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

Betreung:
M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für
Betriebssysteme

ERKLÄRUNG

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

Danksagungen

Bevor der Inhalt dieser Schrifftlichen 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 Selbverständlichkeit 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 heraus, 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 Leidenschafft heraus tun, auch wenn es für sie keine Vorteile hat. Tobias[1] 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 habe[2], er hat sich bei der Korrektur dieser Schrifftlichen 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 Selbverständlichkeit, 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 dagegeben wohl mit der Zeit äußerst kritisch geworden. Das Prof. Dr. Scholl mir zu

[1] Wie auch Prof. Dr. Scholl. Hier geht es aber erstmal um Tobias.
[2] Wofür 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 Selbverständlichkeit, 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 ausgeinandergesetzt 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 perönliches Interessengebiet fällt[6]. Das Aufschreiben dieser Schrifftlichen Ausarbeitung hat mir dagegen eher weniger Spaß gemacht[7]. 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 Programmiesprache kann in der Verwendung ziemlich wilkürlich 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.

[3] Der PicoC-Compiler hätte schließlich mehr Funktionalitäten haben können.
[4] Vielleicht 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.
[6] Womit nicht alle Studenten so viel Glück haben.
[7] Dieses ständige überlegen, wo man möglicherweise eine Erklärlücke hat, ob man nicht was wichtiges ausgelassen hat usw.

Inhaltsverzeichnis

Abbildungsverzeichnis

Codeverzeichnis

Tabellenverzeichnis

Definitionsverzeichnis

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 **Schrifftlichen 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[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

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

Basisname der Datei `program` an eine neue **Dateiendung** `.reti` angehängt wird[2].

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 **Dokumenation** unter Link[3] nachgelesen werden. Die die letzte Spalte gibt den **Standardwert** an, der bei der normalen Nutzung des **PicoC-Compilers** gesetzt ist. In grau ist unter `most_used` des Weiteren der **Standardwert** beim **Shell-Mode** angegeben.

Kommandozeilen-option	Beschreibung	Standard-wert
`-i`, `--intermediate_stages`	Gibt **Zwischenstufen** der Kompilierung in Form der verschiedenen **Tokens**, **Ableitungsbäume**, **Abstrakten Syntaxbäume** der verschiedenen **Passes** in Dateien mit **entsprechenden Dateiendungen** aber gleichem **Basisnamen** aus. Im **Shell-Mode** erfolgt **keine** Ausgabe in Dateien, sondern nur im **Terminal**.	`false`, most_used: `true`
`-p`, `--print`	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)
`-v`, `--verbose`	Fügt den verschiedenen **Zwischenschritten der Kompilierung**, unter anderem auch dem finalen RETI-Code **Kommentare** hinzu, welche eine **Anweisung** oder einen **Befehl** aus einem **vorherigen Pass** beinhalten, der durch die darunterliegenden Anweisungen oder Befehle **ersetzt** wurde. Wenn die `--run`-Option aktivert ist, wird der **Zustand** der virtuellen RETI-CPU **vor** und **nach jedem Befehl** angezeigt.	`false`
`-vv`, `--double_verbose`	Hat **dieselben Effekte**, wie die –**verbose**-Option, aber bewirkt zusätzlich **weitere Effekte**. **PicoC-Knoten** erhalten bei der Ausgabe in den Abstrakten Syntaxbäumen zustätzliche **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 `--intermediate_stages`-Option werden in den dadurch ausgegebenen **Abstrakten Syntaxbäumen** ebenfalls versteckte Attribute, die **Informationen** zu **Datentypen** und für **Fehlermeldungen** beinhalten angezeigt.	`false`
`-h`, `--help`	Zeigt die **Dokumentation**, welche ebenfalls unter Link gefunden werden kann **im Terminal** an. Mit der `--color`-Option kann die **Dokumentation** mit **farblicher Hervorhebung** im Terminal angezeigt werden.	`false`

Tabelle 1.1: *Kommandozeilenoptionen, Teil 1.*

[2] Beim **GCC** wird bei **Nicht-Angabe** eines **Dateinamen** mit der `-o` Option dagegen eine Datei mit der **festen** Bezeichnung `a.out` erstellt.
[3] https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/help-page.txt.

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 Schrifftlichen 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 **Zustsand 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 zusätlich 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 dabei den **PicoC-Compiler** und alle **großgeschriebenen** 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. Die angegebene **Folge von Anweisungen** `<seq-of-stmts>` wird dabei automatisch in eine `main`-Funktion eingefügt: `void main(){<seq-of-stmts>}`.

Mit dem `> compile <cli-options> <filename>`-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.

