

ALBERT LUDWIGS UNIVERSITÄT FREIBURG

TECHNISCHE FAKULTÄT

PicoC-Compiler

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

BACHELORARBEIT

Abgabedatum: 13. September 2022

Autor:

Jürgen Mattheis

Gutachter:

Prof. Dr. Scholl

Betreuung:

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.

Danksagungen

Bevor der Inhalt dieser Schriftlichen Ausarbeitung der Bachelorarbeit anfängt, will ich einigen Personen noch meinen Dank aussprechen.

Ich schreibe die folgenden Danksagungen nicht auf eine bestimmte Weise, wie es sich vielleicht etabliert haben sollte Danksagungen zu schreiben und verwende auch keine künstlichen Floskeln, wie „mein aufrichtigster Dank“ oder „aus tiefstem Herzen“, sondern drücke im Folgenden die Dinge nur so aus, wie ich sie auch wirklich meine.

Estmal, ich hatte selten im Studium das Gefühl irgendwo Kunde zu sein, aber bei dieser Bachelorarbeit und dem vorangegangenen Bachelorprojekt hatte ich genau diese Gefühl, obwohl die Verhältnisse eigentlich genau umgekehrt sein sollten. Die Umgang mit mir wahr echt unglaublich nett und unbürokratisch, was ich als keine Selbstverständlichkeit ansehe und sehr wertgeschätzt habe.

An erster Stelle will ich zu meinem Betreuer M.Sc. Tobias Seufert kommen, der netterweise auch bereits die Betreuung meines Bachelorprojektes übernommen hatte. Wie auch während des Bachelorprojektes, haben wir uns auch bei den Meetings während der Bachelorarbeit hervorragend verstanden. Dabei ging die Freundlichkeit und das Engagement seitens Tobias weit über das hinaus, was man bereits als eine gute Betreuung bezeichnen würde.

Es gibt verschiedene Typen von Menschen, es gibt Leute, die nur genauso viel tun, wie es die Anforderungen verlangen und nichts darüberhinaus tun, wenn es nicht einen eigenen Vorteil für sie hat und es gibt Personen, die sich für nichts zu Schade sind und dies aus einer Philanthropie oder Leidenschaft heraus tun, auch wenn es für sie keine Vorteile hat. Tobias¹ konnte ich während der langen Zeit, die er mein Bachelorprojekt und dann meine Bachelorarbeit betreut hat eindeutig als letzteren Typ Mensch einordnen.

Er war sich nie zu Schade für meine vielen Fragen während der Meetings, auch wenn ich meine Zeit ziemlich oft überzogen habe², er hat sich bei der Korrektur dieser Schriftlichen Ausarbeitung sogar die Mühe gemacht bei den einzelnen Problemstellen längere, wirklich hilfreiche Textkommentare zu verfassen und obendrauf auch noch Tippfehler usw. angemerkt und war sich nicht zu Schade die Rolle des Nachrichtenübermittlers zwischen mir und Prof. Dr. Scholl zu übernehmen. All dies war absolut keine Selbstverständlichkeit, vor allem wenn ich die Betreuung anderer Studenten, die ich kenne mit der vergleiche, die mir zu Teil wurde.

An den Kommentar zu meinem Betreuer Tobias will ich einen Kommentar zu meinem Gutachter Prof. Dr. Scholl anschließen. Wofür ich meinem Gutachter Prof. Dr. Scholl sehr dankbar bin, ist, dass er meine damals sehr ambitionierten Ideen für mögliche Funktionalitäten, die ich in den PicoC-Compiler für die Bachelorarbeit implementierten wollte runtergeschraubt hat. Man erlebt es äußerst selten im Studium, dass Studenten freiwillig weniger Arbeit gegeben wird.

Bei den für die Bachelorarbeit zu implementierenden Funktionalitäten gab es bei der Implementierung viele unerwartete kleine Details, die ich vorher garnicht bedacht hatte, die in ihrer Masse unerwartet viel Zeit zum Implementieren gebraucht haben. Mit den von Prof. Dr. Scholl festgelegten Funktionalitäten für die Bachelorarbeit ist der Zeitplan jedoch ziemlich perfekt aufgegangen. Mit meinen ambitionierten Plänen wäre es bei der Bachelorarbeit dageben wohl mit der Zeit äußerst kritisch geworden. Das Prof. Dr. Scholl mir zu

¹Wie auch Prof. Dr. Scholl. Hier geht es aber erstmal um Tobias.

²Wofür ich mich auch nochmal Entschuldigen will.

seinem eigenen Nachteil^{3 4} weniger Arbeit aufgebrummt hat empfand ich als ich eine äußerst nette Geste, die ich sehr geschätzt habe.

Wie mein Betreuer M.Sc. Tobias Seufert und wahrscheinlich auch mein Gutachter Prof. Dr. Scholl im Verlauf dieser Bachelorarbeit und des vorangegangenen Bachelorprojektes gemerkt haben, kann ich schon manchmal ziemlich eigensinnigen sein, bei der Weise, wie ich bestimmte Dinge umsetzen will. Ich habe es sehr geschätzt, dass mir das durchgehen gelassen wurde. Es ist, wie ich die Universitätswelt als Student erlebe bei Arbeitsvorgaben keine Selbstverständlichkeit, dass dem Studenten überhaupt die Freiheit und das Vertrauen gegeben wird diese auf seine eigenen Weise umzusetzen.

Vor allem, da mein eigenes Vorgehen größtenteils Vorteile für mich hatte, da ich auf diese Weise am meisten über Compilerbau gelernt hab und eher Nachteile für Prof. Dr. Scholl, da mein eigenes Vorgehen entsprechend mehr Zeit brauchte und ich daher als Bachelorarbeit keinen dazu passenden RETI-Emulator mit Graphischer Anzeige implementieren konnte, da die restlichen Funktionalitäten des PicoC-Compilers noch implementiert werden mussten.

Glücklicherweise gibt es aber doch noch einen passenden RETI-Emulator, der den PicoC-Compiler über seine Kommandozeilenargumente aufruft, um ein PicoC-Programm visuell auf einer RETI-CPU auszuführen. Für dessen Implementierung hat sich Michel Giehl netterweise zur Verfügung gestellt. Daher Danke auch an Michel Giehl, dass er sich mit meinem PicoC-Compiler auseinandergesetzt hat und diesen in seinen RETI-Emulator integriert hat, sodass am Ende durch unsere beiden Arbeiten ein anschauliches Lerntool für die kommenden Studentengenerationen entstehen konnte. Vor allem da er auch mir darin vertrauen musste, dass ich mit meinem PicoC-Compiler nicht irgendeinen Misst baue. Der RETI-Emulator von Michel Giehl ist unter [Link](#)⁵ zu finden.

