Bachelorarbeit

Implementierung eines Compilers für die WHILE-Sprache in Haskell

Jonas Cleve
jonas.cleve@fu-berlin.de

17. November 2014

Betreuung und Erstgutachten:
Prof. Dr. Elfriede Fehr

Zweitgutachten:
Prof. Dr. Marcel Kyas

Freie Universität Berlin
Institut für Informatik

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, 17. November 2014

____
Jonas Cleve

Zusammenfassung

Für das Studium des Übersetzerbaus ist es hilfreich, die verschiedenen Phasen der Übersetzung anhand eines konkreten Beispiels nachzuvollziehen. Um hierbei die verwendeten Techniken in den Vordergrund zu stellen, ist es sinnvoll, sich bei der Wahl der zu übersetzenden Sprache auf einen kleinen Sprachumfang zu beschränken. Im Rahmen dieser Bachelorarbeit wurde daher ein Compiler entwickelt, der Programme von der einfachen WHILE-Sprache in Assemblercode für den NASM-Assembler übersetzt. Als Implementierungssprache wurde Haskell verwendet.

Ein besonderes Augenmerk wurde auf einen modularen Aufbau des Compilers gelegt. Klare Schnittstellen spiegeln die verschiedenen Phasen des Übersetzungsprozesses wider und machen sie austauschbar. Zusätzlich kann das Ergebnis jeder Übersetzungsphase durch eine geeignete Visualisierung dargestellt werden, um die jeweiligen Transformationsprozesse der Übersetzungsphasen nachzuvollziehen.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Aufbau der Arbeit . . . 2
- 1.3 Struktur der Implementierung . . . 3

2 Grundlagen **5**
- 2.1 FIRST- und FOLLOW-Mengen . . . 5
- 2.2 Syntaxgerichtete Übersetzungen . . . 6

3 Die WHILE-Sprache **7**
- 3.1 Übersicht . . . 7
- 3.2 Lexikalischer Aufbau . . . 8
- 3.3 Syntaktischer Aufbau . . . 8
  - 3.3.1 Vereinfachte Grammatik . . . 8
  - 3.3.2 Eindeutige Grammatik . . . 9
- 3.4 Semantik . . . 11

4 Lexikalische Analyse **13**
- 4.1 Einführung . . . 13
- 4.2 Implementierung . . . 14

5 Syntaxanalyse **17**
- 5.1 Bottom-Up-Syntaxanalyse . . . 18
  - 5.1.1 Shift-Reduce-Parser . . . 18
  - 5.1.2 LR(0)-Elemente, CLOSURE und GOTO . . . 20
  - 5.1.3 Der LR(0)-Automat . . . 21
  - 5.1.4 (S)LR-Parser . . . 23
  - 5.1.5 Implementierung eines Parsergenerators . . . 25
  - 5.1.6 Der generierte Parser . . . 28

5.2 Funktionale Top-Down-Syntaxanalyse 30
5.2.1 Funktionale Parser und Kombinatoren 31
5.2.2 Elementare Parser 34
5.2.3 Auflösen der Linksrekursion 35
5.3 Optimierung des abstrakten Syntaxbaums 36
5.4 Schlussbemerkung 37

6 Zwischencodeerzeugung **39**
6.1 Eigenschaften des Zwischencodes 39
6.2 Erzeugung des Zwischencodes 40
6.2.1 Umwandlung arithmetischer Ausdrücke 40
6.2.2 Bedingte Anweisungen und Sprünge 41
6.3 Implementierung 44

7 Zwischencodeoptimierung **47**
7.1 Lineare und Guckloch-Optimierung 47
7.1.1 Überflüssige Sprünge 48
7.1.2 Überflüssige Sprungmarken 48
7.1.3 Implementierung 49
7.2 Flussgraph und Datenflussanalyse 49
7.2.1 Lebendigkeitsanalyse 50
7.2.2 Implementierung 51

8 Codeerzeugung **53**
8.1 Registerzuteilung 53
8.1.1 „Linear scan“-Registerzuteilung 54
8.1.2 Verfügbare Register und Ausnahmen 55
8.2 Codeerzeugung 55
8.3 Assembler-Framework 56
8.4 Erzeugung eines ausführbaren Programms 56

9 Bewertung und Ausblick **57**
9.1 Mögliche Verbesserungen 58
9.2 Ausblick 58

A Anhang **61**
A.1 Quellcode 61
A.2 Der Compiler 61

A.3 Compilerausgaben für ein Beispielprogramm 62

B Literaturverzeichnis **69**

C Abbildungsverzeichnis **73**

D Tabellenverzeichnis **75**

E Quelltextverzeichnis **77**

1 Einleitung

1.1 Motivation

Der Übersetzerbau beschäftigt sich mit dem Entwurf und der Entwicklung von Compilern, mit denen Quelltexte von höheren Programmiersprachen in eine (meist niedrigere) Zielsprache umgewandelt werden.

Für das Studium des Übersetzerbaus und das tiefere Verständnis des Themenkomplexes ist es hilfreich, die verschiedenen Phasen der Übersetzung anhand eines konkreten Beispiels nachzuvollziehen. Mit dieser Motivation wurde in dieser Arbeit ein Compiler implementiert, der die gängigen Phasen des Übersetzungsprozesses umfasst. Als Quellsprache wurde mit der WHILE-Sprache eine kleine imperative Sprache gewählt, die die wichtigsten Funktionalitäten zur Programmierung unterstützt. Die Zielsprache ist mit NASM-Assemblercode eine Assemblersprache für aktuelle x86- beziehungsweise AMD64-Architekturen. Dadurch können die vom Compiler erzeugten Programme auf aktuellen Computern ausgeführt und getestet werden.

Die funktionale Programmiersprache Haskell wurde als Implementierungssprache gewählt, um festzustellen, inwiefern funktionale Aspekte bei der Entwicklung hilfreich oder hinderlich sein können.

Als Quellsprache bietet sich die WHILE-Sprache an, da sie nur einen kleinen Sprachumfang besitzt. Dadurch treten bei der Implementierung die verwendeten Techniken in der Vordergrund und es wird eine Fixierung auf besondere Spracheigenschaften der Quellsprache vermieden.

Ein besonderes Ziel der Arbeit war es, die Implementierung möglichst modular zu gestalten und dadurch Wiederverwendbarkeit und Austauschbarkeit der einzelnen Bestandteile zu gewährleisten. Außerdem soll dadurch eine leichte Erweiterbarkeit gegeben werden, sodass mögliche Erweiterungen des Compilers einfach eingebaut werden können.

1.2 Aufbau der Arbeit

Die grundlegende Struktur eines Compilers folgt im Allgemeinen den nötigen Schritten zur Transformation des Quelltextes in die entsprechende Zielsprache. Eine Übersicht über die Phasen des in dieser Arbeit entwickelten Compilers findet sich in Abbildung 1.1. Der inhaltliche Aufbau der Arbeit folgt ebenfalls dieser Struktur.

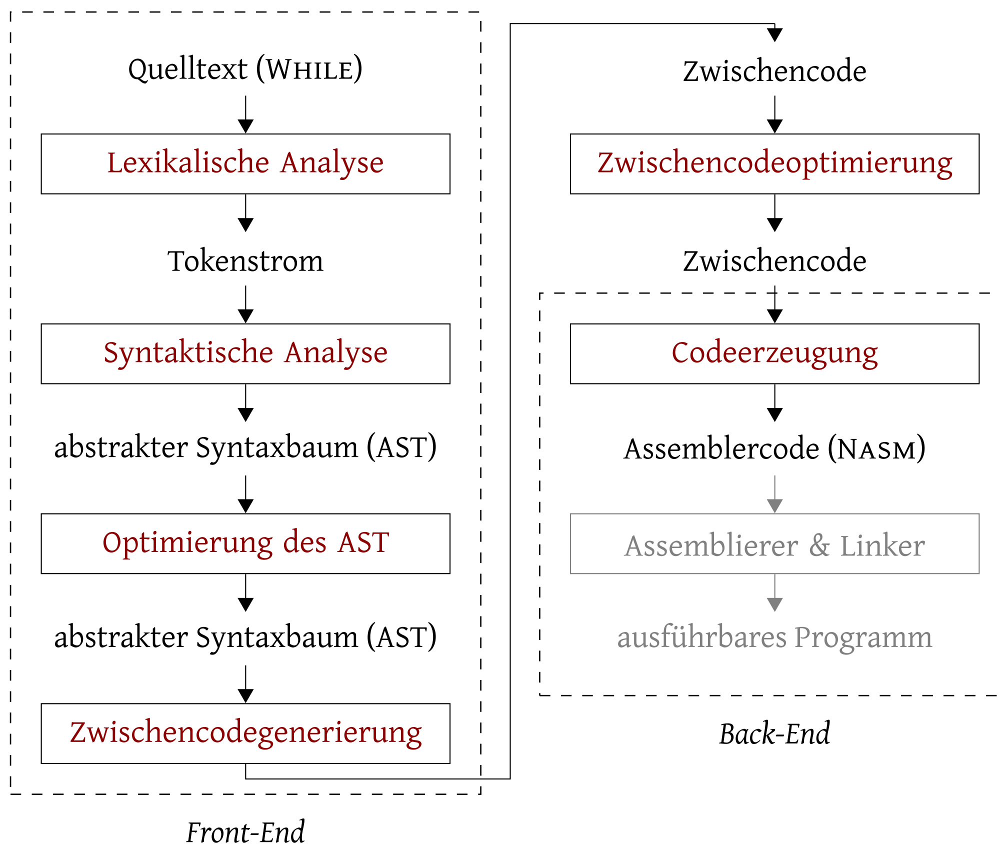


Abbildung 1.1: Übersicht über die Phasen des Compilers – die Phase des Assemblierens und Linkens ist nicht Teil dieser Arbeit

In Kapitel 2 werden verschiedene Grundlagen erörtert, die für den weiteren Teil der Arbeit benötigt werden. Kapitel 3 beschreibt die Quellsprache WHILE und beleuchtet den lexikalischen und syntaktischen Aufbau. Der ersten Phase des Compilers widmet sich Kapitel 4, in welchem die lexikalische Analyse betrachtet wird. Das folgende Kapitel 5 beschreibt zwei verschiedene Möglichkeiten zur syntaktischen Analyse der Eingabeprogramme und geht am Ende kurz auf die implementierten Optimierungen des erzeugten Syntaxbaums ein. Der letzte Teil des Compiler-Front-Ends, die Erzeugung von Code in einer Zwischensprache, wird in Kapitel 6 beschrieben.

Die wichtige Phase der Zwischencodeoptimierung wird in Kapitel 7 dargestellt, wobei verschiedene Möglichkeiten zur Verbesserung des Zwischencodes beschrieben werden. Kapitel 8 widmet sich der Generierung von Assemblercode aus dem Zwischencode.

In Kapitel 9 wird eine abschließende Bewertung vorgenommen und ein Ausblick auf mögliche Erweiterungen und weiterführende Themen gegeben. Anhang A enthält eine beispielhafte Übersicht über die vom Compiler erzeugten Zwischenprodukte während des Kompiliervorgangs.

1.3 Struktur der Implementierung

Die Modulstruktur der Haskell-Implementierung ist in Abbildung 1.2 als Abhängigkeitsgraph zu sehen. Der Übersichtlichkeit halber wurden allgemeine Hilfsmodule sowie die Module mit den Schnittstellendefinitionen ausgelassen. Grundsätzlich wurden die einzelnen Phasen des Compilers in eigene Module ausgelagert. Die von mehreren Phasen benötigten Datenstrukturen und Schnittstellendefinitionen wurden in Untermodule eines `Interface`-Moduls ausgelagert. Somit wird eine möglichst lose Kopplung der Module erreicht und Austauschbarkeit ermöglicht. Für die Syntaxanalyse kann beispielsweise zwischen den beiden in dieser Arbeit entwickelten Parsern gewählt werden, welche untereinander keinerlei Abhängigkeiten besitzen. Auch wäre es möglich, den vorhandenen Lexer durch einen anderen zu ersetzen oder durch minimale Anpassung weitere Optimierungen einzubinden.

Die meisten produktiv verwendeten Compiler führen während der Optimierungsphasen mehrere Optimierungen in einem Schritt aus, um die Laufzeit des Kompiliervorgangs zu reduzieren. Das Vermischen verschiedener Optimierungen führt jedoch dazu, dass die Nachvollziehbarkeit sinkt und die Komplexität steigt. Für den in dieser Arbeit entwickelten Compiler wurde daher ein anderer Ansatz gewählt: Jeder Optimierungsschritt wird einzeln ausgeführt und kann daher leichter nachvollzogen werden. Dies erhöht zwar die Laufzeit des Compilers, da das Hauptaugenmerk jedoch nicht auf der Geschwindigkeit liegt, kann dieser Nachteil zugunsten eines verbesserten Verständnisses und einer erhöhten Wartbarkeit in Kauf genommen werden.

Der Großteil der Optimierungen wird am Zwischencode durchgeführt, dies wird in Kapitel 7 näher erläutert. Es ist jedoch auch sinnvoll, einige Optimierungen am abstrakten Syntaxbaum zu vollziehen. Darauf geht Abschnitt 5.3 ein.

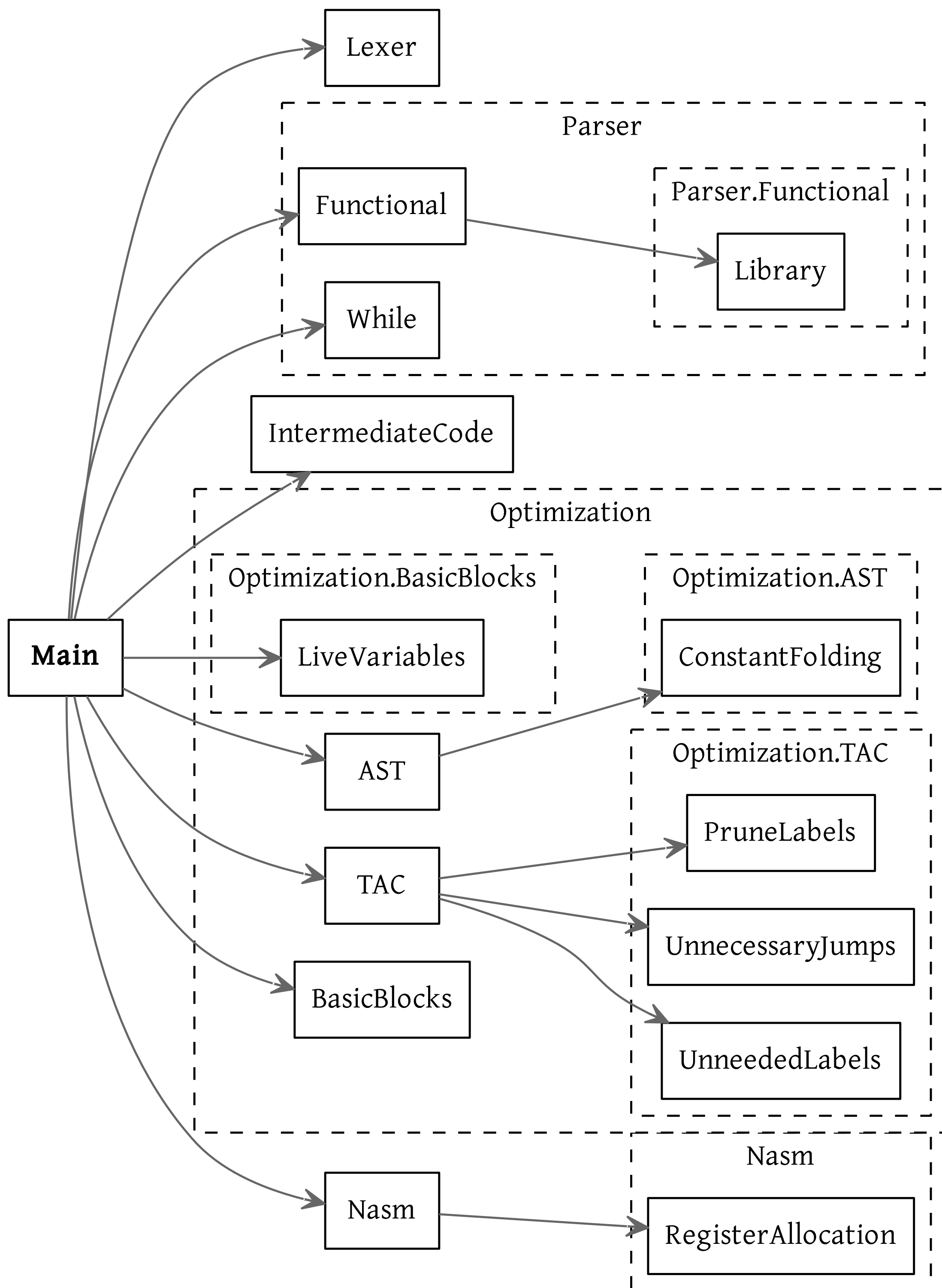


Abbildung 1.2: Struktur der Haskell-Module

2 Grundlagen

Der Inhalt der Arbeit setzt Grundkenntnisse im Bereich kontextfreier Grammatiken voraus. Einige, für den Übersetzerbau benötigte, spezifischere Konstrukte werden im Folgenden kurz erläutert. Für weitere Informationen zu den einzelnen Punkten seien die entsprechenden Kapitel in [2] als Lektüre empfohlen.

Des Weiteren werden Kenntnisse in der funktionalen Programmierung, im Speziellen in Haskell, vorausgesetzt. Kenntnisse über Funktionen höherer Ordnung, do-Notation und Monaden sind vor allem für Abschnitt 5.2, in dem die funktionalen Top-Down-Syntaxanalyse vorgestellt wird, wichtig und werden ebenfalls vorausgesetzt.

2.1 FIRST- und FOLLOW-Mengen

Eine besondere Bedeutung für die Syntaxanalyse und die Konstruktion von Parsern sind die FIRST- und FOLLOW-Mengen von Grammatiksymbolen.

Im Folgenden bezeichnet Σ die Menge der Terminalsymbole, N die Menge der Nichtterminalsymbole und $\stackrel{*}{\Rightarrow}$ eine Ableitung mit beliebig vielen Schritten. Des Weiteren werden Kleinbuchstaben verwendet, um Terminalsymbole, und Großbuchstaben, um Nichtterminalsymbole zu repräsentieren. Eine Ausnahme bilden die letzten Großbuchstaben des Alphabets (X, Y, Z), die sowohl Terminal- als auch Nichtterminalsymbole bezeichnen können.

Die FIRST-Menge eines Grammatiksymbols oder einer Folge von Grammatiksymbolen α besteht aus allen Terminalsymbolen, die bei beliebigen Ableitungen von α an erster Stelle auftreten können. Sollte α auch zum leeren Wort ε abgeleitet werden können, so ist auch dieses in der FIRST-Menge enthalten. Formal ausgedrückt bestimmt sich die FIRST-Menge als

$$\textsc{First}(\alpha) = \left\{ a \in \Sigma \,\middle|\, \alpha \stackrel{*}{\Rightarrow} a\beta \right\} \cup \left\{ \varepsilon \,\middle|\, \alpha \stackrel{*}{\Rightarrow} \varepsilon \right\} \text{ mit } \alpha, \beta \in (\Sigma \cup N)^* .$$

Die FOLLOW-Menge eines Nichtterminalsymbols A hingegen bezeichnet alle Terminalsymbole, die bei beliebigen Ableitungen direkt hinter A auftreten können. Tritt A in

einer Ableitung am Ende des Wortes auf, so ist auch das Sonderzeichen \$, welches das Ende der Eingabe markiert, Teil der FOLLOW-Menge. Die Menge ist formal gegeben als

$$\textsc{Follow}(A) = \left\{ a \in \Sigma \,\middle|\, S \overset{*}{\Rightarrow} \alpha A a \beta \right\} \cup \left\{ \$ \,\middle|\, S \overset{*}{\Rightarrow} \alpha A \right\} \text{ mit } A \in N$$
$$\text{und } \alpha, \beta \in (\Sigma \cup N)^* .$$

Wichtig werden die FOLLOW-Mengen in Abschnitt 5.1 beim Behandeln der Bottom-Up-Syntaxanalyse.

2.2 Syntaxgerichtete Übersetzungen

Im Allgemeinen besteht die Aufgabe der Übersetzung darin, die durch die Syntaxanalyse konstruierten Syntaxbäume mit Informationen zu versehen, diese Informationen im Baum weiterzugeben und zu verwalten. Bei der *syntaxgerichteten Übersetzung* werden diese Informationen und Informationsflüsse zur Definition der Grammatik hinzugefügt. Dabei ergeben sich zwei verschiedene Ausprägungen von syntaxgerichteten Übersetzungen.

Die *syntaxgerichtete Definition* (SDD, englisch für syntax directed definition) definiert zusätzlich zur Grammatik eine Tabelle, in der jeder Produktion *semantische Regeln* zugeordnet werden. Diese Regeln weisen den Nichtterminalsymbolen innerhalb der Produktion Attribute zu, welche sich aus den Terminalsymbolen oder Attributen anderer Nichtterminalsymbole ergeben. Tabelle 6.1 auf Seite 42 zeigt eine in der Zwischencodeerzeugung verwendete SDD. Eine Besonderheit der SDD ist, dass die Auswertungsreihenfolge der semantischen Regeln nicht festgelegt ist, sondern sich aus den Abhängigkeiten der Berechnungen untereinander ergibt.

Im Gegensatz dazu werden bei *Übersetzungsschemata* (SDT, englisch für syntax directed translation scheme) Programmfragmente in die Produktionen eingestreut, wie in Quelltext 6.2 auf Seite 41 zu sehen. Im Allgemeinen werden diese Programmfragmente mit in den Syntaxbaum eingefügt und dann im Rahmen einer Tiefensuche nacheinander ausgeführt.

3 Die WHILE-Sprache

Im Folgenden wird die WHILE-Sprache beschrieben, welche die Quellsprache des entwickelten Compilers ist. Nach einem allgemeinen Überblick wird der lexikalische sowie syntaktische Aufbau der Sprache näher beleuchtet. Zum Schluss wird kurz auf die Semantik der Anweisungen in der WHILE-Sprache eingegangen.

3.1 Übersicht

Die Quellsprache des Compilers ist eine leicht modifizierte Form der WHILE-Sprache, einer kleinen, imperativen Sprache, die vor allem beim Studium der Semantik von Programmiersprachen verwendet wird [7, 21].

Die WHILE-Sprache unterstützt ganzzahlige Variablen und Konstanten, mit denen Rechenoperationen (Addition, Subtraktion, Multiplikation, ganzzahlige Division und Modulo-Rechnung) durchgeführt werden können. Die unterstützen Anweisungen umfassen neben der Wertzuweisung an eine Variable die wichtigsten Sprachkonstrukte zur bedingten Ausführung: While-Schleifen und If-Abfragen.

Mit den Sprachkonstrukten `read`, `output` und `eof` verfügt die WHILE-Sprache über die Möglichkeit, mit dem Benutzer oder anderen Programmen über die Standardein- und -ausgabe zu kommunizieren. Mittels `read` x wird die nächste Zahl aus der Standardeingabe gelesen und der Variablen x zugewiesen, während `output` y den Zahlenwert von y in die Standardausgabe schreibt.

