Albert Ludwigs Universität Freiburg

Technische Fakultät

PicoC-Compiler

Übersetzung einer Untermenge von C in den Befehlssatz der RETI-CPU

Bachelorarbeit

Abgabedatum: 28th April 2022

Author:
Jürgen Mattheis

Gutachter:
Prof. Dr. Scholl

Betreung:
M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für
Betriebssysteme

ERKLÄRUNG

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Definitionen

1 Motivation

1.1 PicoC und RETI

1.2 Aufgabenstellung

1.3 Eigenheiten der Sprache C

1.4 Richtlinien

2 Einführung

2.1 Compiler und Interpreter

Definition 1: Compiler

Definition 2: Interpreter

2.1.1 T-Diagramme

Definition 3: T-Diagram

2.2 Grammatiken

2.3 Grundlagen

Definition 4: Sprache

Definition 5: Chromsky Hierarchie

Definition 6: Grammatik

Definition 7: Reguläre Sprachen

Definition 8: Kontextfreie Sprachen

2.3.1 Mehrdeutige Grammatiken

Definition 9: Ableitungsbaum

Definition 10: Mehrdeutige Grammatik

2.3.2 Präzidenz und Assoziativität

Definition 11: Assoziativität

Definition 12: Präzidenz

2.4 Lexikalische Analyse

Die **Lexikalische Analyse** bildet üblicherweise die erste Ebene innerhalb der **Pipe Architektur** bei der Implementierung von Compilern. Die Aufgabe der lexikalischen Analyse ist vereinfacht gesagt, in einem Inputstring, z.B. dem Inhalt einer Datei, welche in **UTF-8** codiert ist, Folgen endlicher Symbole (auch **Wörter** genannt) zu finden, die bestimmte **Pattern** (Definition 13) matchen, die durch eine **reguläre Grammatik** spezifiziert sind.

Definition 13: Pattern

***Beschreibung** aller möglichen **Lexeme** einer Menge $\mathbb{P}_T$, die einem bestimmten **Token** T zugeordnet werden. Die Menge $\mathbb{P}_T$ ist eine möglicherweise unendliche Menge von **Wörtern**, die sich mit den Produktionen einer **regulären Grammatik** G_{Lex} einer **regulären Sprache** L_{Lex} beschreiben lassen [a], die für die Beschreibung eines **Tokens** T zuständig sind.* [b]

[a] Als Beschreibungswerkzeug können aber auch z.B. reguläre Ausdrücke hergenommen werden.
[b] *What is the difference between a token and a lexeme?*

Diese Folgen endlicher Symoble werden auch **Lexeme** (Definition 14) genannt.

Definition 14: Lexeme

*Ein **Lexeme** ist ein **Wort** aus dem Inputstring, welches das **Pattern** für eines der **Token** T einer **Sprache** L_{Lex} matched.* [a]

[a] *What is the difference between a token and a lexeme?*

Diese **Lexeme** werden vom **Lexer** im **Inputstring** identifziert und **Tokens** T zugeordnet (Definition 15). Die **Tokens** sind es, die letztendlich an die **Syntaktische Analyse** weitergegeben werden.

Definition 15: Lexer (bzw. Scanner)

*Ein **Lexer** ist eine **partielle** Funktion $lex : \Sigma^* \rightharpoonup (N \times W)^*$, welche ein **Wort** aus Σ^* auf ein **Token** T mit einem **Tokennamen** N und einem **Tokenwert** W abbildet, falls diese Folge von Symbolen sich unter der **regulären Grammatik** G_{Lex}, der **regulären Sprache** L_{Lex} ableiten lässt.*[a]

[a] *lecture-notes-2021*.

Ein **Lexer** ist im Allgemeinen eine **partielle Funktion**, da es Zeichenfolgen geben kann, die kein **Pattern** eines **Tokens** der Sprache L_{Lex} matchen. In Bezug auf eine Implementierung, wird, wenn der Lexer Teil der Implementierung eines Compilers ist, in diesem Fall eine **Fehlermeldung** ausgegeben.

Eine weitere Aufgabe der **Lekikalischen Analyse** ist es jegliche für die Weiterverarbeitung unwichtigen Symbole, wie Leerzeichen ␣, Newline `\n`[1] und Tabs `\t` aus dem Inputstring herauszufiltern. Das geschieht mittels des **Lexers**, der allen für die **Syntaktische Analyse** unwichtige Zeichen das leere Wort ϵ zuordnet. Das ist auch im Sinne der Definition, denn $\epsilon \in \Sigma^*$. Nur das, was für die **Syntaktische Analyse** wichtig ist, soll weiterverarbeitet werden, alles andere wird herausgefiltert.