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> <filename>` (oder seine **Abkürzung** `mu`) genutzt werden, um diese Kommandozeilenoptionen mit dem `compile`-Befehl **nicht** jedes mal **selbst** Angeben zu müssen. In der Tabelle 1.1 sind in grau die Werte 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`- und die `--run`-Option beim `most_used`-Befehl aktiviert sind, werden die verschiedenen **Zwischenstufen** der Kompilierung, wie `Tokens`, `Ableitungsbaum` usw., sowie der **Zustand der RETI-CPU** nach der Ausführung des **letzten** Befehls angezeigt. 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 ← und → Pfeiltasten bewegt werden. In der **Befehlshistorie** kann sich mit den ↑ und ↓ Pfeiltasten **rückwarts** und **vorwärts** bewegt werden. Mit Tab 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 Link verwiesen, welche auch über den Befehl `help` 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-Code** zu finden, indem im Terminal eine **virtuelle RETI-CPU** angezeigt wird, welches den **kompletten**

Zustand einer virtuell ausgeführten RETI mit allen **Registern**, **SRAM**, **UART**, **EPROM** und einigen **weiteren Informationen** anzeigt.

Allerdings bringt die Möglichkeit des **Show-Mode**, die **RETI-Befehle** des übersetzten Programmes in **Ausführung zu sehen** 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**, die z.B. unter `<more-options>` noch 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	∅
`TESTNAME`	**Name des Tests**. Alles andere als der **Basisname**, wie die **Dateiendung** wird abgeschnitten	∅
`EXTENSION`	**Dateiendung**, die an `TESTNAME` angehängt werden soll zu `./tests/TESTNAME.EXTENSION`	`reti_states`
`NUM_WINDOWS`	**Anzahl Fenster** auf die ein Dateiinhalt **verteilt** werden soll	`5`
`VERBOSE`	Möglichkeit die **Kommandozeilenoption** `-v` oder `-vv` zu aktivieren für eine **ausführlichere Ausgabe**	∅
`DEBUG`	Möglichkeit die **Kommandozeilenoption** `-d` zu aktivieren, um bei `make test-show TESTNAME=<testname>` den **Debugger** für den entsprechenden **Test** `<testname>` zu starten	∅

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**[4] um einen **Dateiinhalt** 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 `<program>.reti_states` angewandt. Der **Show-Mode** kann aber auch dazu genutzt werden **andere Dateien**, welche verschiedene Zwischenschritte der Kompilierung darstellen anzuzeigen, indem `EXTENSION` auf eine andere **Dateiendung** gesetzt wird.

Im **Show-Mode** wird ein Trick angewandet, indem die verschiedenen **Zustände der RETI-CPU nicht** zur **Laufzeit** des **Show-Mode** berechnet werden, sondern schon 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 Zuständen** der **RETI-CPU vor** und **nach** der Ausführung eines Befehls **angezeigt** werden.