Analog zur Einschränkung der Variablen können auch nur ganze Zahlen eingelesen und ausgegeben werden. Bei der Ausgabe wird jede Zahl durch einen Zeilenumbruch (`\n`) getrennt. In der Eingabe müssen mehrere Zahlen ebenfalls durch Zeilenumbrüche getrennt werden. Erlaubt sind hierbei ein optionales Minus zu Beginn einer Zeile, sowie eine beliebige Folge der Ziffern 0 bis 9. Andere Zeichen führen zu einem Fehler und zum Abbruch des Programms.

Das Beispielprogramm in Quelltext 3.1 zeigt einen Großteil des Sprachumfangs. In einer Schleife werden hier einzelne Zahlen aus der Standardeingabe gelesen und aufsummiert. Zum Schluss gibt das Programm die berechnete Summe aus.

```
1 # Sums all numbers from input and outputs the result
2 sum := 0;
3 while not eof do {
4     read x;
5     sum := sum + x
6 };
7 output sum
```

Quelltext 3.1: Einfaches WHILE-Programm zur Berechnung der Summe aller eingelesenen Zahlen

3.2 Lexikalischer Aufbau

Durch die einfache Syntax der WHILE-Sprache besteht der lexikalische Aufbau hauptsächlich darin, Zahlen, Bezeichner, Schlüsselwörter und zusammengesetzte Operatoren (beispielsweise <= oder :=) zu erkennen und einer lexikalischen Einheit (einem sogenannten Token) zuzuordnen. Hinzu kommt, dass Leerzeichen, Zeilenumbrüche und Kommentare vom Lexer ignoriert werden.

Eine Übersicht der lexikalischen Einheiten findet sich in Tabelle 3.1. Die ersten beiden Zeilen dienen dabei nur der einfacheren Definition und beschreiben keine lexikalischen Einheiten. Wie der Definition von *number* zu entnehmen ist, wird eine beliebige Folge von Ziffern als (positive) Zahl erkannt, führende Nullen sind dabei zulässig. Variablen, für die das Token *identifier* steht, müssen mit einem Buchstaben beginnen und können danach beliebig viele Buchstaben und Zahlen beinhalten. Die verschiedenen Vergleichsoperatoren, sowie mathematisch und logische Operatoren werden jeweils zu Gruppen zusammengefasst. Den verschiedenen reservierten Schlüsselwörtern wie `while` oder `if` wird jeweils ein eigenes Token zugewiesen. Das Token *token* umfasst schließlich alle Token, die nur aus einem einzelnen Zeichen bestehen und in der eindeutigen Grammatik für die Gliederung des Programms benötigt werden.

3.3 Syntaktischer Aufbau

3.3.1 Vereinfachte Grammatik

Eine vereinfachte Grammatik zur Beschreibung der WHILE-Sprache ist in Tabelle 3.2 zu finden. Wie schon angesprochen, umfassen die Anweisungen *cmd* die Konstrukte zur bedingten Ausführung von Programmcode, wobei die If-Anweisung mit und ohne Else-Zweig vorhanden ist. Zusätzlich können die Konstrukte zur Ein- und Ausgabe verwendet

$$
\begin{aligned}
\mathit{digit} &:= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
\mathit{char} &:= \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \texttt{B} \mid \ldots \mid \texttt{Z} \\
\mathit{number} &:= \mathit{digit}^{+} \\
\mathit{identifier} &:= \mathit{char}\ (\mathit{digit} \mid \mathit{char})^{*} \\
\mathit{bool} &:= \texttt{true} \mid \texttt{false} \\
\mathit{relop} &:= \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{!=} \\
\mathit{logop} &:= \texttt{and} \mid \texttt{or} \\
\mathit{mathop} &:= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{mod} \\
\mathit{assign} &:= \texttt{:=} \\
\mathit{token} &:= \texttt{(} \mid \texttt{)} \mid \texttt{\{} \mid \texttt{\}} \mid \texttt{;}
\end{aligned}
$$

Für alle Schlüsselwörter *keyword* existiert zusätzlich eine Regel

$$\mathit{keyword} := \texttt{keyword}$$

Tabelle 3.1: Lexikalischer Aufbau der WHILE-Sprache

sowie mathematische Ausdrücke Variablen zugewiesen werden. Mehrere Anweisungen werden durch Semikola voneinander getrennt.

Mathematische Ausdrücke *expr* können mit den angesprochenen Operatoren verknüpft sowie negiert werden und bestehen atomar entweder aus Zahlenkonstanten oder Variablen. Ähnliches gilt für boolesche Ausdrücke *bexpr*, wobei diese mittels `and`, `or` und `not` verknüpft werden können und atomar aus den booleschen Konstanten `true` und `false` sowie dem Prädikat `eof` bestehen. Zusätzlich können für einen booleschen Ausdruck auch zwei mathematische Ausdrücke verglichen werden.

$$
\begin{aligned}
\mathit{program} &\rightarrow \mathit{cmd} \\
\mathit{cmd} &\rightarrow \mathit{cmd}\ \texttt{;}\ \mathit{cmd} \\
&\mid \texttt{if}\ \mathit{bexpr}\ \texttt{then}\ \mathit{cmd} \\
&\mid \texttt{if}\ \mathit{bexpr}\ \texttt{then}\ \mathit{cmd}\ \texttt{else}\ \mathit{cmd} \\
&\mid \texttt{while}\ \mathit{bexpr}\ \texttt{do}\ \mathit{cmd} \\
&\mid \texttt{read}\ \texttt{identifier} \\
&\mid \texttt{output}\ \mathit{expr} \\
&\mid \texttt{identifier}\ \texttt{:=}\ \mathit{expr}
\end{aligned}
\qquad
\begin{aligned}
\mathit{expr} &\rightarrow \mathit{expr}\ \texttt{mathop}\ \mathit{expr} \\
&\mid \texttt{-}\ \mathit{expr} \\
&\mid \texttt{identifier} \\
&\mid \texttt{number} \\
\mathit{bexpr} &\rightarrow \mathit{bexpr}\ \texttt{logop}\ \mathit{bexpr} \\
&\mid \texttt{not}\ \mathit{bexpr} \\
&\mid \mathit{expr}\ \texttt{relop}\ \mathit{expr} \\
&\mid \texttt{bool} \\
&\mid \texttt{eof}
\end{aligned}
$$

Tabelle 3.2: Simple Grammatik der WHILE-Sprache

3.3.2 Eindeutige Grammatik

Die vorgestellte Grammatik weist jedoch einige Mehrdeutigkeiten auf, weshalb sie nicht direkt geeignet ist, ein WHILE-Programm zu erkennen. Generell kann beispielsweise bei

einem Ausdruck wie `if a then b; c` nicht entschieden werden, ob nur `b` oder auch `c` bedingt ausgeführt werden sollen. Ein doppeltes Problem besteht beim Versuch, mathematische Ausdrücke mit dieser Grammatik zu erkennen. Sowohl die Operatorrangfolge als auch die Assoziativität der mathematischen Operationen wird durch diese Grammatik nicht berücksichtigt. Der Ausdruck `5+3*4` kann sowohl als `(5+3)*4` als auch als `5+(3*4)` erkannt werden und `3-1-2` als `(3-1)-2` oder `3-(1-2)`.

Eine eindeutige Grammatik, die diese Schwachpunkte behebt und Grundlage für die in Kapitel 5 beschriebenen Parser ist, findet sich in Tabelle 3.3. Im Folgenden werden die Unterschiede genauer betrachtet.

$$
\begin{aligned}
\textit{program} &\rightarrow \textit{cmds} \\
\textit{cmds} &\rightarrow \textit{cmds}\ ;\ \textit{cmd} \\
&\mid \textit{cmd} \\
\textit{cmd} &\rightarrow \textit{cmdo} \\
&\mid \textit{cmdc} \\
&\mid \varepsilon \\
\textit{cmdo} &\rightarrow \texttt{if}\ \textit{bexpr}\ \texttt{then}\ \textit{cmd} \\
&\mid \texttt{if}\ \textit{bexpr}\ \texttt{then}\ \textit{cmdc}\ \texttt{else}\ \textit{cmdo} \\
&\mid \texttt{while}\ \textit{bexpr}\ \texttt{do}\ \textit{cmdo} \\
\textit{cmdc} &\rightarrow \texttt{if}\ \textit{bexpr}\ \texttt{then}\ \textit{cmdc}\ \texttt{else}\ \textit{cmdc} \\
&\mid \texttt{while}\ \textit{bexpr}\ \texttt{do}\ \textit{cmdc} \\
&\mid \texttt{read}\ \texttt{identifier} \\
&\mid \texttt{output}\ \textit{expr} \\
&\mid \texttt{identifier}\ :=\ \textit{expr} \\
&\mid \{\ \textit{cmds}\ \} \\
\textit{expr} &\rightarrow \textit{expr}\ +\ \textit{term} \\
&\mid \textit{expr}\ -\ \textit{term} \\
&\mid \textit{term}
\end{aligned}
\qquad
\begin{aligned}
\textit{term} &\rightarrow \textit{term}\ *\ \textit{factor} \\
&\mid \textit{term}\ /\ \textit{factor} \\
&\mid \textit{term}\ \texttt{mod}\ \textit{factor} \\
&\mid \textit{factor} \\
\textit{factor} &\rightarrow (\ \textit{expr}\ ) \\
&\mid -\ \textit{factor} \\
&\mid \texttt{identifier} \\
&\mid \texttt{number} \\
\textit{bexpr} &\rightarrow \textit{bexpr}\ \texttt{or}\ \textit{bterm} \\
&\mid \textit{bterm} \\
\textit{bterm} &\rightarrow \textit{bterm}\ \texttt{and}\ \textit{bfactor} \\
&\mid \textit{bfactor} \\
\textit{bfactor} &\rightarrow (\ \textit{bexpr}\ ) \\
&\mid \texttt{not}\ \textit{bfactor} \\
&\mid \textit{expr}\ \texttt{relop}\ \textit{expr} \\
&\mid \texttt{bool} \\
&\mid \texttt{eof}
\end{aligned}
$$

Tabelle 3.3: Eindeutige Grammatik für die WHILE-Sprache

Damit mathematische und logische Ausdrücke eindeutig erkannt werden können, wird durch zusätzliche beziehungsweise geänderte Produktionen eine Operatorrangfolge inklusive Festlegung der Linksassoziativität für mathematische und boolesche Ausdrücke eingeführt. Grundlagen dazu sind in [2, S. 48–50] zu finden.

Um das Problem der bedingten Ausführung von mehreren Anweisungen zu beheben, wird *cmd* auf einzelne Anweisungen beschränkt und ein neues Nichtterminalsymbol *cmds* eingeführt, um mehrere Anweisungen zu erlauben. Durch geschweifte Klammern können mehrere Anweisungen zu einer Anweisung zusammengefasst werden. Zusätzlich wird die Mehrdeutigkeit der Produktion $\textit{cmd} \rightarrow \textit{cmd}\ ;\ \textit{cmd}$ durch Linksrekursion eliminiert.

Eine weitere Mehrdeutigkeit, die beseitigt werden muss, ist das „dangling else"-Problem. Es besteht darin, dass bei der Eingabe `if a then if b then c else d` der `else`-Teil entweder zum ersten oder zum zweiten `if` gehören kann. Dieses Problem wurde schon in den 60er Jahren erkannt und gelöst. Die von Abrahams [1] vorgeschlagene Lösung besteht darin, zwischen *geschlossenen* und *offenen* Anweisungen zu unterscheiden. Dabei bedeutet *geschlossen*, dass alle `if`-Konstrukte auch einen `else`-Zweig besitzen. Ihnen kann also nicht fälschlicherweise noch ein äußerer `else`-Zweig zugeordnet werden. Wichtig ist hierbei, dass nicht nur für `if`-, sondern auch für `while`-Anweisungen sowohl der geschlossene als auch der offene Fall betrachtet werden muss.

3.4 Semantik

Die Semantik der Ausdrücke ergibt sich in den meisten Fällen aus den Schlüsselwörtern, die in der Mehrzahl der Programmiersprachen mit gleicher Semantik verwendet werden. Der Befehl `read identifier` liest die nächste Zahl aus der Eingabe und weist sie der Variablen, für die das Token `identifier` erzeugt wurde, zu. Sollte zu dem Zeitpunkt noch keine Zahl in der Eingabe stehen, wird gewartet.

Die Anweisung `output` *expr* berechnet den numerischen Wert des Ausdrucks *expr* und gibt das Ergebnis auf der Ausgabe aus. Das Prädikat `eof` wird zu `true` ausgewertet, wenn in der Eingabe keine weiteren Zahlen folgen, sonst zu `false`.

Zur Semantik arithmetischer Ausdrücke und zum Zahlenbereich sind noch Besonderheiten zu benennen. Die Division ist eine ganzzahlige Division, die das Ergebnis zur Null rundet. Die Modulo-Operation erfüllt die Gleichung $(x/y) \cdot x + (x \bmod y) = x$. Das bedeutet, dass der per Modulo ermittelte Rest immer das gleiche Vorzeichen wie der Dividend besitzt.

Das vom Compiler erzeugte Programm verwendet 64 Bit breite, vorzeichenbehaftete Ganzzahlen, sodass der darstellbare Zahlenraum die Menge $\{-2^{63}, \cdots, 0, \cdots, 2^{63} - 1\}$ umfasst. Sollten die Ergebnisse einer Operation den darstellbaren Zahlenraum verlassen, ist das Verhalten nicht definiert.

4 Lexikalische Analyse

Diese Kapitel beschreibt zuerst die allgemeine Idee der lexikalischen Analyse und geht dann genauer darauf ein, wie diese Phase im Compiler implementiert wurde.

4.1 Einführung

Die Aufgabe der lexikalischen Analyse ist, den Quelltext in logisch zusammenhängende Teile (sogenannte *Token*) zu zerteilen. Dadurch vereinfacht sie die Arbeit der nachfolgenden syntaktischen Analyse. Da beispielsweise Kommentare und Leerzeichen für das Programm keinerlei Bedeutung haben, können Sie in der lexikalischen Analyse ignoriert werden. Nur logische Einheiten mit einer semantischen Bedeutung werden vom Lexer erzeugt.

Wie in Abschnitt 3.2 beschrieben, besteht die WHILE-Sprache aus den lexikalischen Einheiten für Ganzzahlen, Schlüsselwörter, Bezeichner (Variablen), boolesche Konstanten, sowie mathematische, logische und Vergleichs-Operatoren. Auch der Zuweisungsoperator := wird als eigenes Token behandelt. In Quelltext 4.1 kann man beispielhaft sehen, in welche lexikalischen Einheiten ein WHILE-Programm zerlegt wird und dass dabei Kommentare, Leerzeichen und Zeilenumbrüche keine Rolle spielen.

```
if a=b then
# Kommentar
output b*3;
```

(a) Eingabeprogramm

```
<if> <id,"a"> <relop,=> <id,"b">
<then> <output> <id,"b"> <mathop,*>
<num,3> <;>
```

(b) Zugehörige Tokenzerlegung

Quelltext 4.1: Einfaches Eingabeprogramm und zugehörige Zerlegung in Token

Token sind in den meisten Fällen relativ einfach aufgebaut, sodass sie durch reguläre Ausdrücke beschrieben werden können. Um die Token zu erkennen, wird ein Lexer in imperativen Programmiersprachen oft über einen Zustandsautomaten implementiert [2]. Dieser gibt beim Aufruf einer Funktion wie `getNextToken` das nächste Token, welches durch Simulation des Zustandsautomaten auf dem Beginn der Eingabe gefunden wurde,

zurück und verwirft die konsumierte Eingabe. Dadurch muss nicht erst der komplette Quelltext in Token umgewandelt und die Liste der Token im Speicher gehalten werden. Der Quelltext wird dadurch immer nur so weit betrachtet, wie es der Parser für die Syntaxanalyse benötigt (vgl. [2, S. 76–85] sowie [12, S. 28–33]).

Dieses Prinzip, immer nur die wirklich benötigten Auswertungen vorzunehmen, wird auch *Bedarfsauswertung* oder „lazy evaluation“ genannt. Die Bedarfsauswertung ist ein Hauptmerkmal von Haskell [14]. Wird beispielsweise eine Liste durch eine Funktion schrittweise erzeugt, so wird die Funktion immer nur für die Listenelemente ausgewertet, auf die zugegriffen wird. Dies wird in der Implementierung der lexikalischen Analyse ausgenutzt.

4.2 Implementierung

Die vom Lexer zu erzeugenden Token lassen sich in zwei Gruppen aufteilen. Zum einen gibt es Token ohne Attribute, wie `<if>` oder `<eof>`. Zum anderen gibt es Token für Bezeichner, Zahlen, Vergleichsoperatoren, etc., die ein zusätzliches Attribut enthalten, beispielsweise `<id,"x">` oder `<num,3>`.

Diese Art von Daten kann in Haskell sinnvollerweise durch einen benutzerdefinierten Datentyp, wie in Quelltext 4.2 zu sehen, dargestellt werden. Die angesprochenen Token `Id`, `Number` und `Boolean` erhalten als Attribut den Wert, für den sie erzeugt wurden. Ebenso die logischen, mathematischen und Vergleichs-Operatoren, welche als Attribut wieder einen benutzerdefinierten Datentyp haben, mit dem die möglichen Werte genau festgelegt sind. So kann beispielsweise der Datentyp `LogOp` genau die beiden Werte `And` und `Or` annehmen. Anhand der Operatoren ist auch erkennbar, dass Token mit ähnlicher Bedeutung zusammengefasst werden.

Das Lexer-Modul exportiert die Funktion `lex :: String -> [PosToken]`, die einen beliebigen String in eine Liste von Token umwandelt. Für den eigentlichen Vorgang der lexikalischen Analyse werden Unterfunktionen aufgerufen, die noch zusätzlich die aktuelle Position im Quelltext mitführen. Dadurch kann bei einem Fehler in der Eingabe die Position des fehlerhaften Zeichens ausgegeben werden. Zusätzlich wird den erzeugten Token ihre Position im Quelltext angeheftet, sodass auch bei einem syntaktischen Fehler, der erst in der Syntaxanalyse erkannt wird, leicht der Ort des Fehlers lokalisiert werden kann.

Quelltext 4.3 zeigt einen Ausschnitt aus dem Lexer, in welchem die Zeichenfolge `:=` als Token `Assign` erkannt wird. Das Beispiel ist insofern vereinfacht, als dass das mitführen der Position der Übersichtlichkeit halber weggelassen wurde. Die Erkennung der Token

```
Interface/Token.hs
45 -- | A data type for the tokens generated during the lexing phase.
46 data Token
47   = Id String      -- ^ All possible variable names (@stuff@, @foo@, @bar@, ...)
48   | Number Int64   -- ^ Integer number (@34@, @132@, @894@, ...)
49   | Boolean Bool   -- ^ Boolean values (@true@ and @false@)
50   | LogOp LogOp    -- ^ Logical operators (@and@ and @or@)
51   | Not            -- ^ Boolean negation (@not@)
52   | RelOp RelOp    -- ^ Relational operators (@=@, @<@, @<=@, @>@, @>=@ and @!=@)
53   | MathOp MathOp  -- ^ Arithmetic operators (@mod@, @+@, @-@, @*@, @/@)
54   | Assign         -- ^ The assignment operator (@:=@)
55   | Eof            -- ^ The eof predicate (@eof@)
56   | Read           -- ^ The read token (@read@)
57   | Output         -- ^ The output token (@output@)
58   | If             -- ^ If
59   | Then           -- ^ Then
60   | Else           -- ^ Else
61   | While          -- ^ While
62   | Do             -- ^ Do
63   | Token Char     -- ^ Single char tokens - used for parentheses, braces, ...
```

Quelltext 4.2: Definition des Datentyps für die generierten Token

ist dabei ein relativ simples Pattern Matching. Entsprechen die ersten zwei Zeichen des Quelltextes `:=`, dann besteht die Liste aller Token aus dem Token `Assign` und der Liste von Token, die entstehen, wenn der Rest des Quelltextes betrachtet wird.

An dieser Stelle ist sehr schön zu sehen, wie Haskells „lazy evaluation“ verwendet wird, um immer nur das nächste benötigte Token zu erzeugen. Wird beispielsweise `lex ":= a"` aufgerufen und das erste erzeugte Token benötigt, so wird es in den Ausdruck `Assign : lex " a"` umgewandelt. Erst wenn `Assign` aus der erzeugten Liste entfernt und das nächste Element benötigt wird, wird `lex` auf der restlichen Eingabe erneut ausgewertet.

Für obiges Beispiel müssen die nächsten zwei Zeichen der Eingabe angeschaut werden, um zu entscheiden, ob das Pattern Matching erfolgreich ist oder nicht. Da für die WHILE-Sprache keine Token existieren, die länger als zwei Zeichen sind, deren Präfix aber selber kein Token sein kann, müssen nie mehr als zwei Zeichen im Voraus gelesen werden. Dadurch ergibt sich ein, hier impliziter, Lookahead des Lexers von zwei Zeichen.

Da Schlüsselwörter wie `output` oder `while` Sonderfälle eines Bezeichners sind, werden diese zusammen erkannt. Der vereinfachte Ausschnitt aus dem Lexer ist in Quelltext 4.4

```
-- Assignment: create token "Assign" when the first two characters are ":="
lex (':':'=':rest) = Assign : lex rest
```

Quelltext 4.3: Erkennung des Zuweisungsoperators im Lexer

zu sehen. Zuerst werden alle zusammenhängenden Zeichenketten erkannt, die mit einem Groß- oder Kleinbuchstaben beginnen und danach beliebig viele Buchstaben oder Zahlen beinhalten. Schließlich wird geprüft, ob der erkannte Bezeichner einem der reservierten Schlüsselwörter entspricht. Ist dies der Fall, wird das entsprechende Token zurückgegeben. Ansonsten wird ein Bezeichner-Token zurückgegeben, das die erkannte Zeichenkette als Attribut beinhaltet.

```
-- Identifiers and keywords
lex (cur:source)
  | isAlpha cur = token : lex source'
    where
      -- Take all following alphanumerical characters from the input into @name@
      -- and put the rest of the source into @source'@
      (name, source') = span isAlphaNumeric source
      token = case cur:name of
        "true"   -> Boolean True
        "false"  -> Boolean False
        "not"    -> Not
        "and"    -> LogOp And
        "or"     -> LogOp Or
        "mod"    -> MathOp Mod
        "eof"    -> Eof
        "read"   -> Read
        "output" -> Output
        "if"     -> If
        "then"   -> Then
        "else"   -> Else
        "while"  -> While
        "do"     -> Do
        _        -> Id (cur:name)
```

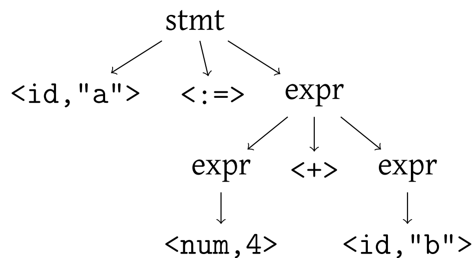
Quelltext 4.4: Erkennung von Bezeichnern und Schlüsselwörtern durch den Lexer

5 Syntaxanalyse

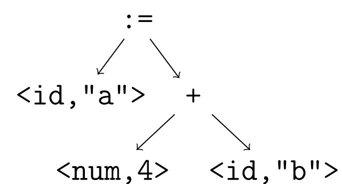
Die Syntax der WHILE-Sprache ist durch eine kontextfreie Grammatik (siehe Abschnitt 3.3) gegeben. In der Phase der Syntaxanalyse wird die lineare Liste der Token in die durch die Grammatik repräsentierte Baumstruktur umgewandelt. In der Regel wird von dieser Struktur noch abstrahiert und der Parsebaum in einen abstrakten Syntaxbaum (AST) umgewandelt [2, 12]. Oft wird der Parsebaum während der Syntaxanalyse gar nicht explizit erzeugt, sondern direkt die Konstruktion des AST vollzogen. Abbildung 5.1 zeigt Tokenstrom, Parsebaum und abstrakten Syntaxbaum einer Zuweisung eines einfachen arithmetischen Ausdrucks.