Mir hat die Implementierung des PicoC-Compilers tatsächlich ziemlich viel Spaß gemacht, da Compilerbau auch in mein persönliches Interessengebiet fällt⁶. Das Aufschreiben dieser Schriftlichen Ausarbeitung hat mir dagegen eher weniger Spaß gemacht⁷. Wobei ich allerdings sagen muss, dass ich eine große Erleichterung verspüre das ganze Wissen über Compilerbau mal aufgeschrieben zu haben, damit ich mir keine Sorgen machen muss dieses ziemlich nützliche Wissen irgendwann wieder zu vergessen. Es hilft einem auch als Programmierer ungemein weiter zu wissen, wie ein Compiler unter der Haube funktioniert, da man sich so viel besser merken, wie eine bestimmte Funktionalität einer Programmiersprache zu verwenden ist. Manch eine Funktionalität einer Programmiersprache kann in der Verwendung ziemlich willkürlich erscheinen, wenn man die technische Umsetzung dahinter im Compiler nicht kennt.

Ich wollte mich daher auch noch dafür Bedanken, dass mir ein so ergiebiges und interessantes Thema als Bachelorarbeit vorgeschlagen wurde und vor allem, dass auch das Vertrauen in mich gesteckt wurde, dass ich am Ende auch einen funktionsfähigen, sauber programmierten und gut durchdachten Compiler implementiere.

Zum Schluss nochmal ein abschließendes Danke an meinen Betreuer M.Sc Seufert und meinen Gutachter Prof. Dr. Scholl für die Betreuung und Bereitstellung dieser interessanten Bachelorarbeit und des vorangegangenen Bachelorprojektes und Michel Giehl für das Integrieren des PicoC-Compilers in seinen RETI-Emulator.

³Der PicoC-Compiler hätte schließlich mehr Funktionalitäten haben können.

⁴Vielleicht finde ich ja noch im nächsten Semester während des Betriebssysteme Tutorats noch etwas Zeit einige weitere Features einzubauen oder möglicherweise im Rahmen eines Masterprojektes 🤔.

⁵<https://github.com/michel-giehl/Reti-Emulator>.

⁶Womit nicht alle Studenten so viel Glück haben.

⁷Dieses ständige überlegen, wo man möglicherweise eine Erklärücke hat, ob man nicht was wichtiges ausgelassen hat usw.

Inhaltsverzeichnis

Abbildungsverzeichnis	I
Codeverzeichnis	II
Tabellenverzeichnis	III
Definitionsverzeichnis	IV
Grammatikverzeichnis	V
Appendix	A
RETI Architektur Details	A
Sonstige Definitionen	C
Bootstrapping	H
Literatur	M

Abbildungsverzeichnis

1.1	Datenpfade der RETI-Architektur aus C. Scholl, „Betriebssysteme“, nicht selbst erstellt. . . .	C
1.2	Cross-Compiler als Bootstrap Compiler.	J
1.3	Iteratives Bootstrapping.	L

Codeverzeichnis

Tabellenverzeichnis

1.1	Load und Store Befehle aus C. Scholl, „Betriebssysteme“, nicht selbst zusammengestellt, leicht abgewandelt.	A
1.2	Compute Befehle aus C. Scholl, „Betriebssysteme“, nicht selbst zusammengestellt, leicht abgewandelt.	B
1.3	Jump Befehle aus C. Scholl, „Betriebssysteme“, nicht selbst zusammengestellt, leicht abgewandelt.	B

Definitionsverzeichnis

1.1	T-Diagram Maschine	C
1.2	Bezeichner (bzw. Identifier)	C
1.3	Label	C
1.4	Assemblersprache (bzw. engl. Assembly Language)	D
1.5	Assembler	D
1.6	Objectcode	D
1.7	Linker	D
1.8	Transpiler (bzw. Source-to-source Compiler)	E
1.9	Rekursiver Abstieg	E
1.10	Linksrekursive Grammatiken	E
1.11	LL(k)-Grammatik	E
1.12	Earley Erkennen	F
1.13	Liveness Analyse	G
1.14	Live Variable	G
1.15	Graph Coloring	G
1.16	Interference Graph	G
1.17	Kontrollflussgraph	G
1.18	Kontrollfluss	H
1.19	Kontrollflussanalyse	H
1.20	Two-Space Copying Collector	H
1.21	Self-compiling Compiler	I
1.22	Bootstrap Compiler	J
1.23	Minimaler Compiler	K
1.24	Bootstrapping	K

Grammatikverzeichnis

Appendix

Dieses Kapitel dient als Lagerstätte für **Definitionen**, **Tabellen**, **Abbildungen** und ganze **Unterkapitel**, die zum Erhalt des **roten Fadens** und des **Lesefflusses** in den vorangegangenen Kapiteln hierher ausgelagert wurden. Im Unterkapitel **RETI Architektur Details** können einige Details der **RETI-Architektur** nachgeschaut werden, die im Kapitel ?? den Leseffluss **stören** würden und zum Verständnis nur **bedingt wichtig** sind. Im Unterkapitel **Sonstige Definitionen** sind einige **Definitionen** ausgelagert, die zum Verständnis der **Implementierung** des PicoC-Compilers **nicht wichtig** sind, aber z.B. an einer bestimmten Stelle in den vorangegangenen Kapiteln **kurz Erwähnung** fanden. Im Unterkapitel **Bootstrapping** wird ein Vorgehen, das **Bootstrapping** erklärt, welches beim PicoC-Compiler **nicht umgesetzt** wurde, es aber erlauben würde aus dem **PicoC-Compiler** einen Compiler für die **RETI-CPU** zu machen, der auf der RETI-CPU selbst läuft.

RETI Architektur Details

Hier wird die **Semantik** der verschiedenen Befehle des **Befehlssatzes** der **RETI-Architektur** mithilfe von Tabelle 1.1, Tabelle 1.2 und Tabelle 1.3 dokumentiert. Des Weiteren sind in Abbildung 1.1 die **Datenpfade** der **RETI-Architektur** dargestellt.

Typ	Modus	Befehl	Wirkung
01	00	LOAD D i	$D := M(\langle i \rangle), \langle PC \rangle := \langle PC \rangle + 1$
01	01	LOADIN S D i	$D := M(\langle S \rangle + i), \langle PC \rangle := \langle PC \rangle + 1$
01	11	LOADI D i	$D := 0^{10}i, \langle PC \rangle := \langle PC \rangle + 1$, bei $D = PC$ wird der PC nicht inkrementiert
10	00	STORE S i	$M(\langle i \rangle) := S, \langle PC \rangle := \langle PC \rangle + 1$
10	01	STOREIN D S i	$M(\langle D \rangle + i) := S, \langle PC \rangle := \langle PC \rangle + 1$
10	11	MOVE S D	$D := S, \langle PC \rangle := \langle PC \rangle + 1$, Move: Bei $D = PC$ wird der PC nicht inkrementiert

Tabelle 1.1: Load und Store Befehle aus C. Scholl, „Betriebssysteme“, nicht selbst zusammengestellt, leicht abgewandelt.