[4] *Home - Neovim.*

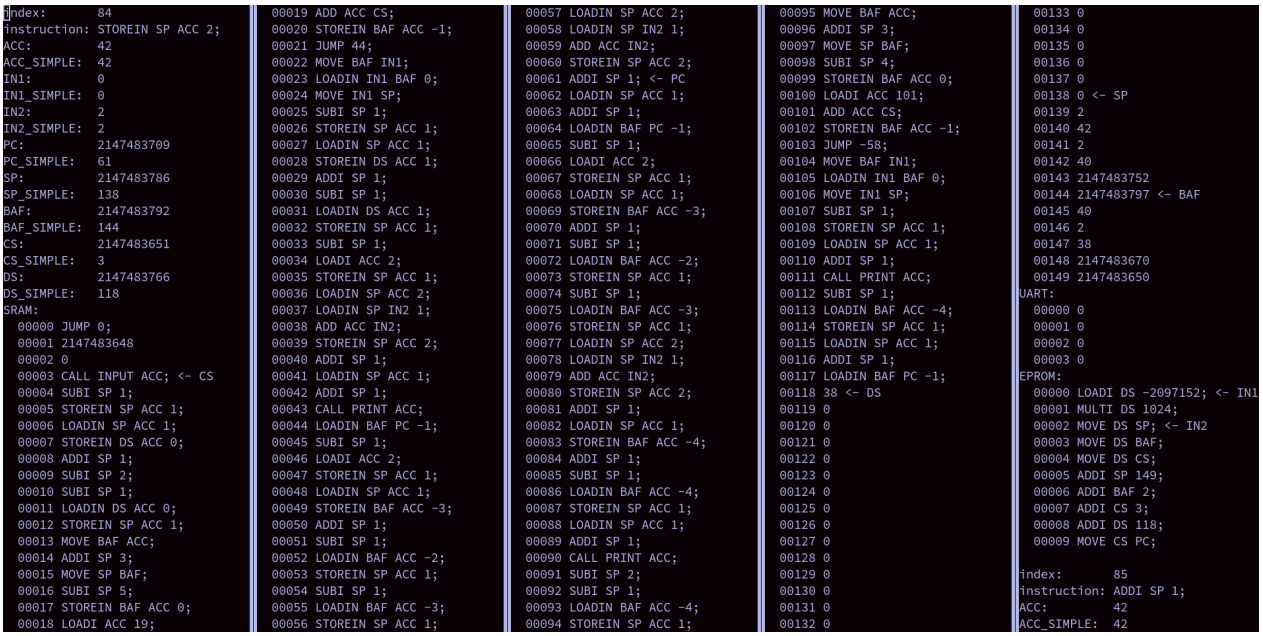


Abbildung 1.1: *Show-Mode in der Verwendung.*

Zur **besseren Orientierung** wird für alle Register ebenfalls ein mit der Registerbezeichnung beschriffteter **Zeiger** `<- REG` an Adressen im **EPROM**, **UART** und **SRAM** angezeigt, je nachdem, ob der **Wert im 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** dieser wieder allerdings auf die **Dokumentation** unter Link 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 **??** in Unterkapitel **??** 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 Syntax**.

Die **Tests** für den **PicoC-Compiler** sind hierbei im Verzeichnis `/tests` bzw. unter Link[5] zu finden. **Eingeteilt** sind die Tests in die folgenden **Kategorien** in Tabelle 1.4.

[5] https://github.com/matthejue/PicoC-Compiler/tree/new_architecture/tests.

Testkategorie	Beschreibung
`basic`	**Einfache** Tests, welche die **grundlegenden Funktionalitäten** des Compilers testen.
`advanced`	Tests, die **Spezialfälle** und **Kombinationen** verschiedener Funktionalitäten des Compilers testen.
`hard`	Tests, die **längere, komplexe Programme** testen, für welche die Funktionaliäten des Compilers in **perfekter Harmonie** miteinander funktionieren müssen.
`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 Verfikation** ausgeführt.
`exclude`	Tests, für welche aufgrund vielfältiger Gründe **keine Verifikation** ausgeführt werden soll.
`thesis`	Tests, die eigentlich vorher **Codebeispiele** für diese Schrifftliche Ausarbeitung der Bachelorarbeit waren.
`tobias`	Tests, die der Betreuer dieser Bachelorarbeit, **Tobias** geschrieben hat.

Tabelle 1.4: *Testkategorien.*

Dass die Programme in beiden Sprachen die **gleiche Semantik** haben, lässt sich mit einer **hohen Wahrscheinlichkeit** gewährleisten, wenn beide die **gleiche Ausgabe** haben und 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 drückt aus, dass jeder Test im `/tests`-Verzeichnis eine `// expected:<space_seperated_output>`-Zeile hat, in welcher der **Schreiber des Tests** die Rolle des entsprechenden **Interpreters**[6] aus Diagramm **??** übernimmt und die **erwartete Ausgabe** seiner eigenen Interpretation des **PicoC-Codes** anstelle von `<space_seperated_output>` hineinschreibt.

Ein Beispiel für einen **Test** ist in Code 1.3 zu sehen. Sobald die Tests mithilfe des Bashcripts `/run_tests.sh` ausgeführt werden oder dieses mithilfe der `/Makefile` mit dem Befehl `> make test` ausgeführt wird, wird als erstes für **jeden** Test das Bashscript `/extract_input_and_expected.sh` ausgeführt, welches die Zeilen `// in:<space_seperated_input>`, `// expected:<space_seperated_output>` und `// datasegment:<datasegment_size>` extrahiert[7] und die entsprechenden Werte in **neu** erstellte Dateien `<program>.in`, `<program>.out_expected` und `<program>.datasegment_size` schreibt. Das **letztere Skript** kann ebenfalls mit dem Befehl `> make extract` ausgeführt werden.

Die Datei `<program>.in` enthält **Eingaben**, welche durch `input()`-Funktionsaufrufe eingelesen werden, die Datei `<program>.out_expected` enthält zu **erwartende Ausgaben** der `print(<exp>)`-Funktionaufrufe, die später eingeführte Datei `<program>.out` enthält die **tatsächlichen Ausgaben** der `print(<exp>)`-Funktionsaufrufe bei der **Ausführung des Tests** und die Datei `<program>.datasegment_size` enthält die **Größe des Datensegments** für die Ausführung des entsprechenden Tests.