<id,"a"> <:=> <num,4> <+> <id,"b">

(a) Der Tokenstrom



(b) Ein möglicher Parsebaum



(c) Abstrakter Syntaxbaum

Abbildung 5.1: Beispiel für die Umwandlung eines Tokenstroms in eine Baumstruktur

Grundlegend gibt es zwei Arten der Syntaxanalyse:

1. Bei der *Top-Down-Syntaxanalyse* wird ausgehend von der Wurzel ein Parsebaum zu einer gegebenen Eingabe gesucht. Die Reihenfolge, in der Kindknoten erzeugt werden, entspricht dabei einer Tiefensuche. Da immer zuerst das am weitesten links stehende Nichtterminalsymbol weiter abgeleitet wird, spricht man auch von einer Linksableitung. Top-Down-Parser werden, da sie die Eingabe von *links* lesen und eine *Links*ableitung erstellen, als LL-Parser bezeichnet.

2. Bei der *Bottom-Up-Syntaxanalyse* wird bei der Konstruktion des Parsebaums von den Blättern ausgegangen und sich Stück für Stück zur Wurzel vorgearbeitet. Dabei werden die aus der Eingabe entstehenden rechten Regelseiten auf das Nichtterminalsymbol ihrer linken Regelseite reduziert. Da die am weitesten rechts stehenden Symbole immer zuletzt reduziert werden, spricht man auch von einer Rechtsreduktion beziehungsweise einer umgekehrten Rechtsableitung. Bottom-Up-Parser werden, da sie die Eingabe von *links* lesen und dabei eine umgekehrte *Rechts*ableitung erstellen, als LR-Parser bezeichnet.

In dieser Arbeit liegt der Fokus zunächst auf der Bottom-Up-Syntaxanalyse. Sie ist zwar aufwändiger zu implementieren, benötigt jedoch keine Änderung an der in Kapitel 3 (und im Speziellen in Tabelle 3.3) vorgestellten vollständigen und eindeutigen Grammatik der WHILE-Sprache.

Im weiteren Verlauf dieses Kapitels wird die Grammatik aus Tabelle 5.1 verwendet, um Beispiele zu illustrieren. Sie beschreibt einfache mathematische Ausdrücke mit den Bezeichnern `id`, den Operatoren + und * sowie der Möglichkeit zur Klammerung von Ausdrücken. Dabei wird die Linksassoziativität und Operatorrangfolge der beiden Operatoren beachtet.

$$E \rightarrow E + T \mid T \qquad T \rightarrow T * F \mid F \qquad F \rightarrow (E) \mid \texttt{id}$$

Tabelle 5.1: Beispiel-Grammatik für einfache mathematische Ausdrücke

5.1 Bottom-Up-Syntaxanalyse

Wie im vorigen Abschnitt schon erwähnt wurde, wird beim Bottom-Up-Parsen der Syntaxbaum von den Blättern aus in Richtung Wurzel aufgebaut. Wie dies exemplarisch aussieht, ist in Abbildung 5.2 zu sehen. Wie in dem Beispiel gut zu sehen ist, werden die Terminalsymbole nach und nach auf Nichtterminalsymbole *reduziert* und dabei so lange ein Baum aufgebaut, bis schließlich nur noch das Startsymbol als Wurzel übrig bleibt.

5.1.1 Shift-Reduce-Parser

Eine unkomplizierte Art und Weise, den Parsevorgang zu implementieren, besteht darin einen Shift-Reduce-Parser zu verwenden. Er arbeitet zusätzlich zur Eingabe mit einem Stack, der sowohl Terminal- als auch Nichtterminalsymbole beinhalten wird. Der Name des Parsers bezieht sich auf die zwei grundlegenden Aktionen, die er durchführt:

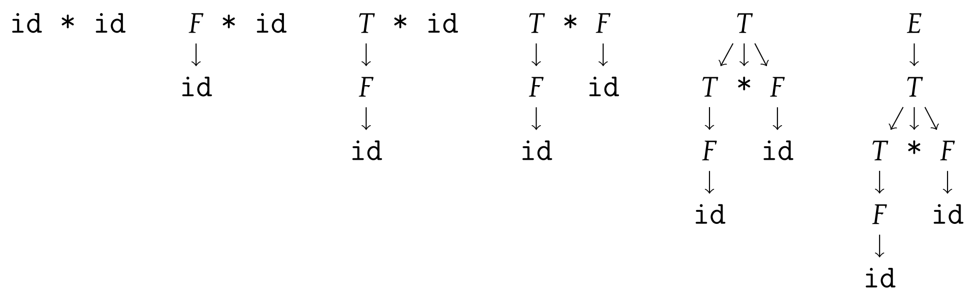


Abbildung 5.2: Parsevorgang auf der Eingabe id ∗ id nach [2, S. 234]

1. *Shift*: Ein Terminalsymbol wird aus der Eingabe entfernt und auf den Stack gelegt.
2. *Reduce*: Liegen ein oder mehrere Symbole oben auf dem Stack, die einer rechten Regelseite entsprechen, so werden sie entfernt und durch das Nichtterminalsymbol auf der linken Regelseite ersetzt.

Stack	Eingabe	Aktion	
$\bot$	id ∗ id \$	*Shift*	
$\bot$ id	∗ id \$	*Reduce* mit Produktion $F \to$ id	$(\star)$
$\bot\ F$	∗ id \$	*Reduce* mit Produktion $T \to F$	
$\bot\ T$	∗ id \$	*Shift*	$(\star\star)$
$\bot\ T\ *$	id \$	*Shift*	
$\bot\ T$ ∗ id	\$	*Reduce* mit Produktion $F \to$ id	
$\bot\ T * F$	\$	*Reduce* mit Produktion $T \to T * F$	
$\bot\ T$	\$	*Reduce* mit Produktion $E \to T$	
$\bot\ E$	\$	*Akzeptiere Eingabe*	

Tabelle 5.2: Parservorgang eines Shift-Reduce-Parsers auf der Eingabe id ∗ id nach [2, S. 237]

Für den in Abbildung 5.2 gezeigten Parsevorgang ist in Tabelle 5.2 der entsprechende Ablauf eines Shift-Reduce-Parsers zu sehen. Dieser Ablauf kann jedoch ohne zusätzliche Informationen nicht deterministisch ausgeführt werden. Beispielsweise wäre es in der mit $(\star\star)$ markierten Zeile auch möglich gewesen, das T mittels $E \to T$ weiter zu reduzieren. Dies wurde hier jedoch nicht gemacht, da danach kein Syntaxbaum mit der kompletten Eingabe mehr hätte erstellt werden können. Zusätzlich hätte in der mit $(\star)$ markierten Zeile eine weitere *Shift*-Aktion ausgeführt werden können. Dann wäre es aber nicht mehr möglich gewesen, den Stack vollständig auf das Startsymbol zu reduzieren.

Wie an dem Beispiel ersichtlich wird, benötigt der Parser weitere Informationen, um eine Eingabe nach dem Shift-Reduce-Prinzip deterministisch erkennen zu können.

5.1.2 LR(0)-Elemente, CLOSURE und GOTO

Die folgenden Absätze geben einen Überblick über die benötigten Hilfsmittel zur Erzeugung eines deterministischen Shift-Reduce-Parsers. Für weitergehende Informationen seien [2, S. 241–246] sowie [12, S. 93–96] empfohlen.

Um den Parsevorgang deterministisch durchführen und an den Stellen mit mehreren Möglichkeiten die richtige Entscheidung treffen zu können, muss der Parser sich merken, wo er sich innerhalb von Produktionen befindet. Dafür werden die Produktionen der Grammatik zu LR(0)-Elementen erweitert: Ein *LR(0)-Element* (kurz *Element*) ist eine Produktion der Grammatik, erweitert durch einen Punkt auf der rechten Regelseite, der besagt, wie weit die Produktion schon erkannt wurde. Die Produktion $F \rightarrow (E)$ hat vier Elemente, da an vier Stellen der rechten Regelseite ein Punkt eingefügt werden kann: $F \rightarrow \bullet (E)$, $F \rightarrow (\bullet E)$, $F \rightarrow (E \bullet)$ und $F \rightarrow (E) \bullet$. Dabei bedeutet $F \rightarrow (\bullet E)$, dass schon eine öffnende Klammer gelesen wurde und erwartet wird, danach einen Ausdruck, der sich auf E reduzieren lässt, sowie eine schließende Klammer zu lesen. $F \rightarrow (E) \bullet$ heißt, dass die komplette rechte Regelseite schon gelesen wurde (und auf dem Stack liegt) und es nun möglich ist, eine Reduktion mit dieser Produktion durchzuführen.

Der Parser merkt sich nun zu jeder Zeit eine Menge an Elementen, die zu seinem momentanen Zustand passen. Diese Elementmengen sind die Zustände des in Abschnitt 5.1.3 konstruierten deterministischen Automaten, der den Parser steuern wird. Um sie zu konstruieren, werden zwei Funktionen, CLOSURE und GOTO, benötigt.

CLOSURE(I) berechnet den Abschluss der Menge I. Er besteht aus der Menge I selber und aus zusätzlichen Elementen nach folgender Vorschrift: Ist in CLOSURE(I) ein Element $A \rightarrow \alpha \bullet B\, \beta$ enthalten, so werden für alle Produktionen $B \rightarrow \gamma$ die Elemente $B \rightarrow \bullet\, \gamma$ zum Abschluss hinzugefügt. Wenn also bei einem Element als nächstes ein Nichtterminalsymbol gelesen werden soll, so gehören alle Elemente mit diesem Nichtterminalsymbol als Regelkopf, bei denen bisher noch nichts gelesen wurde, zum Abschluss. Um die Übersichtlichkeit zu gewährleisten, werden alle durch diese Regel hinzugefügten Elemente nur durch das Nichtterminalsymbol in der Elementmenge repräsentiert, sodass die einzelnen Elemente nicht einzeln aufgeführt werden müssen. Der Vorgang des Hinzufügens wird so lange wiederholt, bis keine neuen Elemente mehr hinzugefügt werden können.

GOTO(I, X) nimmt alle Elemente aus I, in denen als nächstes X gelesen werden soll (also alle Elemente der Form $A \rightarrow \alpha \bullet X\, \beta$), verschiebt den Punkt hinter das X und bestimmt

dann den Abschluss. $\text{GOTO}(\{F \to \bullet\ (\ E\ )\}, ()$ entspricht dann $\text{CLOSURE}(\{F \to (\ \bullet E\ )\})$, was wiederum der Menge $\{F \to (\ \bullet E\ ), E, T, F\}$ entspricht.

5.1.3 Der LR(0)-Automat

Um den im vorigen Abschnitt schon angesprochenen deterministischen Automaten zu konstruieren, fehlt noch ein letzter Schritt: die Erweiterung der Ausgangsgrammatik um ein neues Startsymbol sowie eine Produktion, die das neue zum alten Startsymbol ableitet. In der Beispielgrammatik ist E das Startsymbol. Es wird ein neues Startsymbol S' sowie die Produktion $S' \to E$ hinzugefügt, sodass der Parser beim Reduzieren dieser Produktion weiß, dass der Parsevorgang beendet ist und die Eingabe akzeptiert wurde. Die erweiterte Grammatik ist (inklusive einer später benötigten Nummerierung der Produktionen) in Tabelle 5.3 zu sehen.

$S' \to E$ (0) $\quad E \to E + T$ (1) | T (2) $\quad T \to T * F$ (3) | F (4) $\quad F \to (\ E\ )$ (5) | id (6)

Tabelle 5.3: Erweiterte Beispiel-Grammatik für einfache mathematische Ausdrücke mit Nummerierung der Produktionen

Bei der Konstruktion des Automaten wird vom Startzustand ausgegangen. Dieser Startzustand, als I_0 bezeichnet, ist der Abschluss der Menge $\{S' \to \bullet E\}$, enthält also alle Elemente, die zu einer noch nicht gelesenen Eingabe passen.

Die weiteren Zustände des Automaten und die Übergänge zu ihnen ergeben sich wie folgt: Für einen schon existierenden Zustand I_k und ein Terminal- oder Nichtterminalsymbol X wird die Menge $\text{GOTO}(I_k, X)$ bestimmt. Ist die resultierende Menge nicht leer und noch nicht im Automaten vorhanden, wird sie als neue Menge I_j hinzugefügt. Schließlich wird ein neuer Übergang mit dem Symbol X von I_k nach I_j eingefügt.

Die Startmenge der erweiterten Beispielgrammatik ist $I_0 = \text{CLOSURE}(\{S' \to \bullet E\}) = \{S' \to \bullet E, E, T, F\}$. Wie vorher angesprochen, bedeutet die einfache Aufzählung von E, T und F, dass die Elemente $E \to \bullet E + T$, $E \to \bullet T$, …, $F \to \bullet \texttt{id}$ alle in der Menge enthalten sind. Nun wird beispielsweise $\text{GOTO}(I_0, \texttt{id})$ bestimmt. Da $F \to \bullet \texttt{id}$ das einzige Element in I_0 mit einem id hinter dem Punkt ist, bestimmt sich die Menge zu $\{F \to \texttt{id} \bullet\}$. Diese Menge wird dann als I_1 zum Automaten hinzugefügt und der Übergang von I_0 nach I_1 mit id eingefügt.

Wird dieser Vorgang solange ausgeführt, bis keine neuen Zustände oder Übergänge mehr entstehen, ergibt sich zu der Beispielgrammatik der LR(0)-Automat in Abbildung 5.3. Hier zeigt sich, dass schon Grammatiken mit nur sechs kurzen Produktionen eine relativ

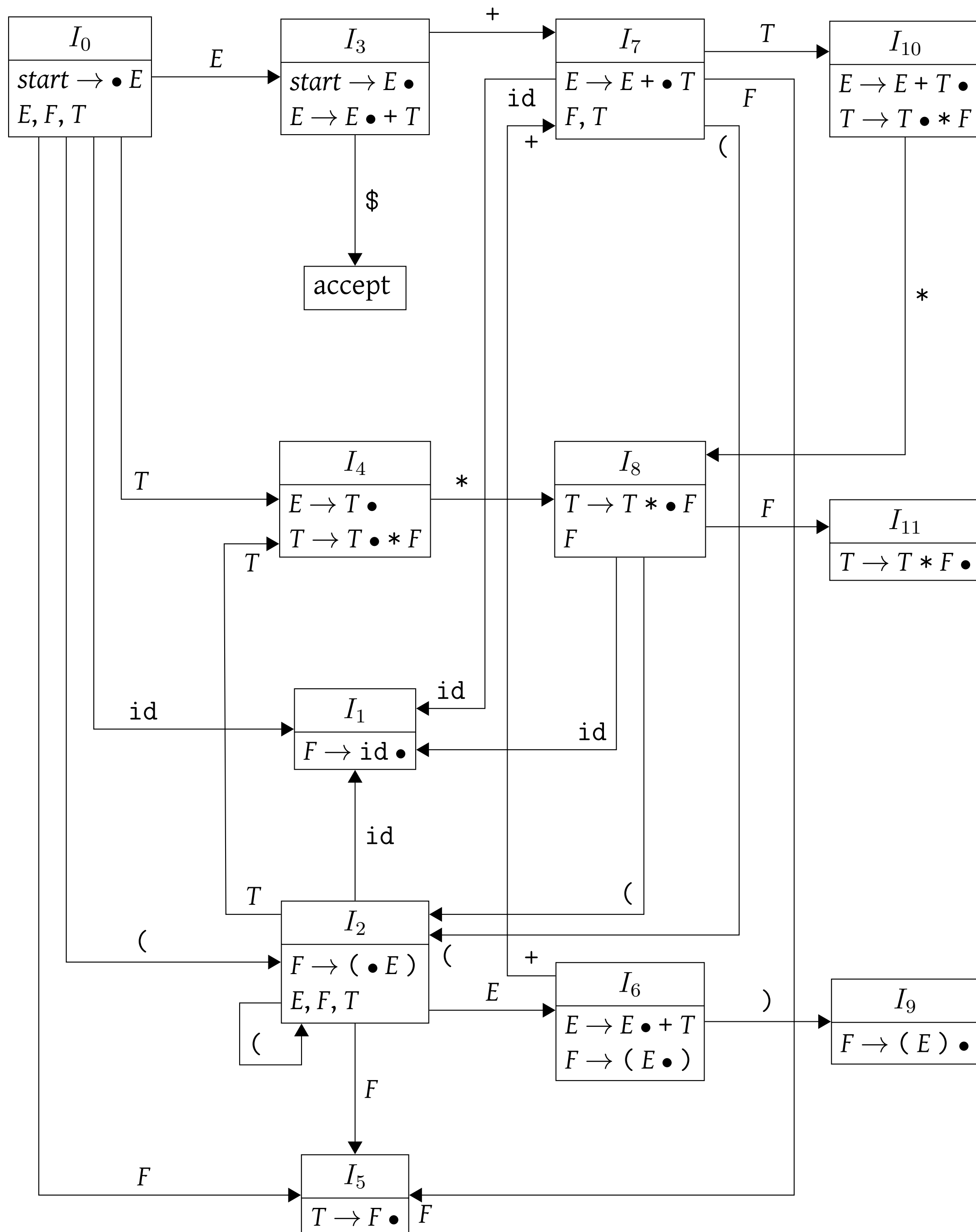


Abbildung 5.3: LR(0)-Automat für die Grammatik aus Tabelle 5.1

große Anzahl an Zuständen besitzen. Die eindeutige Grammatik der WHILE-Sprache kommt mit ihren 36 Produktionen auf 67 Zustände.

5.1.4 (S)LR-Parser

Der schematische Aufbau eines LR-Parsers ist in Abbildung 5.4 zu sehen. Der LR-Parser ist ein Shift-Reduce-Parser und arbeitet daher, wie schon in Abschnitt 5.1.1 angesprochen, neben der Eingabe noch mit einem Stack, um seinen Zustand zu verwalten.

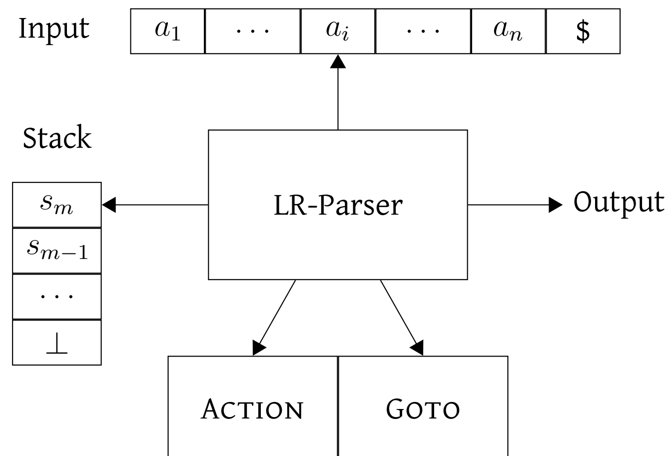


Abbildung 5.4: Schematischer Aufbau eines LR-Parsers nach [2, S. 248]

Das Verhalten des Parsers wird über eine zweiteilige Parsetabelle (ACTION & GOTO) gesteuert. Die Tabelle ist das Äquivalent zum LR(0)-Automaten. In ihr sind die Übergänge zwischen den Zuständen kodiert. Sie beinhaltet zu jedem Zustand k (der zur Elementmenge I_k gehört) und jedem Terminalsymbol a Informationen, welche Aktion der Parser beim Lesen von a ausführen soll.

In Tabelle 5.4 ist die von der Ausdrucksgrammatik erzeugte Parsetabelle zu sehen. Die Einträge in der Tabelle erklären sich wie folgt:

1. *Shift* (sz): Der Parser bewegt sich auf der Eingabe zum nächsten Symbol und legt den Zustand z auf den Zustandsstack, sodass z der neue Zustand wird.

2. *Reduce* (rp): Es wird eine Reduktion mit der Produktion $A \to B_1 \ldots B_n$, welche mit der Zahl p nummeriert wurde, ausgeführt. Dafür werden vom Zustandsstack n Zustände entfernt (n ist die Anzahl der Symbole auf der rechten Seite der Produktion). Für den nun oben auf dem Stack liegenden Zustand wird das Nichtterminalsymbol A im GOTO-Teil der Parsetabelle nachgeschlagen und der dort verzeichnete Zustand auf den Zustandsstack gelegt.

3. *Accept* (acc): Diese Aktion tritt nur auf, wenn der Parser in einem Zustand ist, in dem er die Eingabe akzeptieren kann und wird daher auch nur für das Lesen des Zeichens für das Eingabeende ($) ausgeführt. Der Parser beendet den Parsevorgang.

4. *Error* (kein Eintrag): Jede Stelle, an der in der Tabelle ein leeres Feld ist, bedeutet, dass die Eingabe dieses Terminalsymbols in diesem Zustand nicht gültig geparst werden kann. Daher bleibt dem Parser nichts anderes übrig, als einen Fehler auszugeben und das Parsen abzubrechen.

Es existieren verschiedene LR-Parser, die jedoch alle nach dem gleichen, hier beschriebenen Prinzip arbeiten. Der Unterschied besteht in der Größe der Parsetabelle und der Art ihrer Erzeugung. Drei wichtige Vertreter sind der *kanonische LR*-Parser, auch *LR(1)*-Parser genannt, der *LALR*-Parser, sowie der *Simple LR*-Parser, auch *SLR*-Parser abgekürzt [2, S. 241].

Die Parsetabelle eines LR(1)-Parsers ist deutlich größer, als die der beiden anderen Parser. Dafür kann der LR(1)-Parser alle deterministischen kontextfreien Sprachen erkennen [19]. Der SLR-Parser kann die kleinste Menge an Sprachen erkennen, ist jedoch relativ einfach zu konstruieren und hat eine geringe Anzahl an Zuständen, womit auch die Parsetabelle übersichtlich ausfällt. Der LALR-Parser kann fast so viele Sprachen erkennen wie der LR(1)-Parser, seine Parsetabellen sind aber nur so groß wie die des SLR-Parsers, ihre Konstruktion ist jedoch komplexer.

In dieser Arbeit wurde ein SLR-Parser beziehungsweise -Parsergenerator implementiert. Auf die Details wird im nächsten Abschnitt eingegangen.