Der Grund warum nicht einfach nur die **Lexeme** an die **Syntaktische Analyse** weitergegeben werden und der Grund für die Aufteilung des **Tokens** in **Tokenname** und **Tokenwert** ist, weil z.B. die Bezeichner von Variablen, Konstanten und Funktionen beliebige Zeichenfolgen sein können, wie `my_fun`, `my_var` oder `my_const` und es auch viele verschiedenen Zahlen gibt, wie `42`, `314` oder `12`. Die Überbegriffe bzw. Tokennamen für beliebige Bezeichner von Variablen, Konstanten und Funktionen und beliebige Zahlen sind aber trotz allem z.B. `Zahl` und `Bezeichner`.

Ein **Lexeme** ist damit aber nicht das gleiche, wie der **Tokenwert**, denn z.B. im Falle von PicoC kann z.B. der Wert 99 durch zwei verschiedene Literale darstellt werden, einmal als ASCII-Zeichen `'c'` und des Weiteren auch in Dezimalschreibweise als `99`[2]. Der **Tokenwert** ist jedoch der letztendliche Wert an sich, unabhängig von der Darstellungsform.

Die **Grammatik** G_{Lex}, die zur Beschreibung der Token T einer regulären Sprache L_{Lex} verwendet wird, ist üblicherweise **regulär**, da ein typischer **Lexer** immer nur **ein Symbol** vorausschaut[3], unabhängig davon, was für Symbole davor aufgetaucht sind. Die übliche Implementierung eines **Lexers** merkt sich nicht, was für Symbole davor aufgetaucht sind.

[1] In Unix Systemen wird für Newline das ASCII Symbol **line feed**, in Windows hingegen die ASCII Symbole **carriage return** und **line feed** nacheinander verwendet. Das wird aber meist durch die verwendete Porgrammiersprache, die man zur Inplementierung des Lexers nutzt wegabstrahiert.

[2] Die Programmiersprache Python erlaubt es z.B. diesern Wert auch mit den Literalen `0b1100011` und `0x63` darzustellen.

[3] Man nennt das auch einem **Lookahead** von 1

Um Verwirrung verzubäugen ist es wichtig folgende Unterscheidung hervorzuheben: Wenn von **Symbolen** die Rede ist, so werden in der **Lexikalischen Analyse**, der **Syntaktische Analyse** und der **Code Generierung**, auf diesen verschiedenen Ebenen unterschiedliche Konzepte als Symbole bezeichnet.

In der Lexikalischen Analyse sind einzelne **Zeichen eines Zeichensatzes** die Symbole.

In der Syntaktischen Analyse sind die **Tokennamen** die Symbole.

In der Code Generierung sind die **Bezeichner von Variablen, Konstanten und Funktionnen** die Symbole[a].

[a] Das ist der Grund, warum die Tabelle, in der Informationen zu Identifiern gespeichert werden aus Kapitel 3 Symboltabelle genannt wird.

Definition 16: Literal

*Eine von möglicherweise vielen weiteren **Darstellungsformen** für ein und denselben **Wert**.*

2.5 Syntaktische Analyse

In der **Syntaktischen Analyse** ist für einige Sprachen eine **Kontextfreie Grammatik** G_{Parse} notwendig, um diese Sprache zu beschreiben, da viele Programmiersprachen z.B. für **Funktionsaufrufe** `fun(arg)` und **Codeblöcke** `if(1){}` syntaktische Mittel verwenden, die es notwendig machen sich zu merken wieviele öffnende Klammern `'('` bzw. öffnende geschweifte Klammern `'{'` es momentan gibt, die noch nicht durch eine enstsprechende schließende Klammer `')'` bzw. schließende geschweifte Klammer `'}'` geschlossen wurden.

Die **Syntax**, in welcher der **Inputstring** aufgeschrieben ist, wird auch als **Konkrette Syntax** (Definition 17) bezeichnet. In einem Zwischenschritt, dem **Parsen** wird aus diesem Inputstring mithilfe eines **Parsers** (Definition 19), ein **Derivation Tree** (Definition 18) generiert, der als Zwischenstufe hin zum einem **Abstrakt Syntax Tree** (Definition 24) dient. Für einen ordentlichen Code ist es vor allem im Compilerbau förderlich kleinschrittig vorzugehen, deshalb erst die Generierung des **Derivation Tree** und dann der **Abstrakt Syntax Tree**.

