Lexer-Regel für Label-Erkennung gezeigten Lexer-Regel $LABEL$. $funcDef$ und $varDef$ dagegen sind deutlich komplexere Parser-Regeln und bilden Funktionsdeklarationen und -definitionen und Variablendeklarationen und -initialisierungen ab. Variablen und Funktionen, welche auf diesem Level definiert werden, würden in C, unter Vernachlässigung möglicher Schlüsselwörter, den File-Scope haben. So definierte Variablen und Funktionen wären also in der gesamten Datei zugänglich. Da LimitC immer nur aus einer Datei besteht, entsprechen diese globalen Variablen und Funktionen. Funktionsdefinitionen sind dabei für LimitC, wie in Listing 19 gezeigt, definiert.

```
funcDef:  type ID '(' paramListDef? ')' (codeBlock | ';');
paramListDef: paramDef (',' paramDef)*;
paramDef: (type ID?);
```

Listing 19 Grammatikausschnitt Funktionsdefinition

Wie Listing 19 zeigt, akzeptieren Funktionsdefinitionen und Funktionsdeklarationen in $funcDef$ keine Schlüsselwörter vor dem Rückgabetyp. Der Rückgabetype $type$ matcht dabei nur mit genau einem der von LimitC unterstützten Typen, möglicherweise gefolgt von einem Sternchen zur Definition von Zeigern. Auf dem Rückgabetypen folgt ein Bezeichner ID, der wie in C für einen Bezeichner aus beliebigen Zeichen besteht, welcher nicht mit einer Zahl beginnt (*ISO/IEC 9899:2017 C17 ballot,* S. 334). Wobei LimitC hier tatsächlich nur Buchstaben und Zahlen akzeptiert und keine sonstigen Unicode-Zeichen. Auf den Bezeichner hin muss eine geöffnete runde Klammer folgen und darauf wiederum ein optionales $paramListDef$, welches wiederum aus einer beliebigen Sequenz aus kommagetrennten $paramDef$s besteht, was ein akzeptierter Datentyp gefolgt von einem Bezeichner sein muss. $paramDef$ kann hierbei entweder nur einen Datentyp darstellen, wie er in Funktionsdeklarationen notwendig ist, oder einen Datentyp gefolgt von einem Bezeichner für eine Variablendeklaration, wie es für Funktionsdefinitionen notwendig ist. Nach $paramListDef$ folgt eine geschlossene runde Klammer und dann entweder ein Semikolon oder ein $codeBlock$. Semantisch würde es sich, wenn ein Semikolon folgt, um eine Funktionsdeklaration oder einen Funktionsprototype handeln. Folgt hingegen ein $codeBlock$, handelt es sich um eine Funktionsdefinition. $codeBlock$ definiert dabei eine Parser-Regel, welche sowohl die Ausführungsblöcke von Funktionsdefinitionen matcht als auch mittels Rekursion mögliche Blöcke innerhalb einer Funktion. $codeBlock$ selbst definiert sich dabei wiederum als beliebige List aus weiteren Vorkommen von $codeBlock$, $label$ oder sogenannten $termExpr$, was jede Form von Ausdrücken matcht, die in LimitC akzeptiert werden.

```
codeBlock: '{' codeStateList* '}';

codeStateList: codeBlock | termExpr | label;

termExpr: varDef                # varDefExpression
    | expr ';'                  # looseExpression
    | RETSTAT expr ';'          # returnExpression
    ;
```

Listing 20 Grammatikausschnitt für Codeblöcke mit enthaltenden Elementen

In der Listing 20 gezeigten Parser-Regel $termExpr$ ist auch eine weitere Funktion von ANTLR zu sehen, nämlich benannte Alternativen für sogenannte „$Precise\ Event\ Methods$“ (Parr 2012, S. 112). Dieses Benennen von Alternativen ermöglicht es, beim Traversieren des Baumes auf die genaue übereinstimmende Regel benachrichtigt zu werden, statt nur über die übergeordnete Parser-Regel. Auch wenn diese Funktion keinen Einfluss auf die definierte Sprache hat, so ist es doch für das Abhandeln des Parsing-Ergebnisses eine enorme Vereinfachung. Diese Benennung von Alternativen wird in der LimitC-Grammatik überall dort verwendet, wo es mehrere Alternativen für eine Regel gibt und bei der weiteren Verarbeitung exakt unterschieden werden muss, welche Regel gewählt wurde. Die vollständige LimitC-Grammatik ist im Anhang A dargestellt.

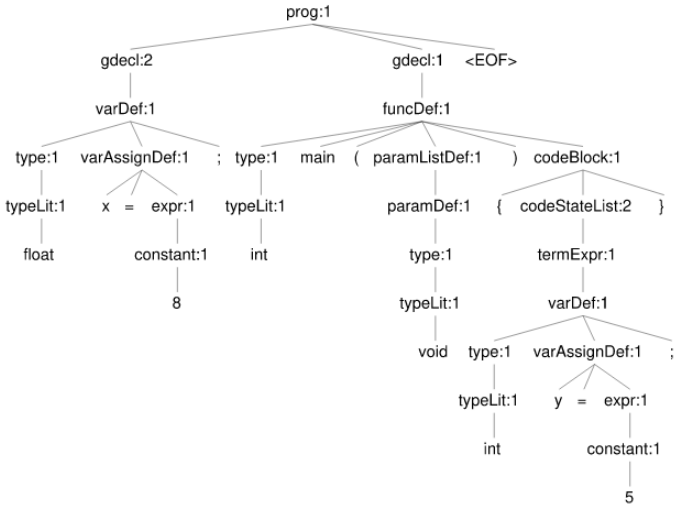
4.5 Implementierung LimitC-Interpreter

Mittels ANTLR lässt sich aus der in Kapitel 4.4 vorgestellten Grammatik nun ein Lexer und ein Parser erzeugen, welche zusammen in der Lage sind, Parsebäume für LimitC Eingaben zu erzeugen. Auf der Grundlage des so erzeugten Parsers wird dann ein Interpreter für LimitC implementiert. Die Erzeugung der entsprechenden Klassen kann für verschiedene Programmiersprachen und mit verschiedenen zusätzlichen Parametern erfolgen. In Listing 21 ist die Konfiguration für den im folgenden genutzten LimitC-Parser zu sehen.

```
java org.antlr.v4.Tool -Dlanguage=CSharp -no-listener -visitor -o ./gen LimitC.g4
```

Listing 21 ANTLR Buildcommand für den LimitC-Parser

Der erste Teil des Befehls stellt den Aufruf zu ANTLR dar[24], danach folgt mittels des Parameters $-Dlanguage$ die Festlegung der Ausgabesprache, hier $CSharp$, für C#. Die weiteren Parameter definieren in dieser Reihenfolge: das Nichterzeugen von Listener-Klassen (Implementierung des Listener-Patterns), das Erzeugen von Visitor-Klassen und die Definition des Ausgabepfads für alle erzeugten Dateien. Als Letztes erfolgt die Angabe der zugehörigen Grammatik, hier benannt als $LimitC.g4$. Mittels des erzeugten Lexers und Parsers können nun für LimitC-Programme Parsebäume erzeugt werden. Diese Parsebäume können zum besseren Verständnis auch grafisch dargestellt werden, wie in Abbildung 6 abgebildet.




```
float x = 8;

int main(void)
{
    int y = 5;
}
```

Abbildung 6 Visualisierung Parse-Baum mit Code

Um diesem Parsebaum und dem zugehörigen Programmcode nun eine semantische Bedeutung zuzuordnen, ist es nötig, den Baum zu traversieren, also die Knoten des Parsebaumes zu besuchen. Beim Besuchen der Knoten werden dann semantische Aktionen ausgeführt, welche in ihrer Gesamtheit dann als Ausführung des Codes angesehen werden. Im Compilerbau wird diese Art der semantischen Analyse als sogenannte „syntaxgerichtete Übersetzung“ bezeichnet (Aho et al. 2008, S. 66 ff.).

[24] Kann sich abhängig von der Installation unterscheiden

Für die Verarbeitung von LimitC wird im Folgenden die ANTLR-Implementierung des Visitor-Patterns verwendet. Bei der Generierung des LimitC-Parsers wurde von ANLTR bereits die nötige Visitor-Basis-Klasse erzeugt, welche Funktionen implementiert, die das Abgehen des Baumes steuern bzw. während dessen aufgerufen werden. Diese Funktionen werden in einer neuen Klasse geerbt und entsprechend mit eigenen Aktionen versehen. Diese Aktionen sollen im Falle von LimitC die Interpretation des Codes sein und enthalten somit eine Vielzahl von verschiedenen Operationen von arithmetischen Berechnungen über Funktionsaufrufe und Variablenzuweisungen bis hin zu impliziten und expliziten Typumwandlungen. Trotz des reduzierten Funktionsumfangs von LimitC gegenüber C stellt die Interpretation eines solchen Codes keine einfache Aufgabe dar. Im Folgenden sollen einige Ausschnitte des LimitC-Interpreters erläutert werden. Der Interpreter wird dabei durch die Implementierung des Visitor-Patterns in Form der Klasse $LimitCVisitor$ umgesetzt.