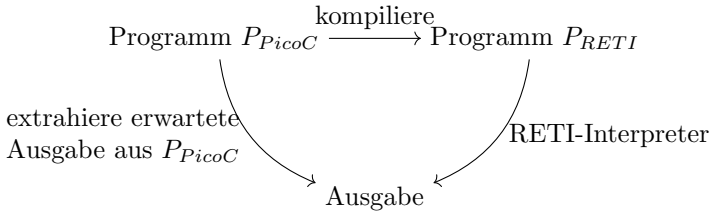
[6] Der die **Semantik** des Tests umsetzt.
[7] Falls vorhanden.

```
// 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 erfüllt, 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 `<program>.out` ausgegeben wird. Ein Test kann mit einer bestimmten Wahrscheinlichkeit die **Korrektheit** des **Teils der Semantik** der Sprache L_{PicoC}, die er abdeckt **verifizieren**, wenn der Inhalt von `<program>.out_expected` und `<program>.out` **identisch** ist.

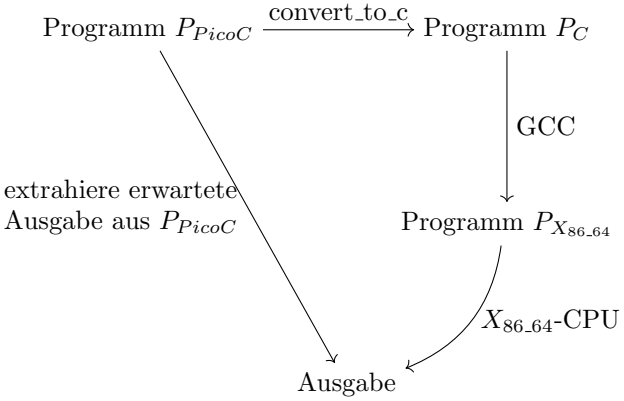


(1.2.1)

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 **Compiler implementiert**. Wenn der **Schreiber** der Tests ein **falsches Verständnis** davon hat, wie das Ergebnis eines Ausdrucks berechnet wird, so wird dieser sowohl im **Test** als auch in seiner **Implementierung** etwas als Ergebnis erwarten bzw. etwas implementieren, was nicht der eigentlichen **Semantik** von L_{PicoC} entspricht[8].

Aus diesem Grund muss hier eine **weitere Maßnahme**, welche in Diagramm 1.2.2 dargestellt ist eingeführt werden, die gewährleistet, dass die **Ausgabe** in Diagramm 1.2.1 sich auf jeden Fall aus der **Semantik** der Sprache L_{PicoC}[9] ergibt. Das wird erreicht, indem wie in Diagramm 1.2.2 dargestellt ist, überprüft wird, ob die **Ausgabe** des Pfades von P_{PicoC} zur Ausgabe mit der **Ausgabe** des Pfades von P_C über $P_{X_{86\_64}}$ **identisch** ist.

[8] Welche ja identisch zu der von L_C sein sollte.
[9] Die eine **Untermenge** von L_C ist.



(1.2.2)

Das Programm P_C ergibt sich dabei aus dem Testprogramm P_{PicoC} durch **Ausführen** des Pythonscripts `/convert_to_c.py`, welches **später näher erläutert** wird. Mithilfe der `/Makefile` und dem Befehl `> make convert` lässt sich dieses Pythonscript auf **alle** Tests anwenden.

Der **Trick** liegt hierbei in der Verwendung des **GCC** für die Kante von P_C zu $P_{X_{86\_64}}$. Beim **GCC** handelt es sich um einen Compiler der Sprache L_C, der somit auch 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[10] [11] 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 kann nun **sichergestellt** werden, dass die Tests **nicht** nur die Interpretation, die der Schreiber der Tests und Implementierer des **PicoC-Compilers** von der Semantik der Sprache L_{PicoC} hat **bestätigen**, sondern die tätsächliche **Einhaltung der Semantik** der Sprache L_{PicoC} testen.

Dazu durchläuft jeder Test, wie in Diagramm 1.2.2 dargestellt ist eine **Verifikation**, in der **verifiziert** wird, 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. Erst dann ist ein Test **verifiziert**, d.h. man kann, wenn der Test **vernünftig definiert** ist mit **hoher Wahrscheinlichkeit** sagen[12], dass wenn dieser Test für den **PicoC-Compiler** durchläuft, der **Teil der Semantik** der Sprache L_{PicoC}, den dieser Test testet vom PicoC-Compiler **korrekt umgesetzt** ist.

Für diese **Verifikation** ist das Bashscript `/verify_tests.sh` verantwortlich, welches mithilfe der `/Makefile` mit dem Befehl `> make verify` ausgeführt wird. Beim Befehl `> make test` wird dieses Bashscript **vor** dem eigentlichen Testen[13] durchgeführt. In Code 1.4 ist ein Testdurchlauf mit `> make test` zu sehen. Wobei `Verified: 50/50` anzeigt, wieviele der Tests **verifizierbar** sind[14], also beim **GCC** ohne Fehlermeldung durchlaufen, `Not verified:` die **nicht verifizierbaren** Tests angibt, `Running through: 88 / 88` anzeigt wieviele Tests mit dem **PicoC-Compiler** durchlaufen, `Not running through:` die **nicht** durchlaufenden Tests angibt, `Passed: 88 / 88` zeigt bei wievielen Tests die Ausgabe mit der erwarteten Ausgabe **identisch** ist, `Not passed:` die Tests anzeigt, bei denen das **nicht** der Fall ist.

[10] *History - GCC Wiki*.
[11] In der langen **Bestehenszeit** und bei der **vielen Verwendung** wurden die **allermeisten kritischen Bugs** wahrscheinlich schon gefunden.
[12] Es besteht allerdings immer eine **Chance**, dass die Ausgabe für den Test nur **zufällig** übereinstimmt. Diese Chance kann allerdings durch **vernünftige Definition** des Tests sehr **gering** gehalten werden.
[13] Prüfen, ob der interpretierte RETI-Code des PicoC-Compilers die **gleiche Ausgabe** hat, wie der Schreiber des Tests **erwartet**.
[14] Also **alle** Tests aus den **Kategorien** `basic`, `advanced`, `hard` und `example`.

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

Die Funktion `print(<exp>)` wird vom Pythonscript `convert_to_c.py` zu `printf("%d", <exp>)` übersetzt. Zuvor muss über `#include<stdio.h>` die **Standard-Input-Output Bibltiothek** `<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** den Wert hin, den die `input()`-Funktionen normalerweise einlesen sollten.

1.3 Erweiterungsideen