Die Erzeugung der in Tabelle 5.4 gezeigten Parsetabelle für den SLR-Parser geschieht wie folgt. Für jeden Zustand i und jedes Terminalsymbol a wird geschaut, ob im LR(0)-

Zustand	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	s1			s2			3	4	5
1		r6	r6		r6	r6			
2	s1			s2			6	4	5
3		s7				acc			
4		r2	s8		r2	r2			
5		r4	r4		r4	r4			
6		s7			s9				
7	s1			s2				10	5
8	s1			s2					11
9		r5	r5		r5	r5			
10		r1	s8		r1	r1			
11		r3	r3		r3	r3			

Tabelle 5.4: Action- & Goto-Tabelle zur erweiterten Beispielgrammatik

Automaten ein Übergang zu einem anderen Zustand j existiert. Ist dies der Fall, wird in der Tabellenzelle (i,a) ein sj notiert, sodass der Parser beim Lesen von a in den Zustand j wechselt. Gleiches wird für jedes Nichtterminalsymbol A getan, wobei hier in der Zelle (i,A) nur der Zielzustand j vermerkt wird.

Nun müssen noch die Zellen befüllt werden, an denen der Parser einen *Reduce*-Schritt durchführen soll. Dafür werden alle Zustände i angeschaut, in denen ein Element vollständig gelesen wurde, also ein Element der Form $A \rightarrow \gamma \bullet$ existiert. Jetzt wird die FOLLOW-Menge von A wichtig, da die Produktion nur dann reduziert werden soll, wenn das nächste Terminalsymbol in FOLLOW(A) enthalten ist, also direkt nach einem A auftreten kann. Für alle Terminalsymbole a aus FOLLOW(A) wird nun in die Tabellenzelle (i,a) ein rp geschrieben, wobei p die Nummer der Produktion $A \rightarrow \gamma$ ist.

Soll die zusätzlich eingefügte Startproduktion (im Beispiel $S' \rightarrow E$) bei einem Lesen von $ reduziert werden, so wird statt einem rp ein acc in die Tabelle geschrieben, damit der Parser weiß, dass jetzt der Parsevorgang abgeschlossen ist.

Es ist möglich, dass nach der Erzeugung der Tabelle in einer Zelle ein sz und ein rp oder ein rp und ein rq stehen. Dies bedeutet, dass ein sogenannter *Shift-Reduce-* beziehungsweise *Reduce-Reduce-Konflikt* aufgetreten ist. Der Parser kann beim Lesen dieses Terminalsymbols nicht entscheiden, welche Aktion beziehungsweise welche Reduktion er ausführen soll. Grammatiken, für die solche Konflikte auftreten, sind nicht SLR-parsebar.

5.1.5 Implementierung eines Parsergenerators

> "So Al agreed to build the tables for the B expression grammar. I remember giving him about 30 grammar rules, and he went up to the stockroom and got a big piece of paper, about 2 by 3 feet, ruled it into squares, and started making entries in it. After an hour of watching him, he said 'this will take a while'. In fact, it took about 2 days!
>
> Finally, Al handed me the paper in triumph, [...] but when I typed the table in and tried to parse, there were errors. Each error we found involved another hour of Al's time and some more rows in the table. Finally, after the third time I asked him 'what are you doing when you make the table?' He told me, and I said 'I could write a program to do that!' And I did..." — Hamilton [13]

Das Zitat ist ein Ausschnitt aus einem Interview, in dem Stephen C. Johnson beschreibt, wie er auf die Idee kam, den Parsergenerator Yacc [18] zu entwickeln. Der Hauptgrund war, dass das Erstellen von Bottom-Up-Parsetabellen und -Automaten von Hand, auch für überschaubare Grammatiken, sehr aufwendig und fehleranfällig ist. Der universelle

Weg besteht darin, einen Parsergenerator zu verwenden, um den Parser automatisch aus einer Grammatik erzeugen zu lassen.

Für Haskell existiert mit Happy [11] ein Parsergenerator, der LALR(1)-Parser erzeugen kann. Um jedoch die Funktionsweise eines Parsergenerators nachzuvollziehen, wurde für diese Arbeit ein kleiner Parsergenerator geschrieben, der aus einer gegebenen Grammatik einen SLR-Parser erzeugen kann. Dies ist für die Grammatik der WHILE-Sprache ausreichend, da sie nicht besonders komplex ist. Die Syntax der Eingabedatei ist dabei sehr stark an die von Happy und damit auch Yacc (dem bekanntesten Parsergenerator) angelehnt.

Parser/While.parser

```
15 %name       parse
16 %tokentype  { PosToken }
17
18 %tokens
19
20 id      { PosToken _ (Id $$) }
21 number  { PosToken _ (Number $$) }
22 bool    { PosToken _ (Boolean $$) }
23 and     { PosToken _ (LogOp And) }
24 or      { PosToken _ (LogOp Or) }
25 not     { PosToken _  Not }
26 relop   { PosToken _ (RelOp $$) }
27 '+'     { PosToken _ (MathOp Plus) }
28 '-'     { PosToken _ (MathOp Minus) }
```

```
74 Expr    :: { AST.Expression }
75         : Expr '+' Term              { AST.Calculation Plus $1 $3 }
76         | Expr '-' Term              { AST.Calculation Minus $1 $3 }
77         | Term                       { $1 }
```

Quelltext 5.1: Ausschnitte aus der WHILE-Grammatikdefinition für den Parsergenerator

Ein Ausschnitt der Eingabedatei für die WHILE-Grammatik ist in Quelltext 5.1 zu sehen. Zu Beginn wird der Funktionsname der Parsers und der Typ der einzulesenden Token festgelegt. Geschweifte Klammern umschließen Haskellcode, der, bis auf kleine Änderungen, wie angegeben später im erzeugten Parser erscheinen wird. Als nächstes folgt die (hier unvollständige) Definition der einzelnen Token. Dies ist einerseits nötig, damit der erzeugte Parser ein Pattern Matching auf den Eingabe-Token ausführen kann. Andererseits werden dadurch Aliase vergeben, wodurch die Token in der Grammatik per `id` oder `'+'` einfacher angesprochen werden können. Die Zeichenfolge $$ innerhalb der Token-Patterns ist eine Besonderheit. Sie besagt, dass an dieser Stelle ein beliebiger Wert

stehen kann und dieser Wert zur weiteren Verwendung in Grammatikregeln bereitgestellt wird.

Im zweiten Ausschnitt werden die Produktionen für das Nichtterminalsymbol *expr* definiert. Da Haskell eine stark getypte Sprache ist, benötigt der Parsergenerator Typinformationen über die für Nichtterminale erzeugten Werte. Dies wird zuerst angegeben, in diesem Fall ein Teilsyntaxbaum `AST` vom Typ `Expression`. Bei den einzelnen Produktionen werden dann Regeln angegeben, wie die Werte der rechten Regelseite zu einem neuen Wert zusammengefasst werden sollen. Abschnitt 5.1.6 geht noch in einem Beispiel näher darauf ein.

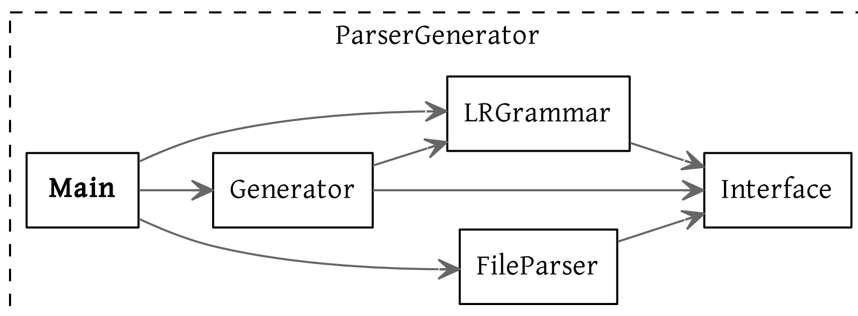


Abbildung 5.5: Modul-Struktur des Parsergenerators

Die Funktionalität des Parsergenerators, dessen Modulstruktur in Abbildung 5.5 als Abhängigkeitsgraph zu sehen ist, ist in drei Phasen aufgeteilt. In der ersten Phase (`FileParser`) wird die Eingabedatei eingelesen, analysiert und in eine geeignete Datenstruktur geschrieben. Durch die Definition der Grammatik besitzt der Parsergenerator danach Informationen über die Terminal- und Nichtterminalsymbole, die in der Grammatik vorkommen, sowie die Produktionen, die definiert wurden.

In der zweiten Phase (`LRGrammar`) ist dann die eigentliche Arbeit des Parsergenerators vollzogen. Aus der in der ersten Phase eingelesenen Grammatik wird nun die Zustandsmenge des LR(0)-Automaten, auch LR(0)-Sammlung genannt, bestimmt. Dies geschieht, wie in den Abschnitten 5.1.2 und 5.1.3 beschrieben. Exemplarisch ist in Quelltext 5.2 ein Ausschnitt zu sehen, in dem die GOTO-Funktion definiert wird. Es ist sehr schön zu sehen, wie der verbal beschriebene Algorithmus fast wörtlich in Haskell umgesetzt werden kann.

Am Ende der zweiten Phase werden die Aktionen des LR(0)-Automaten berechnet, wie es in Abschnitt 5.1.4 beschrieben ist. Statt die Aktionen in einer Tabelle zu speichern, werden sie zunächst ihren Item-Mengen zugeordnet.

Nachdem der LR(0)-Automat und implizit die ACTION- und GOTO-Tabelle berechnet wurden, wird im letzten Schritt (`Generator`) der eigentliche Haskellcode für den Parser

ParserGenerator/LRGrammar.hs

```
204 -- | For the grammar definition @gr@ calculate GOTO(@items@, @symbol@) in 3
205 --   simple steps:
206 --   1. Filter out all items that don't have @symbol@ as next grammar symbol
207 --   2. For all remaining items move the dot one symbol to right
208 --   3. Calculate the closure
209 goto :: LRGrammar -> Symbol -> Set Item -> Set Item
210 goto gr symbol items = closure gr $ Set.map moveDot $
211                                 Set.filter ((== Just symbol) . nextSymbol) items
```

Quelltext 5.2: In Haskell implementierte GOTO-Funktion

erzeugt. Diese Phase besteht prinzipiell nur noch darin, die Elementmengen durchzugehen und mittels Templates die passenden Codefragmente zu erzeugen.

5.1.6 Der generierte Parser

Der vom Parsergenerator erzeugte SLR-Parser arbeitet grundsätzlich mit drei Datenstrukturen: dem Tokenstrom als Eingabe, einem Zustandsstack sowie einem zusätzlichen Wertestack. Der Zustandsstack wird verwendet, wie in Abschnitt 5.1.4 beschrieben. Der Wertestack enthält die Terminalsymbole aus der Eingabe und die für Nichtterminalsymbole erzeugten Teilbäume. Auch die Aktionen sind um die Verwaltung des Wertestacks erweitert:

1. Bei einem *Shift* wird zusätzlich zur Veränderung des Zustandsstacks auch das aktuelle Eingabesymbol auf den Wertestack gelegt, bevor sich der Parser auf der Eingabe weiterbewegt.

2. Bei einem *Reduce* werden auch vom Wertestack die obersten n Elemente heruntergenommen. Sie werden jedoch verwendet, um einen neuen Wert zu erzeugen, welcher zum Nichtterminalsymbol der linken Regelseite gehört.

   In der Grammatikdatei für den SLR-Parser der WHILE-Grammatik ist beispielsweise bei der Produktion $expr \to expr_1 + term$ das Programmfragment `AST.Calculation Plus` $1 $3 gegeben, um den Wert der linken Regelseite zu erzeugen. Dies nimmt die Teilbäume der beiden Nichtterminalsymbole $expr_1$ ($1) und *term* ($3) und fügt sie als Berechnung mit `Plus` zu einem neuen Syntaxteilbaum zusammen.

3. Akzeptiert der Parser die Eingabe mit der Aktion *Accept*, so wird der oben auf dem Wertestack liegende Wert zurückgegeben.

4. Im Falle eines Fehlers durch ein unerwartetes Eingabesymbol wird eine Fehlermeldung ausgegeben, in der alle Terminalsymbole aufgezählt sind, die an dieser Stelle erlaubt gewesen wären.

Der erzeugte Haskell-Quelltext stellt eine Funktion `parse` zur Verfügung, die von außen aufgerufen werden kann. Die Parse-Funktion für den WHILE-Parser nimmt eine Liste von Token entgegen und liefert als Ausgabe einen abstrakten Syntaxbaum, der die Struktur des Programms widerspiegelt.

Parser/While.hs

```
57 -- | The main parsing function. Repeatedly applies a transformation until an
58 --   'accept' definition is reached.
59 parse :: Input -> Output
60 parse inp = parse' ([0], [], inp)
61   where
62     parse' ([], [StackElement_Program result], []) = result
63     parse' tuple = parse' (uncurry3 state tuple)
```

Quelltext 5.3: Die `parse`-Funktion

Die `parse`-Funktion ist in Quelltext 5.3 zu sehen. Das Prinzip ist wie folgt. Auf einer Startkonfiguration (nur Startzustand, leerer Wertestack, komplette Eingabe) wird die Zustandsüberführungsfunktion `state` so lange ausgeführt, bis eine finale Konfiguration erreicht wird. Diese finale Konfiguration enthält nur noch den kompletten Syntaxbaum auf dem Wertestack und gibt ihn dann zurück.

Parser/While.hs

```
365 state states@(8:_) stack input = case headMay input of
366   Just ( PosToken _ (Token ';') ) -> shift 31 states stack input
367   Nothing -> reduce 1 states stack input
368   _ -> error $ "unexpected " ++
369                 if null input then "end of file" else show (head input)
370                 ++ "\nexpecting ';', 'eof'"
```

Quelltext 5.4: Ausschnitt der Zustandsübergänge des erzeugten Parsers

Quelltext 5.4 zeigt einen Ausschnitt aus dem erzeugten Parser. Die Funktion `state` nimmt einen Zustandsstack, einen Wertestack und die Eingabe an und prüft anhand des aktuellen Zustands (in diesem Fall der Zustand mit der Nummer 8) und des ersten Symbols in der Eingabe, welche Aktion ausgeführt werden soll. Wird als nächstes ein `;` gelesen, soll eine *Shift*-Aktion in den Zustand 31 ausgeführt werden. Ist die Resteingabe leer (`Nothing` entspricht damit dem Sonderzeichen $), so soll mittels der ersten Produktion reduziert werden. Dabei sind die Funktionen `shift` und `reduce` nur Hilfsfunktionen,

die eine Konfiguration entgegennehmen und ein 3-Tupel mit der neuen Konfiguration zurückgeben.

Tritt keiner der beiden Fälle ein, so wird der Parsevorgang abgebrochen und ein Fehler ausgegeben. Dabei zeigt die Fehlermeldung nicht nur das gefundene Token, sondern auch, welche Token in diesem Zustand eine gültige Eingabe gewesen wären.

Parser/While.hs
```
108 reduce 16 states
109       (StackElement_Term v3 : _ : StackElement_Expr v1 : stack) input =
110   ( reduceAndGoto 16 states
111   , StackElement_Expr ( AST.Calculation Plus v1 v3 ) : stack
112   , input )
```

Quelltext 5.5: Ausschnitt der Reduktionen des erzeugten Parsers

In Quelltext 5.5 ist die Reduktion für die vorher angesprochene Regel $expr \rightarrow expr_1 + term$ mit dem Codefragment `AST.Calculation Plus` $1 $3 zu sehen. Es ist gut zu sehen, dass zuerst die obersten drei Elemente von Wertestack genommen werden. Dabei erhalten nur der erste und dritte Wert einen Namen, da nur diese beiden Werte im Codefragment referenziert werden. Für die Rückgabe wird dann die im Codefragment angegebene Berechnung ausgeführt und auf den Wertestack gelegt.

5.2 Funktionale Top-Down-Syntaxanalyse

Die Verwendung einer funktionalen Programmiersprache für den Compiler eröffnet – neben den klassischen Verfahren der Bottom-Up- sowie Top-Down-Syntaxanalyse mit tabellengesteuerten Parsern – eine zusätzliche Möglichkeit, die die Idee des rekursiven Abstiegs verwendet. Dabei werden Parser als Funktionen modelliert, die zu einer Eingabe eine Ausgabe und eine eventuelle Resteingabe liefern. Zusätzliche Funktionen höherer Ordnung, in diesem Zusammenhang Kombinatoren genannt, werden verwendet, um Sprachkonstrukte wie Alternativen, sequentielle Ausführung oder Wiederholungen zu implementieren [16]. Dafür werden einem Kombinator ein oder mehrere Parser übergeben und das Ergebnis ist ein neuer Parser.

Parsec [20] stellt für Haskell eine umfangreiche Parser-Kombinator-Bibliothek zur Verfügung. Um den Aufbau und die Funktionsweise nachzuvollziehen wurde für diese Arbeit jedoch, ausgehend von der grundlegenden Literatur [9, 15, 16, 17], ein minimaler funktionaler Parser mit den benötigten Kombinatoren implementiert.

5.2.1 Funktionale Parser und Kombinatoren

Ein Parser kann im Allgemeinen als eine Funktion betrachtet werden, die zu einer Eingabe *input* eine Liste von Tupeln zurückgibt, wobei die Tupel die Ausgabe *output* und die restliche Eingabe *input′* enthalten. Die Verwendung einer Liste als Rückgabetyp erlaubt einerseits, das Fehlschlagen des Parsevorgangs durch eine leere Liste kenntlich zu machen. Andererseits ist es dadurch auch möglich, mehrdeutige Grammatiken zu parsen und in der Liste die Ergebnisse aller verschiedenen Parsevorgänge aufzulisten. Da die hier verwendete Grammatik eindeutig ist, verwendet der implementierte Parser den `Maybe`-Datentyp, um zwischen einem fehlgeschlagenen und einem erfolgreichen Parsevorgang zu unterscheiden. Der komplette Datentyp findet sich in Quelltext 5.6.

Parser/Functional/Library.hs

```haskell
25 -- | Create new parser type with @s@ being the input stream and @r@ being the
26 --   result the parser generates.
27 newtype Parser s r = Parser { runParser :: s -> Maybe (r, s) }
```

Quelltext 5.6: Haskell-Datentyp für den funktionalen Parser

Ein Parser, der einen String mit mathematischen Ausdrücken einliest und den numerischen Wert berechnet, ist beispielsweise ein Parser vom Typ `Parser String Int`, was dem Funktionstyp `String -> Maybe (Int, String)` entspricht. Für die Eingabe `"4+3 müll"` könnte er `Just (7, " müll")` zurückgeben.

Ein Parser besteht im Allgemeinen aus mehreren Teilparsern, die sequentiell ausgeführt werden. Da die do-Notation von Haskell einen komfortablen Weg darstellt, eine sequentielle Ausführung von mehreren Funktionen zu beschreiben, ist es sinnvoll den Parser monadisch zu machen [27].

In Quelltext 5.7 sind die im Folgenden näher beschriebenen Implementierungsschritte zu sehen. Um die do-Notation zu unterstützen und eine Instanz der `Monad`-Klasse zu definieren, müssen für den Parser zwei Funktionen definiert werden. Die erste Funktion ist der Bindungsoperator >>=, welcher intern von der do-Notation verwendet wird, wenn zwei Parser sequentiell ausgeführt werden sollen. Als Funktion höherer Ordnung erwartet er zwei Parser[1] und liefert einen neuen Parser zurück, der dann bei einem Aufruf die beiden anderen Parser ausführt. In der Implementierung wird zuerst der erste Parser auf der Eingabe ausgeführt. Ist er erfolgreich, so wird der zweite Parser auf der Resteingabe ausgeführt und sein Ergebnis zurückgegeben. Ist der erste Parser nicht erfolgreich, wird der zweite Parser gar nicht erst aufgerufen.

[1] Korrekterweise erwartet er einen Parser und eine Funktion, die das Ergebnis eines Parsers entgegennimmt und einen Parser zurückliefert.

Parser/Functional/Library.hs
```
30 -- | Let the parser be an instance of the monad to ease creating and joining
31 --   together parsers.
32 instance Monad (Parser s) where
33   -- | This is a parser which does not use its input and returns what is given
34   return r = Parser (\s -> Just (r, s))
35   -- | Chaining two parsers together is running the first one and then the
36   --   second on the output of the first if it was successful.
37   p >>= f = Parser (\s -> case runParser p s of
38                           Just (r, s') -> runParser (f r) s'
39                           Nothing      -> Nothing)
40
41 -- | Being an instance of the plus monad gives support for choice and failure.
42 instance MonadPlus (Parser s) where
43   -- | Failure: a parser that always fails.
44   mzero = Parser (\_ -> Nothing)
45   -- | Choice: Return the result of the first parser if successful, else the
46   --   result of the second parser.
47   p1 `mplus` p2 = Parser (\s -> case runParser p1 s of
48                                 Nothing -> runParser p2 s
49                                 r       -> r)
```

Quelltext 5.7: Monadische Instanzen des Parsers zur einfacheren Verwendung

Die schematische Abbildung der sequentiellen Ausführung zweier Parser in Abbildung 5.6 zeigt, dass die Resteingabe des ersten Parsers an den zweiten Parser übergeben wird. Am Ende kann sowohl die Rückgabe des ersten als auch die des zweiten Parsers verwendet werden. In der do-Notation wird diese sequentielle Ausführung, ähnlich einer imperativen Programmiersprache, als `do r <- p; q` formuliert.

Die zweite benötigte Funktion `return x` stellt in diesem Kontext einen Parser dar, der die Eingabe gar nicht anschaut, sondern immer den Wert `x` zurückgibt. Wird beispielsweise `return 15` auf der Eingabe `"a+b"` ausgeführt, ist das Ergebnis des Parsers `Just (15, "a+b")`.

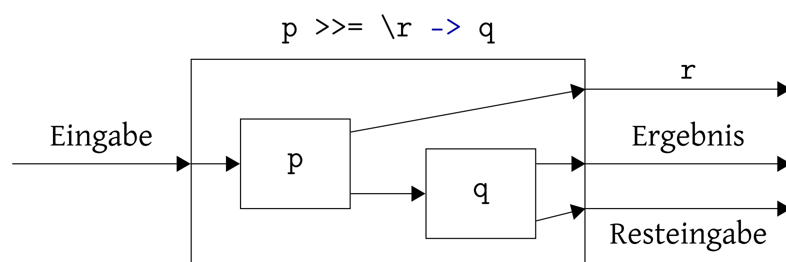


Abbildung 5.6: Schematische Darstellung der sequentiellen Ausführung von p und q – das Ergebnis von p wird an den Bezeichner r gebunden

Neben der Monad-Klasse wird auch die MonadPlus-Klasse mit dem Parser instanziiert. Die beiden von MonadPlus verwendeten Funktionen mzero und mplus haben die semantische Bedeutung eines Fehlschlags und einer Auswahl beziehungsweise Alternative. Daher wird mzero auch als ein Parser implementiert, der bei beliebiger Eingabe fehlschlägt, also Nothing zurückliefert.

