

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 Mist 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
1 Ergebnisse und Ausblick	1
1.1 Funktionsumfang	1
1.1.1 Kommandozeilenoptionen	1
1.1.2 Shell-Mode	3
1.1.3 Show-Mode	5
1.2 Qualitätssicherung	7
1.3 Erweiterungsideen	11
Appendix	A
RETI Architektur Details	A
Sonstige Definitionen	C
Bootstrapping	H
Literatur	L

Abbildungsverzeichnis

1.1	Show-Mode in der Verwendung.	7
2.1	Datenpfade der RETI-Architektur.	C
2.2	Cross-Compiler als Bootstrap Compiler.	I
2.3	Iteratives Bootstrapping.	K

Codeverzeichnis

1.1	Shellaufruf und die Befehle <code>compile</code> und <code>quit</code>	4
1.2	Shell-Mode und der Befehl <code>most_used</code>	5
1.3	Typischer Test.	9
1.4	Testdurchlauf.	11
1.5	Beispiel für Tail Call.	14

Tabellenverzeichnis

1.1	Kommandozeilenoptionen, Teil 1.	2
1.2	Kommandozeilenoptionen, Teil 2.	3
1.3	Makefileoptionen.	6
1.4	Testkategorien.	8
2.1	Load und Store Befehle.	A
2.2	Compute Befehle.	B
2.3	Jump Befehle.	B

Definitionsverzeichnis

2.1	T-Diagram Maschine	C
2.2	Bezeichner (bzw. Identifier)	C
2.3	Label	C
2.4	Assemblersprache (bzw. engl. Assembly Language)	D
2.5	Assembler	D
2.6	Objectcode	D
2.7	Linker	D
2.8	Transpiler (bzw. Source-to-source Compiler)	E
2.9	Rekursiver Abstieg	E
2.10	Linksrekursive Grammatiken	E
2.11	LL(k)-Grammatik	E
2.12	Earley Erkennen	F
2.13	Liveness Analyse	G
2.14	Live Variable	G
2.15	Graph Coloring	G
2.16	Interference Graph	G
2.17	Kontrollflussgraph	G
2.18	Kontrollfluss	H
2.19	Kontrollflussanalyse	H
2.20	Two-Space Copying Collector	H
2.21	Self-compiling Compiler	I
2.22	Minimaler Compiler	J
2.23	Bootstrap Compiler	J
2.24	Bootstrapping	J

Grammatikverzeichnis

1 Ergebnisse und Ausblick

Zum Schluss soll ein **Überblick** über das Resultat dessen, was im Kapitel ?? implementiert wurde gegeben werden. Im Unterkapitel 1.1 wird darauf eingegangen, ob die **versprochenen Funktionalitäten** des **PicoC-Compilers** aus Kapitel ?? alle implementiert werden konnten. Daraufhin wird mithilfe **kurzer Anleitungen** ein grober Einblick gegeben, wie auf diese Funktionalitäten zugegriffen werden kann und es wird auch auf Funktionalitäten **anderer mitimplementierter Tools** eingegangen. Im Unterkapitel 1.2 wird aufgezeigt, was zur **Qualitätssicherung** implementiert wurde, um zu gewährleisten, dass der **PicoC-Compiler** die Kompilierung der **Programmiersprache** L_{PicoC} in **Syntax** und **Semantik identisch** zur entsprechenden **Untermenge** der Programmiersprache L_C umsetzt. Als allerletztes wird im Unterkapitel 1.3 ein Ausblick gegeben, wie der PicoC-Compiler **erweitert** werden könnte.

1.1 Funktionsumfang

Alle Funktionalitäten, die in Kapitel ?? erläutert und versprochen wurden, konnten in dieser **Bachelorarbeit** implementiert werden. In Kapitel ?? wurde die Umsetzung **aller** dieser Funktionalitäten erklärt. Während der **Funktionsumfang** des **PicoC-Compiler** zum Stand des **Bachelorprojektes** noch sehr beschränkt war und einzig eine **Strukturierte Programmierung** mit `if(cond) { } else { }, while(cond) { }` usw. erlaubte und komplexere Programme nur mit **viel Aufwand** und **unübersichtlichem Spaghetticode** implementierbar waren, erlaubt es der **PicoC-Compiler** nachdem er in der **Bachelorarbeit** um **Felder**, **Zeiger**, **Verbunde** und **Funktionen** erweitert wurde mittels der **Funktionen** eine **Prozedurale Programmierung** umzusetzen. **Prozedurale Programmierung** zusammen mit der Möglichkeit **Felder**, **Zeiger** und **Verbunde** zu verwenden trägt zu einem **geordneteren**, **intuitiv verständlicheren** und **übersichtlicheren** Code bei.

Bei der Implementierung des **PicoC-Compilers** wurden verschiedene **Kommandozeilenoptionen** und **Modes** implementiert. Diese werden in den folgenden Unterkapiteln 1.1.1, 1.1.2 und 1.1.3 mithilfe kurzer **Anleitungen** erklärt.

Die kurzen **Anleitungen** in dieser **Schriftlichen Ausarbeitung** der Bachelorarbeit sollen nur zu einem **schnellen, grundlegenden Verständnis** der Verwendung des **PicoC-Compilers** und seiner **Kommandozeilenoptionen** und **Befehle** beihelfen, sowie zum Verständnis der **weiteren implementierten Tools**. Alle weiteren **Kommandozeilenoptionen** und **Befehle** sind für die Verwendung des PicoC-Compilers **unwichtig** und erweisen sich nur in **speziellen Situationen** als nützlich, weshalb für diese auf die **ausführlichere Dokumentation** unter [Link](https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/help-page.txt)¹ verwiesen wird.

1.1.1 Kommandozeilenoptionen

Will man einfach nur ein **Programm** `program.picoc` kompilieren ist das mit dem **PicoC-Compiler** genauso **unkompliziert**, wie mit dem **GCC** durch einfaches **Angeben der Datei**, die kompiliert werden soll: `> picoc_compiler program.picoc`. Als Ergebnis des Kompiliervorgangs wird eine Datei `program.reti` mit dem entsprechenden **RETI-Code** erstellt, wobei für die **Benennung der Datei** einfach nur der

¹https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/help-page.txt.

Basisname der Datei `program` an eine neue **Dateiendung** `.reti` angehängt wird².

Daneben gibt es allerdings auch die Möglichkeit **Kommandozeilenoptionen** `<cli-options>` in der Form `> picoc_compiler <cli-options> program.picoc` mitanzugeben, von denen die **wichtigsten** in Tabelle 1.1 erklärt sind. Alle weiteren **Kommandozeilenoptionen** können in der **Dokumentation** unter [Link](#)³ nachgelesen werden. Die letzte Spalte gibt den **Standardwert** an, der bei der normalen Nutzung des **PicoC-Compilers** gesetzt ist.⁴

Kommandozeilenoption	Beschreibung	Standardwert
<code>-i, --intermediate_stages</code>	Gibt Zwischenstufen der Kompilierung in Form der verschiedenen Tokens, Ableitungsbäume, Abstrakten Syntaxbäume der verschiedenen Passes in Dateien mit entsprechenden Dateieendungen aber gleichem Basisnamen aus. Im Shell-Mode erfolgt keine Ausgabe in Dateien, sondern nur im Terminal .	false , most_used: true
<code>-p, --print</code>	Gibt alle Dateiausgaben auch im Terminal aus. Diese Option ist im Shell-Mode dauerhaft aktiviert.	false (true im Shell-Mode und für den most_used-Befehl)
<code>-v, --verbose</code>	Fügt den verschiedenen Zwischenschritten der Kompilierung , unter anderem auch dem finalen RETI-Code Kommentare hinzu. Diese Kommentare beinhalten eine Anweisung oder einen Befehl aus einem vorherigen Pass , der durch die darunterliegenden Anweisungen oder Befehle ersetzt wurde. Wenn die <code>--run</code> -Option aktiviert ist, wird der Zustand der virtuellen RETI-CPU vor und nach jedem Befehl angezeigt.	false
<code>-vv, --double_verbose</code>	Hat dieselben Effekte , wie die <code>--verbose</code> -Option, aber bewirkt zusätzlich weitere Effekte . PicoC-Knoten erhalten bei der Ausgabe als zusammenhängende Abstrakte Syntaxbäume zusätzliche runde Klammern , sodass direkter abgelesen werden kann, wo ein Knoten anfängt und wo einer aufhört. In Fehlermeldungen werden mehr Tokens angezeigt, die an der Stelle der Fehlermeldung erwartet worden wären. Bei Aktivierung der <code>--intermediate_stages</code> -Option werden in den dadurch ausgegebenen Abstrakten Syntaxbäumen zusätzlich versteckte Attribute , die Informationen zu Datentypen und für Fehlermeldungen beinhalten angezeigt.	false
<code>-h, --help</code>	Zeigt die Dokumentation , welche ebenfalls unter Link gefunden werden kann im Terminal an. Mit der <code>--color</code> -Option kann die Dokumentation mit farblicher Hervorhebung im Terminal angezeigt werden.	false

Tabelle 1.1: Kommandozeilenoptionen, Teil 1.