Die grundlegendste Funktion für Speicherbelegungsprotokolle stellt die Möglichkeit zum Definieren und Zuweisen von Variablen dar. Eine Definition kann dabei optional eine initiale Zuweisung enthalten. Die Definition einer Variable wird dabei von der in Listing 22 gezeigten Funktion $VisitVarDef$ implementiert. Die Definition von mehreren Variablen funktioniert hierbei identisch zu der Definition einer einzelnen Variable. Der Unterschied liegt nur darin, wie oft die $foreach$-Schleife in Zeile 4 durchlaufen wird. Innerhalb dieser Schleife passieren dabei immer folgende Dinge: der Variablenname und der aktuelle Scope wird ermittelt und dann erfolgt das initiale Anlegen der Variable im Speicher. Dabei werden Variablen, welche keine Zeiger sind, immer ein Initial-Wert von 0 gegeben. Dies ist zwar nach der C-Spezifikation nicht für alle Variablen explizit notwendig, sorgt aber dafür, dass jede Variable zumindest einen Wert in seinem gültigen Wertebereich hat, statt $null$ zu sein (*ISO/IEC 9899:2017 C17 ballot,* S. 101). Zeiger werden mit dieser Implementierung entsprechend auf $null$ verweisen, was der Definition von $null$-Zeigern entspricht. Die letzte Aktion einer jeden Variablen-Definition ist hierbei das Abarbeiten einer möglichen initialen Zuweisung für die aktuelle Variable, welches der $Visit$-Aufruf auf Zeile 15 veranlasst.

```
1  public override object? VisitVarDef([NotNull] LimitCParser.VarDefContext context)
2  {
3      var type = context.type().GetText();
4      foreach (var def in context.varAssignDef()) // für jeden Bezeichner der Defintion
5      {
6          var name = def.ID().GetText(); // Name der Variable
7          Scope currentScope = _scopes.Count > 0 ? _scopes.Peek() : _globalScope; // aktueller Scope
8          object? val = null; // Initialisierungswert
9          if (!type.Contains("*")) // Kein Zeiger -> mit 0 initialisieren
10         {
11             val = 0;
12         }
13          // variable in Scope hinzufügen und in Speicher ablegen
14         currentScope.AddVar(name,  MemoryStorage.AddToMemory(type, val));
15         Visit(def); // Zuweisung der Variable
16     }
17     return null;
18 }
```

Listing 22 Visitor-Implementierung Variablen Definition

Jede verarbeitete Variablendefinition führt einem Aufruf der in Listing 23 gezeigten Funktion $VisitVarAssignDef$. Die Funktion wird unabhängig davon aufgerufen, ob eine Initialisierung mit einem Wert vorhanden ist oder nicht. Wenn kein Wert für die Initialisierung angegeben ist, endet die Ausführung der Funktion direkt mit dem $return$ in Zeile 5. Ist jedoch ein Ausdruck für die

Werteinitialisierung angegeben, wird dieser in Zeile 8 ausgewertet und in Zeile 12 der entsprechende Wert in den Speicher geschrieben. Trotz, dass dies der Initialwert ist, wird das Eintragen des Wertes als Update ausgeführt, da das initiale eintragen der Variable in den Speicher bereits passiert ist.

```
1  public override object? VisitVarAssignDef(LimitCParser.VarAssignDefContext context)
2  {
3      var exp = context.expr(); // Zuweisungs Ausdruck
4      if (exp == null) // Wenn keine Zuweisung vorhanden -> return
5          return null;
6      var varName = context.ID().GetText(); // Variablenname holen
7      var currentScope = FindScopeForVar(varName); // aktuellen Scope ermitteln
8      var value = Visit(exp); // Expression bearbeiten
9      //Wenn Expression zu einem zuweisbaren (nicht null) Wert führt -> zuweisen
10     if (value != null)
11     {
12         MemoryStorage.ChangeInMemory(currentScope.GetAdress(varName), value);
13     }
14     return null;
15 }
```

Listing 23 Visitor-Implementierung Wertezuweisung bei Variablendefinition

Bei der Definition und dem Zugriff von Variablen müssen neben der reinen Werte- und Typen-Speicherung auch Gültigkeitsbereiche beachtet werden. Die $LimitCVisitor$ Implementierung eines virtuellen Speichers, in dem Variablen an einer Adresse angelegt werden können, wurde mittels eines $Dictionary$s umgesetzt, welcher als Key einen Integer verwendet und als Value eine Instanz vom Typ $TypedValue$. Diese Speicherstruktur namens $MemoryStorage$ stellt mit den zugehörigen Hilfsfunktionen zum Einfügen, Ändern und Lesen der abgelegten Variablen den Speicher des Interpreters dar und ist in Listing 24 verkürzt dargestellt. Dieser Speicher ist innerhalb einer Ausführung global und wird von den aus dem Visitor zugreifenden Funktionen nur durch die Hilfsfunktionen verändert. Für den genauen abzuspeichernden Wert ist dabei die Klasse $TypedValue$ verantwortlich. Diese Klasse enthält eine einfache Implementierung, welche für die verfügbaren Datentypen simple Type-Umwandlungen durchführt. Dabei wird vorausgesetzt, dass der übergebene Wert auch ohne einen separaten Parse-Vorgang in den angegebenen Typ wandelbar ist. Dies wird durch die Voraussetzung, dass LimitC-Code auch valider C-Code sein muss, sichergestellt. Wird diese Voraussetzung verletzt werden, kann es zu unvorhergesehenem Verhalten oder Abstürzen kommen. Innerhalb der $TypedValue$ Klasse werden die Werte selbst immer entweder als int, $double$ oder $null$ abgespeichert. Dies ermöglicht es bei der Auswertung von arithmetischen Ausdrücken anhand des abgespeicherten Wertes entweder Ganzzahlen- oder Fließkomma-Arithmetik zu verwenden und die entsprechende Operation anhand des erkannten Typen abhängig zu machen. Zwar unterscheidet sich in Standard-C der zugrundeliegende Speichertyp normalerweise, doch für viele Operationen werden Variablen verschiedener Typen dann doch wieder auf einen Typ gebracht. Da für einfache Speicherbelegungsprotokolle kaum typspezifische Operationen benötigt werden, welche sich nicht auf Ganzzahlen und Gleitkommazahl reduzieren lassen, fällt diese Reduzierung in der Praxis nicht auf.

```
public class MemoryStorage
{
    public int MemoryLastPos = 0;
    public Dictionary<int, TypedValue> Memory = new();
    public int AddToMemory(string type, object? val)
    {/* ... */}
    public object? GetFromMemory(int addr)
    {/* ... */}
    public void ChangeInMemory(int addr, object? nval)
    {/* ... */}
}
```

Listing 24 Implementierung Speicherstruktur

Dieser Speicher allein stellt allerdings kein ausreichendes Mittel für die Beachtung von Gültigkeitsbereichen von Variablen dar. Dazu gibt es eine weitere Datenstruktur mit dem Namen $Scope$, welche in gekürzter Form in Listing 25 abgebildet ist. Diese Datenstruktur verwaltet die Variablenbezeichner zusammen mit ihrer zugehörigen Adresse im Speicher. Zudem kann sie für gegebene Variablen, sofern innerhalb des aktuellen Scopes vorhanden, die zugehörige Adresse liefern und prüfen, ob eine Variable im Scope definiert ist oder nicht. Zusätzlich kann jeder Scope beliebig viele Unter-Scopes haben, um damit ineinander verschachtelte Scopes abzubilden. Die in Listing 25 gekürzten Funktionen implementierten dabei einfache Operationen auf diesen Scope, beziehungsweise auf das zugrundeliegenden $vars$-Dictionary. Die Funktion $AddVar$ fügt eine neue Variable zusammen mit ihrer Adresse im Speicher in den Scope ein. Dabei wird auch geprüft, ob die Variable im aktuellen Scope schon definiert war, was eine Ausnahme auslösen würde. $ContainsVar$ prüft, ob eine Variable im aktuellen Scope – oder einem zugehörigen Unter-Scope – definiert ist. Die Funktion $GetAdress$ gibt für eine definierte Variable die zugehörige Speicheradresse zurück, sofern vorhanden.

```
public class Scope
{
    public Dictionary<string, int> vars = new();
    public Stack<Scope> SubScopes = new();
    public void AddSubScope()
    { SubScopes.Push(new Scope()); }
    public void RemoveSubScope()
    { SubScopes.Pop(); }
    public void AddVar(string name, int addr)
    { /* ... */ }
    public bool ContainsVar(string varName)
    { /* ... */ }
    public int GetAdress(string varName)
    { /* ... */ }
}
```

Listing 25 gekürzte Implementierung der Scope-Klasse

Um nun beim Betreten eines neuen Codeblocks einen neuen Scope zu erhalten, wird einfach zum aktuellen Scope ein neuer Sub-Scope hinzugefügt, wie in Listing 26 dargestellt. Beim Verlassen des Blocks, also nach seiner vollständigen Abarbeitung, wird dann der zugehörige Sub-Scope wieder entfernt. Dabei verbleiben die innerhalb des Blocks definierten Variablen im Speicher.

```
public override object? VisitCodeBlock([NotNull] LimitCParser.CodeBlockContext context)
{
    Scope? cs = null;

    if (_scopes.Count == 0)
    {
        _scopes.Push(new Scope());
    }
    else
    {
        cs = _scopes.Peek();
        cs.AddSubScope();
    }

    var ret = VisitChildren(context);

    if (cs == null)
    {
        _scopes.Pop();
    }
    else
    {
        cs.RemoveSubScope();
    }

    return ret;
}
```