Die Funktion mplus ist wieder eine Funktion höherer Ordnung und erwartet zwei Parser. Da mplus das Prinzip der Alternative umsetzt, wird hier zuerst der erste Parser auf der Eingabe ausgeführt. Ist er erfolgreich, so wird sein Ergebnis zurückgegeben. Schlägt er jedoch fehl, so wird der zweite Parser auf der ursprünglichen Eingabe ausgeführt und sein Ergebnis zurückgegeben. Da diese Alternative für die Auswahl zwischen zwei rechten Regelseiten verwendet werden kann, wird der Infix-Operator <|> als Alias der Funktion mplus definiert, sodass die alternative Ausführung der Parser p und q als p <|> q notiert werden kann.

In Abbildung 5.7 ist eine schematische Abbildung des Parsers p <|> q zu sehen. Ein Parser ist hier eine Box mit einer Eingabe und zwei Ausgaben. Die gestrichelten Linien bedeuten, dass dieser Weg nur genommen wird, wenn der erste Weg über p nicht erfolgreich ist.

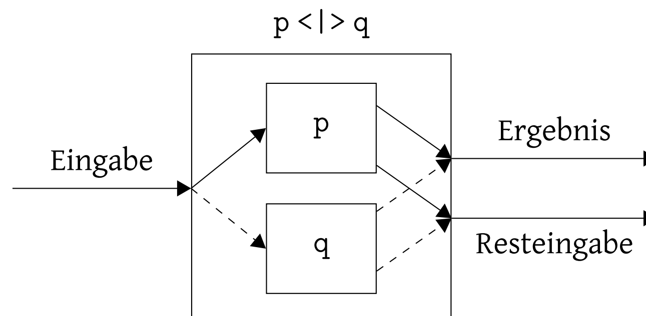


Abbildung 5.7: Grafische Darstellung des Parsers p <|> q

Da bei mplus beim Fehlschlag des ersten Parsers der zweite Parser wieder auf der ursprünglichen Eingabe ausgeführt wird, ist es möglich, dass der erste Parser vor dem Scheitern eine beliebig große Eingabe konsumiert hat. Das bedeutet, dass die auf diese Art und Weise implementierten Parser einen beliebig großen Lookahead besitzen. Dies hat Vor- und Nachteile. Ein Vorteil besteht darin, dass es für Parser mit begrenztem Lookahead oft nötig ist, die Grammatik (zum Beispiel durch Linksfaktorisierung) umzuschreiben. Dieser Schritt entfällt hier. Ein Nachteil besteht darin, dass es bei beliebigem Lookahead schwierig bis unmöglich ist, sinnvolle Fehlermeldungen bei einem Fehlschlagen des Parsers zurückzuliefern.

Des Weiteren können Parser mit beliebigem Lookahead einen großen Speicherverbrauch haben, da der Parser p <|> q so lange seine Eingabe im Speicher halten muss, bis p fertig ist, da bei einem möglichen Fehlschlagen von p der Parser q mit der ursprünglichen Eingabe aufgerufen werden muss. Aus diesen Gründen beschränkt sich beispielsweise *Parsec* auf einen eingeschränkten Lookahead [20, S. 6–7]. Da der hier entwickelte Parserkombinator hauptsächlich zur Veranschaulichung des Konzepts dienen soll, werden die Nachteile nicht berücksichtigt. Die Essenz ist jedoch, dass der funktionale Parser, im Gegensatz zum Bottom-Up-Parser, keine Fehlermeldungen über fehlerhafte Eingaben zurückgibt.

Im Allgemeinen wird beim funktionalen Parsen pro Nichtterminalsymbol ein Parser geschrieben. Die entstehenden Parser werden dann weiter kombiniert und in den Parsern für weitere Nichtterminalsymbole verwendet. Schließlich erhält man einen Parser für das Startsymbol, welcher dann verwendet wird, um beliebige Eingaben zu parsen. Diese Äquivalenz von Parser und Nichtterminalsymbol ist auch im Folgenden wichtig.

5.2.2 Elementare Parser

Zusätzlich zu den vier im letzten Abschnitt beschriebenen einfachen Parsern und Kombinatoren werden zwei einfache Parser zur Verfügung gestellt: Der Parser `eof`, welcher nur bei einer leeren Rest-Eingabe erfolgreich ist, wird verwendet, um sicherzustellen, dass durch einen Parservorgang die ganze Eingabe konsumiert wurde. Der Parser `next` liefert einfach das nächste Element aus der Eingabe zurück.

Mittels dieser Kombinatoren und rudimentären Parser können schon einfache Grammatiken geparst werden. In Quelltext 5.8 sieht man, wie einfach ein Parser für die Produktionen *cmd* → `while` *bexpr* `do` *cmd* und *cmd* → `read identifier` geschrieben werden kann, welcher letztendlich einen Teilbaum des abstrakten Syntaxbaums zurückgibt. Der Parser `token t` ruft intern den Parser `next` auf, um das erste Element der Eingabe zurückzubekommen und vergleicht es mit dem übergebenen Parameter. Sind beide gleich, ist der Parser erfolgreich, ansonsten schlägt er fehl. Steht an der ersten Stelle der Eingabe kein `While`, scheitert die erste Alternative. Dann wird die zweite Alternative, die zuerst versucht ein `Read` einzulesen, ausgeführt.

Wie der erste Teil zeigt, werden innerhalb des Parsers ein weiterer Parser für boolesche Ausdrücke und der *cmd*-Parser selber rekursiv aufgerufen.

```
cmd :: Parser [Token] AST.Command -- Reads a list of tokens as input and returns
                                   -- an abstract syntax tree of type "Command"
cmd = do
    _  <- token While       -- Parse the token <while>
    be <- bexpr             -- Recursively parse a boolean expression
                            -- (bexpr is a parser itself, like cmd)
    _  <- token Do
    c  <- cmd
    return $ AST.While be c -- Return a new abstract syntax tree part for the
                            -- while construct
  <|> do
    _  <- token Read
    id <- identifier
    return $ AST.Read id
  <|> ...
```

Quelltext 5.8: Ausschnitt aus dem funktionalen Parser für das Nichtterminalsymbol *cmd* – die Bereiche zwischen den Alternativen <|> werden sequentiell ausgeführt

5.2.3 Auflösen der Linksrekursion

Ein problematischer Punkt bei Top-Down-Parsern im Allgemeinen ist die Linksrekursion, wie sie auch in der Grammatik der WHILE-Sprache vorkommt. Um beispielsweise die Produktion $expr \rightarrow expr + term$ zu parsen, wird im *expr*-Parser als erstes wieder rekursiv der *expr*-Parser aufgerufen. Diese Rekursion bricht jedoch nicht ab, da dafür vorher feststehen müsste, wie viel + oder - in der Berechnung auftreten. Im Allgemeinen lässt sich das Problem nur durch Veränderungen an der Grammatik lösen, sodass keine Linksrekursion mehr auftritt [2, S. 212–214]. Diese Veränderungen führen jedoch dazu, dass linker und rechter Operand einer mathematisch Operation nicht mehr auf einer Ebene im Parsebaum zu finden sind.

Parser/Functional/Library.hs

```
58 -- | Chains a parser @p@ in a left associative way separated by parser @op@
59 --   which returns the combining function.
60 chainl1 :: Parser s r -> Parser s (r -> r -> r) -> Parser s r
61 p `chainl1` op = p >>= rest
62   where
63     rest x = (do
64         f <- op
65         y <- p
66         rest (f x y)
67       ) <|> return x
```

Quelltext 5.9: Kombinator `chainl1` zur linksassoziativen Verkettung beliebig vieler Parser p

Eine Alternative ergibt sich bei funktionalen Parsern durch das Einführen eines neuen Kombinators `chainl1` (siehe Quelltext 5.9). Seine Funktionsweise wird anhand eines Ausschnitts aus dem funktionalen Parser zur Erkennung des Nichtterminalsymbols *expr* in Quelltext 5.10 erläutert. Der Kombinator `chainl1` nimmt zwei Parser entgegen, in diesem Beispiel `term` und `addop`. Der Parser `term` wird dabei beliebig oft mittels des Parsers `addop` verkettet. Letzterer parst entweder ein + oder ein - und liefert eine Funktion, die zwei Teilsyntaxbäume zu einem neuen Syntaxbaum mittels Addition oder Subtraktion verknüpft.

Die erste geparste Verknüpfungsfunktion wird auf den Teilsyntaxbäumen der ersten beiden `term`-Aufrufe ausgeführt. Das Ergebnis davon wird mittels der zweiten zurückgegebenen Verknüpfungsfunktion mit dem Ergebnis eines dritten `term`-Aufrufs verknüpft. Hier zeigt sich die Linksassoziativität von `chainl1`.

Parser/Functional.hs

```
106 expr :: TokenParser A.Expression
107 expr = term `chainl1` addop
108
109 addop :: TokenParser (A.Expression -> A.Expression -> A.Expression)
110 addop = do { _ <- token $ MathOp Plus ; return $ A.Calculation Plus  }
111     <|> do { _ <- token $ MathOp Minus; return $ A.Calculation Minus }
```

Quelltext 5.10: Verwendung des `chainl1`-Kombinators im *expr*-Parser

Prinzipiell wird durch den Operator `chainl1` eine Produktion der Form $E \longrightarrow E + T \mid T$ in eine Produktion der Form $E \longrightarrow T\ (+\ T)^*$ umgewandelt, also zu einem Vorkommen von T gefolgt von einer beliebigen Wiederholung der Folge $+\ T$. Dies kann jedoch so nicht in einer kontextfreien Grammatik ausgedrückt werden.

5.3 Optimierung des abstrakten Syntaxbaums

Wie in Abschnitt 1.3 angesprochen, erfolgen die Optimierungen des Programms an verschiedenen Stellen. Einige kleine Optimierungen lassen sich effektiv auf dem, durch den Parser erzeugten, abstrakten Syntaxbaum durchführen. Darauf wird im Folgenden näher eingegangen.

Aus verschiedenen Gründen werden in Quelltexten manchmal Berechnungen numerischer Konstanten ausgeschrieben. Dies ist in den meisten Fällen in einer besseren Lesbarkeit oder der Verdeutlichung bestimmter Sachverhalte begründet. Ein Compiler sollte diese Tatsache berücksichtigen und solche Berechnungen schon während der Übersetzung durch ihr numerisches Ergebnis ersetzen.

Des Weiteren ist es möglich, verschiedene mathematische Eigenschaften zu berücksichtigen, sodass in einigen Fällen bestimmte Berechnungen entfallen können. Kandidaten sind hierbei Operationen mit ihrem jeweiligen neutralen Element (Addition und Subtraktion mit 0, Multiplikation und Division mit 1) sowie eine Multiplikation mit 0.

Gleiches gilt für eine mögliche Auswertung von booleschen Bedingungen zur Compile-Zeit. Dadurch können möglicherweise ganze Anweisungen weggelassen oder vereinfacht werden. Beispielsweise kann `if` *bexpr* `then output 10` entweder weggelassen werden, wenn *bexpr* zur Compile-Zeit zu `false` ausgewertet werden kann, oder direkt durch `output 10` ersetzt werden, sollte *bexpr* zu `true` ausgewertet werden.

Sollten durch die Konstantenfaltung Fehlerquellen, wie eine Endlosschleife oder eine Division durch 0 erkannt werden, dann wird eine entsprechende Warnung ausgegeben.

Die Implementierung wurde hier durch Pattern Matching auf dem abstrakten Syntaxbaum realisiert, wie es in Quelltext 5.11 zu sehen ist. Dabei wird diese Funktion bei einem Durchlauf durch den abstrakten Syntaxbaum für jeden Knoten aufgerufen und der Knoten durch den zurückgegebenen Wert ersetzt. Ist beispielsweise einer der beiden Operanden einer Addition die Konstante 0, so wird die Addition durch den anderen Operanden ersetzt.

Optimization/AST/ConstantFolding.hs

```
61 eliminateNeutralElementAndZero :: Expression -> Expression
62 eliminateNeutralElementAndZero (Calculation Plus (Number 0) x) = x
63 eliminateNeutralElementAndZero (Calculation Plus x (Number 0)) = x
64 eliminateNeutralElementAndZero (Calculation Minus (Number 0) x) = Negate x
65 eliminateNeutralElementAndZero (Calculation Minus x (Number 0)) = x
66 eliminateNeutralElementAndZero (Calculation Times (Number 1) x) = x
67 eliminateNeutralElementAndZero (Calculation Times x (Number 1)) = x
68 eliminateNeutralElementAndZero (Calculation Times (Number 0) _) = Number 0
69 eliminateNeutralElementAndZero (Calculation Times _ (Number 0)) = Number 0
70 eliminateNeutralElementAndZero (Calculation DivBy x (Number 1)) = x
71 eliminateNeutralElementAndZero (Calculation DivBy x (Number 0)) =
```

Quelltext 5.11: Eliminierung unnötiger Berechnungen im AST

5.4 Schlussbemerkung

Beide in diesem Kapitel vorgestellten Parser sind im implementierten Compiler vorhanden und können über Kommandozeilenparameter ausgewählt werden. Standardmäßig wird der Bottom-Up-Parser verwendet, da er, im Unterschied zum funktionalen Parser, bei einem syntaktischen Fehler eine sinnvolle Fehlermeldung ausgeben kann.

6 Zwischencodeerzeugung

Die Struktur eines Syntaxbaums unterscheidet sich sehr stark von der linearen Struktur der Zielsprache und ist noch sehr abstrakt. Eine direkte Erzeugung von Code der Zielsprache aus dem Syntaxbaum ist in der Regel nicht gewünscht, da in der Assemblersprache viele architekturspezifische Eigenschaften berücksichtigt werden müssen. Aus diesem Grund wird der abstrakte Syntaxbaum in einen Zwischencode umgewandelt, der Ähnlichkeiten mit Assemblercode hat, aber weiterhin von speziellen Eigenschaften abstrahiert. Je nach Komplexität der Quellsprache und des Compilers werden auch mehrere Zwischencodes verwendet, die jeweils für verschiedene Optimierungen genutzt werden. So verwendet etwa der bekannte GCC-Compiler drei verschiedene Zwischendarstellungen [25, S. 211].

Der Zwischencode ist die Schnittstelle zwischen quellsprachenspezifischem Front- und zielsprachenspezifischem Back-End. Dies wird auch aus der Übersicht über die Phasen des Compilers (Abbildung 1.1 auf Seite 2) ersichtlich. Durch die geeignete Wahl eines Zwischencodes können verschiedene Front- und Back-Ends beliebig kombiniert und dadurch viele Quellsprachen in viele Zielsprachen beziehungsweise Zielarchitekturen übersetzt werden.[1]

6.1 Eigenschaften des Zwischencodes

In dieser Arbeit wird, aufgrund des geringen Sprachumfangs der Quellsprache, nur eine Zwischendarstellung verwendet. Diese ist ein, an Assemblersprachen angelehnter, Drei-Adress-Code mit

- Berechnungen der Form `a = -b` und `a = b + c` (wobei + hier stellvertretend für binäre mathematische Operationen steht),
- Zuweisungen der Form `a = b`,
- benannten Sprungmarken (`marke:`),

[1] Der GCC bringt standardmäßig Front-Ends für C, C++, Objective-C, Fortran, Java, Ada und Go [32] sowie Back-Ends für x86, x86-64, PowerPC, SPARC, ARM und viele andere Architekturen [26] mit.

- unbedingten (`goto marke`) und bedingten Sprüngen (`goto marke if a = b`, mit = als Stellvertreter für Vergleichsoperatoren, sowie `goto marke if (not) a`) und
- den Sprachkonstrukten `output a` sowie `read b`.

Dieser Zwischencode ist einerseits nah am Assemblercode, sodass im Schritt der Codeerzeugung keine großen strukturellen Transformationen vorgenommen werden müssen. Andererseits bleibt er weiterhin abstrakt genug, um sich auf die in Kapitel 7 vorgestellten Optimierungen konzentrieren zu können.

6.2 Erzeugung des Zwischencodes

Die im Folgenden beschriebene Methode der Zwischencodegenerierung folgt im Wesentlichen der Literatur in [2, 12]. Wie dies in der Implementierung umgesetzt wurde, wird in Abschnitt 6.3 beleuchtet.

6.2.1 Umwandlung arithmetischer Ausdrücke

Im Gegensatz zur Quellsprache, bei der arithmetische Ausdrücke beliebig verschachtelt sein können, ist es im Drei-Adress-Code nur möglich, zwei Variablen oder Konstanten mathematisch zu verknüpfen. Daher ist es nötig, die Zwischenergebnisse von Unterausdrücken in temporären Variablen abzulegen. In Quelltext 6.1 werden beispielsweise zwei temporäre Variablen verwendet um die Berechnung durchzuführen.

```
1 a := 3 * b + c + 2
2 # Mit expliziter Klammerung:
3 # a := ((3 * b) + c) + 2
```

(a) Eingabe-Programm

```
1 t1 = 3 * b
2 t2 = t1 + c
3 a = t2 + 2
```

(b) Zugehöriger Zwischencode

Quelltext 6.1: Linearisierung der Baumstruktur eines arithmetischen Ausdrucks

Im Syntaxteilbaum einer mathematischen Operation befindet sich in der Wurzel die Art der Operation (+, ∗, …) und der linke und rechte Unterbaum entspricht jeweils dem mathematischen Ausdruck auf der linken und rechten Seite der Operation. Wie in dem Ausschnitt aus einem möglichen Übersetzungsschema (Quelltext 6.2) zu sehen ist, wird für jeden zusammengesetzten Ausdruck zuerst eine neue temporäre Variable angelegt. Um den Code zur Berechnung des Ausdrucks zu erhalten, wird zuerst der Code zur Berechnung des linken sowie des rechten Teilausdrucks ausgeführt. Zuletzt wird

eine Instruktion erzeugt, die die „Adressen" der beiden Teilausdrücke mathematisch verknüpft und das Ergebnis in die zuvor angelegte temporäre Variable schreibt. Die letzten Zeile des Übersetzungsschemas zeigt, dass eine „Adresse" hierbei sowohl ein Variablenname als auch eine Zahlen-Konstante sein kann.

```
1 C := id := E { C.code = E.code || gen(id "=" E.addr); }
2 E := E1 + T  { E.addr = newTemp();
3                E.code = E1.code || T.code || gen(E.addr "=" E1.addr "+" T.addr);
4              }
5 T := T1 * F  { T.addr = newTemp();
6                T.code = T1.code || F.code || gen(T.addr "=" T1.addr "*" F.addr);
7              }
8 F := id      { E.addr = id ; E.code = ""; }
9 F := num     { E.addr = num; E.code = ""; }
```

Quelltext 6.2: Ausschnitt aus einem Übersetzungsschema (SDT) zur Berechnung arithmetischer Operationen

6.2.2 Bedingte Anweisungen und Sprünge

Die größte strukturelle Änderung bei der Transformation in den Zwischencode findet bei bedingten Anweisungen und Schleifen statt. Beide müssen durch bedingte und unbedingte Sprünge ersetzt werden. Grundlage ist hier die Erzeugung von sogenanntem Springcode. Diese Arbeit beschränkt sich zur Erklärung der Springcodeerzeugung auf einzelne Beispiele. Eine umfassende Übersicht ist beispielsweise in [2, S. 399–407] zu finden.

Die schematische Struktur des Springcodes, der aus den bedingten Anweisungen erzeugt werden soll, ist in Abbildung 6.1 zu betrachten. Sie zeigt den linearen Aufbau des Drei-Adress-Codes sowie die für den Code erzeugten Sprungmarken. Die Pfeile visualisieren Sprünge, die innerhalb der Codeblöcke erzeugt werden und zu den entsprechenden Sprungmarken springen. Gestrichelte Pfeile visualisieren Sprünge aus einem *C*-Block zur entsprechenden *C.next*-Sprungmarke. In Tabelle 6.1 findet sich die dazu passende syntaxgerichtete Definition. Hier werden Ausschnitte aus der vereinfachten WHILE-Grammatik (zu finden in Tabelle 3.2, Kapitel 3) verwendet, wobei die Nichtterminale durch den ersten Buchstaben abgekürzt werden.

Die grundlegende Idee besteht darin, für eine boolesche Bedingung *B* zwei Sprungmarken zu erzeugen, zu denen gesprungen werden soll, wenn *B* zu `true` beziehungsweise `false` ausgewertet wird. Diese beiden Sprungmarken sind ererbte Attribute von *B* und werden daher vor der Berechnung von *B.code* festgesetzt. Somit können sie zum Zeitpunkt der Erzeugung des *B*-Codes als Sprungziele verwendet werden.

Produktion	Semantische Regeln
$C \rightarrow \texttt{if}\ B\ \texttt{then}\ C_1$	$B.true = newLabel()$ $B.false = C_1.next = C.next$ $C.code = B.code \Vert label(B.true) \Vert C_1.code$
$C \rightarrow \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2$	$B.true = newLabel()$ $B.false = newLabel()$ $C_1.next = C_2.next = C.next$ $C.code = B.code \Vert label(B.true) \Vert C_1.code \Vert gen(\text{"goto"}\ C.next)$ $\Vert label(B.false) \Vert C_2.code$
$C \rightarrow \texttt{while}\ B\ \texttt{do}\ C_1$	$begin = newLabel()$ $B.true = newLabel()$ $B.false = C.next$ $C_1.next = begin$ $C.code = label(begin) \Vert B.code \Vert label(B.true)$ $\Vert C_1.code \Vert gen(\text{"goto"}\ begin)$
$B \rightarrow B_1\ \texttt{and}\ B_2$	$B_1.true = newLabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \Vert label(B_1.true) \Vert B_2.code$
$B \rightarrow B_1\ \texttt{or}\ B_2$	$B_1.true = B.false$ $B_1.false = newLabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \Vert label(B_1.false) \Vert B_2.code$
$B \rightarrow \texttt{not}\ B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow \texttt{true}$	$B.code = gen(\text{"goto"}\ B.true)$
$B \rightarrow \texttt{false}$	$B.code = gen(\text{"goto"}\ B.false)$
$B \rightarrow \texttt{eof}$	$B.code = gen(\text{"goto"}\ B.true\ \text{"if eof"}) \Vert gen(\text{"goto"}\ B.false)$
$B \rightarrow E_1\ \texttt{relop}\ E_2$	$B.code = E_1.code \Vert E_2.code$ $\Vert gen(\text{"goto"}\ B.true\ \text{"if"}\ E_2.addr\ \texttt{relop}\ E_2.addr)$ $\Vert gen(\text{"goto"}\ B.false)$

Tabelle 6.1: Ausschnitt aus einer syntaxgerichteten Definition (SDD) zur Erzeugung des Springcodes nach [2, S. 402, 404]

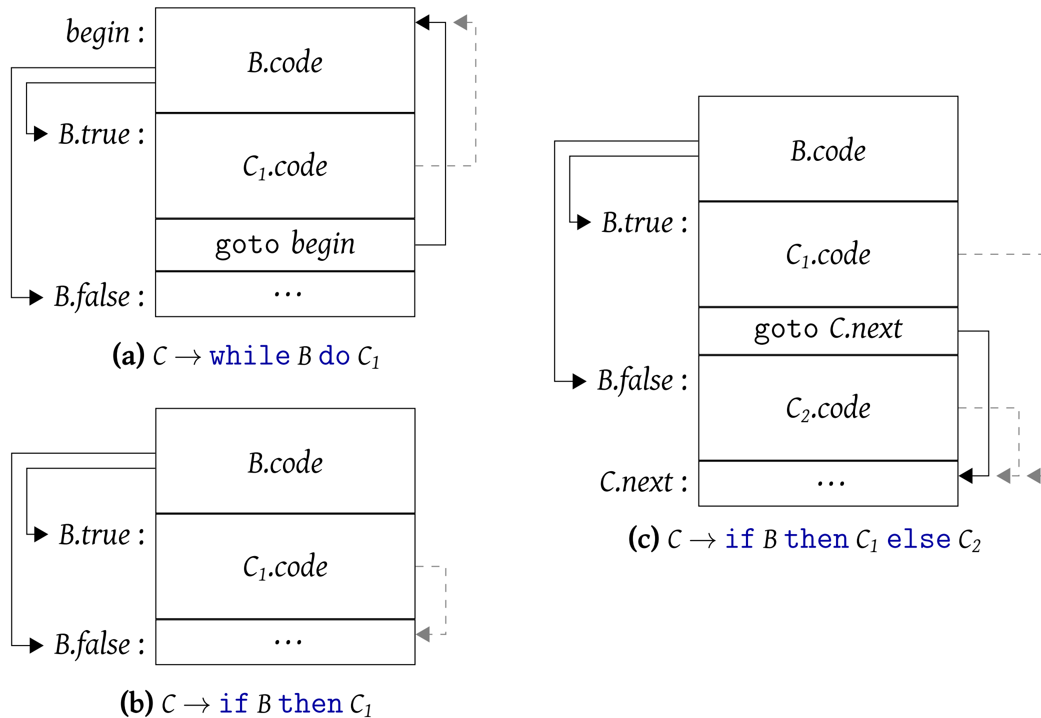


Abbildung 6.1: Schematische Darstellung des Springcodes nach [2, S. 401] und [12, S. 210]

Zusätzlich hat jede WHILE-Anweisung C ein ererbtes Attribut *C.next*, welches ebenfalls eine Sprungmarke ist. Diese Sprungmarke wird immer auf das Ziel festgelegt, welches nach der Anweisung ausgeführt werden soll. Besteht beispielsweise der Code C_1 der `while`-Anweisung in Abbildung 6.1a aus einem einzelnen `if`-`then`-Konstrukt, wird bei Auswertung der `if`-Bedingung zu `false` an das Ende des Konstrukts und damit genau vor `goto` *begin* gesprungen. Durch das korrekte Setzen der Sprungmarke C_1*.next* auf *begin* wird ein doppelter Sprung vermieden und direkt an den Schleifenstart gesprungen.