²Beim **GCC** wird bei **Nicht-Angabe** eines **Dateinamen** mit der `-o` Option dagegen eine Datei mit der **festen** Bezeichnung **a.out** erstellt.

³https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/help-page.txt.

⁴In grau ist unter `most_used` des Weiteren der **Standardwert** bei der Verwendung des `most_used`-Befehls angegeben.

Kommandozeilen-option	Beschreibung	Standard-wert
-l, --lines	Es lässt sich einstellen, wieviele Zeilen rund um die Stelle an welcher ein Fehler aufgetreten ist angezeigt werden sollen.	2
-c, --color	Aktiviert farbige Ausgabe .	false, most_used: true
-t, --thesis	Filtert für die Codebeispiele in dieser Schriftlichen Ausarbeitung der Bachelorarbeit bestimmte Kommentare in den Abstrakten Syntaxbäumen heraus, damit alles übersichtlich bleibt.	false
-R, --run	Führt die RETI-Befehle , die das Ergebnis der Kompilierung sind mit einer virtuellen RETI-CPU aus. Wenn die --intermediate_stages-Option aktiviert ist, wird eine Datei <basename>.reti_states erstellt, welche den Zustand der RETI-CPU nach dem letzten ausgeführten RETI-Befehl enthält. Wenn die --verbose- oder --double_verbose-Option aktiviert ist, wird der Zustand der RETI-CPU vor und nach jedem Befehl auch noch zusätzlich in die Datei <basename>.reti_states ausgegeben.	false, most_used: true
-B, --process_begin	Setzt die relative Adresse , wo der Prozess bzw. das Codesegment für das ausgeführte Programm beginnt .	3
-D, --datasegment_size	Setzt die Größe des Datensegments . Diese Option muss mit Vorsicht gesetzt werden, denn wenn der Wert zu niedrig gesetzt wird, dann können die Globalen Statischen Daten und der Stack miteinander kollidieren .	32

Tabelle 1.2: Kommandozeilenoptionen, Teil 2.

Alle **kleingeschriebenen** Kommandozeilenoptionen, wie -i, -p, -v usw. betreffen den **PicoC-Compiler** und alle **großgeschrieben** Kommandozeilenoptionen, wie -R, -B, -D usw. betreffen den **RETI-Interpreter**.

1.1.2 Shell-Mode

Will man z.B. eine **Folge von Anweisungen** in der Programmiersprache L_{PicoC} **schnell** kompilieren ohne eine Datei erstellen zu müssen, so kann der **PicoC-Compiler** im sogenannten **Shell-Mode** aufgerufen werden. Hierzu wird der PicoC-Compiler **ohne Argumente** `> picoc_compiler` aufgerufen, wie es in Code 1.1 zu sehen ist.

Mit dem `> compile <cli-options> <seq-of-stmts>`-Befehl (oder der **Abkürzung** `cpl`) kann **PicoC-Code** zu **RETI-Code** kompiliert werden. Die Kommandozeilenoptionen <cli-options> sind dieselben, wie wenn der Compiler **direkt** mit Kommandozeilenoptionen aufgerufen wird. Die **wichtigsten** dieser **Kommandozeilenoptionen** sind in Tabelle 1.1 angegeben. Die angegebene **Folge von Anweisungen** <seq-of-stmts> wird dabei automatisch in eine main-Funktion eingefügt: `void main(){<seq-of-stmts>}`.

Mit dem Befehl `> quit` kann der **Shell-Mode** wieder **verlassen** werden.

```

> picoc_compiler
PicoC Shell. Enter `help` (shortcut `?`) to see the manual.
PicoC> cpl "6 * 7;";
----- RETI -----
SUBI SP 1;
LOADI ACC 6;
STOREIN SP ACC 1;
SUBI SP 1;
LOADI ACC 7;
STOREIN SP ACC 1;
LOADIN SP ACC 2;
LOADIN SP IN2 1;
MULT ACC IN2;
STOREIN SP ACC 2;
ADDI SP 1;
LOADIN BAF PC -1;

Compilation successfull

PicoC> quit

```

Code 1.1: Shellaufruf und die Befehle *compile* und *quit*.

Wenn man möglichst alle nützlichen **Kommandozeilenoptionen** direkt aktiviert haben will, bei denen es **keinen** Grund gibt sie nicht mitanzugeben, kann der Befehl `> most_used <cli-options> <seq-of-stmts>` (oder seine **Abkürzung** `mu`) genutzt werden. Auf diese Weise müssen diese Kommandozeilenoptionen **nicht** wie beim `compile`-Befehl jedes mal **selbst** angegeben werden. In Tabelle 1.1 sind in grau die **Standardwerte** der einzelnen **Kommandozeilenoptionen** angegeben, die bei dem Befehl `most_used` gesetzt werden. In Code 1.2 ist der `most_used`-Befehl in seiner Verwendung zu sehen.






Dadurch, dass die `--intermediate_stages-`, `print-` und die `--run-`Option beim `most_used`-Befehl aktiviert sind, werden die verschiedenen **Zwischenstufen** der Kompilierung, wie **Tokens**, **Ableitungsbaum**, **Passes** usw., sowie der **Zustand der RETI-CPU** nach der Ausführung des **letzten** Befehls in das Terminal **ausgegeben**. Aus **Platzgründen** ist das meiste allerdings mit `'...'` ausgelassen.

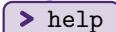
```

PicoC> mu "int var = 42;";
----- Code -----
// stdin.picoc:
void main() {int var = 42;}
----- Tokens -----
...
----- Derivation Tree -----
...
----- Derivation Tree Simple -----
...
----- Abstract Syntax Tree -----
...
----- PicoC Shrink -----
...
----- PicoC Blocks -----
...
----- PicoC Mon -----
...
----- Symbol Table -----
...
----- RETI Blocks -----
...
----- RETI Patch -----
...
----- RETI -----
SUBI SP 1;
LOADI ACC 42;
STOREIN SP ACC 1;
LOADIN SP ACC 1;
STOREIN DS ACC 0;
ADDI SP 1;
LOADIN BAF PC -1;
----- RETI Run -----
...
Compilation successfull

```

Code 1.2: Shell-Mode und der Befehl *most_used*.

Im **Shell-Mode** kann der **Cursor** mit den Pfeiltasten  und  bewegt werden. In der **Befehlshistorie** kann sich mit den Pfeiltasten  und  **rückwärts** und **vorwärts** bewegt werden. Mit  kann ein Befehl **automatisch vervollständigt** werden.

Es gibt für den **Shell-Mode** noch **weitere Befehle**, wie `color_toggle`, `history` etc. und **kleinere Funktionalitäten** für die Shell, die sich in der ein oder anderen Situation als **nützlich** erweisen können. Für die **Erklärung** dieser wird allerdings auf die **Dokumentation** unter [Link⁵](https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/help-page.txt) verwiesen, welche auch über den Befehl  angezeigt werden kann.

1.1.3 Show-Mode

Der **Show-Mode** ist ein Nebenprodukt der Implementierung des **PicoC-Compilers**. Dieser **Mode** wurde eigentlich nur implementiert, um beim **Testen** des PicoC-Compilers **Bugs** bei der Generierung des **RETI-**

⁵https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/help-page.txt.

Code zu **inspizieren**. Das ganze ist so umgesetzt, dass im Terminal eine **virtuelle RETI-CPU** angezeigt wird, mit dem kompletten, momentanen **Zustand** in Form aller **Register**, **SRAM**, **UART**, **EPROM** und einigen **weiteren Informationen**.

Die Möglichkeit des **Show-Mode**, die **RETI-Befehle** des übersetzten Programmes in **Ausführung zu sehen**, bringt auch einen großen **Lerneffekt** mit sich, weshalb der **Show-Mode** noch **weiterentwickelt** wurde, sodass auch **Studenten** ihn auf unkomplizierte Weise nutzen können.

Der **Show-Mode** kann auf die **einfachste Weise** mittels der `/Makefile` des **PicoC-Compilers** mit dem Befehl `> make show FILEPATH=<path-to-file> <more-options>` gestartet werden. Alle **einstellbaren Optionen** `<more-options>`, die für die **Makefile** gesetzt werden können, sind in Tabelle 1.3 aufgelistet.