Definition 17: Konkrette Syntax

***Syntax** einer **Sprache**, die durch die **Grammatiken** G_{Lex} und G_{Parse} zusammengenommen beschrieben wird.*

*Ein **Programm** in seiner **Textrepräsentation**, wie es in einer Textdatei nach den Produktionen der **Grammatiken** G_{Lex} und G_{Parse} abgeleitet steht, bevor man es kompiliert, ist in **Konkretter Syntax** aufgeschrieben.*[a]

[a] *Course Webpage for Compilers (P423, P523, E313, and E513).*

Definition 18: Derivation Tree (bzw. Parse Tree)

Compilerinterne Darstellung *eines in* ***Konkretter Syntax*** *geschriebenen Inputstrings als* ***Baumdatenstruktur****, in der* ***Nichtterminalsymbole*** *die* ***Inneren Knoten*** *des Baumes und* ***Terminalsymbole*** *die* ***Blätter*** *des Baumes bilden. Jede* ***Produktions*** *der* ***Grammatik*** G_{Parse}*, die ein Teil der* ***Konkrette Syntax*** *ist, wird zu einem eigenen* ***Knoten****.*

Der ***Derivation Tree*** *wird optimalerweise immer so konstruiert bzw. die* ***Konkrette Syntax*** *immer so definiert, dass sich möglichst einfach ein* ***Abstrakt Syntax Tree*** *daraus konstruieren lässt.*

Definition 19: Parser

Ein Programm, dass eine ***Eingabe*** *in eine für die* ***Weiterverbeitung*** *taugliche Form bringt.*

19.1: *In Bezug auf Compilerbau ist ein* ***Parser*** *ein Programm, dass einen Inputstring von* ***Konkretter Syntax*** *in die compilerinterne Darstellung eines* ***Derivation Tree*** *übersetzt, was auch als* ***Parsen*** *bezeichnet wird*[a].[b]

[a] Es gibt allerdings auch alternative Definitionen, denen nach ein Parser in Bezug auf Compilerbau ein Programm ist, dass einen Inputstring von **Konkretter Syntax** in **Abstrakte Syntax** übersetzt. Im Folgenden wird allerdings die obigte Definition 19.1 verendet.

[b] *Compiler Design - Phases of Compiler*.

An dieser Stelle könnte möglicherweise eine Begriffsverwirrung enstehen, ob ein **Lexer** nach der obigen Definition nicht auch ein **Parser** ist.

In Bezug auf Compilerbau ist ein **Lexer** ein Teil eines Parsers. Der Parser vereinigt sowohl die **Lexikalische Analyse**, als auch einen Teil der **Syntaktischen Analyse** in sich. Aber für sich isoliert, ohne Bezug zu Compilerbau betrachtet, ist ein Lexer nach Definition 19 ebenfalls ein Parser. Aber im Compilerbau hat **Parser** eine spezifischere Definition und hier überwiegt beim **Lexer** seine Funktionalität, dass er den Inputstring lexikalisch weiterverarbeitet, um ihn als Lexer zu bezeichnen, der Teil eines Parsers ist.

Die vom **Lexer** im Inputstring identifizierten **Token** werden in der **Syntaktischen Analyse** vom **Parser** (Definition 19) als **Wegweiser** verwendet, da je nachdem, in welcher Reihenfolge die **Token** auftauchen, dies einer anderen Ableitung in der **Grammatik** G_{Parse} entspricht. Dabei wird in der Grammatik nach dem **Tokennamen** unterschieden und nicht nach dem Tokenwert, da es nur von Interesse ist, ob an einer bestimmten Stelle z.B. eine `Zahl` steht und nicht, welchen konkretten Wert diese `Zahl` hat. Der **Tokenwert** ist erst später in der **Code Generierung** in 2.6 relevant.