Mit dem **Funktionsumfang** des **PicoC-Compilers**, der in Unterkapitel 1.2 erläutert wurde muss allerdings das Ende der Fahnenstange noch **nicht** erreicht sein. Weitere Ideen, die im **PicoC-Compiler**[15] implementiert werden könnten, wären:

- **Register Allokation:** Variablen werden nicht nur **Adressen** im **Hauptspeicher** zugewiesen, sondern an erster Stelle **Registern** und erst wenn alle Register **voll** sind werden Variablen an Adressen auf dem **Hauptspeicher** gespeichert. Da hat den Grund, dass der **Zugriff auf Register** deutlich **schneller** ist, als der **Zugriff auf den Hauptspeicher**. Um die Variablen möglichst optimal **Locations**

[15] Möglicherweise ja im Rahmen eines **Masterprojektes** ☺.

(Definition **??**) zuzuweisen wird mithilfe einer **Liveness Analyse** (Defintion 2.13) ein **Interferenzgraph** (Definition 2.16) aufgebaut. Auf den **Interferenzgraph** wird ein **Graph Coloring** Algorithmus (Definition 2.15) angewandt, der den **Locations** Zahlen zuordnet. Die **ersten** Zahlen entsprechen **Registern**, aber ab einem bestimmten Zahlenwert, wenn alle Register zugeordnet sind, entsprechen die Zahlen **Adressen auf dem Hauptspeicher**. Des Weiteren muss die **Liveness Analyse** nach Ansätzen der **Kontrollflussnalayse** (Definition 2.19) **iterativ** unter Verwendung eines **Kontrollflussgraphen** (Definition 2.17) auf die verschiedenen **Blöcke** angewendet werden, bis sich an den Live Variablen **nichts** mehr **ändert**.[16]

- **Tail Call:** Wenn ein Funktionsaufruf die **letzte** Anweisung in einem Funktionsblock ist, wird der Stackframe dieser aufrufenden Funktion **nicht** mehr **gebraucht**, da **nicht** mehr in diese Funktion zurückgekehrt werden muss[17]. 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 nur eine **konstante Menge** an **Speicherplatz** auf dem Stack verbraucht. In Code 1.5 sind **zwei Tail Calls** markiert.

- **Partielle Evaluation:** Bei Ausdrücken wie `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 Abstrakten Syntaxbaum in eine neue Abstrakte Syntax der Sprache L_{Picoc_Eval} umformt. In der Abstrakten Syntax der Sprache L_{PicoC_Eval} sind **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** produziert wird und weniger **RETI-Code** bedeutet wiederum eine **schnellere Programmausführung**.

- **Lazy Evaluation:** Bei Ausdrücken wie `var1 && 42 / 0` oder `var2 || 42 / 0`, wobei `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 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**[18] 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** werden zusammen mit **Attributen** in **mehrere Dateien** aufgeteilt, welche **seperat** programmiert und kompiliert werden können. Für die **Deklaration** von **Funktionen** und **Attributen** werden `.h`-**Headerdateien** verwendet, für die Definition sind `.c`-**Quellcodedateien** da. Hierbei ist der **Basisname** einer `.h`-**Headerdatei identisch** zur entsprechenden `.c`-**Quellcodedatei** mit den entsprechenden Definitionen. Dateien werden über `#include "file"` eingebunden, was einem **direkten einfügen** des entsprechenden Codes der eingebundenen Datei entspricht. Über einen **Linker** (Definition 2.7) können die **kompilierten** `.o`-**Objektdateien** (Definition 2.6) zusammengefügt werden, wobei der **Linker** darauf achtet **keinen doppelten Code** zuzulassen.

- `malloc` **und** `free`**:** Es wird eine **Bibltiothek** mit den Funktionen `malloc` und `free`, wie in der Bibltiothek

[16] Die 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 und vom Vorgehen beim **PicoC-Compiler** abgegrenzt werden kann.

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

[18] Es gibt hierfür leider keinen **deutschen Begriff**, der geläufig ist.

`stdlib`[19] implementiert, deren `.h`-Headerdatei mittels `#include "malloc_and_free.h"` eingebunden werden muss. 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 diesen **zurückgegeben**. Dieser **Speicherplatz** kann von der `free`-Funktion wieder **freigegeben** werden. Um zu wissen, wo und wieviel Speicherplatz 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, der sonst unmöglich in der Zukunft mehr genutzt werden würde freigegeben wird. 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 defininiert 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 $L_{PicocPython}$ **wiederverwendet** werden und viele **Passes** müssten nur erweitert werden.

- **Call by Reference:** Über das wiederverwenden des `&`-Symbols für **Parameter** bei **Funktiondeklaration** 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** unabängig 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** für die **RETI-CPU** gemacht, der auf der RETI-CPU selbst läuft.

[19] Auch engl. **General Purpose Standard Library** genannt.