Typ	M	RO	F	Befehl	Wirkung
00	0	0	000	ADDI D i	$[D] := [D] + [i], \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	001	SUBI D i	$[D] := [D] - [i], \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	010	MULI D i	$[D] := [D] * [i], \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	011	DIVI D i	$[D] := [D] / [i], \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	100	MODI D i	$[D] := [D] \% [i], \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	101	OPLUSI D i	$[D] := [D] \oplus 0^{10}i, \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	110	ORI D i	$[D] := [D] \vee 0^{10}i, \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	101	ANDI D i	$[D] := [D] \wedge 0^{10}i, \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	000	ADD D i	$[D] := [D] + [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	001	SUB D i	$[D] := [D] - [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	010	MUL D i	$[D] := [D] * [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	011	DIV D i	$[D] := [D] / [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	100	MOD D i	$[D] := [D] \% [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	101	OPLUS D i	$D := D \oplus M(\langle i \rangle), \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	110	OR D i	$D := D \vee M(\langle i \rangle), \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	101	AND D i	$D := D \wedge M(\langle i \rangle), \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	000	ADD D S	$[D] := [D] + [S], \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	001	SUB D S	$[D] := [D] - [S], \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	010	MUL D S	$[D] := [D] * [S], \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	011	DIV D S	$[D] := [D] / [S], \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	100	MOD D S	$[D] := [D] \% [S], \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	101	OPLUS D S	$D := D \oplus S, \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	110	OR D S	$D := D \vee S, \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	101	AND D S	$D := D \wedge S, \langle PC \rangle := \langle PC \rangle + 1$

Tabelle 1.2: Compute Befehle aus C. Scholl, „Betriebssysteme“, nicht selbst zusammengestellt, leicht abgewandelt.

Type	Condition	J	Befehl	Wirkung
11	000	00	NOP	$\langle PC \rangle := \langle PC \rangle + 1$
11	001	00	JUMP _{>} i	Falls $[ACC] > 0$: $\langle PC \rangle := \langle PC \rangle + [i]$, sonst: $\langle PC \rangle := \langle PC \rangle + 1$
11	010	00	JUMP ₌ i	Falls $[ACC] = 0$: $\langle PC \rangle := \langle PC \rangle + [i]$, sonst: $\langle PC \rangle := \langle PC \rangle + 1$
11	011	00	JUMP _≥ i	Falls $[ACC] ≥ 0$: $\langle PC \rangle := \langle PC \rangle + [i]$, sonst: $\langle PC \rangle := \langle PC \rangle + 1$
11	100	00	JUMP _{<} i	Falls $[ACC] < 0$: $\langle PC \rangle := \langle PC \rangle + [i]$, sonst: $\langle PC \rangle := \langle PC \rangle + 1$
11	101	00	JUMP _≠ i	Falls $[ACC] ≠ 0$: $\langle PC \rangle := \langle PC \rangle + [i]$, sonst: $\langle PC \rangle := \langle PC \rangle + 1$
11	110	00	JUMP _≤ i	Falls $[ACC] ≤ 0$: $\langle PC \rangle := \langle PC \rangle + [i]$, sonst: $\langle PC \rangle := \langle PC \rangle + 1$ $\langle PC \rangle := \langle PC \rangle + [i]$
11	111	00	JUMPi	$\langle PC \rangle := \langle PC \rangle + [i]$
11	*	01	INT i	$\langle PC \rangle := IVT[i]$ Interrupt Nr.i wird Ausgeführt
11	*	10	RTI	Rücksprungadresse vom Stack entfernt, in PC geladen, Wechsel in Usermodus

Tabelle 1.3: Jump Befehle aus C. Scholl, „Betriebssysteme“, nicht selbst zusammengestellt, leicht abgewandelt.

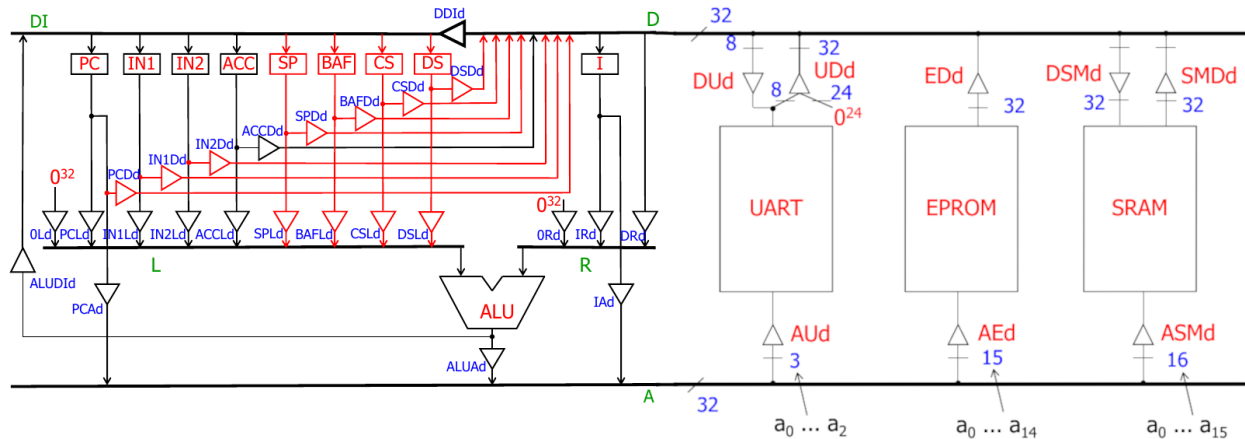


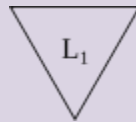
Abbildung 1.1: Datenpfade der RETI-Architektur aus C. Scholl, „Betriebssysteme“, nicht selbst erstellt.

Sonstige Definitionen

Im Folgenden sind einige Definitionen aufgelistet, die in den vorangegangenen Kapiteln **Erwähnung** fanden und zur Beibehaltung des **roten Fadens** und des **Leseflusses** in dieses Unterkapitel **ausgelagert** wurden. Die Definitionen in diesem Unterkapitel vermitteln **Theorie über Compilerbau**, die über das **hinausgeht**, was zum Verständnis der Implementierung des PicoC-Compilers notwendig ist.

Definition 1.1: T-Diagram Maschine

Repräsentiert eine **Maschine**, welche ein **Programm** in **Maschinensprache** L_1 ausführt.^{a,b}



^aWenn die Maschine **Programme** in einer höheren Sprache als **Maschinensprache** ausführt, ist es auch erlaubt diese Notation zu verwenden, dann handelt es sich um eine **Abstrakte Maschine**, wie z.B. die **Python Virtual Machine** (PVM) oder **Java Virtual Machine** (JVM).