Ein **Parser** ist einerseits ein erweiterter **Recognizer**, denn zum einen hat der **Parser** die Aufgabe eines **Recognizers** (Definition 20), und des Weiteren

Definition 20: Recognizer (bzw. Erkenner)

Entspricht dem Maschinenmodell eines ***Automaten****. Im Bezug auf Compilerbau entspricht der* ***Recognizer*** *einem* ***Kellerautomaten****, in dem* ***Wörter*** *bestimmter* ***Kontextfreier Sprachen*** *erkannt werden. Der* ***Recognizer*** *erkennt, ob ein Iputstring bzw. Wort sich mit den Produktionen der* ***Konkrette Syntax*** *ableiten lässt, also ob er bzw. es Teil der Sprache ist, die von der* ***Konkretten Syntax*** *beschrieben wird oder nicht.*

Für das **Parsen** gibt es verschiedene Ansätze:

- **Top-Down Parsing:** asdf.
  - asd
- **Bottom-Up Parsing:** Es wird mit dem Inputstring gestartet und versucht diesen durch Rückwärtsanwenden der **Produktionen** der **Konkretten Syntax** bis zum **Startsymbol** zurückzuverfolgen.
  - LR Parsing:

Der **Abstrakt Syntax Tree** wird mithilfe von **Transformern** (Definition 21) und **Visitors** (Definition 22) generiert und ist das Endprodukt der **Syntaktischen Analyse**. Wenn man die gesamte **Syntaktische Analyse** betrachtet, so übersetzt diese einen Inpustring von der **Konkretten Syntax** in die **Abstrakte Syntax** (Definition 23).

Die **Baumdatenstruktur** des **Derivation Tree** und **Abstrakt Syntax Tree** ermöglicht es die Operationen, die der Compiler bei der Weiterverarbeitung des Inputstrings ausführen muss möglichst **effizient** auszuführen.

Definition 21: Transformer

*Ein Programm, dass von **unten-nach-oben**, nach dem **Breadth First Search** Prinzip alle Knoten des **Derivation Tree** besucht und in Bezug zu Compilerbau, beim Antreffen eines bestimmten Knoten des **Derivation Tree** einen entsprechenden Knoten des **Abstrakt Syntax Tree** erzeugt und diesen anstelle des Knotens des **Derivation Tree** setzt und so Stück für Stück den **Abstrak Syntax Tree** konstruiert.*

Definition 22: Visitor

*Ein Programm, dass von **unten-nach-oben**, nach dem **Breadth First Search** Prinzip alle Knoten des **Derivation Tree** besucht und in Bezug zu Compilerbau, beim Antreffen eines bestimmten **Knoten** des Derivation Tree, diesen **in-place** mit anderen Knoten **tauscht** oder **manipuliert**, um den Derivation Tree für die weitere Verarbeitung durch z.B. einen Transformer zu vereinfachen.*

*Kann theoretisch auch zur Konstruktion eines **Abstrakt Syntax Tree** verwendet werden, wenn z.B. eine externe Klasse verwendet wird, welches für die Konstruktion des **Abstrakt Syntax Tree** verantwortlich ist, aber dafür ist ein **Transformer** besser geeignet.*

Definition 23: Abstrakte Syntax

***Syntax**, die beschreibt, was für Arten von **Komposition** bei den **Knoten** eines **Abstrakt Syntax Trees** möglich sind.*[a]

*Jene Produktionen, die in der **Konkretten Syntax** für die Umsetzung von **Präzidenz** notwendig waren, sind in der **Abstrakten Syntax** abgeflacht.*

[a] *Course Webpage for Compilers (P423, P523, E313, and E513).*

Definition 24: Abstrakte Syntax Tree

Compilerinterne Darstellung *eines Programs, in welcher sich anhand der Knoten auf dem Pfad von der Wurzel zu einem* ***Blatt*** *nicht mehr direkt nachvollziehen lässt, durch welche* ***Produktionen*** *dieses Blatt abgeleitet wurde.*

Der ***Abstrakt Syntax Tree*** *hat einmal den Zweck, dass die Kompositionen, die die Konten bilden können* ***semantisch*** *näher an den* ***Instructions eines Assemblers*** *dran sind und, dass man mit ihm bei der Betrachtung eines* ***Knoten****, der für einen Teil des Programms steht, möglichst schnell die Frage beantworten kann, welche* ***Funktionalität*** *der Sprache dieser umsetzt, welche* ***Bestandteile*** *er hat und welche Funktionalität der Sprache diese Bestandteile umsetzen usw.*[a]

[a] *Course Webpage for Compilers (P423, P523, E313, and E513).*

Je weiter **unten**[4] und **links** ein Knoten im **Abstrakt Syntax Tree** liegt, desto eher wird dieser Knoten komplett abgearbeitet sein, da in der **Code Generierung** die Knoten nach dem **Depth First Search Prinzip** abgearbeitet werden.