Die Sprünge zu den jeweiligen Sprungmarken werden im Springcode für die booleschen Ausdrücke generiert. Der simpelste Code wird für die Konstanten `true` und `false` erzeugt, nämlich ein unbedingter Sprung zu der korrekten Sprungmarke. Interessanter wird es, wenn man die logischen Verknüpfungen betrachtet. Bei einer Negation etwa wird die Umkehrung der Logik durch eine einfache Vertauschung der beiden Sprungmarken erreicht.

Bei den logischen Verknüpfungen `and` und `or` wird eine Bedarfsauswertung vorgenommen. Wird bei einem logischen Oder der linke Teilausdruck zu `true` ausgewertet, so kann direkt zur *true*-Sprungmarke des Gesamtausdrucks gesprungen werden. Auf diese Art und Weise werden unnötige Berechnungen vermieden und die spätere Ausführung

```
1 if x > 6 or (x < 3 and x = y) then
2     x := 5
```

(a) Eingabe-Programm

```
1     goto L2 if x > 6
2     goto L3
3 L3: goto L4 if x < 3
4     goto L1
5 L4: goto L2 if x == y
6     goto L1
7 L2: x = 5
8 L1:
```

(b) Zugehöriger Zwischencode

Quelltext 6.3: Zwischencode für eine einfache `if`-Anweisung

wird beschleunigt. In Quelltext 6.3 kann man gut die beiden Abkürzungen in den Zeilen 1 und 4 des Zwischencodes beobachten. Ist `x > 6` kann direkt zur Zuweisung gesprungen werden. Ist in dem rechten Teilausdruck die erste Bedingung nicht erfüllt, so wird direkt hinter die Zuweisung gesprungen, ohne die zweite Bedingung auszuwerten.

6.3 Implementierung

Die Zwischencodeerzeugung ist in dem Haskell-Modul `IntermediateCode` implementiert, welches wiederum auf die beiden Schnittstellen `Interface.AST` für den abstrakten Syntaxbaum und `Interface.TAC` für den Drei-Adress-Code (Three-Address-Code) zugreift.

Die in Tabelle 6.1 gezeigte syntaxgerichtete Definition ist eine sogenannte L-Attributierung. Das bedeutet, dass alle Attribute der Knoten in einem Durchgang einer Tiefensuche, bei der jeder Knoten einmal vor und einmal nach der Berechnung der Kindknoten besucht wird, berechnet werden können. Für die Produktion $A \rightarrow B_1 \ldots B_k$ werden beim ersten Besuch des Knotens A die ererbten Attribute aller B_i berechnet und dann nacheinander die Knoten B_i besucht. Zuletzt werden die synthetisierten Attribute von A bestimmt.

Dieses Verhalten lässt sich durch geeignete Funktionen nachbilden, ohne die Attribute explizit an die Knoten zu binden. In Quelltext 6.4 sind die Deklarationen der in der Implementierung verwendeten Funktionen zu sehen. Wie leicht nachvollzogen werden kann, werden hier zusätzlich zu dem Teilbaum (C, E oder B) die ererbten Attribute (hier sind es nur die Sprungmarken) übergeben. Zurückgegeben wird der erzeugte Code sowie im Fall der mathematischen Ausdrücke zusätzlich die „Adresse“ des Ausdrucks.

Innerhalb der Funktionen werden zuerst eventuell benötigte Sprungmarken oder temporäre Variablen erzeugt und dann die passenden Funktionen verwendet, um den Code für die Unterausdrücke berechnen zu lassen.

```
        -- C       -> C.next -> C.code
command :: Command -> Label  -> TAC

           -- E           -> (E.code, E.addr)
expression :: Expression -> (TAC    , Data  )

               -- B               -> B.true -> B.false -> B.code
boolExpression :: BoolExpression -> Label  -> Label   -> TAC
```

Quelltext 6.4: Funktionsdeklarationen zur Drei-Adress-Code-Erzeugung

Quelltext 6.5 zeigt einen Ausschnitt aus der Drei-Adress-Code-Erzeugung, in dem Code für die logischen Verknüpfungen erzeugt wird. Hier werden zuerst die von den Unterknoten benötigten Sprungmarken erzeugt und dann die Generierung der jeweiligen Teilbäume rekursiv aufgerufen. Schließlich werden die beiden Drei-Adress-Codes und die neue erzeugte Sprungmarke verkettet und zurückgegeben. Die Umsetzung der SDD aus Tabelle 6.1 ist deutlich zu erkennen.

Zur Erzeugung der nummerierten Sprungmarken (und der temporären Variablen bei den mathematischen Ausdrücken) muss gespeichert werden, wie viele Sprungmarken und temporäre Variablen zu einem bestimmten Zeitpunkt schon erzeugt wurden. Um einen solchen globalen Zustand in Haskell zu realisieren, wird die gesamte Berechnung des Drei-Adress-Codes innerhalb einer Zustandsmonade ausgeführt.

IntermediateCode.hs

```
134 -- | Generates three address code for one boolean expression in the AST
135 -- (possibly generating code for boolean subexpressions first).
136 boolExpression :: AST.BoolExpression -> TAC.Label -> TAC.Label
137                -> State GenState TAC.TAC
138 boolExpression bexpr lTrue lFalse = case bexpr of
139   AST.LogOp op b1 b2 -> case op of
140     T.And -> do
141       b1True <- newLabel
142       tac1 <- boolExpression b1 b1True lFalse
143       tac2 <- boolExpression b2 lTrue lFalse
144       return $ tac1 ++ [TAC.Label b1True] ++ tac2
145     T.Or -> do
146       b1False <- newLabel
147       tac1 <- boolExpression b1 lTrue b1False
148       tac2 <- boolExpression b2 lTrue lFalse
149       return $ tac1 ++ [TAC.Label b1False] ++ tac2
```

Quelltext 6.5: Ausschnitt aus dem Modul zur Erzeugung des Drei-Adress-Code

7 Zwischencodeoptimierung

> “The term ‘optimization’ in compiler design refers to the attempts that a compiler makes to produce code that is more efficient than the obvious code. ‘Optimization’ is thus a misnomer, since there is no way that the code produced by a compiler can be guaranteed to be as fast or faster than any other code that performs the same task.” — Aho u. a. [2, S. 15]

Das größte Potenzial der Optimierung des zu erzeugenden Programms liegt in der Optimierung des Zwischencodes. Hierbei ist jedoch wichtig immer darauf zu achten, bei einer Optimierung nicht die Semantik des Programms zu verändern. Im Folgenden werden die einzelnen Komponenten der im Compiler implementierten Optimierungen beleuchtet.

Einige der möglichen beziehungsweise nötigen Optimierungen ergeben sich aus der Art der Zwischencodeerzeugung, wie in Kapitel 6 beschrieben. Vor allem der Springcode (Abschnitt 6.2.2) lässt Raum für schnelle und einfache Optimierungen.

Im Allgemeinen lassen sich bei der Zwischencodeoptimierung zwei verschiedene Strategien unterscheiden. Zum einen können Optimierungen anhand des linearen Drei-Adress-Codes durchgeführt werden. Ein Spezialfall ist hier die sogenannte *Guckloch-Optimierung*. Zum anderen kann das Programm in einen gerichteten Graphen (den Flussgraphen) zerlegt werden, in dem verschiedene Arten von Datenflüssen analysiert werden. Mit den Ergebnissen dieser *Datenflussanalyse* können schließlich (globale) Optimierungsentscheidungen getroffen werden.

7.1 Lineare und Guckloch-Optimierung

Die Guckloch-Optimierung (engl. *peephole optimization*) beschreibt ein Verfahren, bei dem Optimierungsentscheidungen basierend auf der direkten Umgebung einer Drei-Adress-Code-Anweisung getroffen werden. Der Name leitet sich davon ab, dass immer nur ein sehr eingeschränkter Bereich des Programms, beispielsweise vier Zeilen, zusammen betrachtet wird.

Die allgemeinere Optimierung anhand des Drei-Adress-Codes kann hingegen auch globale Entscheidungen treffen, wie es in Abschnitt 7.1.2 vorgenommen wird.

7.1.1 Überflüssige Sprünge

Bedingte Anweisungen wie `if a = b then c := d + 1` werden bei der Springcodeerzeugung zu Zwischencode, wie er in Quelltext 7.1a zu sehen ist, transformiert. Dieser Code kann semantikerhaltend zu einer Version mit nur einem Sprungbefehl, gezeigt in Quelltext 7.1b, umgewandelt werden, indem die Sprungbedingung negiert wird. Die Sprungmarke `label2:` wird dabei nicht entfernt, da sie möglicherweise noch von einem anderen Befehl angesprungen wird (siehe Abschnitt 7.1.2).

```
1 goto label2 if a == b
2 goto label1
3 label2:
4 c = d + 1
5 label1:
```

(a) vor der Optimierung

```
1 goto label1 if a != b
2
3 label2:
4 c = d + 1
5 label1:
```

(b) nach der Optimierung

Quelltext 7.1: Zwischencodeoptimierung zu `if a = b then c := d + 1`

Umgekehrt kann eine Anweisung wie `if not a = b then c := d + 1` dazu führen, dass unbedingte Sprünge zur direkt folgenden Zeile erzeugt werden, wie in Quelltext 7.2 zu sehen ist. Dies passiert, da der Teil, in dem die Sprungmarken in den Code eingefügt werden, und der Teil, in dem die Sprünge erzeugt werden, unabhängig voneinander sind. Hier kann der unbedingte Sprung vollständig entfernt werden; auf das Entfernen der Sprungmarke wird jedoch aus denselben Gründen wie oben verzichtet.

```
1 goto label12 if a == b
2 goto label15
3 label15:
4 c = d + 1
5 label12:
```

(a) vor der Optimierung

```
1 goto label12 if a == b
2
3 label15:
4 c = d + 1
5 label12:
```

(b) nach der Optimierung

Quelltext 7.2: Überflüssiger unbedingter Sprung

7.1.2 Überflüssige Sprungmarken

Zusätzlich zu den überflüssigen Sprungbefehlen können in der Phase der Zwischencodegenerierung zwei Arten von überflüssigen Sprungmarken enstehen.

Einerseits können mehrere Sprungmarken an der gleichen Instruktion auftreten. Offensichtlich können diese Sprungmarken alle durch eine einzelne ersetzt werden. Zusätzlich müssen alle Sprünge zu einer der entfernten Marken auf die neue Marke geändert werden, sodass dieser Optimierungsschritt zwei Durchläufe über den kompletten Zwischencode benötigt.

Andererseits wird pro Anweisung im Quellcode eine Sprungmarke generiert. Die meisten dieser Marken werden jedoch von keinem Sprungbefehl angesprungen und können daher einfach entfernt werden. Dies ist besonders für die Konstruktion des Flussgraphen in Abschnitt 7.2 sinnvoll.

7.1.3 Implementierung

Die Implementierung der Guckloch-Optimierung besteht darin, die Liste der Drei-Adress-Code-Befehle zu durchlaufen und dabei die angesprochenen Muster von Sprüngen und Sprungmarken zu erkennen und zu ersetzen. In Quelltext 7.3 ist ein Ausschnitt zu sehen, in dem unbedingte Sprünge mit direkt folgenden Sprungmarken erkannt werden. Sollten dann das Ziel des Sprungs und die Sprungmarke identisch sein, wird der Sprung aus dem Code einfach weggelassen.

Optimization/TAC/UnnecessaryJumps.hs

```
40 -- Remove every goto instruction that jumps to a directly following label
41 removeUnnecessaryJumps (Goto l1 : Label l2 : tac)
42   | l1 == l2 = Label l2 : removeUnnecessaryJumps tac
```

Quelltext 7.3: Entfernen unbedingter Sprünge auf direkt nachfolgende Sprungmarken

Für die globalen Optimierungen, wie das Vereinigen mehrerer Sprungmarken und das Entfernen von nicht angesprungenen Marken, müssen in einem ersten Schritt Informationen über den Drei-Adress-Code gesammelt werden. In Quelltext 7.4 wird beispielsweise in einem Durchlauf eine Liste aller Sprungmarken angelegt, die von einer Sprunganweisung verwendet werden. Danach werden alle Marken entfernt, die in dieser Liste nicht vorkommen.

7.2 Flussgraph und Datenflussanalyse

Die Datenflussanalyse ist für die Optimierung von Programmen sehr wichtig, da durch sie verschiedene Informationen über Informationsflüsse im Programm gewonnen werden können. Mithilfe dieser Informationen können dann beispielsweise Lebendigkeitsbereiche

Optimization/TAC/UnneededLabels.hs

```
29 -- | Removes all labels from the TAC that are never jumped to by some goto
30 --   instruction.
31 removeUnneededLabels :: TAC -> TAC
32 removeUnneededLabels tac = removeUnneededLabels' tac
33   where
34     removeUnneededLabels' :: TAC -> TAC
35     removeUnneededLabels' [] = []
36     removeUnneededLabels' (Label l : tac_)
37       | l `notElem` jumpTargets = removeUnneededLabels' tac_
38     removeUnneededLabels' (t:tac_) = t : removeUnneededLabels' tac_
39     -- Get all labels that are jumped to by some goto instruction
40     jumpTargets :: [Label]
41     jumpTargets = nub $ getLabelFromGoto <$> filter isGoto tac
```

Quelltext 7.4: Entfernen von Sprungmarken, die nie angesprungen werden

von Variablen erkannt werden. Dafür muss die lineare Struktur des Zwischencodes jedoch zuerst in einen gerichteten Graphen transformiert werden.

Der Transformationsprozess gestaltet sich wie folgt: Zuerst wird der Zwischencode in Grundblöcke zerteilt, wobei ein Grundblock eine maximale Folge von Anweisungen ist, die auf jeden Fall hintereinander ausgeführt werden. Das bedeutet, dass Sprünge von woanders höchsten am Anfang und Sprünge woanders hin höchstens am Ende eines Grundblocks auftreten dürfen. Daher beginnen sowohl alle Anweisungen hinter einem Sprungbefehl als auch Sprungmarken einen neuen Grundblock. Wären zu diesem Zeitpunkt noch überflüssige Sprungmarken vorhanden (siehe Abschnitt 7.1.2), würden mehr Grundblöcke erzeugt als nötig.

Nachdem die Grundblöcke aufgeteilt wurden, werden gerichtete Kanten für alle Übergänge eingefügt, die das Programm nehmen kann. Bis auf finale Grundblöcke haben damit alle Grundblöcke eine ausgehende Kante (bei einem unbedingten oder gar keinem Sprung am Ende) oder zwei ausgehende Kanten (bei einem bedingten Sprung am Ende). Am Ende werden noch ein Startknoten ENTRY sowie ein Endknoten EXIT eingefügt. Ein Beispiel eines erzeugten Flussgraphen ist in Anhang A.3.6 zu sehen.

7.2.1 Lebendigkeitsanalyse

Ein wichtiger Analyseprozess, sowohl für die Registerzuteilung in Abschnitt 8.1 als auch für weitere lokale Optimierungen, ist die Analyse der *Lebendigkeit* von Variablen. Dabei wird für jede Variable bestimmt, in welchen Blöcken sie gelesen und in welchen Blöcken sie geschrieben wird. Wird eine Variable in einem Block beispielsweise nur gelesen, bedeutet dies, dass sie zu Beginn des Blocks und zum Ende der vorherigen Blöcke lebendig ist.

Ihr Inhalt muss im Speicher oder einem Register gehalten und darf nicht überschrieben werden.

In Anhang A.3.7 ist gut zu sehen, dass die beiden Variablen `min` und `max` zu (fast) jeder Zeit im Programm lebendig sind. Die Variable `x` hingegen wird nur in den Blöcken `B3` bis `B6` verwendet.

Mithilfe der Lebendigkeitsanalyse kann auch festgestellt werden, ob es Variablen gibt, die innerhalb eines Programms gelesen werden, obwohl sie möglicherweise vorher noch nicht geschrieben wurden. Ist die Menge der zu Beginn des ersten Blocks lebendigen Variablen (die Menge `IN` von `B1`) nicht leer, ist es möglich, dass diese Variablen ohne vorherige Initialisierung gelesen werden. Da sie aber nicht zwingend vor einem ersten Schreiben gelesen werden, gibt der Compiler hier nur eine Warnung aus, um den Benutzer darauf hinzuweisen.

7.2.2 Implementierung

Die Implementierung der Datenflussanalyse stellt sich in Haskell auf den ersten Blick schwieriger dar, als es beispielsweise in imperativen Programmiersprachen der Fall wäre. Dies liegt darin begründet, dass es in funktionalen Programmiersprachen nicht so einfach möglich ist, Datenstrukturen in-place zu verändern. Auch Zeiger auf Datenstrukturen können gar nicht oder nicht wie in imperativen Programmiersprachen verwendet werden. Mithilfe der *Functional Graph Library* [5, 6] ist es jedoch auch in Haskell sehr komfortabel und effizient möglich, Graphen zu erstellen, sie zu traversieren und zu verändern.

Dadurch ist es möglich, den in [2] dargestellten Algorithmus zur Datenflussanalyse zu implementieren. Dafür wird in einem ersten Schritt die Liste der Drei-Adress-Code-Befehle durchlaufen. Dabei werden die einzelnen Grundblöcke erzeugt und inklusive der Kanten in einen Graphen eingefügt. Für die Lebendigkeitsanalyse werden nun zu Beginn zwei Mengen pro Block konstruiert: die Menge der innerhalb des Blocks gelesener und die Menge der innerhalb des Blocks geschriebener Variablen. Sie werden im weiteren Verlauf noch benötigt. Die beiden in der Datenflussanalyse berechneten Mengen `IN` und `OUT`, die für jeden Block die am Blockeingang beziehungsweise Blockausgang lebendigen Variablen enthalten, werden initial leer gelassen.

Bei der Lebendigkeitsanalyse handelt es sich um eine sogenannte Rückwärtsanalyse. Die Informationen fließen entgegen der Richtung der Programmausführung. Bei der Implementierung der Lebendigkeitsanalyse bestimmt sich daher die Menge der am Blockausgang lebendigen Variablen `OUT` aus allen Variablen, die bei einem Nachfolgeknoten am Eingang lebendig sind. Ist eine Variable am Ausgang lebendig, so muss sie auch am

Blockeingang lebendig und in der Menge IN enthalten sein, es sei denn sie wird innerhalb des Blocks geschrieben. Zusätzlich gehen alle Variablen, die nur gelesen oder vor einem ersten Schreiben gelesen werden, in die Menge IN ein.

Da sich die OUT-Menge eines Knotens aus den IN-Mengen der Nachfolger ergibt und gleichzeitig die IN-Menge des Knotens selber verändern kann, muss der Graph mehrfach traversiert werden, um die Mengen zu aktualisieren. Erst wenn sich die Mengen durch eine Traversierung nicht mehr verändern, kann abgebrochen und der entstandene Graph zurückgegeben werden.

Ist die entsprechende Option beim Aufruf des Compiler gesetzt, werden die bei der Datenflussanalyse erzeugten Graphen auch als PDF-Dateien ausgegeben. Dafür wird die Software Graphviz [10] verwendet, für die auch ein Haskell-Paket existiert [24]. Mit dessen Hilfe ist es komfortabel möglich, die interne Graphdarstellung in eine PDF-Datei zu konvertieren.

8 Codeerzeugung

Die Phase der Codeerzeugung ist der letzte Schritt des Übersetzungsprozesses. Hier geht es darum, den relativ abstrakten Drei-Adress-Code in architekturspezifische Assemblerbefehle zu übersetzen. Dabei ist nicht nur wichtig, welche Befehle mit welchen Operanden von der gewählten Zielarchitektur unterstützt werden, sondern vor allem auch, wie Variablen im Speicher und in Registern gespeichert werden und wie auf sie zugegriffen werden kann.

Der entwickelte Compiler übersetzt in Assemblercode für den NASM-Assembler [29] für die AMD64-Architektur unter Linux.

8.1 Registerzuteilung

Eine der wichtigsten Aufgaben der Codeerzeugung besteht darin, die im Programm genutzten Variablen auf die vorhandenen Register zu verteilen, sodass Berechnungen möglichst schnell durchgeführt werden können und Zugriffe auf den Speicher nur selten vorkommen. Wie in [22] gezeigt wurde, ist das Problem der Registerzuteilung NP-vollständig, es ist also schwer, eine optimale Lösung in akzeptabler Laufzeit zu bestimmen. Nichtsdestotrotz gibt es verschiedene Möglichkeiten, gute Lösungen für das Problem zu finden.

Die am weitesten verbreitete Möglichkeit zur Registerzuteilung ist, das Problem auf das Graphfärben zu reduzieren [3]. In dieser Arbeit wurde jedoch das Verfahren „Linear scan“ [23] implementiert. Dieses Verfahren ist weniger komplex als das Graphfärben und kann daher in kürzerer Laufzeit ähnlich gute Registerzuteilungen erreichen. Da bei Programmen in der WHILE-Sprache im Allgemeinen nicht so viele Variablen gleichzeitig lebendig sind und die gewählte Zielarchitektur eine relativ große Anzahl an Registern zur Verfügung stellt, ist das „Linear scan“-Verfahren eine gute Wahl für den entwickelten Compiler.