Kommandozeilenoption	Beschreibung	Standardwert
FILEPATH	Pfad zur Datei, die im Show-Mode angezeigt werden soll.	<code>()</code>
TESTNAME	Name des Tests. Alles andere als der Basisname , wie die Dateiendung wird abgeschnitten.	<code>()</code>
EXTENSION	Dateiendung , die an TESTNAME angehängt werden soll, damit daraus z.B. <code>./tests/TESTNAME.EXTENSION</code> wird.	<code>reti_states</code>
NUM_WINDOWS	Anzahl Fenster auf die ein Dateiinhalte verteilt werden soll.	<code>5</code>
VERBOSE	Möglichkeit für eine ausführlichere Ausgabe die Kommandozeilenoption <code>-v</code> oder <code>-vv</code> zu aktivieren.	<code>()</code>
DEBUG	Möglichkeit die Kommandozeilenoption <code>-d</code> zu aktivieren, um bei <code>make test-show TESTNAME=<testname></code> den Debugger für den entsprechenden Test <code><testname></code> zu starten.	<code>()</code>

Tabelle 1.3: Makefileoptionen.

Alternativ kann der **Show-Mode** mit dem Befehl `make test-show TESTNAME=<testname> <more-options>` auch für einen der geschriebenen **Tests** im Ordner `/tests` gestartet werden. Der **Test** wird bei diesem Befehl **erst ausgeführt** und dann der **Show-Mode** gestartet.

Der **Show-Mode** nutzt den Terminal Texteditor **Neovim**⁶, um einen **Dateiinhalte** über mehrere **Fenster** verteilt anzuzeigen, so wie es in Abbildung 1.1 zu sehen ist. Für den **Show-Mode** wird eine eigene **Konfiguration für Neovim** verwendet, welche in der **Konfigurationsdatei** `/interpr_showcase.vim` spezifiziert ist.

Gedacht ist der **Show-Mode** vor allem dafür, etwas ähnliches wie ein **RETI-Debugger** zu sein und wird daher standardmäßig bei **Nicht-Angabe** einer **EXTENSION** auf die **Datei** oder den **Test** `<program>.reti_states` angewandt. Der **Show-Mode** kann aber auch dazu genutzt werden **andere Dateien**, welche verschiedene Zwischenschritte der Kompilierung darstellen, über mehrere Fenster **verteilt** anzuzeigen, indem **EXTENSION** auf eine andere **Dateiendung** gesetzt wird.

Im **Show-Mode** wird ein Trick angewandt, indem die verschiedenen **Zustände** der RETI-CPU **nicht** zur **Laufzeit** des **Show-Mode** berechnet werden, sondern schon **zuvor** berechnet wurden und nacheinander in die Datei `<program>.reti_states` ausgegeben wurden. Der **Show-Mode** macht nichts anderes, als immer an die Stelle zu springen, an welcher der **nächste Zustand** anfängt. Durch Drücken von `Tab` und `↑ -Tab` können auf diese Weise die **verschiedenen Zustände** der RETI-CPU **vor** und **nach** der Ausführung eines

⁶Home - Neovim.

Befehls **angezeigt** werden.

Index: 84	00019 ADD ACC CS;	00057 LOADIN SP ACC 2;	00095 MOVE BAF ACC;	00133 0
Instruction: STOREIN SP ACC 2;	00020 STOREIN BAF ACC -1;	00058 LOADIN SP IN2 1;	00096 ADDI SP 3;	00134 0
ACC: 42	00021 JUMP 44;	00059 ADD ACC IN2;	00097 MOVE SP BAF;	00135 0
ACC_SIMPLE: 42	00022 MOVE BAF IN1;	00060 STOREIN SP ACC 2;	00098 SUBI SP 4;	00136 0
IN1: 0	00023 LOADIN IN1 BAF 0;	00061 ADDI SP 1; <- PC	00099 STOREIN BAF ACC 0;	00137 0
IN1_SIMPLE: 0	00024 MOVE IN1 SP;	00062 LOADIN SP ACC 1;	00100 LOADI ACC 101;	00138 0 <- SP
IN2: 2	00025 SUBI SP 1;	00063 ADDI SP 1;	00101 ADD ACC CS;	00139 2
IN2_SIMPLE: 2	00026 STOREIN SP ACC 1;	00064 LOADIN BAF PC -1;	00102 STOREIN BAF ACC -1;	00140 42
PC: 2147483709	00027 LOADIN SP ACC 1;	00065 SUBI SP 1;	00103 JUMP -58;	00141 2
PC_SIMPLE: 61	00028 STOREIN DS ACC 1;	00066 LOADI ACC 2;	00104 MOVE BAF IN1;	00142 40
SP: 2147483786	00029 ADDI SP 1;	00067 STOREIN SP ACC 1;	00105 LOADIN IN1 BAF 0;	00143 2147483752
SP_SIMPLE: 138	00030 SUBI SP 1;	00068 LOADIN SP ACC 1;	00106 MOVE IN1 SP;	00144 2147483797 <- BAF
BAF: 2147483792	00031 LOADIN DS ACC 1;	00069 STOREIN BAF ACC -3;	00107 SUBI SP 1;	00145 40
BAF_SIMPLE: 144	00032 STOREIN SP ACC 1;	00070 ADDI SP 1;	00108 STOREIN SP ACC 1;	00146 2
CS: 2147483651	00033 SUBI SP 1;	00071 SUBI SP 1;	00109 LOADIN SP ACC 1;	00147 38
CS_SIMPLE: 3	00034 LOADI ACC 2;	00072 LOADIN BAF ACC -2;	00110 ADDI SP 1;	00148 2147483670
DS: 2147483766	00035 STOREIN SP ACC 1;	00073 STOREIN SP ACC 1;	00111 CALL PRINT ACC;	00149 2147483650
DS_SIMPLE: 118	00036 LOADIN SP ACC 2;	00074 SUBI SP 1;	00112 SUBI SP 1;	UART:
SRAM:	00037 LOADIN SP IN2 1;	00075 LOADIN BAF ACC -3;	00113 LOADIN BAF ACC -4;	00000 0
00000 JUMP 0;	00038 ADD ACC IN2;	00076 STOREIN SP ACC 1;	00114 STOREIN SP ACC 1;	00001 0
00001 2147483648	00039 STOREIN SP ACC 2;	00077 LOADIN SP ACC 2;	00115 LOADIN SP ACC 1;	00002 0
00002 0	00040 ADDI SP 1;	00078 LOADIN SP IN2 1;	00116 ADDI SP 1;	00003 0
00003 CALL INPUT ACC; <- CS	00041 LOADIN SP ACC 1;	00079 ADD ACC IN2;	00117 LOADIN BAF PC -1;	EPROM:
00004 SUBI SP 1;	00042 ADDI SP 1;	00080 STOREIN SP ACC 2;	00118 38 <- DS	00000 LOADI DS -2097152; <- IN1
00005 STOREIN SP ACC 1;	00043 CALL PRINT ACC;	00081 ADDI SP 1;	00119 0	00001 MULTI DS 1024;
00006 LOADIN SP ACC 1;	00044 LOADIN BAF PC -1;	00082 LOADIN SP ACC 1;	00120 0	00002 MOVE DS SP; <- IN2
00007 STOREIN DS ACC 0;	00045 SUBI SP 1;	00083 STOREIN BAF ACC -4;	00121 0	00003 MOVE DS BAF;
00008 ADDI SP 1;	00046 LOADI ACC 2;	00084 ADDI SP 1;	00122 0	00004 MOVE DS CS;
00009 SUBI SP 2;	00047 STOREIN SP ACC 1;	00085 SUBI SP 1;	00123 0	00005 ADDI SP 149;
00010 SUBI SP 1;	00048 LOADIN SP ACC 1;	00086 LOADIN BAF ACC -4;	00124 0	00006 ADDI BAF 2;
00011 LOADIN DS ACC 0;	00049 STOREIN BAF ACC -3;	00087 STOREIN SP ACC 1;	00125 0	00007 ADDI CS 3;
00012 STOREIN SP ACC 1;	00050 ADDI SP 1;	00088 LOADIN SP ACC 1;	00126 0	00008 ADDI DS 118;
00013 MOVE BAF ACC;	00051 SUBI SP 1;	00089 ADDI SP 1;	00127 0	00009 MOVE CS PC;
00014 ADDI SP 3;	00052 LOADIN BAF ACC -2;	00090 CALL PRINT ACC;	00128 0	
00015 MOVE SP BAF;	00053 STOREIN SP ACC 1;	00091 SUBI SP 2;	00129 0	Index: 85
00016 SUBI SP 5;	00054 SUBI SP 1;	00092 SUBI SP 1;	00130 0	Instruction: ADDI SP 1;
00017 STOREIN BAF ACC 0;	00055 LOADIN BAF ACC -3;	00093 LOADIN BAF ACC -4;	00131 0	ACC: 42
00018 LOADI ACC 19;	00056 STOREIN SP ACC 1;	00094 STOREIN SP ACC 1;	00132 0	ACC_SIMPLE: 42

Abbildung 1.1: Show-Mode in der Verwendung.