Listing 26 Visit-Implementierung für Scopes bei Codeblöcken

Somit werden innerhalb eines Blocks alle neu definierten Variablen in diesem neuen Scope definiert. Beim Abrufen einer Variable wird nun der $Stack$ von Scopes von oben nach unten abgelaufen und auf das Vorhandensein der gesuchten Variable geprüft. Wird beim Abgehen des Stacks keine entsprechende Variable gefunden, wird als Letztes der globale Scope überprüft, welcher ein separater Scope ist. Sollte die Variable auch dort nicht definiert sein, wurde sie entweder nicht oder außerhalb des gültigen Scopes definiert. Dadurch, dass beim Finden einer gültigen Definition die Funktion den zugehörigen Scope direkt zurückgibt, wird auch das „Überdecken“ von Variablen in darunterliegenden Scopes korrekt abgebildet. Eine Besonderheit bei diesem Ansatz stellen Funktions-Scopes dar, weil Funktionen immer über ihren eigenen Scope verfügen und nicht über den Scope in dem sie aufgerufen werden. Der globale Scope und der Speicher sind aber natürlich die gleichen. Statt wie beim Betreten eines Codeblocks wird beim Aufruf einer Funktion kein neuer Sub-Scope hinzugefügt, sondern ein komplett neuer Scope erzeugt. Vor dem Abarbeiten des entsprechenden Funktionsbaumes werden diesem neuen Scope noch die übergebenen Variablen kopiert und dem neu definierten lokalen Scope hinzugefügt. Dadurch hat der Funktionsaufruf den gleichen Speicher und globalen Scope zur Verfügung. Zusätzlich verfügt er dann über seinen eigenen Scope, welcher natürlich im Verlauf der Abarbeitung auch wieder beliebig viele Sub-Scopes haben kann. Implementierungstechnisch bekommt jeder Funktionsaufruf eine neue Objektinstanz vom Typ $LimitCSolver$, wie in Listing 27 dargestellt ist.

```
public override object? VisitFuncCall(LimitCParser.FuncCallContext context)
{
    object? ret = null;
    var fname = context.ID().GetText(); // Funktionsname
    if (FunctionDefs.TryGetValue(fname, out var functionDef)) // Funktion definiert?
    {
        var nlc = new Scope();  // neuen Scope erstellen
        if (context.paramList() != null) // wurden Parameter übergeben?
        {
            var parameters = context.paramList().param(); // Liste der übergebenen Parameter
            for (var index = 0; index < parameters.Length; index++)
            {
                var paramContext = parameters[index];
                var val = Visit(paramContext.expr()); // Ausdruck auswerten (Variable auflösen, Constant auswerten, etc.)
                var arg = functionDef.Arguments[index];
                nlc.AddVar(arg.name, MemoryStorage.AddToMemory(arg.type, val)); // Variable in neuen Scope einfügen und im Speicher neu ablegen
            }
        }
        var nlcv =  new LimitCVisitor(FunctionDefs, _globalScope, nlc, MemoryStorage); // neuer Visitor
        nlcv.LabelCheckPointReached += LabelCheckPointReached; // Erreignisse an EventHandler durchreichen
        ret = nlcv.Visit(functionDef.ParseTree); // Abarbeitung Funktion durchführen
    }
    return ret; // Rückgabewerte zurückgeben
}
```

Listing 27 Visitor-Implementierung Funktionsaufrufe

Diese Implementierung ermöglicht es, Funktionsausführungen jederzeit per $return$ komplett zu stoppen und so den Parsebaum wieder nach oben zu steigen, was andernfalls nur schwierig umzusetzen ist. Allerdings macht es diese Implementierung auch notwendig, dass Funktionen bereits definiert sind, bevor sie aufgerufen werden können. In Standard-C müssen Funktionen zwar auch vor ihrem Aufruf definiert worden sein, hier reicht jedoch die Existenz eines vorgelagerten Funktionsprototyps aus. Da aber beim Funktionsaufruf der komplette zur Funktion gehörende Parsebaum abgearbeitet werden soll, reicht die Kenntnis über die Existenz der Funktion und ihrer Parameter in dieser Form nicht aus. Um diese Funktionsbäume zur Ausführungszeit bereits zu kennen, wird die Erkennung von Funktionen in einen Schritt vor der Ausführung der $main$-Funktion verlagert. In diesem – in Listing 28 gezeigten - Schritt wird mittels des gleichen Parsers, aber eines anderen Walkers, eine Liste von Funktionsdefinitionen erstellt, welche dann dem $LimitCVisitor$ der die $main$-Funktion ausführt, mit übergeben wird.

```
var LimitCContext = LimitCParser.prog(); // erstellen des Parsebaumes

var functionDetector = new LimitCFunctionTreeBuilder(); // Funktions-Detektor
functionDetector.Visit(LimitCContext); // erkennen von erkannten Funktionen

var visitor = new LimitCVisitor(functionDetector.FunctionDefs, new Scope()); // main-Visitor
visitor.Visit(LimitCContext); // abarbeitung starten
```

Listing 28 Implementierung der globalen Funktions-Erkennung

Der verwendete $LimitCFunctionTreeBuilder$ ist eine weitere Implementierung des $LimitCBaseVisitor$s und besitzt nur einen einzigen Visitor für Funktionsdefinitionen und befüllt beim Erkennen einer Funktion ein $Dictionary < string, FunctionDef >$ mit den nötigen Informationen. Die Implementierung ist leicht gekürzt in Listing 29 zu sehen.

```
public partial class LimitCFunctionTreeBuilder : LimitCBaseVisitor<object?>
{
    public Dictionary<string, FunctionDef> FunctionDefs = new();
    public override object? VisitFuncDef(LimitCParser.FuncDefContext context)
    {
        if (context.codeBlock() != null) // ignorieren von Prototypen oder Deklarationen
        {
            var parameters = new List<(string type, string name)>();
            var paramDef = context.paramListDef()?.paramDef(); // Liste von Funktionsparametern
            if (paramDef != null)
                foreach (var paramDefContext in paramDef) // für jeden Parameter in der Definition
                {
                    if(paramDefContext.type().GetText() != "void")
                        parameters.Add((paramDefContext.type().GetText(), paramDefContext.ID().GetText())); // Parameterliste mit Parametername und Typ befüllen
                }
            // erkannte Funnktionsdefinition der Liste hinzufügen
            FunctionDefs.Add(context.ID().GetText(), new FunctionDef(context.ID().GetText(), context.type().GetText(), parameters, context.codeBlock()));
            Console.WriteLine($"detect function definition: {context.ID().GetText()}");
        }
        return null;
    }
}
```

Listing 29 Visitor-Implementierung für Funktionserkennung

Die in Listing 29 dargestellte Implementierung zeigt auch gut, dass durch die Verwendung des Visitor-Patterns ein Eingriff in die Abarbeitung des Parsebaumes möglich ist. Statt nämlich den Parsebaum unterhalb der Funktionsdefinition abzuarbeiten, wird dieser zusammen mit dem Funktionsbezeichner abgespeichert, um später von der Hauptfunktion abgearbeitet werden zu können. Dies bedeutet, dass alle Nodes innerhalb der Funktionsdefinition auch tatsächlich nicht besucht werden. Das Visitor-Pattern ermöglicht es, Nodes gar nicht zu besuchen oder die Evaluationsreihenfolge zu verändern, was wiederum große Aufmerksamkeit bei der Implementierung erfordert, da es schnell passieren kann, dass die Evaluationsreihenfolge unabsichtlich verändert wird oder Child-Nodes unbeabsichtigt nicht besucht werden.

Neben Funktionen, Sichtbarkeitsbereichen und einfachen Zuweisungen können natürlich auch arithmetische Ausdrücke in Speicherbelegungsprotokollen eine Rolle spielen. Als Beispiel wird in Listing 30 die Implementierung der Multiplikation dargestellt. Die Alternativrechnungen für Division, Addition und Subtraktion sind weitestgehend identisch. Die Zeilen 3 und 4 evaluieren den Content der beiden Operanten mit dem Rückgabetypen $object?$, was bedeutet, dass beide Operanten theoretisch auch $null$ sein könnten. Diese $null$-Zulässigkeit stellt aber nur ein Implementierungsdetail dar und keine echte Rechenmöglichkeit. Wäre mindestens einer der Operanten $null$,würde die Rechnung mit einer Fehlermeldung fehlschlagen. Durch die Verschleierung der exakten Typen der Operanten ist es nötig, diese im Folgenden zu testen, um korrekte Berechnungen vorzunehmen und auch den korrekten Typ zurückgeben zu können. Dies beruht auch darau,f dass auch in C# arithmetische Operationen direkt auf den zugehörigen Typen definiert ist. Eine Division auf die Typen $object?$ ist also nicht möglich, selbst wenn sich innerhalb ein int oder ein $float$ befindet. Daher reicht es also auch nicht aus, zu testen, ob einer der beiden Operanten ein int oder $float$ ist, da die genaue Rechenweise nur unter Kenntnis beider Typen bestimmt werden kann. Die if-Anweisungen in Zeile 7,11,15 und 19 prüfen genau diese Möglichkeiten und führen bei Übereinstimmung direkt eine Variableninitialisierung mit dem entsprechenden Typ aus. Dadurch ist der C#-Compiler in der Lage die Berechnung auf den bestimmten Typen durchzuführen.

```
1  public override object VisitMulDivExpression([NotNull] LimitCParser.MulDivExpressionContext context)
2  {
3      object? l = Visit(context.expr(0)), // evaluate left expression
4              r = Visit(context.expr(1)); // evaluate right expression
5      if (context.AST() != null)
6      {
7          if (l is int lint && r is int rint) // beides ints
8          {
9              return lint * rint; // returns int
10         }
11         if (l is double ldouble && r is double rdouble) // beides double
12         {
13             return ldouble * rdouble; // returns double
14         }
15         if (l is double ldouble2 && r is int rint2) // links ist double
16         {
17             return ldouble2 * rint2; // returns double
18         }
19         if (l is int lint2 && r is double rdouble2) // rechts is double
20         {
21             return lint2 * rdouble2; // returns double
22         }
23     }
24     else if (context.DIVOP())
25     {/* DIVISION */}
26     throw new InvalidOperationException(/* Error für invalide Rechnung */);
27 }
```

Listing 30 Implementierung Visit-Funktion für Multiplikations-Ausdrücke

Ebenfalls demonstriert diese Parse-Regeln den Umgang mit mehreren Übereinstimmungsmöglichkeiten innerhalb einer Regel. Die Grammatikregel für Multiplikations- und Divisionsausdrücke, die dem in Listing 30 gezeigten Visitor zu Grunde liegt, ist in Listing 31 dargestellt. Die Regel besteht aus zwei Ausdrücken, die mittels $expr$ gematcht werden, und sind verbunden mittels

eines Vorkommens von AST oder $DIVOP$. Die Lexer-Bezeichner AST und $DIVOP$ stehen hierbei für ein Sternchen „$*$“ beziehungsweise ein Slash „/“. Sie stehen in diesem Kontext für den Multiplikations- oder Divisionsoperator.

```
1       expr (AST | DIVOP) expr          # mulDivExpression
```

Listing 31 Parser-Regel für Multiplikations- und Divisions-Ausdrücke

Für die Unterscheidung, welcher Operator tatsächlich gematcht wurde, kann die Rückgabe der entsprechend erkennenden Funktion gegen $null$ gecheckt werden, wie es in Zeile 5 von Listing 30 zu sehen ist. Resultiert der Vergleich darin, dass die Funktion $null$ zurückgegeben hat, so ist klar, dass diese Option nicht gematcht wurde. Rein funktional ist diese Umsetzung identisch zu einer Formulierung von einzelnen Parser-Regeln für die Multiplikation und Division, mit jeweils eigenen Funktionen.