8.1.1 „Linear scan“-Registerzuteilung

Die Idee des Verfahrens besteht darin, dass jeder im Programm verwendeten Variable x in einer vorherigen Lebendigkeitsanalyse (siehe Abschnitt 7.2.1) ein Lebendigkeitsbereich zugewiesen wurde. Dieser Bereich bezieht sich auf die Drei-Adress-Code-Instruktionen und geht von der ersten Instruktion, in der x verwendet wird, bis zur letzten Instruktion, in der x vorkommt. Die berechneten Lebendigkeitsbereiche aller Variablen werden nun anhand des Beginns sortiert. Danach wird diese Liste von vorne nach hinten durchlaufen und dabei jeder Variablen ein freies Register zugeteilt. Ist ein Lebendigkeitsbereich beendet, wird das der entsprechenden Variable zugeteilte Register wieder der Menge der freien Register hinzugefügt. In Abbildung 8.1 sind vier Variablen mit ihrem Lebendigkeitsbereich zu sehen. Hier würden drei Register ausreichen, da den Variablen p und s das gleiche Register zugeteilt werden kann.

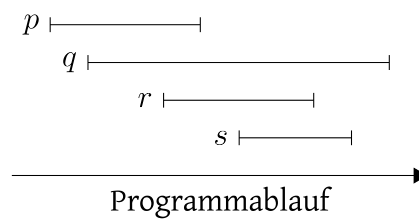


Abbildung 8.1: Beispielhafter Lebendigkeitsbereich von vier Variablen

Sollten zu einem bestimmten Zeitpunkt keine freien Register mehr vorhanden sein, so wird der Variablen, deren Lebendigkeitsbereich noch am längsten andauert, stattdessen ein Platz im Speicher reserviert und das vorher möglicherweise schon zugeteilte Register wieder freigegeben. Die Heuristik, die Variable mit dem längsten Gültigkeitsbereich in den Speicher zu schreiben, stammt ursprünglich von der Idee der statischen Einzelzuweisung, bei der jede Variable zu Beginn einmal geschrieben und dann einmal gelesen wird. Nach Aussage von Poletto und Sarkar [23, S. 899–900] scheint diese Heuristik jedoch auch für mehrfache Zugriffe gut zu funktionieren. Wären für Abbildung 8.1 nur zwei Register vorhanden, würde dies beim Versuch, r ein neues Register zuzuweisen, bemerkt. Unter den drei zu diesem Zeitpunkt aktiven Variablen endet der Lebendigkeitsbereich von q als letztes. Daher wird q ein Platz im Speicher reserviert und das vorher zugeteilte Register wird an r gegeben. Der Variablen s kann wie im ersten Beispiel einfach das dann freie Register von p zugewiesen werden.

8.1.2 Verfügbare Register und Ausnahmen

Die AMD64-Architektur stellt 16 Register zur Verfügung [28, S. 75]. Zwei dieser Register (`rbp` und `rsp`) werden jedoch zur Verwaltung des Programmstacks und der Variablen, die im Speicher liegen, benötigt. Sie können daher nicht einfach überschrieben werden. Genauso werden die beiden Register `rax` und `rdx` vom Compiler nicht zugeteilt, da für Divisionen und Modulo-Berechnungen immer diese beiden Register beschrieben werden müssen. Trotz dieser Einschränkungen bleibt eine relativ große Anzahl von zwölf Registern übrig, die den einzelnen Variablen zugewiesen werden können.

8.2 Codeerzeugung

Da der Drei-Adress-Code schon sehr nah am Assemblercode ist, sind bei der Codeerzeugung keine großen Transformationen mehr nötig. Ein Unterschied, der Berücksichtigung findet, ist jedoch, dass die verwendete Architektur für viele arithmetische Operationen nur Instruktionen in der Form `a = a + b` zur Verfügung stellt. Für eine Berechnung der Art `a = b + c` bedeutet das, dass erst ein Kopier- und dann ein Additionsbefehl erzeugt werden müssen. Weitere Besonderheiten, wie etwa die im vorigen Abschnitt erwähnte, feste Registerbelegung für die Division und Modulo-Berechnung, müssen bei der Erzeugung berücksichtigt werden.

Die für den Assemblercode verwendeten Instruktionen benötigen unterschiedlich lange für eine Ausführung. Heutige Prozessoren können beispielsweise Additionen und Subtraktionen sehr schnell, das heißt in einem Prozessortakt, berechnen. Auch Multiplikationen sind relativ schnell, wohingegen Divisionen im Vergleich dazu sehr langsam sind und ein Vielfaches an Prozessortakten dauern können [8, S. 174]. Die Multiplikationen und Divisionen mit beziehungsweise durch Zweierpotenzen entsprechen jedoch einer Verschiebung der Bits nach links oder rechts. Sollte also der eine Operand eine Konstante mit einer Zweierpotenz sein, kann der Compiler statt einer Multiplikation eine Assemblerinstruktion für eine Bitverschiebung nach links erzeugen. Somit wird die Ausführung des Programms beschleunigt. Bei Division muss zusätzlich berücksichtigt werden, dass eine Bitverschiebung nach rechts für die negativen Zahlen in Richtung $-\infty$ rundet. Dies widerspricht der in Abschnitt 3.4 beschriebenen Semantik, nach der immer in Richtung 0 gerundet wird. Das Problem lässt sich jedoch umgehen, indem vor der Bitverschiebung um n Bits auf alle negative Zahlen $2^n - 1$ aufaddiert wird.

Die Operanden der Assembler-Instruktionen müssen besonders berücksichtigt werden. Die Instruktionen unterstützen in den meisten Fällen nicht alle möglichen Kombinationen

von Registern, Speicherplätzen und direkten Zahlwerten. So ist es beispielsweise mit der `add`-Instruktion nicht möglich, einen Wert aus dem Speicher auf einen anderen Wert aus dem Speicher aufzuaddieren. In solchen Fällen wird einer der Operanden temporär in ein freies Register geschrieben.

8.3 Assembler-Framework

Für Interaktion, beispielsweise mit dem Benutzer, bringt die WHILE-Sprache die Anweisungen `read` und `output`, sowie das Sprachkonstrukt `eof` mit.

Eine einfache Möglichkeit, die Ein- und Ausgabe zu unterstützen, wäre, die in der Standard-C-Bibliothek vorhandenen Funktionen `printf` für die Ausgabe und `scanf` für die Eingabe zu verwenden [33, 35]. Dies würde jedoch bedeuten, dass nur für diese zwei Funktionen eine Abhängigkeit zur Standard-C-Bibliothek hergestellt würde. Da die Implementierung der WHILE-Sprache nur die Ein- und Ausgabe von ganzen Zahlen erlaubt, wurde die Funktionalität der drei Sprachkonstrukte komplett in NASM nachgebildet.

Für eine Interaktion mit dem Ein- bzw. Ausgabekanal stehen die beiden Systemaufrufe `sys_read` und `sys_write` zur Verfügung [4, 34, 36]. Mit `sys_read` kann ein Puffer mit den nächsten Zeichen der Eingabe befüllt werden, wobei eventuell blockierend gewartet wird. Zusätzlich wird zurückgegeben, ob der Eingabestrom zu Ende ist oder nicht. Mit `sys_write` wird der Inhalt eines Puffers in einen Ausgabestrom geschrieben.

Mithilfe dieser beiden Systemaufrufe, eines Puffers, um die Ein- beziehungsweise Ausgabe zwischenzuspeichern, sowie etwas Assemblercode, um den ganzzahligen Wert eines Registers in einen String (und umgekehrt) umzuwandeln, werden die Assembler-Unterfunktionen `input_number`, `output_number` und `eof` bereitgestellt. Die Umwandlung der entsprechenden Drei-Adress-Code-Konstrukte ist dadurch sehr leicht und besteht nur noch in der Erzeugung eines Funktionsaufrufs inklusive passender Wertzuweisung von Ein- oder Ausgabeparameter.

8.4 Erzeugung eines ausführbaren Programms

Die eigentliche Arbeit des Compilers endet nach der Erzeugung des Assemblercodes. Der Compiler nimmt jedoch dem Benutzer den letzten Schritt, aus dem Assemblercode eine ausführbare Binärdatei zu erzeugen, noch ab. Daher ruft der Compiler nach dem Schreiben einer Datei mit Assemblercode zuerst den NASM-Assembler auf, um eine Objektdatei zu erzeugen. Zuletzt wird der Linker auf der Objektdatei ausgeführt. Dadurch wird ein ausführbares Programm erzeugt.

9 Bewertung und Ausblick

Im Rahmen dieser Arbeit wurde ein Compiler entwickelt, der von der WHILE-Sprache in Assemblercode übersetzt. Dafür wurde unter anderem ein Parsergenerator entwickelt, um das aufwendige, händische Erstellen eines Bottom-Up-Parsers zu automatisieren. Der Compiler selber enthält alle für den Übersetzungsprozess wichtigen Phasen. Die einzelnen Übersetzungsphasen umfassen zum heutigen Forschungsstand jedoch große Themenbereiche, sodass beispielsweise viele mögliche Optimierungen in dieser Arbeit nicht angesprochen geschweige denn implementiert werden konnten, da es sonst den Umfang der Arbeit überstiegen hätte.

Die Arbeit hat gezeigt, dass es auch in einer funktionalen Programmiersprache sehr gut möglich ist, einen Compiler zu implementieren. Teilweise, wie beim Lexer (siehe Kapitel 4) und beim funktionalen Parser (siehe Abschnitt 5.2), bieten funktionale Sprachen auch Lösungsmöglichkeiten, die auf diese Art und Weise nicht im imperativen Programmierparadigma hätten umgesetzt werden können. Einige Phasen boten jedoch auch Schwierigkeiten auf, da die Algorithmen für verschiedene Problemlösungen in der Regel im imperativen Stil gegeben sind. Hier war es teilweise nötig, die Algorithmen abzuwandeln und sie dem funktionalen Paradigma anzupassen.

Mit Blick auf die beiden in der Arbeit implementierten Parser lässt sich sagen, dass der Bottom-Up-Parser deutlich komplexer zu implementieren war. Er kann dafür jedoch eine relativ große Menge an Sprachen erkennen und bei syntaktischen Fehlern sinnvolle Fehlermeldungen ausgeben.

Der funktionale Parser besticht hingegen durch die Idee, einzelne kleine Parser durch verschiedene Kombinatoren zu verbinden und daraus letztlich einen Parser für die vollständige WHILE-Grammatik zu gewinnen. Auch die Implementierung ist simpler und durch geeignete Tricks, wie den `chainl1`-Kombinator, können Nachteile der Top-Down-Syntaxanalyse, wie die Probleme bei Linksrekursion, umgangen werden.

9.1 Mögliche Verbesserungen

Wie schon zu Beginn der Arbeit angesprochen, war die Geschwindigkeit des Compilers kein Hauptziel der Implementierung. Dies schlägt sich beispielsweise in den Optimierungsphasen nieder, in denen teilweise mehrere Durchläufe über den Drei-Adress-Code benötigt werden, da jede Phase für sich arbeitet. Hier wäre es durch ein Verschmelzen der Optimierungen möglich, die Durchläufe durch das Programm zu verringern. Darunter würde jedoch die Übersichtlichkeit und starke Trennung der einzelnen Teile leiden.

Außerdem wurden daher in der Implementierung nicht unbedingt die optimalen Datenstrukturen verwendet. Durch die Verwendung von optimalen Datenstrukturen und einer stärkeren Anpassung der verwendeten Algorithmen auf die spezifischen Anforderungen funktionaler Programmierung wäre es mit Sicherheit möglich, die Laufzeit des Compilers zu verringern.

Für komplexere (als die im Rahmen der Arbeit implementierten) Programme in der WHILE-Sprache ist es möglich, dass der verwendete Algorithmus zur Registerzuteilung keine optimalen Ergebnisse mehr liefert. Dann wäre es möglich, das „Linear scan"-Verfahren durch die deutlich komplexere Methode der Registerzuteilung durch Graphfärben [3] zu ersetzen.

Aufgrund der Art, wie die Anweisung `read` in Assembler implementiert wurde, wartet das erzeugte Programm nach Eingabe einer Zahl auf die Eingabe einer weiteren Zahl oder das Ende der Eingabe (unter Linux per `Strg+D` möglich). Dies liegt daran, dass für die korrekte Funktionalität von `eof` durch einen erneuten Aufruf von `sys_read` geprüft werden muss, ob die Eingabe schon zu Ende ist, oder noch weitere Zahlen folgen. Durch geeignete Verbesserungen ist es mit Sicherheit möglich, dieses unschöne Verhalten zu verhindern.

9.2 Ausblick

Für eine eventuelle Weiterentwicklung des Compilers stehen viele Möglichkeiten offen. Beispielsweise wäre es wünschenswert, den Sprachumfang der WHILE-Sprache zu erweitern und Arrays, Verbundtypen (structs) und Gleitkommazahlen als Datentypen zu ermöglichen. Dafür müsste unter anderem eine Symboltabelle eingeführt werden, um die Datentypen der Variablen zu speichern. Eine weitere sinnvolle Erweiterung der Sprache wäre das Einführen von Funktionen, was unter anderem auch rekursive Problemlösungen erstmals erlauben würde.

Auch im Bereich der Optimierungen bestehen noch einige Möglichkeiten, die ausgeschöpft werden könnten. Zum Beispiel wäre es möglich, gemeinsame Teilausdrücke in arithmetischen Ausdrücken zu erkennen und dadurch nicht doppelt zu berechnen. Ein weiterer sinnvoller Optimierungsschritt besteht darin, Berechnungen, die innerhalb einer Schleife stattfinden, soweit es geht aus der Schleife herauszunehmen. Wird beispielsweise in jedem Schleifendurchgang ein Wert berechnet, der sich nicht ändert, da seine Komponenten innerhalb der Schleife nicht verändert werden, so könnte man die Berechnung einmal vor der Schleife ausführen und danach nur noch auf den vorberechneten Wert zurückgreifen.

Durch den modularen Aufbau des Compilers sollte es möglich sein, diese Erweiterungen ohne große Änderungen an den bestehenden Modulen einzubinden.

A Anhang

A.1 Quellcode

Der Haskell-Quellcode sowie eine ausführbare Version des Compilers und des entwickelten Parsergenerators sind auf der beiliegenden CD zu finden.[1] Entwickelt wurde mit dem Glasgow Haskell Compiler (GHC) [31] in der Version 7.8.3. Weitere Informationen zu den benötigten Paketen und anderen Abhängigkeiten sind in `compiler.cabal`, der zugehörigen Konfigurationsdatei für Cabal[2], zu finden.

A.2 Der Compiler

Der Anhang enthält zunächst in Anhang A.2.1 eine Übersicht der vom Compiler unterstützten Kommandozeilenargumente. Im Allgemeinen wird dem Compiler eine WHILE-Datei übergeben, welche dann in eine ausführbare Datei kompiliert wird. Verschiedene Optionen können das Verhalten und die Ausgaben des Compilers beeinflussen.

```
Usage: compiler [OPTION ..] file
  -h        --help           Show this help and exit
  -o FILE   --output=FILE    Output file (default: derived from input file)
  -i        --no-cleanup     Do not clean up intermediate .asm and .o files
  -d        --debug          Output debug information
  -g        --graphs         Output graphs for various compiling phases
  -f        --dot-files      Output graphs as graphviz dot files
  -f        --p-functional   Use the functional parser
  -b        --p-bottom-up    Use the bottom-up parser (default)
```

Anhang A.2.1: Kommandozeilenparameter des Compilers

[1] Alternativ unter `https://page.mi.fu-berlin.de/jonascleve/ba-compiler.tar.gz` mit der SHA-1-Prüfsumme `ffc6e333902d64af5e1ff9584a945c0be5fede02`.

[2] Cabal [30] ist eine Software, die das Bauen und Verwalten von Haskell-Paketen und -Programmen unterstützt und vereinfacht.

A.3 Compilerausgaben für ein Beispielprogramm

Anhang A.3.1 zeigt ein Beispielprogramm, welches für alle über die Standardeingabe eingegebenen Zahlen das Maximum und Minimum bestimmt und am Ende ausgibt. Im Folgenden werden die einzelnen vom Compiler für dieses Programm erzeugten Zwischenschritte gezeigt und kurz erläutert.

```
1  # Outputs the minimum and maximum of all numbers in input. It assumes that there
2  # is at least one number < 1000000 and one number > -1000000.
3  max := -1000000;
4  min :=  1000000;
5  while not eof do {
6      read x;
7      if x > max then max := x;
8      if x < min then min := x;
9  };
10 output min;
11 output max;
```

Anhang A.3.1: Beispielprogramm zur Ermittlung des größten und kleinsten Werts aller eingelesenen Zahlen

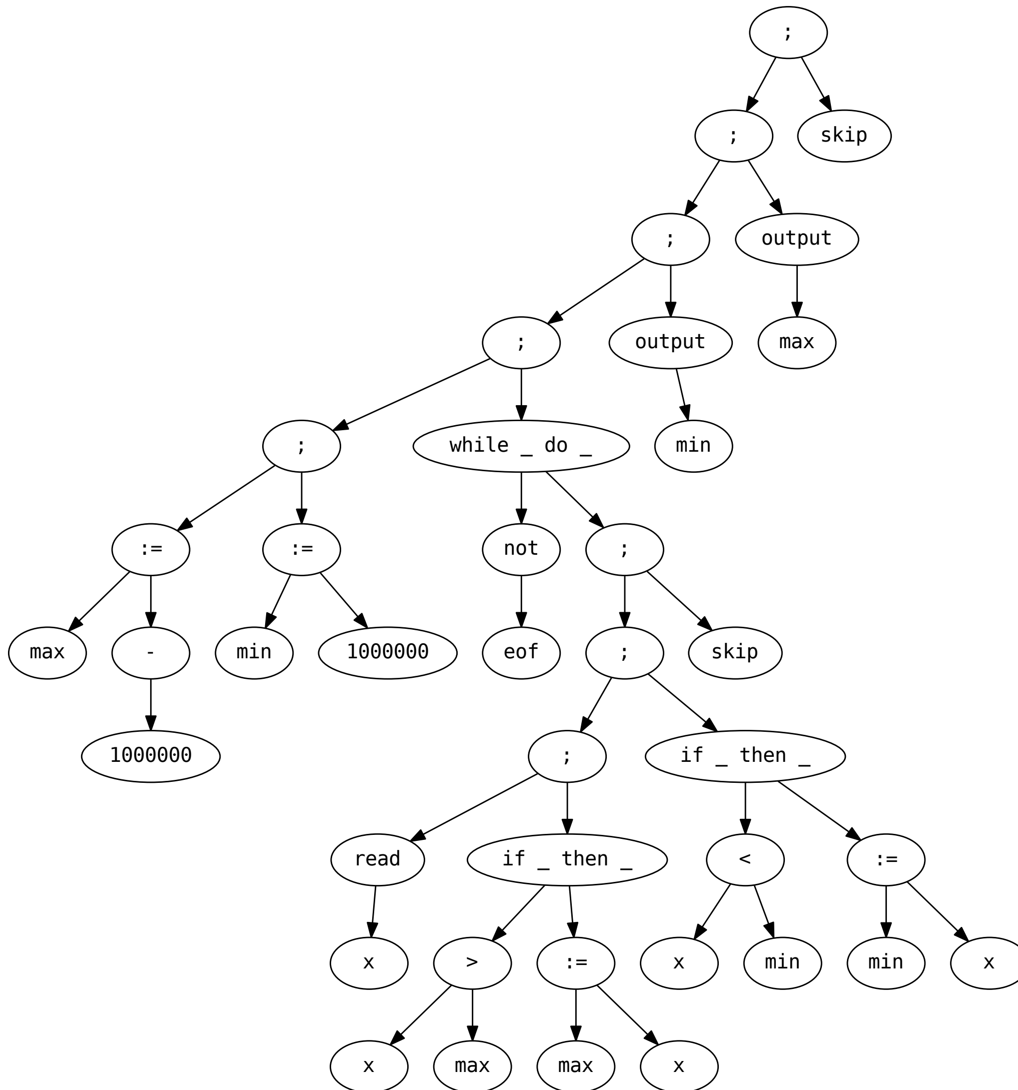
Zuerst kann in Anhang A.3.2 der von der lexikalischen Analyse erzeugte Tokenstrom betrachtet werden. Es ist gut zu sehen, dass jegliche Leerzeichen und Kommentare ignoriert werden und nur noch semantisch bedeutsame Teile des Quelltextes als Token vorhanden sind.

```
<id,"max"> <:=> <mathop,-> <num,1000000> <;> <id,"min"> <:=> <num,1000000> <;>
<while> <not> <eof> <do> <{> <read> <id,"x"> <;> <if> <id,"x"> <relop,>>
<id,"max"> <then> <id,"max"> <:=> <id,"x"> <;> <if> <id,"x"> <relop,<>
<id,"min"> <then> <id,"min"> <:=> <id,"x"> <;> <}> <;> <output> <id,"min"> <;>
<output> <id,"max"> <;>
```

Anhang A.3.2: Aufteilung in Token

Der nächste Schritt nach der lexikalischen Analyse ist die syntaktische Analyse oder der Parsevorgang. Ein während des Parsens erzeugter abstrakter Syntaxbaum ist in Anhang A.3.3 zu sehen. Er enthält jedoch noch Optimierungsmöglichkeiten, wie etwa die überflüssigen `skip`-Anweisungen, die durch überflüssige Semikola im Quelltext entstanden sind.

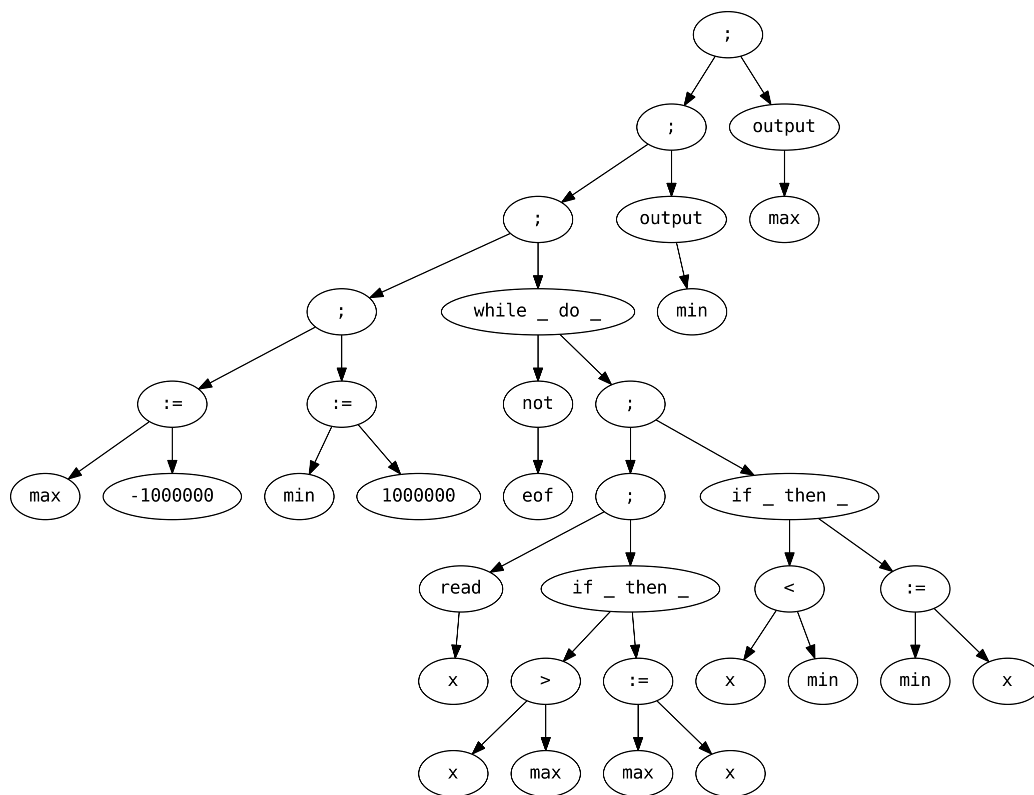
Anhang A.3.4 zeigt den optimierten Syntaxbaum, auf den die in Abschnitt 5.3 angesprochenen Optimierungen angewendet worden sind.