```
// in:42
// expected:0

int ret0() {
  return 0;
}

int ret1() {
  return 1;
}

int tail_call_fun(int bool_val) {
  if (bool_val) {
    return ret0();
  }
  return ret1();
}

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

Code 1.5: *Beispiel für Tail Call.*

Anmerkung

Partielle Evaluation und **Lazy Evaluation** wurden im PicoC-Compiler **nicht impelementiert**, da dieser als **Lerntool** gedacht ist und dieses 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 ausgelaggert 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** ausgelaggert, 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 **Bootsrapping** 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$_\geq$i	Falls $[ACC] \geq 0$: $\langle PC \rangle := \langle PC \rangle + [i]$, 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$_\neq$i	Falls $[ACC] \neq 0$: $\langle PC \rangle := \langle PC \rangle + [i]$, sonst: $\langle PC \rangle := \langle PC \rangle + 1$
11	110	00	JUMP$_\leq$i	Falls $[ACC] \leq 0$: $\langle PC \rangle := \langle PC \rangle + [i]$, 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 2.3: *Jump Befehle.*

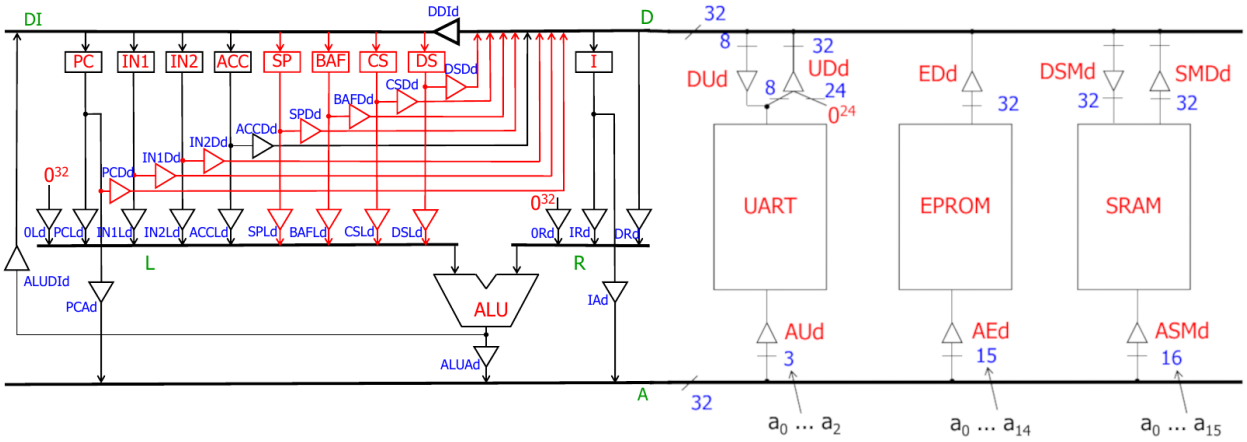


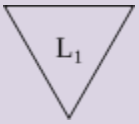
Abbildung 2.1: *Datenpfade der RETI-Architektur.*

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 tuen haben.

Definition 2.1: T-Diagram Maschine

*Repräsentiert eine **Maschine**, welche ein **Programm** in **Maschinensprache** L_1 ausführt.*[a][b]



[a] Wenn 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).
[b] J. 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]

[a] Bzw. **Tokenwert**.
[b] Außer wenn z.B. bei Funktionen die Programmiersprache das **Überladen** erlaubt usw. In diesem Fall wird die **Signatur** der Funktion als weiteres Unterschiedungsmerkmal hinzugenommen, damit es eindeutig ist.
[c] Thiemann, „Einführung in die Programmierung“.

Definition 2.3: Label

*Durch einen **Bezeichner eindeutig** zuordenbares **Sprungziel** im Programmcode.*[a]

[a] Thiemann, „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]

[a] Befehle der Assemblersprache, die mehreren Maschinenbefehlen entsprechen werden auch als **Pseudo-Befehle** bezeichnet und entsprechen dem, was man im allgemeinen als Macro bezeichnet.
[b] Z.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.
[c] Z.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.
[d] P. 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 Maschienencode zusammengesetzt wird ausführbar ist.