^bJ. Earley und Sturgis, „A formalism for translator interactions“.

Definition 1.2: Bezeichner (bzw. Identifier)

Zeichenfolge^a, die eine **Konstante**, **Variable**, **Funktion** usw. innerhalb ihres Sichtbarkeitsbereichs **eindeutig** benennt.^{b,c}

^aBzw. **Tokenwert**.

^bAußer wenn z.B. bei Funktionen die Programmiersprache das **Überladen** erlaubt usw. In diesem Fall wird die **Signatur** der Funktion als weiteres Unterscheidungsmerkmal hinzugenommen, damit es eindeutig ist.

^cThiemann, „Einführung in die Programmierung“.

Definition 1.3: Label

Durch einen **Bezeichner** **eindeutig** zuordenbares **Sprungziel** im Programmcode.^a

^aThiemann, „Compilerbau“.

Definition 1.4: Assemblersprache (bzw. engl. Assembly Language)

Eine sehr *hardwarenahe* Programmiersprache, deren *Befehle* eine starke Entsprechung zu bestimmten Maschinenbefehlen bzw. Folgen von Maschinenbefehlen^a haben. Viele *Befehle* haben eine ähnliche übliche Struktur *Operation* <Operanden>, mit einer *Operation*, die einem *Opcode* eines Maschinenbefehls bezeichnet und keinen oder mehreren *Operanden*, wie die späteren Maschinenbefehle, denen sie entsprechen. Allerdings gibt es oftmals noch viel „syntaktischen Zucker“ innerhalb^b der Befehle und drumherum^c.^d

^aBefehle der Assemblersprache, die mehreren Maschinenbefehlen entsprechen werden auch als **Pseudo-Befehle** bezeichnet und entsprechen dem, was man im allgemeinen als Macro bezeichnet.

^bZ.B. erlaubt die Assemblersprache des **GCC** für die **X_{86_64}-Architektur** für manche Operanden die Syntax **n(%r)**, die einen **Speicherzugriff** mit **Offset** *n* zur Adresse, die im **Register** **%r** steht durchführt, wobei z.B. die Klammern () usw. nur „syntaktischer Zucker“ sind und natürlich nicht mitkodiert werden.

^cZ.B. sind im **X_{86_64} Assembler** die Befehle in **Blöcken** untergebracht, die ein **Label** haben und zu denen mittels **jmp <label>** gesprungen werden kann. Ein solches Konstrukt, was vor allem auch noch relativ beliebig wählbare Bezeichner verwendet hat keine direkte Entsprechung in einem handelsüblichen Prozessor und Hauptspeicher.

^dP. Scholl, „Einführung in Embedded Systems“.

Ein **Assembler** (Definition 1.5) ist in üblichen Compilern in einer bestimmten Form meist schon integriert, da Compiler üblicherweise direkt **Maschinencode** bzw. **Objectcode** (Definition 1.6) erzeugen. Ein **Compiler** soll möglichst viel von seiner internen Funktionsweise und der damit verbundenen Theorie für den Benutzer abstrahieren und dem Benutzer daher standardmäßig einfach nur die Ausgabe liefern, welche er in den allermeisten Fällen haben will, nämlich den **Maschinencode** bzw. **Objectcode**, der direkt ausführbar ist bzw. wenn er später mit dem **Linker** (Definition 1.7) zu Maschinencode zusammengesetzt wird ausführbar ist.

Definition 1.5: Assembler

Übersetzt im allgemeinen **Assemblercode**, der in **Assemblersprache** geschrieben ist zu **Maschinencode** bzw. **Objectcode** in **binärer Repräsentation**, der in **Maschinsprache** geschrieben ist.^a

^aP. Scholl, „Einführung in Embedded Systems“.

Definition 1.6: Objectcode

Bei Komplexeren Compilern, die es erlauben den Programmcode in *mehrere Dateien* aufzuteilen wird häufig **Objectcode** erzeugt, der neben der Folge von Maschinenbefehlen in **binärer Repräsentation** auch noch Informationen für den **Linker** enthält, die im späteren **Maschiendencode** nicht mehr enthalten sind, sobald der **Linker** die Objektdateien zum Maschinencode zusammengesetzt hat.^a

^aP. Scholl, „Einführung in Embedded Systems“.

Definition 1.7: Linker

Programm, dass **Objektcode** aus mehreren Objektdateien zu ausführbarem **Maschinencode** in eine ausführbare Datei oder Bibliotheksdatei **linkt** bzw. zusammenfügt, sodass unter anderem kein vermeidbarer **doppelter Code** darin vorkommt.^a

^aP. Scholl, „Einführung in Embedded Systems“.

Definition 1.8: Transpiler (bzw. Source-to-source Compiler)

*Kompiliert zwischen Sprachen, die ungefähr auf dem **gleichen** Level an **Abstraktion** arbeiten^{ab}*

^aDie Programmiersprache **TypeScript** will als **Obermenge** von **JavaScript** die Sprache Javascript **erweitern** und gleichzeitig die **syntaktischen Mittel** von JavaScript unterstützen. Daher bietet es sich Typescript zu Javascript zu **transpilieren**.

^bThiemann, „Compilerbau“.

Definition 1.9: Rekursiver Abstieg

*Es wird jedem **Nicht-Terminalsymbol** eine **Prozedur** zugeordnet, welche die **Produktionen** dieses Nicht-Terminalsymbols umsetzt. **Prozeduren** rufen sich dabei wechselseitig entsprechend der Produktionen, welche sie jeweils umsetzen gegenseitig auf.*

Bei manchen **Ansätzen** für das **Parsen** eines Programmes, ist es notwendig eine **LL(k)-Grammatik** (Definition 1.11) vorliegen zu haben. Bei diesen Ansätzen, die meist die Methode des **Rekursiven Abstiegs** (Definition 1.9) verwenden lässt sich eine bessere minimale **Laufzeit** garantieren, da aufgrund der **LL(k)-Eigenschaft** ausgeschlossen werden kann, dass **Backtracking** notwendig ist¹.

Manche der **Ansätze** für das **Parsen** eines Programmes haben ein Problem, wenn die Grammatik, die für das Programm zur Entscheidung des **Wortproblems** verwendet wird, eine **Linksrekursive Grammatik** (Definition 1.10) ist².