Neben der Verarbeitung von Code in C-äquivalenter Form gibt es eine Ausnahme von dieser Regel, nämlich den Label-Mechanismus zur Definition der Kontrollpunkte. Dieser hat zwar selbst keinen direkten Einfluss auf den Inhalt des LimitC-Programms, stellt aber den Stopppunkt für den Abgleich mit einem gegebenen Speicherbelegungsprotokoll dar. Und damit auch den Punkt, an welchem die Berechnung für fehlerberücksichtigende Lösungen startet. Die zulässige Form von Label-Definitionen wurde ja bereits in Kapitel 4.4 vorgestellt. Da Label vom Lexer bereits als vollständiges Element gematcht werden, beschränkt sich die zugehörige Parser-Regel auf genau dieses Label-Token, wie in Listing 32 gezeigt.

```
1   label: LABEL;
```

Listing 32 Parser-Regel für Label

Das Vorkommen eines solchen Labels löst im $LimitCVisitor$ die Funktion $VisitLabel$ aus, deren Anfang in Listing 33 dargestellt ist. Die Funktion hat primär die Aufgabe, die Programmausführung zu unterbrechen und das Vorkommen eines Labels mit allen nötigen Informationen nach außen zu geben. Außen bedeutet hierbei, es aus der aktuellen Klasseninstanz des $LimitCVisitor$s herauszugeben, um es in dem aufrufenden Programm verarbeiten zu können. Um eine Verarbeitung eines Kontrollpunktes realisieren zu können, wird unter anderem die Position des Kontrollpunktes in Form seiner Nummer nötig. Da das Label wegen der Überschneidung mit mehrzeiligen Kommentaren bereits im Lexer vollständig erfasst wird, liegt es im Parser nur als ein Token vor. Der Aufbau eines Labels ist allerdings so einfach, dass es mit einem sehr kurzen regulären Ausdruck erneut gematcht werden kann. Die Definition des Ausdrucks ist in Listing 33 auf Zeile 1 zu sehen und matcht unter Vernachlässigung von Groß- und Kleinschreibung das Wort „Label“ gefolgt von möglichen Leerzeichen und einer Zahl aus mindestens einer Ziffer. Schlägt dieser Abgleich fehl oder kann die Zahlensequenz nicht in einen int geparst werden, endet die Funktion direkt, da eine weitere Verarbeitung ohne die Label-Nummer nicht möglich ist.

```
private readonly Regex _labelRegex = new("Label[\\s]*([0-9]+)", RegexOptions.Compiled | RegexOptions.IgnoreCase);

public override object? VisitLabel([NotNull] LimitCParser.LabelContext context)
{

    var m = _labelRegex.Match(context.LABEL().GetText());
    if (!m.Success)
        return null;

    int n;
    if (int.TryParse(m.Groups[1].Value, out n) && n <= 0)
        return null;
```

Listing 33 Implementierung Label-Visitor Teil 1

Neben der Label-Nummer sind noch die Variablendaten der aktuellen Ausführung relevant, deren Sammeln in Listing 34 zu sehen ist. Zum Ermitteln aller sichtbaren Variablen wird über alle verfügbaren Scopes iteriert und ein Verzeichnis aus Variablennamen und der zugehörigen Adresse erstellt. Dieses Verzeichnis nutzt den Variablennamen als Schlüssel und gewährleistet so, dass jede sichtbare Variable, die in einem späteren Scope erneut vorkommt, einen vorherigen Eintrag überschreiben würde. Dadurch enthält das Verzeichnis l am Ende der dritten $foreach$-Schleife alle sichtbaren Variablen und deren Adressen. Da diese Liste aber keinerlei Werte- oder Typangaben enthält, wird zusätzlich die Instanz des Speichers mit übergeben. Diese gesammelten Daten werden in einer Klasse vom Typ $LabelCheckPointEventArgs$ vereint und dann das Event $LabelCheckPointReached$ ausgelöst. Somit kann ein Eventhandler, der an dieses Event geknüpft ist, eine Evaluation des aktuellen Standes bis zur erreichten Position des Labels durchführen.

```

    var cscope = _scopes.Peek(); // aktueller Scope
    var l = new Dictionary<string, int>(); // Liste der Variablen und ihren Adressen
    foreach (var (name, addr) in _globalScope.vars) // globale Variablen
    {
        l[name] = addr;
    }
    foreach (var (name, addr) in cscope.vars) // locale Variablen
    {
        l[name] = addr;
    }
    foreach (var scope in cscope.SubScopes) // locale SubScope variablen
    {
        foreach (var (name, addr) in scope.vars)
        {
            l[name] = addr;
        }
    }
    // Event auslösen
    LabelCheckPointReached?.Invoke(this, new LabelCheckPointEventArgs(n, l, MemoryStorage));
    return null;
}
```

Listing 34 Implementierung Label-Visitor Teil 2

4.6 Praktische Umsetzung des Lösers

Mit dem in Kapitel 4.5 vorgestellten Interpreter können LimitC Programme ausgeführt werden, da allerdings LimitC keinerlei Eingabe- oder Ausgabemechanismen unterstützt, ist eine reine Ausführung des Codes nicht zielführend. Außerdem wird eine Oberfläche zum Eingeben des Codes, dem Einladen von gegebenen Protokollen und dem Abbilden des Ergebnisses benötigt. Das Backend dieser Anwendung wurde wie auch der Interpreter selbst in C# mit .NET6 geschrieben. Des Weiteren wurde für die Benutzeroberfläche ein WPF-Frontend genutzt. Abseits der Ermöglichung der Interaktion mit dem Interpreter setzt auch diese Anwendung den in Kapitel vorgestellten Algorithmus zur Berechnung der fehlerbasierten Lösungen um, indem er bei Erreichen eines Kontrollpunktes den Abgleich mit dem gegebenen Protokoll durchführt und bei Erkennen eines Fehlers die fehlerhaften Daten direkt in die Laufzeitdaten des Interpreters einfügt. Zusätzlich implementiert die Anwendung ein Punktesystem, welches für korrekt angegebene Einträge Punkte vergibt. Die Anzahl der Punkte pro Match ist konfigurierbar, ebenso wie die Notwendigkeit von Variablentypen im Protokoll. Die Punktvergabe berücksichtigt dabei auch den Fall, dass zwischendurch eine Angabe falsch war, dann aber im weiteren Verlauf wieder zur Originallösung zurückgefunden wurde. Die Anwendung kennzeichnet die richtig und falsch erkannten Werte farblich, um eine bessere Verdeutlichung der Bewertung zu geben.

Benutzeroberfläche

Die Benutzeroberfläche der Anwendung ist in Abbildung 7 dargestellt. Der linke Teil der Anwendung enthält den Code der Aufgabenstellung, einen Namen für die Aufgabe sowie Buttons zum Speichern und Laden der Aufgabenstellung und für die Berechnung einer Lösung. Die gespeicherten Aufgabenstellungen sind auch die Ausgangsdateien für die Protokolleingabetool. Im mittleren Teil der Anwendung befinden sich im oberen Teil zwei Buttons zum Laden und Prüfen von Protokollen. Direkt darunter ist das zu prüfende Protokoll und nach einer Prüfung die berechneten Punkte. Unterhalb dessen wird die berechnete Originallösung angezeigt. Auf der rechten Seite des Fensters sind im oberen Bereich zwei Konfigurationsmöglichkeiten. Zum einen wird hier konfiguriert, ob Typen geprüft werden sollen und zum anderen kann der Punktewert pro Übereinstimmung angepasst werden. Die Einstellung für die Notwendigkeit von Typen wird beim Abspeichern der Aufgabenstellung mit übernommen, so dass das Eingabetool Typen abfordern kann oder nicht. Auch die zu vergebenden Punkte werden mit abgespeichert, spielen aber bei der Eingabe der Protokolle keine Rolle.