Definition 2.5: Assembler

*Übersetzt im allgemeinen **Assemblercode**, der in **Assemblersprache** geschrieben ist zu **Maschinencode** bzw. **Objectcode** in **binärerer Repräsentation**, der in **Maschinensprache** geschrieben ist.*[a]

[a] P. 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]

[a] P. 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]

[a] P. 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*[a][b]

[a] Die Programmiersprache **TypeScript** will als **Obermenge** von **JavaScript** die Sprachhe Javascript **erweitern** und gleichzeitig die **syntaktischen Mittel** von JavaScript unterstützen. Daher bietet es sich Typescript zu Javascript zu **transpilieren**.
[b] Thiemann, „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)-Eigenschafft** ausgeschlossen werden kann, dass **Backtracking** notwendig ist[1].

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) ist[2].

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]

[a] Also eine beliebige Folge von **Terminalsymbolen** und **Nicht-Terminalsymbolen**.
[b] *Parsers — 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 **l**eft-to-right und **l**eftmost-derivation, da das **Eingabewort** von **links nach rechts** geparst und immer **Linksableitungen** genommen werden müssen*[b]*, damit die obige Bedingung mit den **nächsten** k Symbolen gilt.*[c]

[a] Das wird auch als **Lookahead** von k bezeichnet.
[b] Wobei 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.
[c] Nebel, „Theoretische Informatik“.

[1] Mehr **Erklärung** hierzu findet sich im Unterkapitel **??**.
[2] Für 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** $G_{Parse} = \langle N, \Sigma, P, S \rangle$.*
- ***Ausgabe:** 0 wenn $w \notin L(G_{Parse})$[d] und 1 wenn $w \in L(G_{Parse})$.*

*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**:*
   - ***Voraussage:** 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* $\Rightarrow w \notin L(G_{Parse})$.

[a] Jay Earley, „An efiicient 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.
[e] Also eine Folge von **Terminalsymbolen** und **Nicht-Terminalsymbolen**.

Definition 2.13: Liveness Analyse

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

[a] G. Siek, *Essentials of Compilation*.

Definition 2.14: Live Variable

Eine Location, deren momentaner Wert ***später*** *im Programmablauf noch* ***verwendet*** *wird. Man sagt auch die Location ist* ***live.***[a b]

[a] Es gibt leider **kein** allgemein verwendetes **deutsches** Wort für **Live Variable**.
[b] G. 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* ***adjazente Knoten*** *die* ***gleiche Zahl*** *haben und* ***möglichst wenige*** *unterschiedliche Zahlen gebraucht werden.*[c d]

[a] In Bezug zu Compilerbau ein **Ungerichteter Graph**.
[b] Bzw. **Farben**.
[c] Es gibt leider **kein** allgemein verwendetes **deutsches** Wort für **Graph Coloring**.
[d] G. Siek, *Essentials of Compilation*.

Definition 2.16: Interference Graph

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

[a] Bzw. **Farbe**.
[b] G. Siek, *Essentials of Compilation*.

Definition 2.17: Kontrollflussgraph

Gerichteter Graph*, der den Kontrollfluss eines Programmes beschreibt.*[a]

[a] G. Siek, *Essentials of Compilation*.

Definition 2.18: Kontrollfluss

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

[a] Man geht hier von einem **imperativen** Programm aus.

Definition 2.19: Kontrollflussanalyse

***Analyse** des **Kontrollflusses** (Defintion 2.18) 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]

[a] Im Bezug zu Compilerbau die **Linveness Analayse**.
[b] Bis diese sich **stabilisiert** haben
[c] G. Siek, *Essentials of Compilation*.

Definition 2.20: Two-Space Copying Collector

*Ein **Garbabe 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]

[a] G. 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** $C^{PicoC}_{RETI\_PicoC}$ (Defintion 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 **Bootrapping** 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 $C^{o\_j}_{i\_w\_k\_min}$ 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 $C^{o\_j}_{w\_k}$ geschrieben. Falls es sich um einen **minimalen Compiler** handelt (Definition 2.22) kann man das als C_{min} schreiben.

[3] Ein **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]

[a] J. 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** $C^{PicoC}_{RETI\_PicoC}$ implementieren und hat bereits den gesamtem **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ 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** $C^{PicoC}_{RETI\_PicoC}$ auf der **Maschine** M_{RETI} zu kompilieren bereits einen kompilierten **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$, 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** $C^{PicoC}_{RETI\_PicoC}$ nicht selbst komplett in der Maschinensprache B_{RETI} schreiben will, wäre eine Möglichkeit, dass man den **Cross-Compiler** C^{Python}_{PicoC}, 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** $C^{PicoC}_{RETI\_PicoC}$ für die Maschine M_{RETI} kompiliert bzw. **bootstraped** und man den kompilierten **RETI-Maschiendencode** dann einfach von der Maschine M_{other} auf die Maschine M_{RETI} kopiert.[5]