Zur **besseren Orientierung** wird für **alle Register** ein mit der Registerbezeichnung beschrifteter **Zeiger** <- REG an Adressen im **EPROM**, **UART** und **SRAM** angezeigt, je nachdem, ob der **Wert** im entsprechenden Register nach der **Memory Map** dem **Adressbereich** von **EPROM**, **UART** oder **SRAM** entspricht.

Durch Drücken von **Esc** oder **q** kann der **Show-Mode** wieder verlassen werden. Es gibt für den **Show-Mode** noch viele **weitere Tastenkürzel**, die sich in der ein oder anderen Situation als **nützlich** erweisen können. Für die **Erklärung** aller **weiteren Tastenkürzel** wird allerdings auf die **Dokumentation** unter [Link⁷](#) verwiesen. Des Weiteren stehen durch die Nutzung des Terminal Texteditors **Neovim** auch alle **Funktionalitäten** dieses mächtigen Terminal Texteditors zur Verfügung, welche mittels der Eingabe von **:help** **nachgelesen** werden können oder mittels der Eingabe von **:Tutor** mithilfe einer kurzen **Einführungsanleitung** **erlernt** werden können.

1.2 Qualitätssicherung

Um verifizieren zu können, dass der **PicoC-Compiler** sich genauso verhält, wie er soll, müssen die **Beziehungen** aus Diagramm ??⁸ genauso für den **PicoC-Compiler** gelten. Für den **PicoC-Compiler** lässt sich ein ebensolches Diagramm 1.2.1 definieren. Ein **beliebiges** Testprogramm P_{PicoC} in der Sprache L_{PicoC} muss die **gleiche Semantik** haben, wie das entsprechend **kompilierte** Programm P_{RETI} in der Sprache L_{RETI} , trotz der **unterschiedlichen Sprache**.

Die Beziehungen im Diagramm 1.2.1 werden mithilfe von **Tests** verifiziert. Die **Tests** für den **PicoC-Compiler** sind hierbei im Verzeichnis **/tests** bzw. unter [Link⁹](#) zu finden. **Eingeteilt** sind die Tests in die folgenden **Kategorien** in Tabelle 1.4.

⁷https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/help-page.txt.

⁸In Unterkapitel ??.

⁹https://github.com/matthejue/PicoC-Compiler/tree/new_architecture/tests.

Testkategorie	Beschreibung
basic	Einfache Tests, welche die grundlegenden Funktionalitäten des PicoC-Compilers testen.
advanced	Tests, die Spezialfälle und Kombinationen verschiedener Funktionalitäten des PicoC-Compilers testen.
hard	Tests, die lang und komplex sind. Für diese Tests müssen die Funktionalitäten des PicoC-Compilers in perfekter Harmonie miteinander funktionieren.
example	Tests, die bekannte Algorithmen darstellen und daher als gutes, repräsentatives Beispiel für die Funktionsfähigkeit des PicoC-Compilers dienen.
error	Tests, die Fehlermeldungen testen. Für diese Tests wird keine Verifikation ausgeführt.
exclude	Tests, für welche aufgrund vielfältiger Gründe keine Verifikation ausgeführt werden soll.
thesis	Tests, die vorher Codebeispiele für diese Schriftliche Ausarbeitung der Bachelorarbeit waren und etwas umgeschrieben wurden, damit nicht nur das Durchlaufen dieser Tests getestet wird.
tobias	Tests, die der Betreuer dieser Bachelorarbeit, M.Sc. Tobias Seufert geschrieben hat.

Tabelle 1.4: Testkategorien.

Dass ein Programm P_{PicoC} und das Programm P_{RETI} , welches das **kompilierte** P_{PicoC} ist nach Diagramm 1.2.1 die **gleiche Semantik** haben, lässt sich mit einer **hohen Wahrscheinlichkeit** gewährleisten, wenn die Tests so konstruiert sind, dass es sehr **unwahrscheinlich** ist, zufällig bei der gewählten Eingabe die spezifische Ausgabe zu erhalten. Wenn **immer mehr Tests**, die alle einen unterschiedlichen **Teil der Semantik** der Sprache L_{PicoC} abdecken vorliegen, bei denen die jeweiligen Programme P_{PicoC} und P_{RETI} interpretiert die **gleiche Ausgabe** haben, dann kann mit **immer höherer Wahrscheinlichkeit** von einem **funktionierenden** Compiler ausgegangen werden.

Die Kante vom Testprogramm P_{PicoC} zur *Ausgabe* aus Diagramm 1.2.1 ist so umgesetzt, dass jeder Test im `/tests`-Verzeichnis eine `// expected:<space_seperated_output>`-Zeile hat. Der **Schreiber des Tests** übernimmt die Rolle des entsprechenden **Interpreters** aus Diagramm ???. Die **erwartete Ausgabe** `<space_seperated_output>` ist seine eigene Interpretation des **PicoC-Codes**.

Ein Beispiel für einen **Test** ist in Code 1.3 zu sehen. Die Tests werden mithilfe des Bashskripts `/run_tests.sh` **ausgeführt** oder mithilfe der `/Makefile` mit dem Befehl `> make test`, welcher einfach nur dieses Bashskript ausführt. Bei der **Ausführung** des Bashskripts `/run_tests.sh`, wird als erstes für **jeden** Test das Bashskript `/extract_input_and_expected.sh` ausgeführt, welches die Zeilen `// in:<space_seperated_input>`, `// expected:<space_seperated_output>` und `// datasegment:<datasegment_size>` extrahiert¹⁰ und die entsprechenden Zeichenketten `<space_seperated_input>`, `<space_seperated_output>` und `<datasegment_size>` in **neu** erstellte Dateien `<program>.in`, `<program>.out_expected` und `<program>.datasegment_size` schreibt. Das **letzte Skript** kann ebenfalls mit dem Befehl `> make extract` ausgeführt werden.

Die Datei `<program>.in` enthält **Eingaben**, welche durch `input()`-Funktionsaufrufe im Programm P_{PicoC} eingelesen werden. Die Datei `<program>.out_expected` enthält zu **erwartende Ausgaben**, welche durch `print(<exp>)`-Funktionsaufrufe im Programm P_{PicoC} ausgegeben werden. Die Datei `<program>.out`, die später genauer erläutert wird, enthält die **tatsächlichen Ausgaben** der `print(<exp>)`-Funktionsaufrufe bei der **Ausführung des Testprogramms** P_{PicoC} . Die Datei `<program>.datasegment_size` enthält die **Größe des**

¹⁰Falls vorhanden.

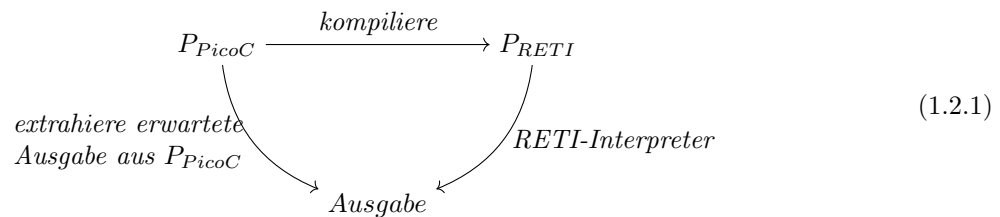
Datensegments für die Ausführung des entsprechenden Tests.

```
// in:21 2 6 7
// expected:42 42
// datasegment:4

void main() {
    print(input() * input());
    print(input() * input());
}
```

Code 1.3: *Typischer Test.*

Die Kante vom Programm P_{RETI} zur *Ausgabe* aus Abbildung 1.2.1 ist dadurch umgesetzt, dass das Programm P_{RETI} vom **RETI-Interpreter** interpretiert wird und jedes mal beim Antreffen des **RETI-Befehls** CALL PRINT ACC, der entsprechende **Inhalt** des ACC-Registers in die Datei $\langle\text{program}\rangle.out$ ausgegeben wird. Ein Test kann¹¹ die **Korrektheit** des **Teils der Semantik** der Sprache L_{PicoC} , die er abdeckt verifizieren, wenn der **Inhalt** von $\langle\text{program}\rangle.out_expected$ und $\langle\text{program}\rangle.out$ **identisch** ist.



Allerdings gibt es bei dem Testverfahren, welches in Diagramm 1.2.1 dargestellt ist ein **Problem**, denn der **Schreiber der Tests** ist in diesem Fall die **gleiche Person**, die auch den **PicoC-Compiler implementiert** hat. Wenn der **Schreiber** der Tests bzw. **Implementierer** des PicoC-Compilers ein **falsches Verständnis** davon hat, wie das Ergebnis eines Ausdrucks berechnet wird, so wird dieser sowohl in den **Tests** als auch in seiner **Implementierung** etwas als Ergebnis erwarten bzw. etwas implementieren, was nicht der eigentlichen **Semantik** von L_{PicoC} entspricht¹². Die **Tests** können dann nur **bestätigen**, dass der PicoC-Compiler so implementiert wurde, wie der **Implementierer** sich die **Semenatik** der Sprache L_{PicoC} vorstellt.