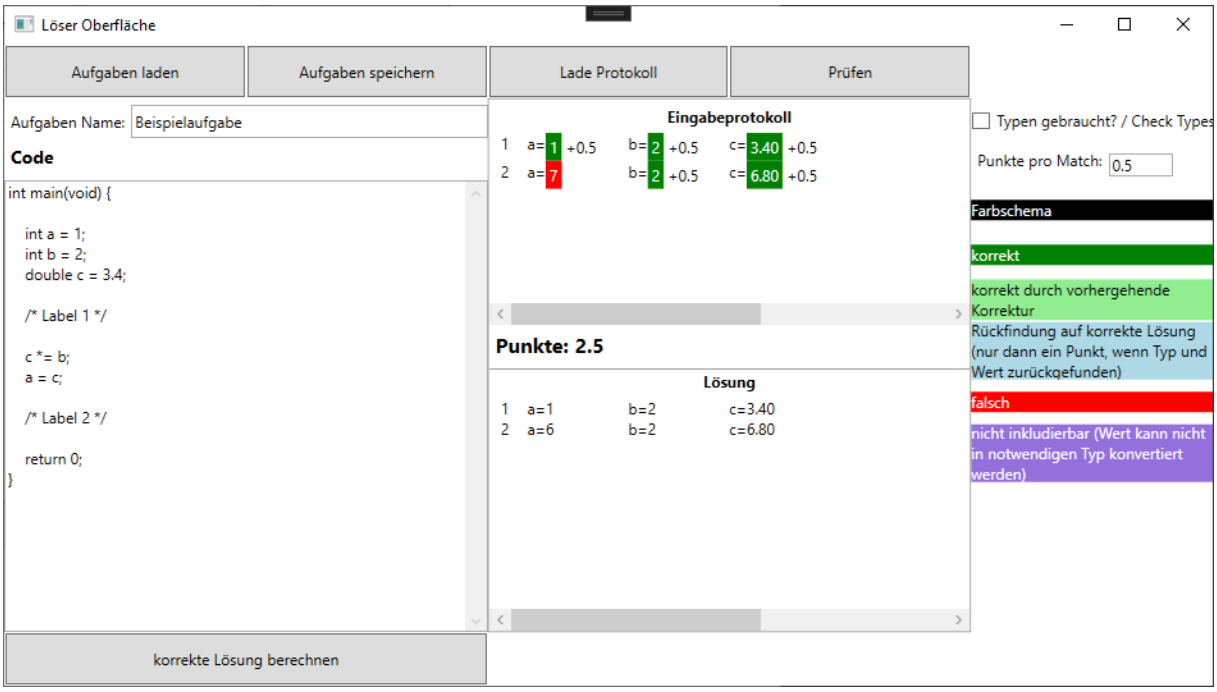


Abbildung 7 Benutzeroberfläche Löser

Backend

Die Prüfung eines geladenen Protokolls erfolgt immer Schritt für Schritt beim Erreichen eines Label-Kommentars durch den Interpreter. Beim Erreichen eines Labels löst dieser wie in Kapitel 4.5 beschrieben ein Event aus, welches durch den Eventhandler bearbeitet wird. Der Eventhandler startet mit der Prüfung, ob alle anwendungsseitigen Daten, die benötigt werden, vorhanden sind: abzugleichendes Protokoll und berechnete Originallösung. Ist beides vorhanden, wird aus beiden die zum ausgelösten Label gehörige Zeile mit Variableneinträgen geladen. Bevor der eigentliche Vergleich beginnt, werden die Variablen, die vom Interpreter übergeben wurden, einmal umsortiert, um sicherzustellen, dass erst alle Variablen von Basisdatentypen geprüft werden und dann erst zugehörige Zeiger. Dieser Schritt ist notwendig, um die korrekte Bewertung von Zeiger-Zielen gewährleisten zu können und findet bereits mit Zugriff auf den Ausführungsspeicher statt. Diese Anweisungen werden bei jedem neu erreichten Label genau einmal ausgeführt und sind in Listing 35 dargestellt.

```
private void VisitorOnLabelCheckPointReachedCheckProtokol(object? sender, LabelCheckPointEventArgs e)
{

    if (GivenProtokol == null)
    {
        Error($"Label {e.LabelNum} in Überprüfung erreicht, aber kein Protokoll zum Abgleich geladen?");
        return;
    }

    var vars = e.VisibleVars.OrderBy(se => e.MemoryStorage.Memory[se.Value].Type.Contains('*'));

    var protokollEntry = GivenProtokol.Entrys.FirstOrDefault(pe => pe.Num == e.LabelNum);
    if (protokollEntry == null)
    {
        Error($"Label {e.LabelNum} in Überprüfung erreicht,
        aber im Protokoll scheint kein entsprechender Eintrag vorhanden zu sein!");
        return;
    }

    var absSltV = CalcedSolution?.Entrys.FirstOrDefault(pe => pe.Num == e.LabelNum);

```

Listing 35 Implementierung Protokoll Überprüfung Teil 1 – Prüfung der Voraussetzungen

Nach diesen Schritten startet die Iteration über die vorher umsortierten Variablen. Da diese Variablenliste aus Scopes zusammengestellt wurde, enthalten diese nur den Namen der Variable und die zugehörige Adresse im Speicher. Für einen Typen- und Wertevergleich wird nun die zur Variable gehörige Instanz vom Typ $TypedValue$ aus dem Speicher gelesen, sowie die zu vergleichenden Werte aus dem gegebenen Protokoll und aus der Originallösung geladen. Beim Auslesen der äquivalenten Werte aus dem gegebenen Protokoll sowie aus der Lösung müssen Zeiger-Typen bereits beachtet werden, da für die Lesbarkeit dessen, was angegeben werden soll, die Dereferenzierungsoperatoren in den Protokollen vor dem Namen stehen. Die Implementierung dieser Variablendefinition ist in Listing 36 zu sehen.

```
foreach (var (name, addr) in vars)
{

    TypedValue memVal = e.MemoryStorage.Memory[addr];
    var p = new string('*', memVal.Type.Count(c => c == '*'));

    var absVarVal = absSltV?.VarEntrys.First(x => x.Name == $"{p}{name}").Value ?? "";
    var absVarType = absSltV?.VarEntrys.First(x => x.Name == $"{p}{name}").Type ?? "";

    var protVar = protokollEntry.VarEntrys.FirstOrDefault(pv => pv.Name == $"{p}{name}");

    if (protVar == null)
    {
        Error($"Fehler bei Label {e.LabelNum}, die sichtbare Variable {name} scheint" +
              $" im Protokoll an der entsprechenden Stelle nicht definiert zu sein!");
        return;
    }
```

Listing 36 Implementierung Protokoll Überprüfung Teil 2 – Protokolldaten laden

Nachdem alle nötigen Vergleichswerte definiert sind, beginnt der eigentliche Abgleich dieser. Als erstes erfolgt der in Listing 37 dargestellte Typencheck, sofern dieser per Konfiguration erforderlich ist. Ist dieser notwendig, wird im Folgenden der Type des Anwendungs-Datensatzes mit dem Typen aus dem gegebenen Protokoll verglichen und in der Zugehörigen Variable des geladenen Protokolls das Ergebnis vermerkt. Schlägt dieser Typencheck fehl, wird zusätzlich gegen die Originallösung verglichen und das

Ergebnis abgespeichert. Diese zweite Prüfung ist notwendig, da der erste Vergleich mit den aktuellen Anwendungsergebnissen berechnet wurde, welche potenziell schon von der Originallösung abweichen können.

```
if (!CurrentConfig.NeedTypes)
{
    protVar.TypeCheck = true;
}
else
{
    // Typed needed
    protVar.TypeCheck = memVal.Type == protVar.Type;
    if (protVar.TypeCheck == false)
    {
        // Type war nicht der den der aktuelle Durchlauf berechnet hat (könnte bereits korrigiert sein)
        protVar.TypeCheck = absVarType == protVar.Type;
        if (protVar.TypeCheck == true)
        {
            // korrigiert berechnete Lösung Falsch, aber korrekt zur absoluten Lösung
            protVar.AbsCorrectedType = true;
        }
    }
}
```

Listing 37 Implementierung Protokoll Überprüfung Teil 3 - Typencheck

Nach dem Typencheck erfolgt der Vergleich des angegebenen Wertes. Handelt es sich bei der aktuellen Variable um einen Zeiger, so muss der Zeiger erst noch aufgelöst werden, da im Speicherbelegungsprotokoll nicht die Adresse des Zeigers angegeben wird, sondern der Wert seines Verweiszieles. Der Zielwert des Zeigers wird, wie in Listing 38 dargestellt, mittels wiederholendem Loopkup in den Ausführungsspeicher ermittelt. Dabei wird keine weitere Typenprüfung der Zeigerziele vorgenommen, da dies darauf beruht, dass es sich bei dem Ausführungscode um einen validen LimitC-Code handelt, welcher keine Null-Zeiger zulässt und somit keine Zeiger zwischen verschiedenen oder unbekannten Typen möglich sind.

```
var memV = memVal.Value; // Variablen Wert oder Ziel-Adresse (wenn Zeiger)
var memA = -1;
if (memVal.Type.Contains('*')) // aktuelle Variable ein Zeiger? -> memV == Adresse ?? null
{
    if (memV != null) // kein Null-Pointer -> memV == Adresse
    {
        TypedValue? vov = null;
        if (e.MemoryStorage.Memory.ContainsKey((int)memV)) // Adresse definiert?
        {
            vov = e.MemoryStorage.Memory[(int)memV]; // Lookup Adresse im Speicher
            memA = (int)memV;                        // Zeilwert oder neue Adresse
            for (int i = 1; i < p.Length; i++) // Vorgang für Zeigertiefe {p.Length} wiederholen
            {
                // Abbruch, bei vorzeitigem Null-Zeiger oder nicht auflösbarem Ziel.
                if (vov.Value == null || !e.MemoryStorage.Memory.ContainsKey((int)vov.Value))
                {
                    vov = null;
                    break;
                }
                memA = (int)vov.Value;
                vov = e.MemoryStorage.Memory[(int)vov.Value];
            }
        }
        memV = vov?.Value ?? null; // zuletzt gefundenen Wert übernehmen
    }
}
```

Listing 38 Implementierung Protokoll Überprüfung Teil 4 - Auflösung Zeiger-Werte

Nach dem Auflösen des Zeigers ist der Wert, der tatsächlich verglichen werden soll, bekannt und kann weiterverarbeitet werden. Unabhängig davon, ob der aktuelle Variablen-Wert aus einem Zeigerverweis oder direkt aus einer Variable kommt, wird dieser Wert nun auf eine vergleichsfähige String-

Repräsentation reduziert, um eine Vergleichbarkeit mit den Werten aus dem eingelesenen Protokoll herzustellen. Die String-Darstellung vereinfacht es, datentypenabhängige Formate zu vergleichen. So kann für $char$s oder Zeiger auf solche sowohl die numerische Darstellung verglichen werden als auch die des äquivalenten Zeichens. Auch bei Gleitkommazahlen kann so eine spezifische Genauigkeit verglichen werden, statt einer spezifischen Zahl, deren Genauigkeit wohlmöglich durch nötiges Parsing Probleme bereiten könnte. In der in Listing 39 gezeigten Implementierung dieser Stringkonvertierung ist zu sehen, dass Gleitkommazahlen – welche intern immer als $double$ gespeichert werden – hier auf eine feste Genauigkeit von zwei Nachkommastellen reduziert werden. Auch die doppelte Abspeicherung von $char$ Werten für einen Integer- und einen Zeichenvergleich ist ersichtlich, sowie eine Präsentation für $null$-Werte, welche theoretisch in Form von $null$-Zeigern auftreten könnten.

```
string valval = "";
string valval2 = "";
if (memV is int memValInt)
{
    valval = memValInt.ToString();
    if (memVal.Type.Contains("char"))
        valval2 = ((char)memValInt).ToString();
}
else if (memV is double memValDouble)
{
    valval = memValDouble.ToString("F2", CultureInfo.InvariantCulture);
}
else if (memV is null)
{
    valval = "NULL";
}
```

Listing 39 Implementierung Protokoll Überprüfung Teil 5 - Konvertierung in Stringdarstellung

Nach der Umwandlung wird der Wertevergleich angestellt und geprüft, ob die Eingabe mit der Berechnung übereinstimmt. Schlägt diese Überprüfung fehl, wird auch hier wieder mit der Originallösung verglichen.