Definition 1.10: Linksrekursive Grammatiken

*Eine **Grammatik** ist **linksrekursiv**, wenn sie ein **Nicht-Terminalsymbol** enthält, dass **linksrekursiv** ist.*

*Ein **Nicht-Terminalsymbol** ist **linksrekursiv**, wenn das **linkeste Symbol** in einer seiner **Produktionen** es selbst ist oder zu sich selbst gemacht werden kann durch eine Folge von Ableitungen:*

$$A \Rightarrow^* Aa,$$

*wobei a eine beliebige Folge von **Grammatiksymbolen**^a ist.^b*

^aAlso eine beliebige Folge von **Terminalsymbolen** und **Nicht-Terminalsymbolen**.

^b*Parsers — Lark documentation.*

Definition 1.11: LL(k)-Grammatik

*Eine Grammatik ist **LL(k)** für $k \in \mathbb{N}$, falls jeder Ableitungsschritt eindeutig durch die **nächsten k Tokentypen** der **Tokens**, welche aus dem **Eingabewort** generiert wurden zu bestimmen ist^a. Dabei steht **LL** für **left-to-right** und **leftmost-derivation**, da das **Eingabewort** von **links nach rechts** geparkt und immer **Linksableitungen** genommen werden müssen^b, damit die obige Bedingung mit den **nächsten k Symbolen** gilt.^c*

^aDas wird auch als **Lookahead** von k bezeichnet.

^bWobei sich das mit den **Linksableitungen** automatisch ergibt, wenn man das Eingabewort von **links-nach-rechts** parsed und jeder der nächsten k **Ableitungsschritte** eindeutig sein soll.

^cNebel, „Theoretische Informatik“.

¹Mehr **Erklärung** hierzu findet sich im Unterkapitel ??.

²Für den im **PicoC-Compiler** verwendeten **Earley Parsers** stellt dies allerdings **kein** Problem dar.

Definition 1.12: Earley Erkenner



Ist ein **Erkenner**, der für alle **Kontextfreien Sprachen** das **Wortproblem** entscheiden kann und dies mittels **Dynamischer Programmierung** mit dem **Top-Down Ansatz** umsetzt.^{a b c}

Eingabe und **Ausgabe** des Algorithmus sind:

- **Eingabe:** Eingabewort w und **Konkrete Grammatik** $G_{\text{Parse}} = \langle N, \Sigma, P, S \rangle$.
- **Ausgabe:** 0 wenn $w \notin L(G_{\text{Parse}})$ ^d und 1 wenn $w \in L(G_{\text{Parse}})$.

Bevor dieser **Algorithmus** erklärt wird müssen noch einige **Symbole** und **Notationen** erklärt werden:

- α, β, γ stellen eine **beliebige Folge** von **Grammatiksymbolen**^e dar.
- A und B stellen **Nicht-Terminalsymbole** dar.
- a stellt ein **Terminalsymbol** dar.
- **Earley's Punktnotation:** $A ::= \alpha \bullet \beta$ stellt eine **Produktion**, in der α **bereits geparst** wurde und β **noch geparst** werden muss.
- Die **Indexierung** ist informell ausgedrückt so umgesetzt, dass die **Indices zwischen Tokentypen** liegen, also **Index 0 vor** dem ersten **Tokentyp** verortet ist, **Index 1 nach** dem ersten **Tokentyp** verortet ist und **Index n nach** dem **letzten Tokentyp** verortet ist.

und davor müssen noch einige **Begriffe definiert** werden:

- **Zustandsmenge:** Für jeden der $n + 1$ **Indices** j wird eine **Zustandsmenge** $Z(j)$ generiert.
- **Zustand einer Zustandsmenge:** Ist ein **Tupel** $(A ::= \alpha \bullet \beta, i)$, wobei $A ::= \alpha \bullet \beta$ die **aktuelle Produktion** ist, die bis **Punkt \bullet** geparst wurde und i der **Index** ist, ab welchem der Versuch der Erkennung eines **Teilworts** des **Eingabeworts** mithilfe dieser **Produktion** begann.

Der **Ablauf** des Algorithmus ist wie folgt:

1. **initialisiere** $Z(0)$ mit der **Produktion**, welches das **Startsymbol** S auf der **linken Seite** des $::=-$ Symbols hat.
2. es werden in der **aktuellen Zustandsmenge** $Z(j)$ die folgenden **Operationen ausgeführt**:
 - **Vorausage:** Für jeden **Zustand** in der **Zustandsmenge** $Z(j)$, der die Form $(A ::= \alpha \bullet B\gamma, i)$ hat, wird für jede **Produktion** $(B ::= \beta)$ in der Konkreten Grammatik, die ein B auf der **linken Seite** des $::=-$ Symbols hat ein **Zustand** $(B ::= \bullet\beta, j)$ zur **Zustandsmenge** $Z(j)$ hinzugefügt.
 - **Überprüfung:** Für jeden **Zustand** in der **Zustandsmenge** $Z(j)$, der die Form $(A ::= \alpha \bullet a\gamma, i)$ hat wird der **Zustand** $(A ::= \alpha a \bullet \gamma, i)$ zur **Zustandsmenge** $Z(j + 1)$ hinzugefügt.
 - **Vervollständigung:** Für jeden **Zustand** in der **Zustandsmenge** $Z(j)$, der die Form $(B ::= \beta \bullet, i)$ hat werden alle **Zustände** in $Z(i)$ gesucht, welche die Form $(A ::= \alpha \bullet B\gamma, i)$ haben und es wird der **Zustand** $(A ::= \alpha B \bullet \gamma, i)$ zur **Zustandsmenge** $Z(j)$ hinzugefügt.

bis:

- der **Zustand** $(A ::= \beta \bullet, 0)$ in der **Zustandsmenge** $Z(n)$ auftaucht, wobei A das **Startsym-**

bol S ist $\Rightarrow w \in L(G_{Parse})$.

- ***keine Zustände*** mehr hinzugefügt werden können $\Rightarrow w \notin L(G_{Parse})$.

^aJay Earley, „An efficient context-free parsing“.

^b**Erklärweise** wurde von der Webseite *Earley parser* übernommen.

^c*Earley Parser*.

^d $L(G_{Parse})$ ist die **Sprache**, welche durch die **Konkrete Grammatik** G_{Parse} beschrieben wird.

^eAlso eine Folge von **Terminalsymbolen** und **Nicht-Terminalsymbolen**.

Definition 1.13: Liveness Analyse



Findet heraus, welche **Variablen** in welchen **Regionen** eines Programmes **verwendet** werden.^a

^aG. Siek, *Essentials of Compilation*.

Definition 1.14: Live Variable



Eine **Variable** (Definition ??), deren momentaner Wert **später** im Programmablauf noch **verwendet** wird. Man sagt auch die Variable ist **live**.^{a,b}

^aEs gibt leider **kein** allgemein verwendetes **deutsches** Wort für **Live Variable**.

^bG. Siek, *Essentials of Compilation*.