Aus diesem Grund muss hier eine **weitere Maßnahme** eingeführt werden, welche in Diagramm 1.2.2 dargestellt ist. Diese Maßnahme gewährleistet, dass die *Ausgabe* sich auf jeden Fall aus der tatsächlichen **Semantik** der Sprache L_{PicoC} ¹³ ergeben muss. Das wird erreicht, indem wie in Diagramm 1.2.2 dargestellt ist, überprüft wird, ob die *Ausgabe* des Pfades $(P_{PicoC}, \text{Ausgabe})$ mit der *Ausgabe* des Pfades von $(P_C, P_{X_{86.64}}, \text{Ausgabe})$ **identisch** ist.

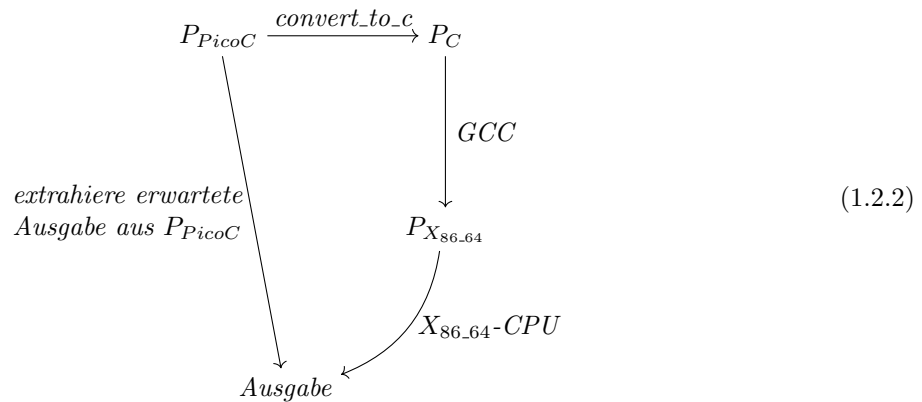
Im Diagramm 1.2.2 hat die Kante *extrahiere erwartete Ausgabe aus P_{PicoC}* die **gleiche Umsetzung**, wie die entsprechende Kante in Diagramm 1.2.1, welche bereits erklärt wurde. Die Kante **GCC**¹⁴ zur **Kompilierung** des Programms P_C von der Programmiersprache L_C in die Maschinensprache $L_{X_{86.64}}$ zu $P_{X_{86.64}}$ verwendet wird. Die Kante $X_{86.64}-CPU$ ist so umgesetzt, dass sie das Programm $P_{X_{86.64}}$ auf einer $X_{86.64}-CPU$ **ausführt**, wobei hierfür zumindestens beim Computer des **Implementierers des PicoC-Compilers** eine $X_{86.64}-CPU$ verwendet wird.

¹¹Mit einer **bestimmten Wahrscheinlichkeit**.

¹²Welche **identisch** zu einer Teilmenge von L_C ist.

¹³Die eine **Untermenge** von L_C ist.

¹⁴*GCC, the GNU Compiler Collection - GNU Project.*



Das Programm P_C ergibt sich aus dem Testprogramm P_{PicoC} durch **Ausführen** des Pythonskripts `/convert_to_c.py`, welches **später näher erläutert** wird. Dieses Pythonskript lässt sich ebenfalls mithilfe der `/Makefile` und dem Befehl `> make convert` ausführen.

Der **Trick** liegt hierbei in der Verwendung des **GCC** für die Kante $(P_C, P_{X86.64})$. Beim **GCC** handelt es sich um einen Compiler der Sprache L_C , der somit mit Ausnahme der `print()` und `input()`-Funktionen auch die Sprache L_{PicoC} kompilieren kann. Der **GCC** setzt aufgrund seiner bekanntermaßen **vielfachen Verwendung** auf der Welt und seinem **sehr langem Bestehen** seit 1987^{15 16} die **Semantik** der Sprache L_C , vor allem für die kleine Untermenge, welche L_{PicoC} darstellt mit sehr hoher Wahrscheinlichkeit **korrekt** um.

Durch das **Abgleichen** mit dem **GCC** in Diagramm 1.2.2 wird etwas wichtiges **sichergestellt**. Durch diese zweifache Überprüfung bestätigen die Tests nicht nur die Interpretation, die der **Schreiber** der Tests und **Implementierer** des PicoC-Compilers von der Semantik der Sprache L_{PicoC} hat, sondern stellen die tatsächliche **Einhaltung der Semantik** der Sprache L_{PicoC} sicher.

Für die zweifache Überprüfung durchläuft jeder Test eine **Verifikation**, wie sie in Diagramm 1.2.2 dargestellt ist. In dieser wird **verifiziert**, ob bei der **Kompilierung** des Testprogramms P_C mit dem **GCC** und **Ausführung** des hieraus generierten $X_{86.64}$ -Maschinencodes die Ausgabe **identisch** zur erwarteten Ausgabe `// expected:<space_seperated_output>` des Testschreibers ist.

Für die **Verifikation** ist das Bashskript `/verify_tests.sh` verantwortlich, welches mithilfe der `/Makefile` mit dem Befehl `> make verify` ausgeführt werden kann. Beim Befehl `> make test` wird dieses Bashskript **vor** dem eigentlichen Testen¹⁷ ausgeführt. In Code 1.4 ist ein **Testdurchlauf** mit `> make test` zu sehen.

Hierbei zeigt **Verified: 80/80** an, wieviele der Tests, die überhaupt verifizierbar sind¹⁸ sich **verifizieren** lassen, indem sie mit dem **GCC** ohne Fehlermeldung **durchlaufen** und die **erwartete Ausgabe erfüllen**. **Not verified:** gibt die **nicht** mit dem GCC **verifizierten** Tests an. **Running through: 118 / 118** zeigt an, wieviele Tests mit dem **PicoC-Compiler durchlaufen**. **Not running through:** gibt die **nicht** mit dem PicoC-Compiler **durchlaufenden** Tests an. **Passed: 118 / 118** zeigt an, bei wievielen Tests die Ausgabe beim Ausführen mit der erwarteten Ausgabe **identisch** ist. **Not passed:** zeigt die Tests an, bei denen das **nicht** der Fall ist.

¹⁵History - GCC Wiki.

¹⁶In der langen **Bestehenszeit** und bei der **vielen Verwendung** wurden die **allermeisten kritischen Bugs** wahrscheinlich schon gefunden.

¹⁷Das **eigentliche Testen** ist hier das **Überprüfen**, ob der interpretierte RETI-Code des Tests, der vom PicoC-Compiler kompiliert wurde die **gleiche Ausgabe** hat, wie der Schreiber des Tests **erwartet**.

¹⁸Also **alle** Tests aus den **Kategorien basic, advanced, hard, example, thesis** und **tobias**.

```

> make test
=====
= ./tests/basic_array_init.picoc =
=====
...
=====
=          Verification          =
=====
./tests/basic_array_init.c
...
=====
=          Results              =
=====
Verified: 80 / 80
Not verified:
Running through: 118 / 118
Not running through:
Passed: 118 / 118
Not passed:

```

Code 1.4: Testdurchlauf.

Der Befehl `> make test <more-options>` lässt sich ebenfalls mit den **Makefileoptionen** `<more-options>` TESTNAME, VERBOSE und DEBUG aus Tabelle 1.3 kombinieren.

Das Pythonskript `/convert_to_c.py` ist notwendig, da L_{PicoC} sich bei den Funktionen `print(<exp>)` und `input()` von der **Syntax** der Sprache L_C **unterscheidet**. Es muss z.B. `printf("%d", 12)` anstelle von `print(12)` geschrieben werden. Für die Sprache L_{PicoC} erfüllen die Funktionen `print(<exp>)` und `input()` nur den **Zweck**, dass sie zum **Testen des PicoC-Compilers** gebraucht werden. Über die Funktion `input()` soll es möglich sein, für eine bestimmte **Eingabe** die **Ausgabe** über die Funktion `print(<exp>)` testen können. Aus diesem Grund ist es notwendig die **Syntax** dieser Funktionen in L_C zu übersetzen.

Die Funktion `print(<exp>)` wird vom Pythonskript `/convert_to_c.py` zu `printf("%d", <exp>)` **übersetzt**. Zuvor muss über `#include<stdio.h>` die **Standard-Input-Output Bibliothek** `<stdio.h>` eingebunden werden. Bei der Funktion `input()` wurde **nicht** der aufwändige **Umweg** genommen, die Funktion `input()` durch ihre entsprechende Funktion in der Sprache L_C zu ersetzen. Es geht viel direkter, indem **nacheinander** die `input()`-Funktionen durch entsprechende Eingaben aus der Datei `<program>.in` ersetzt werden. Man schreibt einfach **direkt** die Werte hin, welche die `input()`-Funktionen normalerweise einlesen sollten.