Anhang A.3.3: Abstrakter Syntaxbaum

Aus dem optimierten Syntaxbaum wird dann der in Anhang A.3.5a gezeigte Zwischencode erzeugt. Auf diesen Zwischencode wird zuerst die lineare und die Guckloch-Optimierung angewendet, wodurch Anhang A.3.5b entsteht.

Nach dem ersten Optimierungsschritt wird das Programm in einen Flussgraphen umgewandelt, wie er in Anhang A.3.6 zu sehen ist. An diesem sind Verzweigungen und Schleifen im Programm gut zu erkennen.



Anhang A.3.4: Optimierter abstrakter Syntaxbaum

Der Flussgraph wird daraufhin verwendet, um die Lebensdauer von Variablen zu analysieren. Anhang A.3.7 zeigt das Ergebnis der Analyse. Jeder Block enthält zusätzlich eine Menge von Variablen, die am Blockeingang lebendig sind, sowie eine Menge von Variablen, die am Blockausgang lebendig sind.

Zum Schluss wird der Flussgraph wieder linearisiert und dann für jede Drei-Adress-Code-Anweisung einzeln der Assembler-Code erzeugt. Zusätzlich zu dem erzeugten Assembler-Code wird jeweils die Drei-Adress-Code-Anweisung als Kommentar ausgegeben, sodass die Codeerzeugung einfach nachvollzogen werden kann. Für die Registerzuteilung werden die Daten der Lebensdauer-Analyse verwendet. Der erzeugte Assembler-Code findet sich in Anhang A.3.8.

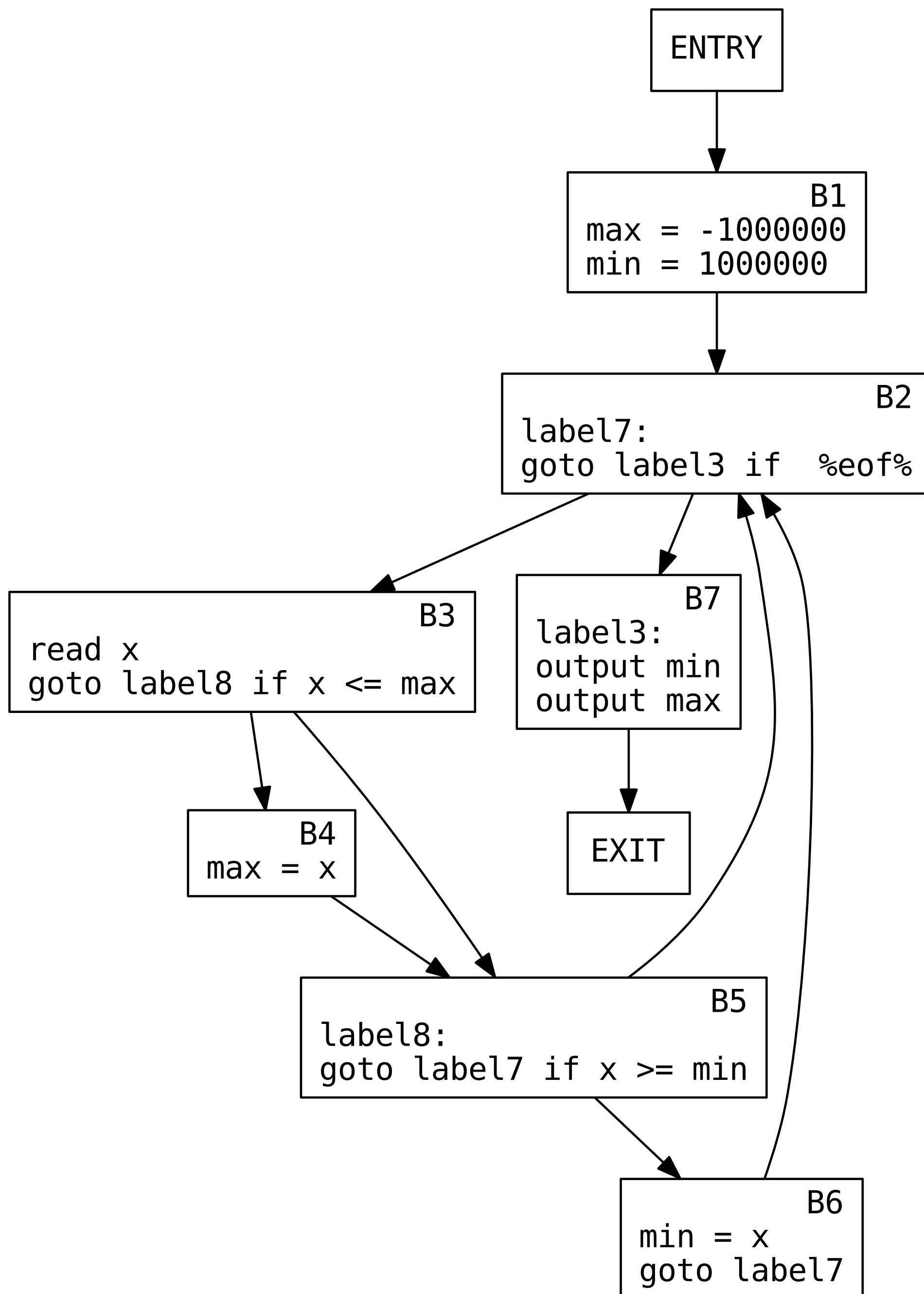
```
1  max = -1000000
2  label5:
3  min = 1000000
4  label4:
5  label7:
6  goto label3 if  %eof%
7  goto label6
8  label6:
9  read x
10 label9:
11 goto label10 if x > max
12 goto label8
13 label10:
14 max = x
15 label8:
16 goto label11 if x < min
17 goto label7
18 label11:
19 min = x
20 goto label7
21 label3:
22 output min
23 label2:
24 output max
25 label1:
```

(a) ohne Optimierung

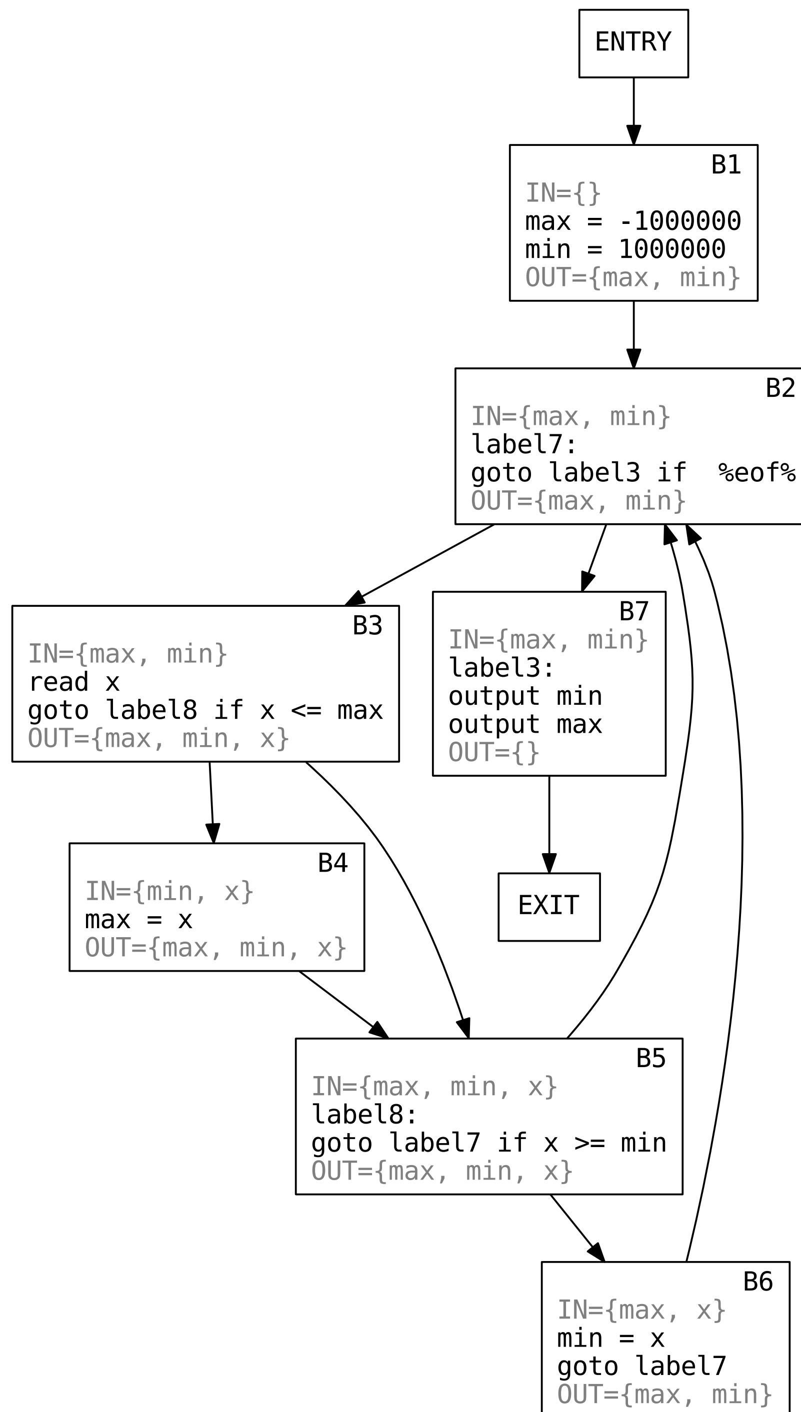
```
1  max = -1000000
2  min = 1000000
3  label7:
4  goto label3 if  %eof%
5  read x
6  goto label8 if x <= max
7  max = x
8  label8:
9  goto label7 if x >= min
10 min = x
11 goto label7
12 label3:
13 output min
14 output max
```

(b) nach der linearen Optimierung

Anhang A.3.5: Erzeugter Zwischencode



Anhang A.3.6: Flussgraph des Programms



Anhang A.3.7: Lebensdauer-Analyse der Variablen

```
mov rbp, rsp
; max = -1000000
mov rbx, -1000000
; min = 1000000
mov rcx, 1000000
; label7:
.label7:
; goto label3 if  %eof%
call eof
cmp QWORD rax, 0
jne .label3
; read x
call input_number
mov rsi, rax
; goto label8 if x <= max
cmp rsi, rbx
jle .label8
; max = x
mov rbx, rsi
; label8:
.label8:
; goto label7 if x >= min
cmp rsi, rcx
jge .label7
; min = x
mov rcx, rsi
; goto label7
jmp .label7
; label3:
.label3:
; output min
mov rax, rcx
call output_number
; output max
mov rax, rbx
call output_number
```

Anhang A.3.8: Das fertige NASM-Programm

B Literaturverzeichnis

[1] Paul W. Abrahams. „A Final Solution to the Dangling else of ALGOL 60 and Related Languages“. In: *Communications of the ACM* 9.9 (Sep. 1966), S. 679–682. ISSN: 0001-0782. DOI: 10.1145/365813.365821.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi und Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2. Aufl. Boston: Addison Wesley, 2007. ISBN: 978-0-321-48681-3.

[3] Preston Briggs. „Register Allocation via Graph Coloring“. Diss. Houston, TX, USA: Rice University, 1992.

[4] Ryan A. Chapman. *Linux System Call Table for x86_64*. 29. Nov. 2012. URL: http://blog.rchapman.org/post/36801038863/linux-system-call-table-for-x86-64 (besucht am 04. 09. 2014).

[5] Martin Erwig. „Inductive Graphs and Functional Graph Algorithms“. In: *Journal of Functional Programming* 11.5 (Sep. 2001), S. 467–492. ISSN: 0956-7968. DOI: 10.1017/S0956796801004075.

[6] Martin Erwig und Ivan Lazar Miljenovic. *fgl: Martin Erwig's Functional Graph Library*. Version 5.5.0.1. 28. Apr. 2014. URL: http://hackage.haskell.org/package/fgl (besucht am 02. 11. 2014).

[7] Elfriede Fehr. *Semantik von Programmiersprachen*. Berlin: Springer, 7. März 1989. 202 S. ISBN: 978-3-540-15163-0.

[8] Agner Fog. *4. Instruction tables. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. Technical University of Denmark, 2014. URL: http://www.agner.org/optimize/instruction_tables.pdf (besucht am 29. 10. 2014).

[9] Jeroen Fokker. „Functional parsers“. In: *Advanced Functional Programming*. Hrsg. von Johan Jeuring und Erik Meijer. Bd. 925. Lecture Notes in Computer Science. Berlin: Springer, 1995, S. 1–23. ISBN: 978-3-540-59451-2. DOI: 10.1007/3-540-59451-5_1.

[10] Emden R. Gansner und Stephen C. North. „An open graph visualization system and its applications to software engineering". In: *Software Practice and Experience* 30.11 (2000), S. 1203–1233. URL: http://www.graphviz.org/Documentation/GN99.pdf (besucht am 02. 11. 2014).

[11] Andy Gill und Simon Marlow. *Happy: The Parser Generator for Haskell*. URL: http://www.haskell.org/happy/ (besucht am 15. 10. 2014).

[12] Ralf Hartmut Güting und Martin Erwig. *Übersetzerbau - Techniken, Werkzeuge, Anwendungen*. Auflage: 1999. Berlin: Springer, 1999. 384 S. ISBN: 978-3-540-65389-9.

[13] Naomi Hamilton. *The A-Z of Programming Languages: YACC*. TechWorld. 9. Juli 2008. URL: http://www.techworld.com.au/article/252319/a-z_programming_languages_yacc/ (besucht am 15. 10. 2014).

[14] Paul Hudak, John Hughes, Simon Peyton Jones und Philip Wadler. „A History of Haskell: Being Lazy with Class". In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: ACM, 2007, S. 12.1–12.55. ISBN: 978-1-59593-766-7. DOI: 10.1145/1238844.1238856.

[15] Graham Hutton. *Programming in Haskell*. Cambridge: Cambridge University Press, 15. Jan. 2007. 184 S. ISBN: 978-1-139-46122-1.

[16] Graham Hutton und Erik Meijer. *Monadic parser combinators*. Nottingham: Department of Computer Science, University of Nottingham, 1996. URL: http://eprints.nottingham.ac.uk/237/ (besucht am 04. 09. 2014).

[17] Graham Hutton und Erik Meijer. „Monadic parsing in Haskell". In: *Journal of Functional Programming* 8.04 (1998), S. 437–444. DOI: 10.1017/S0956796898003050.

[18] Stephen C. Johnson. *Yacc: Yet Another Compiler-Compiler*. Murray Hill, New Jersey: Bell Laboratories, 31. Juli 1978. URL: http://www.cs.umanitoba.ca/~comp3290/Docs/yacc.pdf (besucht am 16. 10. 2014).

[19] Donald E. Knuth. „On the translation of languages from left to right". In: *Information and Control* 8.6 (1965), S. 607–639. ISSN: 0019-9958. DOI: 10.1016/S0019-9958(65)90426-2.

[20] Daan Leijen und Erik Meijer. *Parsec: Direct style monadic parser combinators for the real world*. UU-CS-2001-27. Utrecht: Department of Computer Science, Universiteit Utrecht, 2001. URL: http://research.microsoft.com/pubs/65201/parsec-paper-letter.pdf (besucht am 04. 09. 2014).

[21] Flemming Nielson, Hanne R. Nielson und Chris Hankin. *Principles of Program Analysis*. Berlin: Springer, 2005. 482 S. ISBN: 978-3-540-65410-0.

[22] Fernando Magno Quintão Pereira und Jens Palsberg. „Register allocation after classical SSA elimination is NP-complete“. In: *Foundations of Software Science and Computation Structures*. Springer, 2006, S. 79–93. ISBN: 978-3-540-33045-5. DOI: 10.1007/11690634_6.

[23] Massimiliano Poletto und Vivek Sarkar. „Linear scan register allocation“. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21.5 (1999), S. 895–913. DOI: 10.1145/330249.330250.

[24] Matthew Sackman und Ivan Lazar Miljenovic. *graphviz: Bindings to Graphviz for graph visualisation*. Version 2999.17.0.1. 19. Mai 2014. URL: http://hackage.haskell.org/package/graphviz (besucht am 02. 11. 2014).

[25] Richard M. Stallman und The GCC Developer Community. *GNU Compiler Collection Internals. For gcc version 5.0.0 (pre-release)*. Free Software Foundation, Inc., 2014. URL: https://gcc.gnu.org/onlinedocs/gccint.pdf (besucht am 07. 10. 2014).

[26] Richard M. Stallman und The GCC Developer Community. *Using the GNU Compiler Collection. For gcc version 4.9.1*. Free Software Foundation, Inc., 2014. URL: https://gcc.gnu.org/onlinedocs/gcc-4.9.1/gcc.pdf (besucht am 08. 10. 2014).

[27] Philip Wadler. „Monads for functional programming“. In: *Advanced Functional Programming*. Hrsg. von Johan Jeuring und Erik Meijer. Bd. 925. Lecture Notes in Computer Science. Berlin: Springer, 1995, S. 24–52. ISBN: 978-3-540-59451-2. DOI: 10.1007/3-540-59451-5_2.

[28] *AMD64 Architecture Programmer's Manual - Volume 3: General-Purpose and System Instructions*. Advanced Micro Devices, Inc. Mai 2013. URL: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2008/10/24594_APM_v3.pdf (besucht am 28. 09. 2014).

[29] *NASM — The Netwide Assembler*. 21. Mai 2014. URL: http://www.nasm.us/xdoc/2.11.05/nasmdoc.pdf (besucht am 03. 10. 2014).

[30] *The Haskell Cabal*. Common Architecture for Building Applications and Libraries. URL: https://www.haskell.org/cabal/.

[31] *The Glasgow Haskell Compiler*. URL: https://www.haskell.org/ghc/.

[32] *GCC Front Ends*. 12. Juni 2014. URL: https://gcc.gnu.org/frontends.html (besucht am 08. 10. 2014).

[33] *printf(3): formatted output conversion - Linux man page*. URL: http://linux.die.net/man/3/printf (besucht am 02. 09. 2014).

[34] *read(2): read from file descriptor - Linux man page*. URL: http://linux.die.net/man/2/read (besucht am 02. 09. 2014).

[35] *scanf(3): input format conversion - Linux man page*. URL: http://linux.die.net/man/3/scanf (besucht am 02. 09. 2014).

[36] *write(2): write to file descriptor - Linux man page*. URL: http://linux.die.net/man/2/write (besucht am 02. 09. 2014).

C Abbildungsverzeichnis

1.1 Übersicht über die Phasen des Compilers 2
1.2 Struktur der Haskell-Module 4

5.1 Beispiel für die Umwandlung eines Tokenstroms in eine Baumstruktur . 17
5.2 Parsevorgang auf der Eingabe `id * id` 19
5.3 LR(0)-Automat für die Grammatik aus Tabelle 5.1 22
5.4 Schematischer Aufbau eines LR-Parsers 23
5.5 Modul-Struktur des Parsergenerators 27
5.6 Schematische Darstellung der sequentiellen Ausführung von `p` und `q` . . 32
5.7 Grafische Darstellung des Parsers `p <|> q` 33

6.1 Schematische Darstellung des Springcodes 43

8.1 Beispielhafter Lebendigkeitsbereich von vier Variablen 54

D Tabellenverzeichnis

3.1 Lexikalischer Aufbau der WHILE-Sprache 9
3.2 Simple Grammatik der WHILE-Sprache 9
3.3 Eindeutige Grammatik für die WHILE-Sprache 10

5.1 Beispiel-Grammatik für einfache mathematische Ausdrücke 18
5.2 Parservorgang eines Shift-Reduce-Parsers auf der Eingabe `id * id` 19
5.3 Erweiterte Beispiel-Grammatik für einfache mathematische Ausdrücke mit Nummerierung der Produktionen 21
5.4 ACTION- & GOTO-Tabelle zur erweiterten Beispielgrammatik 24

6.1 Ausschnitt aus einer SDD zur Erzeugung des Springcodes 42

E Quelltextverzeichnis

3.1 WHILE-Programm zur Berechnung der Summe der eingelesenen Zahlen 8

4.1 Einfaches Eingabeprogramm und zugehörige Zerlegung in Token 13
4.2 Definition des Datentyps für die generierten Token 15
4.3 Erkennung des Zuweisungsoperators im Lexer 15
4.4 Erkennung von Bezeichnern und Schlüsselwörtern durch den Lexer . . . 16

5.1 Ausschnitte aus der WHILE-Grammatikdefinition für den Parsergenerator 26
5.2 In Haskell implementierte GOTO-Funktion 28
5.3 Die `parse`-Funktion 29
5.4 Ausschnitt der Zustandsübergänge des erzeugten Parsers 29
5.5 Ausschnitt der Reduktionen des erzeugten Parsers 30
5.6 Haskell-Datentyp für den funktionalen Parser 31
5.7 Monadische Instanzen des Parsers zur einfacheren Verwendung 32
5.8 Ausschnitt aus dem funktionalen Parser für *cmd* 35
5.9 Kombinator `chainl1` zur linksassoziativen Verkettung 35
5.10 Verwendung des `chainl1`-Kombinators im *expr*-Parser 36
5.11 Eliminierung unnötiger Berechnungen im AST 37

6.1 Linearisierung der Baumstruktur eines arithmetischen Ausdrucks 40
6.2 Ausschnitt aus einem SDT zur Berechnung arithmetischer Operationen . 41
6.3 Zwischencode für eine einfache `if`-Anweisung 44
6.4 Funktionsdeklarationen zur Drei-Adress-Code-Erzeugung 45
6.5 Ausschnitt aus dem Modul zur Erzeugung des Drei-Adress-Code 45

7.1 Zwischencodeoptimierung zu `if a = b then c := d + 1` 48
7.2 Überflüssiger unbedingter Sprung 48
7.3 Entfernen unbedingter Sprünge auf direkt nachfolgende Sprungmarken 49
7.4 Entfernen von Sprungmarken, die nie angesprungen werden 50