Definition 1.15: Graph Coloring



Problem, bei dem den **Knoten** eines Graphen^a **Zahlen**^b zugewiesen werden sollen, sodass **keine** zwei **adjazente Knoten** die **gleiche Zahl** haben und **möglichst wenige** unterschiedliche Zahlen gebraucht werden.^{c,d}

^aIn Bezug zu Compilerbau ein **Ungerichteter Graph**.

^bBzw. **Farben**.

^cEs gibt leider **kein** allgemein verwendetes **deutsches** Wort für **Graph Coloring**.

^dG. Siek, *Essentials of Compilation*.

Definition 1.16: Interference Graph



Ein **ungerichteter Graph** mit **Variablen** als **Knoten**, der eine **Kante** zwischen zwei Variablen hat, wenn es sich bei beiden Variablen **zu dem Zeitpunkt** um **Live Variablen** (Definition 1.14) handelt. In Bezug auf **Graph Coloring** bedeutet eine **Kante**, dass diese zwei Variablen **nicht** die **gleiche Zahl**^a zugewiesen bekommen dürfen.^b

^aBzw. **Farbe**.

^bG. Siek, *Essentials of Compilation*.

Definition 1.17: Kontrollflussgraph



Gerichteter Graph, der den **Kontrollfluss** (Definition 1.17) eines Programmes beschreibt.^a

^aG. Siek, *Essentials of Compilation*.

Definition 1.18: Kontrollfluss

Die **Reihenfolge** in der z.B. **Anweisungen**, **Funktionsaufrufe** usw. eines Programmes ausgewertet werden^a.

^aMan geht hier von einem **imperativen** Programm aus.

Definition 1.19: Kontrollflussanalyse

Analyse des **Kontrollflusses** (Definition 1.18) eines **Programmes**, um herauszufinden zwischen welchen Teilen des Programms **Daten ausgetauscht** werden und welche **Abhängigkeiten** sich daraus ergeben.

Der **simpelste Ansatz** ist es in einen Kontrollflussgraph **iterativ** einen Algorithmus^a anzuwenden, bis sich an den Werten der Knoten **nichts** mehr **ändert**^{b, c}.

^aIm Bezug zu Compilerbau die **Linveness Analyse**.

^bBis diese sich **stabilisiert** haben

^cG. Siek, *Essentials of Compilation*.

Definition 1.20: Two-Space Copying Collector

Ein **Garbage Collector** bei dem der **Heap** in **FromSpace** und **ToSpace** unterteilt wird und bei **nicht ausreichendem** Speicherplatz auf dem **Heap** alle Variablen, die in Zukunft noch verwendet werden vom **FromSpace** zum **ToSpace** kopiert werden. Der aktuelle **ToSpace** wird danach zum neuen **FromSpace** und der aktuelle **FromSpace** wird danach zum neuen **ToSpace**.^a

^aG. Siek, *Essentials of Compilation*.

Bootstrapping

Wenn eines Tages eine **RETI-CPU** auf einem **FPGA** implementiert werden sollte, sodass ein **provisorisches Betriebssystem** darauf laufen könnte, dann wäre der nächste Schritt einen **Self-Compiling Compiler** $C_{PicoC_RETI}^{PicoC}$ (Definition 1.21) zu schreiben. Durch einen **Self-Compiling Compiler** kann die **Unabhängigkeit** von der Programmiersprache L_{Python} , in welcher der **PicoC-Compiler** $C_{PicoC_RETI}^{Python}$ bisher implementiert ist erreicht werden. Des Weiteren kann die **Unabhängigkeit** von einer **anderen Maschine**, die bisher immer für das **Cross-Compiling** (Definition ??) notwendig war erreicht werden. Mittels **Bootstrapping** wird aus dem **PicoC-Compiler** ein „richtiger Compiler“³ für die **RETI-CPU** gemacht, der auf der RETI-CPU selbst läuft.

Anmerkung

Im Folgenden wird ein **voll ausgeschriebenes Compilerkürzel** als $C_{E_A_k_min}^{I-j}$ geschrieben, wobei:

- $C_{E_A_k_min}^{I-j} \hat{=}$ **Eingabesprache**.
- $C_{E_A_k_min}^{I-j} \hat{=}$ **Ausgabesprache**.
- $C_{E_A_k_min}^{I-j} \hat{=}$ Version k der **Eingabesprache**.

³Ein **üblicher Compiler**, wie ihn ein Programmierer verwendet, wie der **GCC** oder **Clang** läuft üblicherweise selbst auf der Maschine für welche er kompiliert.

- $C_{E_A_k_min}^{I-j} \hat{=}$ Implementierungssprache.
- $C_{E_A_k_min}^{I-j}$ Version j der Implementierungssprache.
- $C_{E_A_k_min}^{I-j} \hat{=}$ Minimaler Compiler.

bedeuten.

Definition 1.21: Self-compiling Compiler



Compiler C_L^L , der in der Sprache L **implementiert** ist, die er **kompiliert**. Also ein Compiler, der sich **selbst** kompilieren könnte.^a

^aJ. Earley und Sturgis, „A formalism for translator interactions“.

Will man für eine Maschine M_{RETI} , auf der bisher **keine anderen** Programmiersprachen mittels **Bootstrapping** (Definition 1.24) zum laufen gebracht wurden, den gerade beschriebenen **Self-compiling Compiler** $C_{PicoC_RETI}^{PicoC}$ implementieren und hat bereits den gesamten **Self-compiling Compiler** $C_{PicoC_RETI}^{PicoC}$ in der Sprache L_{PicoC} **implementiert**, so stösst man auf ein Problem, dass auf das **Henne-Ei-Problem**⁴ reduziert werden kann. Man bräuchte, um den **Self-compiling Compiler** $C_{PicoC_RETI}^{PicoC}$ auf der **Maschine** M_{PicoC} zu **kompilieren** bereits einen **kompilierten Self-compiling Compiler** $C_{PicoC_RETI}^{RETI}$. Es liegt eine **zirkulare Abhängigkeit** vor, die man nur auflösen kann, indem eine **externe Entität** zur Hilfe nimmt.

Eine Möglichkeit diese **zirkulare Abhängigkeit** zu brechen, wäre, dass man den **Cross-Compiler** $C_{PicoC_RETI}^{Python}$, den man bereits in der Programmiersprache L_{Python} implementiert hat auf einer anderen Maschine M_{other} dazu nutzt, um den **Self-compiling Compiler** $C_{PicoC_RETI}^{PicoC}$ für die Maschine M_{RETI} zu kompilieren bzw. zu **bootstrappen**. Der **Cross-Compiler** $C_{PicoC_RETI}^{Python}$ stellt in diesem Fall einen **Bootstrap Compiler** (Definition 1.22) dar. Den **kompilierten Compiler** $C_{PicoC_RETI}^{RETI}$ kann man dann einfach von der Maschine M_{other} auf die Maschine M_{RETI} **kopieren**⁵. In Abbildung 1.2 ist das ganze in einem **T-Diagramm** (siehe Unterkapitel ??) dargestellt.