1.3 Erweiterungsideen

Mit dem **Funktionsumfang** des **PicoC-Compilers**, der in Unterkapitel 1.1 erläutert wurde muss allerdings das Ende der Fahnenstange noch **nicht** erreicht sein. Weitere Ideen, die im **PicoC-Compiler**¹⁹ implementiert werden könnten, wären:

- **Register Allokation:** Variablen werden nicht nur **Adressen** im **Hauptspeicher** zugewiesen, sondern an erster Stelle **Registern**. Erst wenn alle Register **voll** sind, werden Variablen an Adressen im **Hauptspeicher** gespeichert. Da hat den Grund, dass der **Zugriff auf Register** deutlich **schneller**

¹⁹Möglicherweise ja im Rahmen eines **Masterprojektes** 😊.

ist, als der **Zugriff auf den Hauptspeicher**. Um die Variablen möglichst optimal **Locations** (Definition ??) zuzuweisen, wird mithilfe einer **Liveness Analyse** (Definition 2.13) ein **Interferenzgraph** (Definition 2.16) mit **Variablen** als Knoten aufgebaut. Auf den **Interferenzgraph** wird ein **Graph Coloring** Algorithmus (Definition 2.15) angewandt, der den **Variablen** Zahlen zuordnet. Die **ersten** Zahlen entsprechen **Registern**, aber ab einem bestimmten Zahlenwert, wenn alle Register zugeordnet sind, entsprechen die Zahlen **Adressen auf dem Hauptspeicher**. Sobald eine Programmiersprache es erfordert für die Kompilierung **Blöcke** in den Passes einzuführen²⁰, muss die **Liveness Analyse** nach Ansätzen der **Kontrollflussanalyse** (Definition 2.19) **iterativ** unter Verwendung eines **Kontrollflussgraphen** (Definition 2.17) separat auf die verschiedenen **Blöcke** angewendet werden, bis sich an den Live Variablen **nichts** mehr **ändert**.²¹

- **Tail Call:** Wenn ein Funktionsaufruf der **letzte ausgeführte Ausdruck** in einem Funktionsblock ist, wird der **Stackframe** dieser aufrufenden Funktion **nicht** mehr **gebraucht**, da **nicht** mehr in diese Funktion **zurückgekehrt** werden muss²². Daher kann der **Stackframe** der aufrufenden Funktion **entfernt** werden, **bevor** der **Funktionsaufruf** getätigt wird. Der **Vorteil** ist, dass eine rekursive Funktion, die **nur Tail Calls** ausführt, mit **Stackframes** nur eine **konstante Menge** an **Speicherplatz** auf dem **Stack** verbraucht. In Code 1.5 sind **zwei Tail Calls** markiert.
- **Partielle Evaluation:** Bei Ausdrücken, wie z.B. `4 + input() - 2`, `input() * 1` oder `0 + input() * 2` können **Teilausdrücke** bereits **während** des **Kompilierens partiell** zu `2 + input()`, `input()` und `input() * 2` berechnet werden. Dies kann durch einen neuen **PicoC-Eval Pass** umgesetzt werden, der **vor** oder **nach** dem **PicoC-Shrink Pass** den jeweiligen Abstrakten Syntaxbaum in eine neue Abstrakte Syntax der Sprache L_{PicoC_Eval} umformt. In der **Abstrakten Grammatik** der Sprache L_{PicoC_Eval} sind z.B. **binäre Operationen** zwischen zwei `Num(str)`-PicoC-Knoten **nicht möglich**. Diese **partielle Vorberechnung** kann auch auf **Konstanten** und **Variablen** ausgeweitet werden. Der **Vorteil** ist, dass hierdurch weniger **RETI-Code** generiert wird und weniger **RETI-Code** bedeutet wiederum eine **schnellere Programmausführung**.
- **Lazy Evaluation:** Bei Ausdrücken, wie z.B. `var1 && 42 / 0` oder `var2 || 42 / 0`, wobei z.B. `var1 = 0` und `var2 = 1` müssen diese Ausdrücke nur **soweit** berechnet werden, wie es **benötigt** wird. Sobald bei einer Aneinanderreihung von `&&`-Operationen einmal eine 0 auftaucht, muss der Rest des Ausdrucks **nicht** mehr berechnet werden, da mit dem Auftauchen der 0 bereits klar ist, dass dieser Ausdruck sich zu 0 auswertet. Genauso für eine Aneinanderreihung von `||`-Operationen und dem Auftauchen einer 1. Daher kommt es in beiden gerade gebrachten Beispielen aufgrund der Division durch 0 **nicht** zu einer **DivisionByZero-Fehlermeldung**, da die Ausdrücke garnicht so weit ausgewertet werden. Im Unterschied zur **Partiellen Evaluation** läuft **Lazy Evaluation**²³ zur **Laufzeit** ab.
- **Objektorientierung:** Wie in der Programmiersprache L_{C++} müssen **Klassen** und `new`-, `new[]`-, `delete`-, `delete[]`- und `::`-Operatoren eingeführt werden. Die Speicherung eines **Objekts** ist ähnlich wie bei **Verbunden**.
- **Mehrere Dateien:** **Funktionen** und **Attribute** werden in **mehrere Dateien** aufgeteilt, welche **separat** programmiert und kompiliert werden können. Für die **Deklaration** von **Funktionen** und **Attributen** werden **.h-Headerdateien** verwendet und für deren **Definitionen** sind **.c-Quellcodedateien** da. Hierbei ist der **Basisname** einer **.h-Headerdatei** **identisch** zu dem der entsprechenden **.c-Quellcodedatei**. Dateien werden über `#include "file"` eingebunden, was einem **direkten einfügen** des entsprechenden Codes der eingebundenen Datei an genau dieser Stelle in die einbindende Datei entspricht. Über einen **Linker** (Definition 2.7) können **kompilierte .o-Objektdaten** (Definition 2.6) zusammengefügt werden. Der **Linker** achtet darauf **keinen doppelten Code** zuzulassen.

²⁰Das ist **notwendig**, sobald es sich um eine **Strukturierte Programmiersprache** (Definition ??) handelt.

²¹Die in diesem **Unterpunkt** erwähnten **Begriffe** werden nur **grob** erläutert, da sie für den **PicoC-Compiler** keine Rolle spielen. Aber sie wurden erwähnt, damit in dieser **Bachelorarbeit** auch das übliche Vorgehen Erwähnung findet.

²²Was der Grund ist, warum ein **Stackframe** überhaupt angelegt wird, damit später beim **Rücksprung** aus der **aufgerufenen Funktion** die Ausführung mit allen Variablen, wie **vor dem Funktionsaufruf** fortgesetzt werden kann.

²³Es gibt hierfür leider keinen **deutschen Begriff**, der geläufig ist.