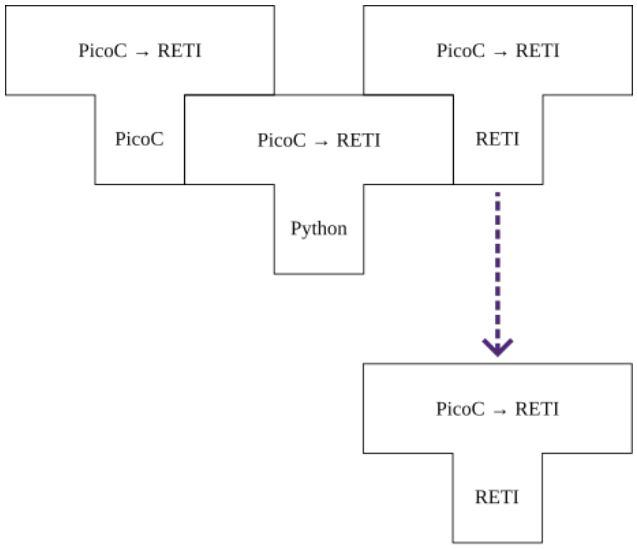


Abbildung 2.2: *Cross-Compiler als Bootstrap Compiler.*

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

[4] Beschreibt 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.

[5] Im 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 mithife des **Cross-Compilers** C^{Python}_{PicoC} 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 allersten 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.

[a] In diesem Fall, dem **Cross-Compiler** C^{Python}_{PicoC}.

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.*[ab]

[a] Den **PicoC-Compiler** könnte man auch als einen **minimalen Compiler** ansehen.
[b] Thiemann, „Compilerbau“.

Definition 2.23: Boostrap Compiler

*Compiler C^o_w, der es ermöglicht einen **Self-compiling Compiler** C^w_w zu **boostrapen**, indem der Self-compiling Compiler C^w_w mit dem **Bootstrap Compiler** C^o_w **kompiliert** wird*[a]*. Der Bootstrapping Compiler stellt die **externe Entität** dar, die es ermöglicht die **zirkulare Abhängigkeit**, dass initial ein **Self-compiling Compiler** C^w_w bereits kompiliert vorliegen müsste, um sich selbst kompilieren zu können, zu brechen.*[b]

[a] Dabei kann es sich um einen **lokal** auf der Maschine selbst laufenden Compiler oder auch um einen **Cross-Compiler** handeln.
[b] Thiemann, „Compilerbau“.

Aufbauend auf dem **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$, 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** $C^{PicoC}_{RETI\_PicoC}$, 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*[abcd]*. 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.*[efg]

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

[a] Z.B. mithilfe eines **Bootstrap Compilers**.
[b] Der 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.
[c] Hat 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 weiterentwicklern, 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.
[d] Einen 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.
[e] Es 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.
[f] Der 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.
[g] J. Earley und Sturgis, „A formalism for translator interactions“.

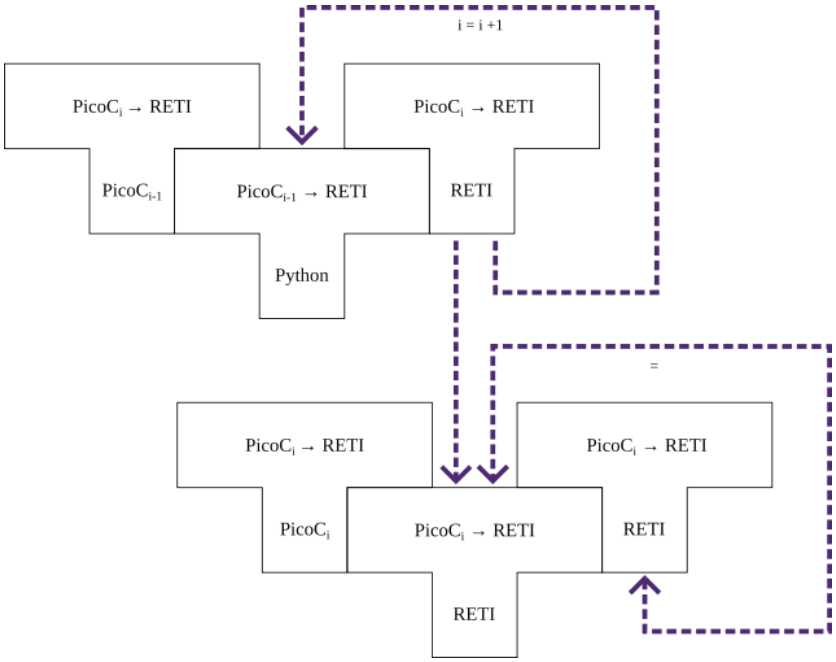


Abbildung 2.3: *Iteratives Bootstrapping.*

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 $C_{w_i}^{w_{i-1}}$ 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).
- *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.
- Earley, Jay. „An efiicient 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).

- Thiemann, Peter. „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).