⁴Beschreibt die Situation, wenn ein System sich **selbst** als **Abhängigkeit** hat, damit es überhaupt einen **Anfang** für dieses System geben kann. Dafür steht das Problem mit der **Henne** und dem **Ei** sinnbildlich, da hier die Frage ist, wie das ganze seinen Anfang genommen hat, da beides **zirkular** voneinander abhängt.

⁵Im Fall, dass auf der Maschine M_{RETI} die Programmiersprache L_{Python} bereits mittels **Bootstrapping** zum Laufen gebracht wurde, könnte der **Self-compiling Compiler** $C_{PicoC_RETI}^{PicoC}$ auch mithilfe des **Cross-Compilers** $C_{PicoC_RETI}^{Python}$ als **externe Entität** auf der Maschine M_{RETI} **selbst** kompiliert werden.

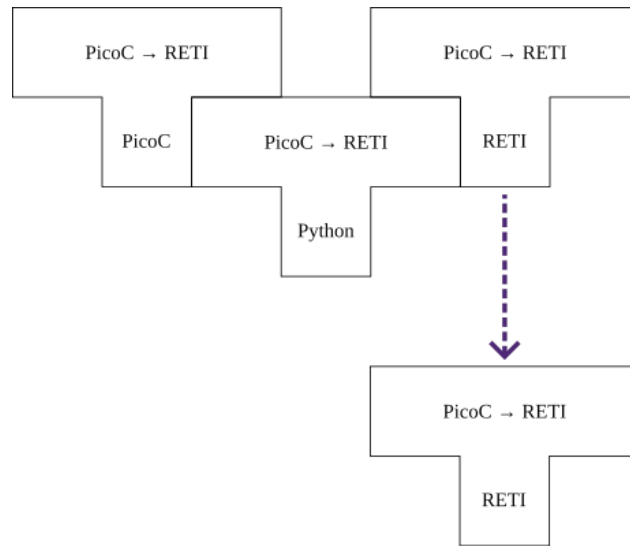


Abbildung 1.2: Cross-Compiler als Bootstrap Compiler.

Definition 1.22: Bootstrap Compiler

Compiler C_{L-M}^O , der es ermöglicht einen **Self-compiling Compiler** C_{L-M}^L zu **bootstrafen** (Definition 1.24). Dies wird ermöglicht, indem der **Self-compiling Compiler** C_{L-M}^L mit dem **Bootstrap Compiler** C_{L-M}^O **kompiliert** wird^a. Der **Bootstrap Compiler** C_{L-M}^O stellt eine **externe Entität** dar, die es ermöglicht die **zirkulare Abhängigkeit** zu brechen, dass initial ein **kompilierter Self-compiling Compiler** C_{L-M}^M bereits vorliegen müsste, um sich selbst kompilieren zu können.^b

^aDabei kann es sich um einen **lokal** auf der Maschine selbst laufenden Compiler oder auch um einen **Cross-Compiler** handeln.

^bThiemann, „Compilerbau“.

Aufbauend auf dem **Self-compiling Compiler** $C_{RETI_PicoC}^{PicoC}$, der einen **minimalen Compiler** (Definition 1.23) für eine Teilmenge der **Programmiersprache** C bzw. L_C darstellt, könnte man auch noch weitere Teile der Programmiersprache C bzw. L_C für die Maschine M_{RETI} mittels **Bootstrap** implementieren.⁶

Das bewerkstelligt man, indem man **iterativ** auf der Zielmaschine M_{RETI} selbst, aufbauend auf diesem **minimalen Compiler** $C_{RETI_PicoC}^{PicoC}$, wie in Subdefinition 1.24.1 den minimalen Compiler schrittweise zu einem immer vollständigeren **C-Compiler** C_C weiterentwickelt.

Anmerkung

Einen ersten **minimalen Compiler** (Definition 1.23) $C_{w.2.min}$ für eine Maschine M_2 und Wunschsprache L_w kann man entweder mittels eines **externen Bootstrap Compilers** C_w^O kompilieren^a oder man schreibt ihn direkt in der **Maschinensprache** L_{M_2} bzw. wenn ein **Assembler** vorhanden ist, in der **Assemblesprache** L_{A_2} .

Die letzte Option wäre allerdings nur beim allerersten Compiler C_{first} für eine allererste **abstraktere Programmiersprache** L_{first} mit Schleifen, Verzweigungen usw. notwendig gewesen. Ansonsten hätte man immer eine Kette, die beim allerersten Compiler C_{first} anfängt fortführen können, in der ein Compiler einen anderen Compiler kompiliert bzw. einen ersten minimalen Compiler kompiliert und

⁶Natürlich könnte man aber auch einfach den **Cross-Compiler** C_{PicoC}^{Python} um weitere Funktionalitäten von L_C erweitern, hat dann aber weiterhin eine **Abhängigkeit** von der Programmiersprache L_{Python} .

dieser minimale Compiler dann eine umfangreichere Version von sich kompiliert usw.

^aIn diesem Fall, dem **Cross-Compiler** $C_{PicoC.RETI}^{Python}$.

Definition 1.23: Minimaler Compiler



Compiler $C_{w,min}$, der nur die **notwendigsten Funktionalitäten** einer Wunschsprache L_w , wie **Schleifen**, **Verzweigungen** kompiliert, die für die Implementierung eines **Self-compiling Compilers** C_w^w oder einer **ersten Version** $C_{w_i}^{w_i}$ des Self-compiling Compilers C_w^w wichtig sind.^{a,b}

^aDen **PicoC-Compiler** könnte man auch als einen **minimalen Compiler** ansehen.

^bThiemann, „Compilerbau“.

Definition 1.24: Bootstrapping



Wenn man einen **Self-compiling Compiler** C_w^w einer Wunschsprache L_w auf einer **Zielmaschine** M zum laufen bringt^{a,b,c,d}. Dabei ist die Art von **Bootstrapping** in 1.24.1 nochmal gesondert hervorzuheben:

1.24.1: Wenn man die **aktuelle Version** eines **Self-compiling Compilers** $C_{w_i}^{w_i}$ der Wunschsprache L_{w_i} mithilfe von **früheren Versionen** seiner selbst kompiliert. Man schreibt also z.B. die aktuelle Version des Self-compiling Compilers in der Sprache $L_{w_{i-1}}$, welche von der früheren Version des Compilers, dem Self-compiling Compiler $C_{w_{i-1}}^{w_{i-1}}$ kompiliert wird und schafft es so **iterativ** immer umfangreichere Compiler zu bauen.^{e,f,g}

^aZ.B. mithilfe eines **Bootstrap Compilers**.