- **malloc und free:** Es wird eine **Bibliothek**, wie die Bibliothek `stdlib`²⁴ mit den Funktionen `malloc` und `free` implementiert, deren `.h`-Headerdatei mittels `#include "malloc_and_free.h"` eingebunden wird. Es braucht eine neue **Kommandozeilenoption** `-l`, um dem **Linker** verwendete Bibliotheken mitzuteilen. Aufgrund der Einführung von `malloc` und `free`, wird im **Datensegment** der Abschnitt nach den **Globalen Statischen Daten** als **Heap** bezeichnet, der mit dem **Stack** kollidieren kann. Im **Heap** wird von der `malloc`-Funktion **Speicherplatz allokiert** und ein **Zeiger** auf den allokierten Speicherplatz **zurückgegeben**. Dieser **Speicherplatz** kann von der `free`-Funktion wieder **freigegeben** werden. Um zu wissen, wo und wieviel Speicherplatz an diesen Stellen im **Heap** zur **Allokation** frei ist, muss dies in einer **Datenstruktur** abgespeichert werden.
- **Garbage Collector:** Anstelle der `free`-Funktion kann auch einfach die `malloc`-Funktion direkt so implementiert werden, dass sobald der Speicherplatz auf dem **Heap** knapp wird, Speicherplatz **freigegeben** wird. Es soll Speicherplatz freigegeben werden, der sowieso **unmöglich** in der **Zukunft** mehr **gebraucht** werden würde. Auf eine sehr einfache Weise lässt sich dies mit dem **Two-Space Copying Collector** (Definition 2.20) implementieren.
- **stdio.h:** Die Funktionen `print` und `input` werden nicht über den **Trick** einen eigenen **RETI-Befehl** `CALL (PRINT | INPUT) ACC` für den **RETI-Interpreter** zu definieren, der einfach **direkt** das **Ausgeben** und **Eingaben entgegennehmen** übernimmt gelöst, sondern über eine eigene **stdio-Bibliothek** mit `print`- und `input`-Funktionen, welche die **UART** verwenden, um z.B. an einem simpel gehaltenen simulierten **Monitor** Daten zu übertragen, die dieser anzeigt.
- **Feld mit Länge:** Man könnte in einer **Bibliothek** einen eigenen **Felddatentyp**, wie in der Programmiersprache L_{C++} mit dem Datentyp `std::vector` über eine **Klasse** implementieren, der seine **Anzahl Elemente** an den **Anfang** des Felds speichert, sodass über eine **Methode** `size` die **Anzahl Elemente** direkt über die **Variable des Felds** selbst ausgelesen werden kann (z.B. `vec.var.size`) und **nicht** in einer **seperaten Variable** gespeichert werden muss.
- **Maschinencode in binärer Repräsentation:** Maschinencode wird nicht, wie momentan beim **PicoC-Compiler** in **menschenlesbarer Repräsentation** ausgegeben, sondern in **binärer Repräsentation** nach dem **Intruktionsformat**, welches in der Vorlesung C. Scholl, „Betriebssysteme“ festgelegt wurde.
- **PicoPython:** Da das **Lark Parsing Toolkit** verwendet wurde, welches das **Parsen** über eine selbst angegebene **Konkrete Grammatik** übernimmt, könnte mit **relativ geringem Aufwand** ein Konkrete Grammatik definiert werden, die eine zur Programmiersprache L_{Python} **ähnliche Konkrete Syntax** beschreibt. Die **Konkrete Syntax** einer Programmiersprache lässt sich durch Austauschen der Konkreten Grammatik **sehr einfach** ändern, nur die **Semanatik** zu ändern kann **deutlich aufwändiger** sein. Viele der **PicoC-Knoten** könnten für die Programmiersprache $L_{PicoPython}$ **wiederverwendet** werden und viele **Passes** müssten nur erweitert werden.
- **Call by Reference:** Über das wiederverwenden des `&`-Symbols für **Parameter** bei **Funktionsdeklaration** und **Funktionsdefinition**, wie es in der Vorlesung P. Scholl, „Einführung in Embedded Systems“ erklärt wurde.
- **PicoC-Debugger:** Es wird eine neue **Kommandozeilenoption**, z.B. `-g` eingeführt durch welche spezielle **Informationen** in den RETI-Code geschrieben werden, die einem **Debugger** unter anderem mitteilen, wo die **RETI-Befehle** für eine Anweisungen **beginnen** und wo sie **aufhören** usw., damit der **Debugger** weiß, bis wohin er die **RETI-Befehle** ausführen soll, damit er eine Anweisung abgearbeitet hat.
- **Bootstrapping:** Mittels **Bootstrapping** lässt sich der **PicoC-Compiler** unabhängig von der Sprache L_{Python} und der **Maschine**, die das **cross-compilen** (Definition ??) übernimmt machen. Im Unterkapitel 1.3 wird genauer hierauf eingegangen. Hierdurch wird der **PicoC-Compiler** zum einem **Compiler**

²⁴Auch engl. **General Purpose Standard Library** genannt.

für die **RETI-CPU** gemacht, der auf der RETI-CPU selbst läuft.

```
1 // in:42
2 // expected:1
3
4 int ret1() {
5     return 1;
6 }
7
8 int ret0() {
9     return 0;
10 }
11
12 int tail_call_fun(int bool_val) {
13     if (bool_val) {
14         return ret1();
15     }
16     return ret0();
17 }
18
19 void main() {
20     print(tail_call_fun(input()));
21 }
```

Code 1.5: Beispiel für Tail Call.

Anmerkung 🔍

Partielle Evaluation und Lazy Evaluation wurden im PicoC-Compiler **nicht** implementiert, da dieser als **Lerntool** gedacht ist und diese Funktionalitäten den **RETI-Code** für Studenten **schwerer verständlich** machen könnten, da die **Codeschnipsel** und damit verbundene **Paradigmen** aus der Vorlesung **nicht** mehr so einfach **nachvollzogen** werden können und das **schwerere Ausmachen** können von **Orientierungspunkten** und **Fehlen erwarteter Codeschnipsel** leichter zur **Verwirrung** bei den Studenten führen könnte.

Appendix

Dieses Kapitel dient als Lagerstätte für **Definitionen**, **Tabellen**, **Abbildungen** und ganze **Unterkapitel**, die zum Erhalt des **roten Fadens** und des **Leseflusses** in den vorangegangenen Kapiteln hierher ausgelagert wurden. Im Unterkapitel **RETI Architektur Details** können einige Details der **RETI-Architektur** nachgeschaut werden, die im Kapitel ?? den Lesefluss **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

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 2.1: Load und Store Befehle.

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 2.2: Compute Befehle.

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: $\langle PC \rangle := \langle PC \rangle + 1$
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 2.3: Jump Befehle.

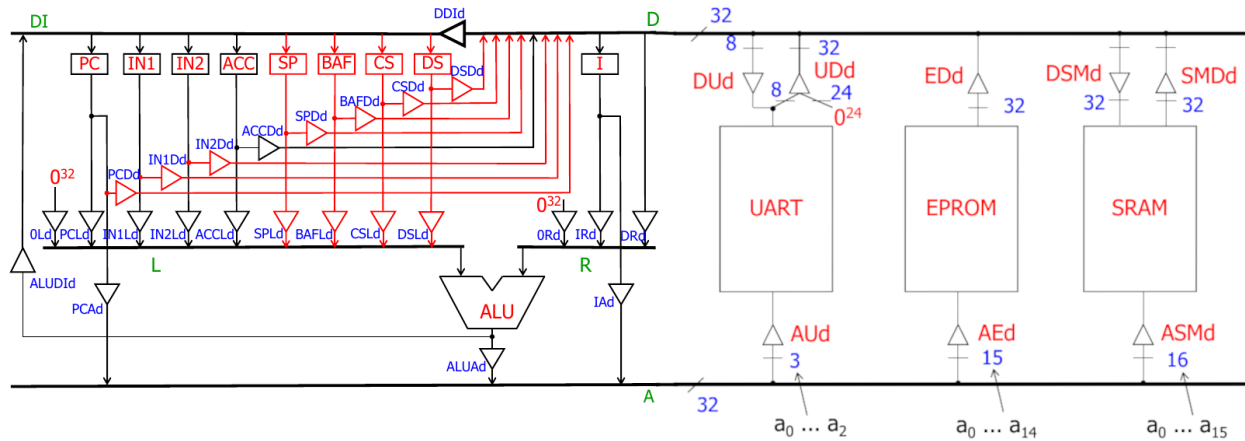


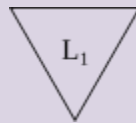
Abbildung 2.1: Datenpfade der RETI-Architektur.

Sonstige Definitionen

Im Folgenden sind einige Definitionen aufgelistet, die zur **Erklärung** der Vorgehensweise zur Implementierung eines **üblichen Compilers** referenziert werden, aber **nichts** mit dem Vorgehen zur Implementierung des **PicoC-Compilers** zu tun haben.

Definition 2.1: 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 2.2: 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 2.3: Label

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

^aThiemann, „Compilerbau“.

Definition 2.4: 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 2.5) ist in üblichen Compilern in einer bestimmten Form meist schon integriert, da Compiler üblicherweise direkt **Maschinencode** bzw. **Objectcode** (Definition 2.6) 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 2.7) zu Maschinencode zusammengesetzt wird ausführbar ist.

Definition 2.5: Assembler

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

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

Definition 2.6: 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 2.7: 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 2.8: 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 2.9: 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 2.11) vorliegen zu haben. Bei diesen Ansätzen, die meist die Methode des **Rekursiven Abstiegs** (Definition 2.9) 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 2.10) ist².

Definition 2.10: 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**.

^bParsers — Lark documentation.

Definition 2.11: 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 2.12: 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 2.13: Liveness Analyse



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

^aG. Siek, *Essentials of Compilation*.

Definition 2.14: Live Variable



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

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

^bG. Siek, *Essentials of Compilation*.

Definition 2.15: Graph Coloring



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

^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 2.16: 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 2.14) 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 2.17: Kontrollflussgraph



Gerichteter Graph, der den **Kontrollfluss** (Definition 2.17) eines Programmes beschreibt.^a

^aG. Siek, *Essentials of Compilation*.

Definition 2.18: Kontrollfluss

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

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

Definition 2.19: Kontrollflussanalyse

Analyse des **Kontrollflusses** (Definition 2.18) eines **Programmes**, um herauszufinden zwischen welchen Teilen des Programms **Daten ausgetauscht** werden und welche **Abhängigkeiten** sich daraus ergeben.