Wird bei einer der beiden Prüfungen eine Falscheingabe entdeckt, erfolgt nun die Anpassung des aktuellen Ausführungszustandes für die Berechnung von Folgefehlerlösungen, wie in Listing 40. Um den falschen Wert nun für die weitere Ausführung verwenden zu können, muss dieser noch in den richtigen Typ gewandelt werden. Das ist entweder der im Protokoll angegebene Typ, wenn dieser verlangt war, oder der Typ aus dem Ausführungsspeicher. Schlägt diese Konvertierung fehl, ist eine Einbeziehung des Wertes in die weitere Ausführung nicht möglich. War die Konvertierung erfolgreich, wird der Wert direkt in den Speicher der aktuellen Ausführung geschrieben und die Variable – genauer gesagt die Adresse - als korrigiert markiert.

```
object? nval = memV;
string ntype = memVal.Type;
if (protVar.TypeCheck == false && protVar.Type is "int" or "short" or "long" or "float" or "double" or "char")
{
    ntype = protVar.Type;
}
try
{
    if (protVar.ValueCheck == false)
    {
        if (ntype is "int" or "short" or "long")
        {
            nval = int.Parse(protVar.Value);
        }
        else if (ntype is "float" or "double")
        {
            nval = double.Parse(protVar.Value, CultureInfo.InvariantCulture);
        }
        else if (ntype is "char")
        {
            int tci;
            if (int.TryParse(protVar.Value, out tci))
            {
                // numerische Darstellung
                nval = (char)tci;
            }
            else
            {
                nval = Convert.ToChar(protVar.Value.Replace("'", ""));
            }
        }
    }
    e.MemoryStorage.Memory[addr] = new TypedValue(ntype, nval);
    CorrectedVars.Add((name, addr));
}
catch (Exception exception)
{
    Error(exception.Message);
    protVar.FailedToInclude = true;
}
```

Listing 40 Implementierung Protokoll Überprüfung Teil 5 - Korrektur des Ausführungsspeichers für Folgefehlerberechnung

Ist bei dem beschriebenen Vergleichsprozess kein Fehler aufgetreten, also die aktuelle Variable korrekt – entweder korrekt nach zurückliegender Folgefehlererkennung oder korrekt gegenüber der Originallösung – wird ein entsprechender Punktwert vergeben und die nächste Variable geprüft. Nachdem alle Variablen in dieser Art verarbeitet wurden und somit auch potenziell auch der aktuelle Anwendungsspeicher des Interpreters angepasst wurde, endet der Eventhandler und der Interpreter setzt die Bearbeitung des Codes fort. Entweder bis das nächste Label auftritt oder der Code vollständig abgearbeitet ist. Durch die Markierung von Variablen und Adressen wird bereits während des Interpretationsvorganges in der GUI eine entsprechende farbliche Markierung vorgenommen. Die Markierungen können insgesamt 4 Zustände abbilden:

- ➔ Die Variable wurde korrekt erkannt und zuvor auch noch nicht korrigiert.
- ➔ Die Variable wurde korrekt erkannt, wurde aber vorher bereits korrigiert (Folgefehler).
- ➔ Die Variable ist korrekt zur Originallösung, weicht aber von der aktuellen Ausführung ab.
- ➔ Die Variable ist falsch angegeben.

Alle vier möglichen Zustände sind beispielhaft in Abbildung 8 demonstriert, ebenso wie die Punktevergabe. Bei der Punktevergabe ist auch zu sehen, dass es sowohl Punkte für Folgefehler gibt wie bei Variable b bei Label 2 und 3, also auch für Übereinstimmungen mit der Originallösung, nach Anpassung durch einen erkannten Fehler für Variablen a und c an den Labeln 2 und 3.

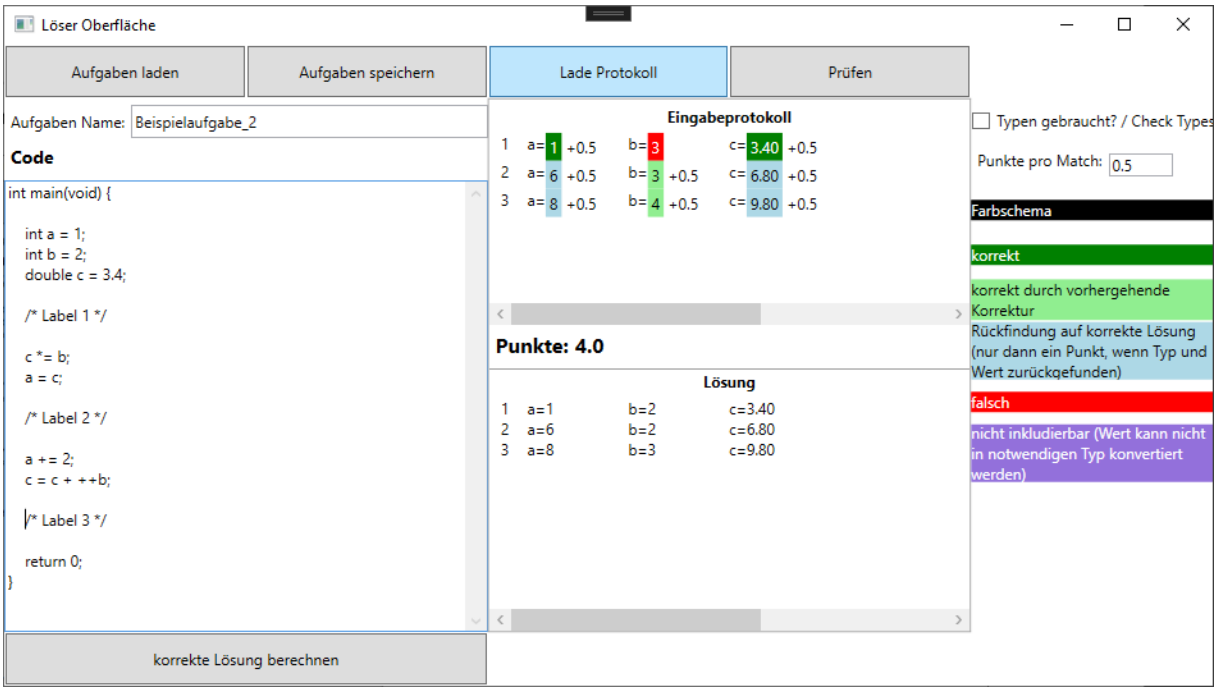


Abbildung 8 Anwendungsoberfläche mit allen Bewertungszuständen

Wenn Typen gefordert sind, werden diese je nach Erfolg der Typenprüfung ebenso gekennzeichnet wie die Variablen selbst. Dies führt dazu, dass zwar ein Wert richtig erkannt worden sein kann, aber so wie in Abbildung 9 der zugehörige Typ nicht. Im Falle von geforderten Typen bringt nur eine korrekte Übereinstimmung von Wert und Typ eine korrekte Bewertung.

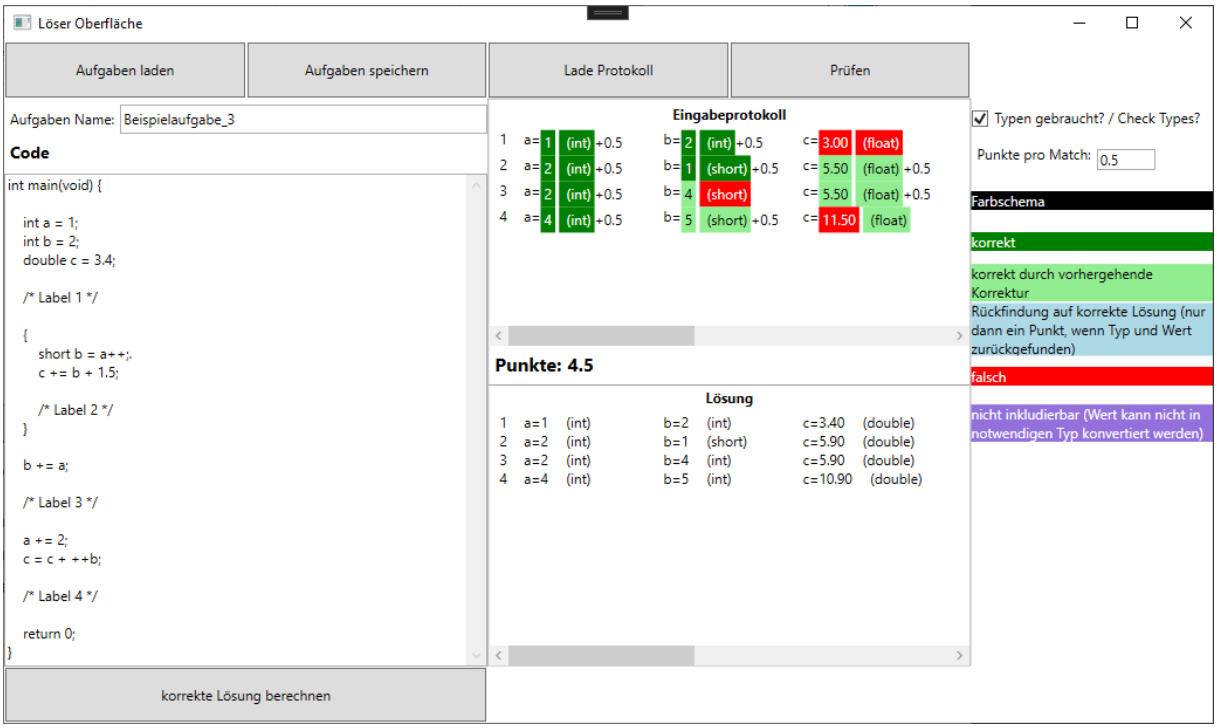


Abbildung 9 Anwendungsoberfläche mit Ergebnis und Typenprüfung

Die Anwendung erwartet zum Laden der Protokolle ein spezifisches Datenformat, welches dann in die interne Darstellung eines Protokolls geladen werden kann. Um den Prozess des Protokollerstellens möglichst einfach zu halten, wurde auch dafür eine Anwendung erstellt, welche die Aufgabendateien