^bDer Begriff hat seinen Ursprung in der englischen **Redewendung** „pulling yourself up by your own bootstraps“, was im deutschen ungefähr der aus den **Lügengeschichten des Freiherrn von Münchhausen** bekannten Redewendung „sich am eigenen Schopf aus dem Sumpf ziehen“ entspricht.

^cHat man einmal einen solchen **Self-compiling Compiler** C_w^w auf der Maschine M zum laufen gebracht, so kann man den Compiler auf der Maschine M weiterentwickeln, ohne von externen Entitäten, wie einer bestimmten Sprache L_o , in der der Compiler oder eine frühere Version des Compilers ursprünglich geschrieben war abhängig zu sein.

^dEinen Compiler in der Sprache zu schreiben, die er selbst kompiliert und diesen Compiler dann sich selbst kompilieren zu lassen, kann eine gute **Probe aufs Exempel** darstellen, dass der Compiler auch wirklich funktioniert.

^eEs ist hierbei theoretisch nicht notwendig den **letzten** Self-compiling Compiler $C_{w_{i-1}}^{w_{i-1}}$ für das Kompilieren des **neuen** Self-compiling Compilers $C_{w_i}^{w_i}$ zu verwenden, wenn z.B. der **Self-compiling Compiler** $C_{w_{i-3}}^{w_{i-3}}$ auch bereits alle Funktionalitäten, die beim Schreiben des **Self-compiling Compilers** C_w^w verwendet werden kompilieren kann.

^fDer Begriff ist sinnverwandt mit dem **Booten** eines Computers, wo die wichtigste Software, der **Kernel** zuerst in den Speicher geladen wird und darauf aufbauend von diesem dann das Betriebssysteme, welches bei Bedarf dann **Systemsoftware**, Software, die das Ausführen von Anwendungssoftware ermöglicht oder unterstützt, wie z.B. Treiber. und **Anwendungssoftware**, Software, deren Anwendung darin besteht, dass sie dem Benutzer unmittelbar eine Dienstleistung zur Verfügung stellt, lädt.

^gJ. Earley und Sturgis, „A formalism for translator interactions“.

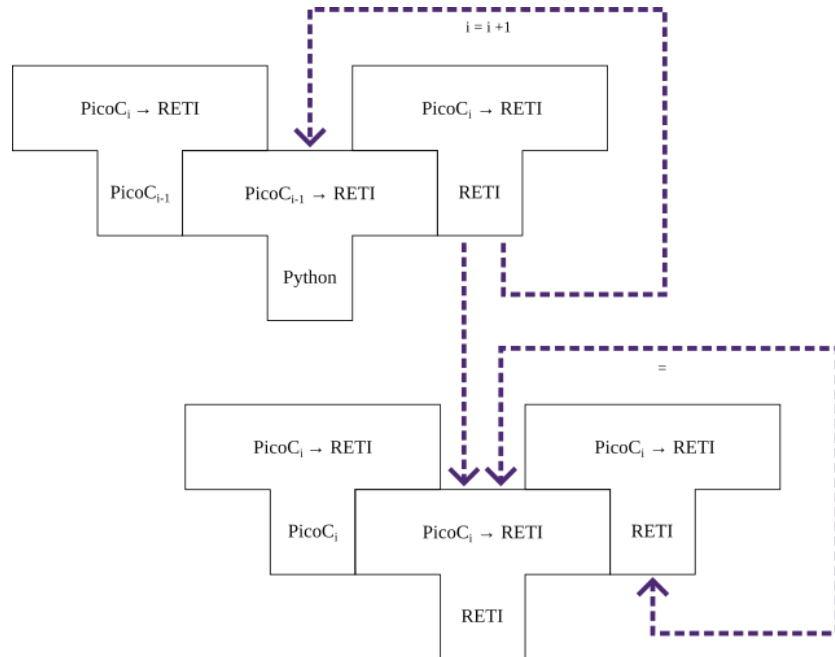


Abbildung 1.3: Iteratives Bootstrapping.

Anmerkung 🔍

Auch wenn ein **Self-compiling Compiler** $C_{w_i}^{w_i}$ in der Subdefinition 1.24.1 selbst in einer früheren Version $L_{w_{i-1}}$ der Programmiersprache L_{w_i} geschrieben wird, wird dieser nicht mit $C_{w_i}^{w_{i-1}}$ bezeichnet, sondern mit $C_{w_i}^{w_i}$, da es bei **Self-compiling Compilern** darum geht, dass diese zwar in der Subdefinition 1.24.1 eine frühere Version $C_{w_{i-1}}^{w_{i-1}}$ nutzen, um sich selbst kompilieren zu lassen, aber sie auch in der Lage sind sich selber zu kompilieren.

Literatur

Online

- *Earley Parser*. URL: <https://rahul.gopinath.org/post/2021/02/06/earley-parsing/> (besucht am 20.06.2022).
- *Parsers — Lark documentation*. URL: <https://lark-parser.readthedocs.io/en/latest/parsers.html> (besucht am 20.06.2022).

Bücher

- G. Siek, Jeremy. *Essentials of Compilation*. 28. Jan. 2022. URL: <https://iucompilercourse.github.io/IU-Fall-2021/> (besucht am 28.01.2022).

Artikel

- Earley, J. und Howard E. Sturgis. „A formalism for translator interactions“. In: *CACM* (1970). DOI: 10.1145/355598.362740.
- Earley, Jay. „An efficient context-free parsing“. In: 13 (1968). URL: <https://web.archive.org/web/20040708052627/http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/cmt-55/lti/Courses/711/Class-notes/p94-earley.pdf> (besucht am 10.08.2022).

Vorlesungen

- Nebel, Bernhard. „Theoretische Informatik“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: http://gki.informatik.uni-freiburg.de/teaching/ss20/info3/index_de.html (besucht am 09.07.2022).
- Scholl, Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157 (besucht am 09.07.2022).
- Scholl, Philipp. „Einführung in Embedded Systems“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: <https://earth.informatik.uni-freiburg.de/uploads/es-2122/> (besucht am 09.07.2022).
- Thiemann, Peter. „Compilerbau“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: <http://proglang.informatik.uni-freiburg.de/teaching/compilerbau/2021ws/> (besucht am 09.07.2022).
- — „Einführung in die Programmierung“. Vorlesung. Vorlesung. Universität Freiburg, 2018. URL: <http://proglang.informatik.uni-freiburg.de/teaching/info1/2018/> (besucht am 09.07.2022).

Sonstige Quellen

- *Earley parser*. In: *Wikipedia*. Page Version ID: 1090848932. 31. Mai 2022. URL: https://en.wikipedia.org/w/index.php?title=Earley_parser&oldid=1090848932 (besucht am 15.08.2022).