Der **simpleste 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 2.20: 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** $C_{RETI_PicoC}^{PicoC}$ (Definition 2.21) zu schreiben. Dadurch kann die **Unabhängigkeit** von der Programmiersprache L_{Python} , in der der momentane Compiler C_{PicoC} für L_{PicoC} implementiert ist und die Unabhängigkeit von einer **anderen Maschine**, die bisher immer für das Cross-Compiling 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 ausgeschriebener **Compiler** als $C_{i_w_k_min}^{o-j}$ geschrieben, wobei C_w die **Sprache** bezeichnet, die der Compiler als **Input** nimmt und zu einer nicht näher spezifizierten Maschinensprache L_{B_i} einer Maschine M_i kompiliert. Falls die Notwendigkeit besteht, die **Maschine** M_i anzugeben, zu dessen **Maschinensprache** L_{B_i} der Compiler kompiliert, wird das als C_i geschrieben. Falls die Notwendigkeit besteht die **Sprache** L_o anzugeben, in der der Compiler selbst geschrieben ist, wird das als C^o geschrieben. Falls die Notwendigkeit besteht die Version der Sprache, in die der Compiler kompiliert (L_{w_k}) oder in der er selbst geschrieben ist (L_{o_j}) anzugeben, wird das als $C_{w_k}^{o-j}$ geschrieben. Falls es sich um einen **minimalen Compiler** handelt (Definition 2.22) kann man das als C_{min} schreiben.

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

Definition 2.21: Self-compiling Compiler

Compiler C_w^w , der in der Sprache L_w *geschrieben* ist, die er *selbst* kompiliert. Also ein Compiler, der sich *selbst* kompilieren kann.^a

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

Will man nun für eine Maschine M_{RETI} , auf der bisher keine anderen Programmiersprachen mittels **Bootstrapping** (Definition 2.24) zum laufen gebracht wurden, den gerade beschriebenen **Self-compiling Compiler** $C_{RETI_PicoC}^{PicoC}$ implementieren und hat bereits den gesamten **Self-compiling Compiler** $C_{RETI_PicoC}^{PicoC}$ in der Sprache L_{PicoC} geschrieben, so stösst man auf ein Problem, dass auf das **Henne-Ei-Problem**⁴ reduziert werden kann. Man bräuchte, um den **Self-compiling Compiler** $C_{RETI_PicoC}^{PicoC}$ auf der **Maschine** M_{RETI} zu kompilieren bereits einen kompilierten **Self-compiling Compiler** $C_{RETI_PicoC}^{PicoC}$, der mit der Maschinensprache B_{RETI} läuft. Es liegt eine **zirkulare Abhängigkeit** vor, die man nur auflösen kann, indem eine **externe Entität** zur Hilfe nimmt.

Da man den gesamten **Self-compiling Compiler** $C_{RETI_PicoC}^{PicoC}$ nicht selbst komplett in der Maschinensprache B_{RETI} schreiben will, wäre eine Möglichkeit, dass man den **Cross-Compiler** C_{PicoC}^{Python} , den man bereits in der Programmiersprache L_{Python} implementiert hat, der in diesem Fall einen **Bootstrapping Compiler** (Definition 2.23) darstellt, auf einer anderen Maschine M_{other} dafür nutzt, damit dieser den **Self-compiling Compiler** $C_{RETI_PicoC}^{PicoC}$ für die Maschine M_{RETI} kompiliert bzw. **bootstrapped** und man den kompilierten **RETI-Maschiendencode** dann einfach von der Maschine M_{other} auf die Maschine M_{RETI} kopiert.⁵

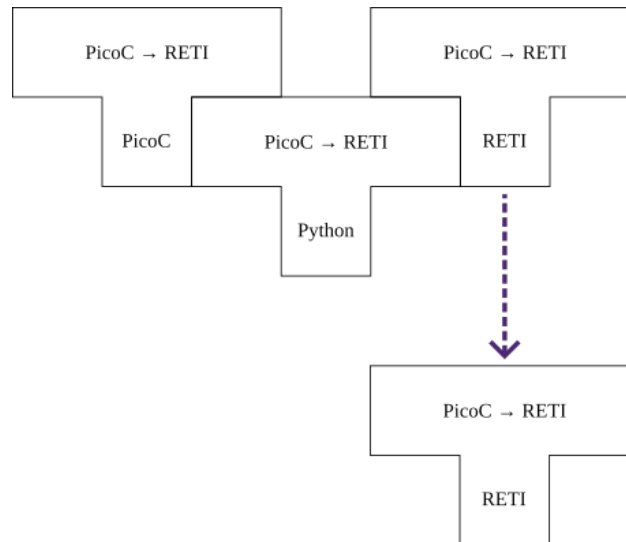


Abbildung 2.2: Cross-Compiler als Bootstrap Compiler.

Anmerkung

Einen ersten **minimalen Compiler** $C_{2_w_min}$ 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

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** $C_{RETI_PicoC}^{PicoC}$ auch mithilfe des **Cross-Compilers** C_{PicoC}^{Python} als **externe Entität** und der Programmiersprache L_{Python} auf der Maschine M_{RETI} selbst kompiliert werden.

in der **Maschinensprache** B_2 bzw. wenn ein **Assembler** vorhanden ist, in der **Assemblesprache** 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 dieser minimale Compiler dann eine umfangreichere Version von sich kompiliert usw.

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

Definition 2.22: 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 2.23: Bootstrap Compiler

Compiler C_w^o , der es ermöglicht einen **Self-compiling Compiler** C_w^w zu **bootstrappen**, indem der Self-compiling Compiler C_w^w mit dem **Bootstrap Compiler** C_w^o **kompiliert** wird^a. Der Bootstrapping Compiler stellt die **externe Entität** dar, die es ermöglicht die **zirkuläre Abhängigkeit**, dass initial ein **Self-compiling Compiler** C_w^w bereits kompiliert vorliegen müsste, um sich selbst kompilieren zu können, zu brechen.^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 2.22) 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 **Bootstrapping** 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 2.24.1 den minimalen Compiler schrittweise zu einem immer vollständigeren **C-Compiler** C_C weiterentwickelt.

Definition 2.24: 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 2.24.1 nochmal gesondert hervorzuheben:

2.24.1: 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}

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} .

^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 **Lügend Geschichten 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“.

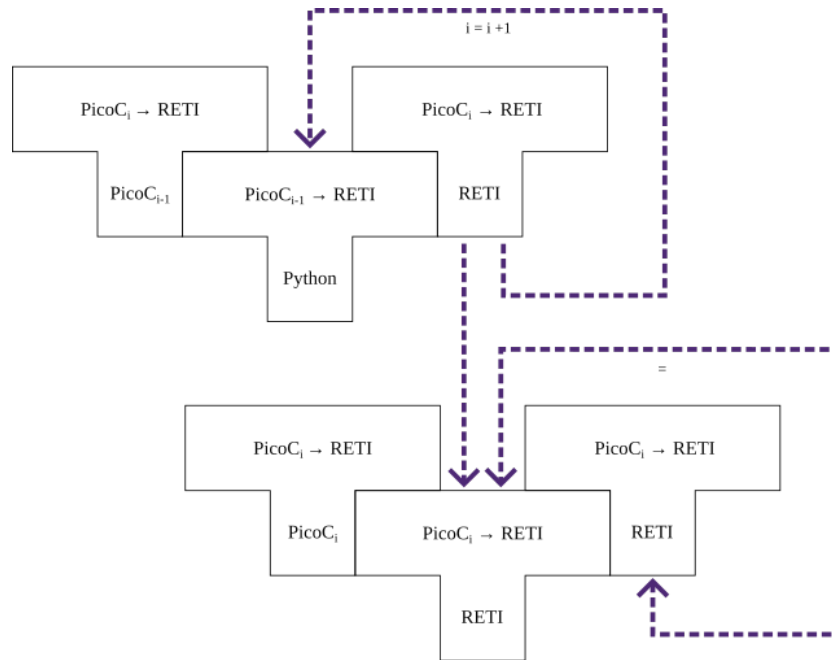


Abbildung 2.3: Iteratives Bootstrapping.

Anmerkung

Auch wenn ein **Self-compiling Compiler** $C_{w_i}^{w_i}$ in der Subdefinition 2.24.1 selbst in einer früheren Version $L_{w_{i-1}}$ der Programmiersprache L_{w_i} geschrieben wird, wird dieser nicht mit $C_{w_{i-1}}^{w_{i-1}}$ bezeichnet, sondern mit $C_{w_i}^{w_i}$, da es bei **Self-compiling Compilern** darum geht, dass diese zwar in der Subdefinition 2.24.1 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).
- *GCC, the GNU Compiler Collection - GNU Project*. URL: <https://gcc.gnu.org/> (besucht am 13.07.2022).
- *History - GCC Wiki*. URL: <https://gcc.gnu.org/wiki/History> (besucht am 06.08.2022).
- *Home - Neovim*. URL: <http://neovim.io/> (besucht am 04.08.2022).
- *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](https://doi.org/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).