2.6 Code Generierung

Definition 25: Pass

2.7 Fehlermeldungen

Definition 26: Fehlermeldung

[4] In der Informatik wachsen Bäume von oben-nach-unten. Die Wurzel ist also oben.

3 Implementierung

3.1 Lexikalische Analyse

3.1.1 Verwendung von Lark

3.1.2 Basic Parser

3.2 Syntaktische Analyse

3.2.1 Verwendung von Lark

3.2.2 Umsetzung von Präzidenz

Die PicoC Sprache hat dieselben Präzidenzregeln implementiert, wie die Sprache C[1]. Die Präzidenzregeln von PicoC sind in Tabelle 3.2.2 aufgelistet.

Präzidenz	Operator	Beschreibung	Assoziativität

1	`a()`
	`a[]`
	`a.b`
		Funktionsaufruf
		Indexzugriff
		Attributzugriff
			Links, dann rechts →

2	`-a`
	`!a ~a`
	`*a &a`
		Unäres Minus
		Logisches NOT und Bitweise NOT
		Dereferenz und Referenz, auch Adresse-von
			Rechts, dann links ←

3	`a*b a/b a%b`	Multiplikation, Division und Modulo	Links, dann rechts →
4	`a+b a-b`	Addition und Subtraktion	
5	`a<b a<=b` `a>b a>=b`	Kleiner, Kleiner Gleich, Größer, Größer gleich	
6	`a==b a!=b`	Gleichheit und Ungleichheit	
7	`a&b`	Bitweise UND	
8	`a^b`	Bitweise XOR (exclusive or)	
9	`a|b`	Bitweise ODER (inclusive or)	
10	`a&&b`	Logiches UND	
11	$a \| \| b$	Logisches ODER	
12	`a=b`	Zuweisung	Rechts, dann links ←
13	`a,b`	Komma	Links, dann rechts →

Tabelle 3.1: Präzidenzregeln von PicoC

[1] *C Operator Precedence - cppreference.com*.

3.2.3 Derivation Tree Generierung

3.2.4 Early Parser

3.2.5 Derivation Tree Vereinfachung

3.2.6 Abstrakt Syntax Tree Generierung

ASTNode

PicoC Nodes

RETI Nodes

3.3 Code Generierung

3.3.1 Passes

PicoC-Shrink Pass

PicoC-Blocks Pass

PicoC-Mon Pass

Definition 27: Symboltabelle

RETI-Blocks Pass

RETI-Patch Pass

RETI Pass

3.3.2 Umsetzung von Pointern und Arrays

3.3.3 Umsetzung von Structs

3.3.4 Umsetzung von Funktionen

3.3.5 Umsetzung kleinerer Details

3.4 Fehlermeldungen

3.4.1 Error Handler

4 Ergebnisse und Ausblick

4.1 Funktionsumfang

4.2 Qualitätskontrolle

4.3 Kommentierter Kompiliervorgang

4.4 Erweiterungsideen

A Appendix

A.1 Konkrette und Abstrakte Syntax

A.2 Bedienungsanleitungen

A.2.1 PicoC-Compiler

A.2.2 Showmode

A.2.3 Entwicklertools

Literatur

Online

- *C Operator Precedence - cppreference.com*. URL: https://en.cppreference.com/w/c/language/operator_precedence (besucht am 27. 04. 2022).
- *Compiler Design - Phases of Compiler*. URL: https://www.tutorialspoint.com/compiler_design/compiler_design_phases_of_compiler.htm (besucht am 19. 06. 2022).
- *Course Webpage for Compilers (P423, P523, E313, and E513)*. 28. Jan. 2022. URL: https://iucompilercourse.github.io/IU-Fall-2021/ (besucht am 28. 01. 2022).
- *lecture-notes-2021*. 20. Jan. 2022. URL: https://github.com/Compiler-Construction-Uni-Freiburg/lecture-notes-2021/blob/56300e6649e32f0594bbbd046a2e19351c57dd0c/material/lexical-analysis.pdf (besucht am 28. 04. 2022).
- *What is the difference between a token and a lexeme?* NewbeDEV. URL: http://newbedev.com/what-is-the-difference-between-a-token-and-a-lexeme (besucht am 17. 06. 2022).