des Lösers als Ausgangslage nimmt. Es erlaubt dem Nutzer, das vorgegebene Protokoll zu befüllen. Da die Aufgabendatei auch die Konfiguration bezüglich Notwendigkeit von Dateitypen enthält, kann die Eingabeanwendung die Oberfläche danach anpassen und zeigt die Typenfelder auch nur dann an, wenn sie nötig sind, wie im Beispiel in Abbildung 10 zu sehen. Beim Öffnen einer Aufgabendatei wird das Format des erwarteten Protokolls vollständig vorgegeben, was die Frage nach der Sichtbarkeit vorwegnimmt. Dadurch wird die Eingabe um ein deutliches vereinfacht. Zusätzlich enthält das Eingabetool eine Validierungsfunktion, welche gewährleistet, dass sofern Typen verlangt werden, auch nur Eingaben getätigt werden, die im entsprechenden Typ abbildbar sind. Dies vereinfacht die Erkennung in der Lösungsanwendung und sorgt dafür, dass es die Eingaben auch verwertet werden können. Da für das Verständnis von Speicherbelegungsprotokollen bereits ein grundlegendes Verständnis von Datentypen erforderlich ist, sollte dies kaum eine zusätzliche Hilfestellung darstellen. Die Eingabevalidierung findet nur statt, wenn Typen angegeben werden sollen.

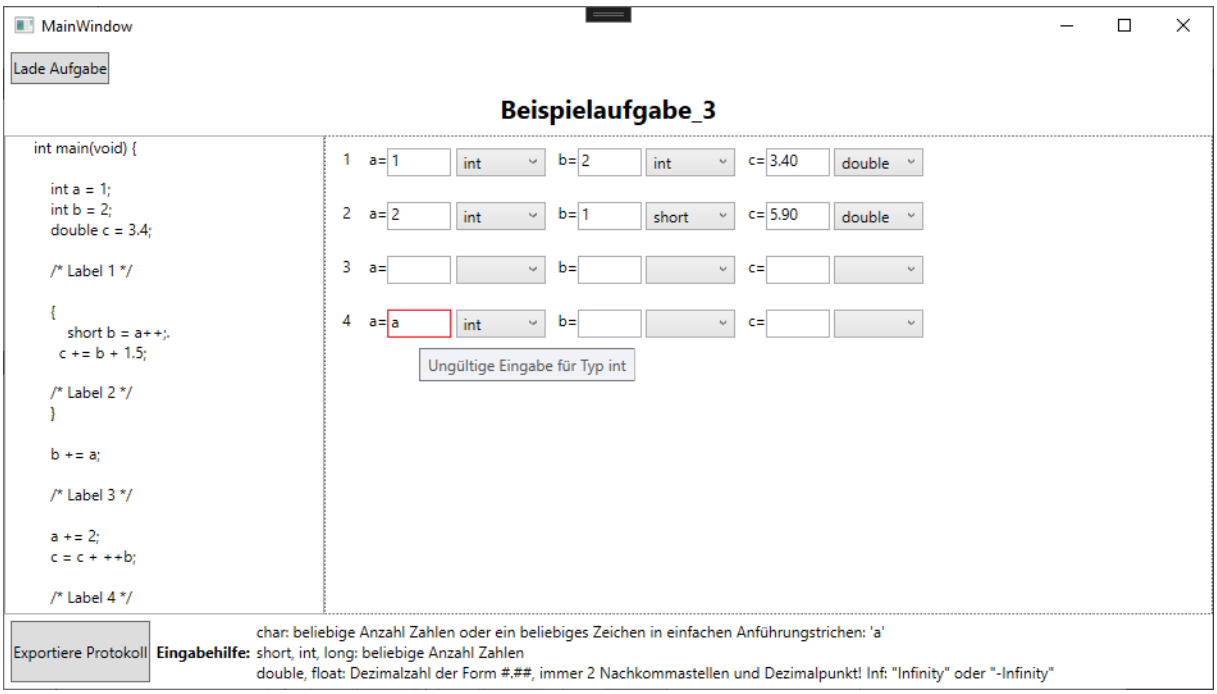


Abbildung 10 Protokoll-Eingabetool beispielhaft mit Typen und Eingabevalidierung

Protokolle für den Löser können auch ohne die Nutzung des Eingabetools erzeugt werden und als JSON-Datei importiert werden. Da das Format für den Austausch zwischen Löser und Eingabetool mehrere Felder aufweist, welche für ein anderweitiges Erstellen weder notwendig noch sinnvoll wären, wurde hierfür ein reduziertes JSON-Schema erstellt, welches beispielhaft Listing 41 dargestellt ist.

```json
[
    {
        "Label": 1,
        "Vars": [
            {
                "Name": "a",
                "Type": "int",
                "Value": "1235"
            },
            {
                "Name": "b",
                "Type": "float",
                "Value": "-Infinity"
            },
            {
                "Name": "c",
                "Type": "char",
                "Value": "'c'"
            }
        ]
    },
    {
        "Label": 2,
        "Vars": [
            /* gekürzt */
        ]
    }
]
```

Listing 41 JSON-Beispiel für Speicherbelegungsprotokoll

Für die formale Definition des erwarteten JSON wurde eine standardisierte JSON-Schema-Definition erstellt. Die Schema-Definition des JSONs definiert dabei nur die Felder, welche zum Abbilden der Protokolle notwendig sind. Alle Felder sind dabei obligatorisch und müssen angegeben werden. Strukturell betrachtet ist das Format ein Array von Objekten, welche über ein Feld $Label$ verfügen und über ein Array mit dem Namen „$Vars$“. Das $Vars$-Array besteht dabei wiederum aus Objekten mit den drei String-Properties $Name$, $Type$ und $Value$. Wenn keine Typen verlangt werden, kann das $Type$-Feld leer sein, muss aber trotzdem mit angegeben werden. Diese Definition weicht von dem Format ab, welches beim Protokollexport aus dem Eingabetool erzeigt wird. Das durch das Eingabetool erzeugte Protokoll enthält mehr Felder, welche nicht notwendig sind für den Import. Diese werden aber mit exportiert, weil die serialisierte Klasse auch Teil der Aufgabenstellung ist, wofür die Felder dann wiederrum relevant sind. Das zusätzlich definierte Importformat ist vollständig kompatibel und wird beim Import wieder in die interne Darstellung konvertiert. Die formale JSON-Definition ist

5 Fazit

5.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde eine Lösungsanwendung entwickelt, welche in der Lage ist, teilkorrekte Speicherbelegungsprotokolle einzulesen und zu verarbeiten. Anhand des zu einem Speicherbelegungsprotokoll gehörenden Programmcodes kann es aufgetretene Fehler und daraus resultierende Folgefehler korrekt erkennen und eine entsprechende Bewertung erstellen. Ebenso kann für den Programmcode das korrekte Speicherbelegungsprotokoll berechnet werden. Der unterstützte Programmcode wurde auf einen Ausschnitt der Programmiersprache C beschränkt, welcher unter dem Namen LimitC abgegrenzt wurde. Um die Verarbeitung des Programmcodes zu ermöglichen, wurde eine Grammatik für den in LimitC unterstützten Programmumfang erstellt. Anhand der erstellten Grammatik wurde mit Hilfe des Parsergenerators ANTLR ein funktionaler Parser generiert. Auf der Grundlage dieses generierten Parsers wurde ein Interpreter für LimitC erstellt, der in der Lage ist, entsprechenden Code auszuführen und an den für die Kontrolle der Protokolle nötigen Stellen entsprechende Ereignisse zu generieren. Dieser Interpreter wird vom Lösungsprogramm verwendet, um sowohl korrekte Lösungen als auch Folgefehler berücksichtigende Programmabläufe zu erstellen und anhand dieser das Speicherbelegungsprotokoll zu bewerten. Für die Erstellung von Lösungen unter Berücksichtigung von Folgefehlern wurde ein entsprechender Algorithmus beschrieben und dann innerhalb der Lösungsanwendung umgesetzt.

Für die Verarbeitung von Programmcode innerhalb der Lösungsanwendung wurden relevante Grundlagen des Compiler- und Interpreterbaus vorgestellt. Die betrachteten Techniken wurden im Folgenden zusammen mit den Grundlagen aus dem Themengebiet der formalen Sprachen angewandt, um die Grammatik und den Parser für den C-Ausschnitt LimitC zu erstellen sowie mit dem Generieren eines Parsers praktisch angewandt. Als Grundlage für diese Techniken wurde das Themengebiet der formalen Sprachen vorgestellt und definiert.

Zusätzlich zur Lösungsanwendung wurde ein Eingabetool gebaut, welches das Erfassen und Exportieren von Speicherbelegungsprotokollen für die Lösungsanwendung ermöglicht. Für den Import von Speicherbelegungsprotokollen in die Lösungsanwendung wurde zusätzlich ein formales JSON-Schema entwickelt, was einen Import aus weiteren Quellen ermöglicht.

5.2 Ausblick

Die Implementierung der Lösungsanwendung ist ein Prototyp und keine vollumfängliche Anwendung. Zwar ist die Implementierung des Lösungsalgorithmus funktional und als solche auch zweckdienlich, aber gerade im Bereich der Usability ist noch Potenzial vorhanden. Für eine praktische Verwendung der Anwendung im Lehrbetrieb wäre eventuell eine bessere Aufteilung der Benutzeroberfläche möglich. Auch die Punktevergabe, die aktuell nur eine Veränderung des Punktwertes pro Übereinstimmung zulässt, könnte erweitert werden und zum Beispiel unterschiedliche Fehler unterschiedlich bewerten. Ebenso könnte eine manuelle Eingabemöglichkeit für Speicherbelegungsprotokolle innerhalb der Lösungsanwendung sinnvoll sein. Insbesondere wenn es nötig wäre, zum Beispiel papierhafte Abgaben zu überprüfen, ohne vorher einen Umweg über ein zusätzliches Eingabetool nehmen zu müssen. In diesem Kontext wäre auch eine Weiterentwicklung des Eingabetools für die Herausgabe an die Schüler oder Studenten denkbar. Damit wäre es möglich, dass die Protokollierenden ihre Protokolle direkt mit diesem Tool erfassen und dann an die Lehrkraft weitergeben. Hier wäre zum Beispiel auch eine Onlineanbindung denkbar, bei welchen Aufgabenstellungen zentral verteilt werden und Lösungseingaben direkt an die Lehrkraft weitergeleitet werden könnten. Diesem Gedanken folgend wäre es auch möglich, beide Anwendungen statt als Desktop-Anwendung als Web-Anwendung zu konzipieren. Dies könnte eine direkte Selbstkontrolle oder Selbstbewertung, die durch Schüler oder Studenten vorgenommen wird, ermöglichen.

Für die praktische Verwendung wäre auch in beiden Anwendungen ein Syntaxhighlighting denkbar, welches die Lesbarkeit des Protokollcodes erhöhen würde.

Mit einem Austausch der Benutzeroberfläche wäre es auch möglich, beide Anwendungen für andere Betriebssysteme als Windows zu erstellen. Zwar ist die gewählte Frontendtechnik mit WPF auf Windows beschränkt, doch die zugrundliegende Plattform ist .NET6 mit C#, womit mit geringem Mehraufwand auch Apps für andere Plattformen möglich sind.

Der zugrundeliegende LimitC Interpreter kann je nach Anforderung angepasst werden. Sinnvoll ist zum Beispiel die Erweiterung des LimitC-Umfangs um Schleifen und Bedingte-Anweisungen. Solche Erweiterungen der unterstützten Sprache sind auch ohne Anpassen der Lösungsanwendung oder des Eingabetools möglich, da nur die Lösungsanwendung mit dem Interpreter interagiert und diese Interaktion unabhängig vom Sprachumfang agiert. Durch Austausch des Interpreters wäre es auch möglich Unterstützung für weitere Programmiersprachen hinzuzufügen.

Literaturverzeichnis

Aho AV, Lam MS, Sethi R, Ullman JD, Leuschel M. Compiler. Prinzipien, Techniken und Werkzeuge. München, Boston, San Francisco, Harlow, England, Don Mills, Ontario, Sydney, Mexico City, Madrid, Amsterdam: Pearson Studium 2008.

Backus JW, Bauer FL, Green J, Katz C, McCarthy J, Naur P, Perlis AJ, Rutishauser H, Samelson K, Vauquois B, Wegstein JH, van Wijngaarden A, Woodger M. Report on the algorithmic language ALGOL 60. Numer. Math. 1960; 2(2): 106–136.

Bison - GNU Project - Free Software Foundation. 31 August 2023. https://www.gnu.org/software/bison/ (31 August 2023).

Böckenhauer H-J, Hromkovic J. Formale Sprachen. Wiesbaden: Springer Fachmedien Wiesbaden 2013.

Erez Shinan (erezsh). lark-parser/lark: Lark is a parsing toolkit for Python, built with a focus on ergonomics, performance and modularity. 31 August 2023. https://github.com/lark-parser/lark (31 August 2023).

Hedtstück U. Einführung in die theoretische Informatik. Formale sprachen und Automatentheorie. Munich, Germany: Oldenbourg Verlag 2012.

Hoffmann DW. Theoretische Informatik. Mit 22 Tabellen und 78 Aufgaben. München: Hanser 2009.

Hopcroft JE, Motwani R, Ullman JD. Einführung in Automatentheorie, formale Sprachen und Berechenbarkeit. München: Pearson Studium 2011.

Hromkovič J. Theoretische Informatik. Formale Sprachen, Berechenbarkeit, Komplexitätstheorie, Algorithmik, Kommunikation und Kryptographie. Wiesbaden: Springer Vieweg 2014.

IETF. Augmented BNF for Syntax Specifications: ABNF. RFC Editor; (5234): Request for Comments. https://www.rfc-editor.org/info/rfc5234.

ISO/IEC. Information technology - Syntactic metalanguage - Extended BNF 1996; (14977): 15 Dezember 1996 15. Dezember 1996. https://www.iso.org/standard/26153.html (6 August 2023).

ISO/IEC 9899:2017 C17 ballot. Information technology - Programming languages - C (working document); (9899:2017): https://teaching.csse.uwa.edu.au/units/CITS2002/resources/n2176.pdf (20 September 2023).

Louden KC. Programmiersprachen. Grundlagen, Konzepte, Entwurf. Bonn: Internat. Thomson Publ 1994.

Parr T. The definitive ANTLR 4 reference. Dallas, Tex.: Pragmatic Bookshelf 2012.

Parr T, Harwell S, Fisher K. Adaptive LL(*) Parsing: The Power of Dynamic Analysis.

Socher R. Theoretische Grundlagen der Informatik. Mit 31 Tabellen, 36 Beispielen und 75 Aufgaben mit Lösungen. München: Hanser 2008.

Wagenknecht C, Hielscher M. Formale Sprachen, abstrakte Automaten und Compiler. Lehr- und Arbeitsbuch mit FLACI für Grundstudium und Fortbildung. Wiesbaden, Heidelberg: Springer Vieweg 2022.

Wilhelm R, Seidl H, Hack S. Übersetzerbau. Berlin, Heidelberg: Springer Vieweg 2012.

Wirth N. Grundlagen und Techniken des Compilerbaus. München: Oldenbourg 2008.

Wolf J, Krooß R. Grundkurs C. Bonn: Rheinwerk Verlag 2020.

Anhang A

ANTLR LimitC Grammatik Teil 1

```
grammar LimitC;

prog: gdecl* EOF;

gdecl: (funcDef | varDef | label);

funcDef:  type ID '(' paramListDef? ')' (codeBlock | ';');

paramListDef: paramDef (',' paramDef)*;
paramDef: (type ID?);

varDef: type varAssignDef (',' varAssignDef)* ';';

varAssignDef: ID ('=' expr)?;

funcCall: ID '(' paramList? ')';
paramList: param (',' param)*;
param: expr;

expr: constant                          # constantExpression
    | lvalue                            # lValExpression
    | funcCall                          # funcCallExpression
    | '(' expr ')'                      # parenthesesExpression
    | ADDOP expr                        # unaryPlusExpression
    | SUBOP expr                        # unaryNegationExpression
    | '(' type ')' expr                 # castExpression
    | expr (AST | DIVOP) expr           # mulDivExpression
    | expr (ADDOP | SUBOP) expr         # addSubExpression
    | assOp                             # additionalAssignmentExpression
    ;

lvalue: ID                                  # varExpression
    | INCR lvalue                           # preIncrementExpression
    | DECR lvalue                           # preDecrementExpression
    | '(' lvalue ')'                        # lparExpression
    | lvalue INCR                           # postIncrementExpression
    | lvalue DECR                           # postDecrementExpression
    | AMP lvalue                            # ampExpression
    | AST lvalue                            # astExpression
    ;

constant: INTEGER               # integerConstant
    | DOUBLE                    # doubleConstant
    | CHAR                      # charConstant
    | STR                       # stringConstant
    ;

codeBlock: '{' codeStateList* '}';

codeStateList: codeBlock | termExpr | label;

termExpr: varDef                    # varDefExpression
    | expr ';'                  # looseExpression
    | RETSTAT expr ';'          # returnExpression
    ;

assOp: lvalue '=' expr            # varAssignment
    | lvalue '+=' expr          # addAssignment
    | lvalue '-=' expr          # subAssignment
    | lvalue '*=' expr          # multAssignment
    | lvalue '/=' expr          # divAssignment
    ;

type: typeLit AST*;

typeLit : 'void'
    |   'char'
    |   'short'
    |   'int'
    |   'long'
    |   'float'
    |   'double'
    ;

label: LABEL;
```

ANTLR LimitC Grammatik Teil 2

```
76
77  STR: '"' .*? '"';
78  CHAR: '\'' .*? '\'';
79  DOUBLE: NUM* '.' NUM+;
80  INTEGER: NUM+;
81
82  AST: '*';
83  AMP: '&';
84
85  INCR: '++';
86  DECR: '--';
87  DIVOP: '/';
88  ADDOP: '+';
89  SUBOP: '-';
90
91  RETSTAT: 'return';
92
93  ID:   LETTER (LETTER | NUM)*;
94
95  LETTER: [a-zA-Z_];
96
97  NUM: [0-9];
98
99  COMSTART: '/*';
100 COMEND: '*/';
101
102 WS: [ \t] -> skip;
103
104 LINEBREAK: ('\r' '\n'? | '\n') -> skip;
105
106 LABEL: COMSTART WS* LABELLIT WS* INTEGER WS* COMEND;
107 LABELLIT: [Ll] [Aa] [Bb] [Ee] [Ll];
108
109 MULTILINECOMMENT: COMSTART .*? COMEND -> skip;
110
111 COMMENT: '//' .*? LINEBREAK -> skip;
112
113 DIR: '#' .*? LINEBREAK -> skip;
```