Entwicklung eines Parsers zur Normalisierung und Aufarbeitung heterogener Build-Protokolle

Projektarbeit IIb (T3_2000)

Im Rahmen der Prüfung:
Bachelor of Science (B. Sc.)

des Studienganges Informatik
an der Dualen Hochschule Baden-Württemberg Karlsruhe

von
Lukas Bailey

Abgabedatum	15. September 2025
Bearbeitungszeitraum	30.06.2025 - 15.09.2025
Matrikelnummer, Kurs	8232296, TINF23B2
Ausbildungsfirma	SAP SE Dietmar-Hopp-Allee 16 69190 Walldorf, Deutschland
Betreuer der Ausbildungsfirma	Philipp Degler
Gutachter der Dualen Hochschule	Prof. Dr. Sebastian Ritterbusch

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Projektarbeit IIb (T3_2000) mit dem Thema:

> *Entwicklung eines Parsers zur Normalisierung und Aufarbeitung heterogener Build-Protokolle*

gemäß § 5 der „Studien- und Prüfungsordnung DHBW Technik“ vom 29. September 2017 selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Karlsruhe, 25. Februar 2026

gez.: Bailey, Lukas

Sperrvermerk

Die nachfolgende Arbeit enthält vertrauliche Daten der:

SAP SE
Dietmar-Hopp-Allee 16
69190 Walldorf
Deutschland

Der Inhalt dieser Arbeit darf weder als Ganzes noch in Auszügen Personen außerhalb des Prüfungsprozesses und des Evaluationsverfahrens zugänglich gemacht werden, sofern keine anderslautende Genehmigung vom Dualen Partner vorliegt.

Abstract

- Deutsch -

In dieser Arbeit wird ein Parser zur Normalisierung der Unterschiede in von unterschiedlichen Werkzeugen generierten Build-Protokollen vorgestellt. Dieser Parser, `log-jitsu` genannt, liest wichtige Diagnostikausgaben von unterschiedlichen Build-Prozessen unabhängig davon ein, welche Build-Systeme und Compiler verwendet wurden, und verarbeitet sie in ein Format, das wesentliche Informationen kontextuell darstellt. Um diese Anforderungen zu erfüllen, implementiert `log-jitsu` in Rust einen mehrstufigen Prozess: Zunächst wird das Protokoll eingelesen und identifiziert, welches Build-System für den Bauprozess verantwortlich war. Daraufhin werden die Eigenheiten dieses Systems normalisiert; es entstehen Zeilen, die eindeutig einem Projekt und dem Werkzeug, das sie geschrieben hat, zugeordnet werden können. Jede dieser Zeilen wird in einem Parse-Prozess verarbeitet, der unabhängig davon, wer sie produziert hat, Diagnostiken erkennen kann und alle ihre Komponenten in einem Datenformat zurückgibt, das ihren semantischen Zusammenhang wahrt. Dieses Datenformat ist ideal für die Verarbeitung und Darstellung in einer HTML-Ausgabe, die zusätzlich gezeigt wird. Es wird festgestellt, dass Rust und besonders der Parsergenerator Nom ideale Werkzeuge für diese Aufgabe sind und die Implementierung von `log-jitsu` eine gute Lösung für die Problemstellung leistet. Diese Arbeit erläutert alle nötigen Grundlagen, diskutiert die genaue Funktionsweise von `log-jitsu` und analysiert die Lösung auf einen möglichen Produktiveinsatz.

Inhaltsverzeichnis

Quellcodeverzeichnis

1 Einleitung

1.1 Motivation

In der Welt der Software-Entwicklung existiert eine Vielzahl von Möglichkeiten, Änderungen an Programmcode auf Probleme zu untersuchen und Anwendungen nach Änderungen zu aktualisieren. Viele dieser Möglichkeiten generieren ein Protokoll, das dokumentiert, welche Aktionen durchgeführt und welche Probleme dabei gefunden wurden. Bei der automatischen Aufbereitung der Informationen in diesem Protokoll eröffnet sich folgende Problemstellung: Unterschiedliche Protokolle besitzen unterschiedliche Syntaxen. Eine automatische Verarbeitung dieser setzt also eine Normalisierung dieser Unterschiede voraus, um in einer einzelnen Anwendung verschiedene Protokolle lesen zu können. Bei der Entwicklung in SAP-Systemen stellt sich das gleiche Problem. Der ABAP-Kernel, der als Schnittstelle zwischen Anwendungen und Betriebssystem das Herz des SAP-Systems bildet, wird regelmäßig weiterentwickelt. Die dabei entstehenden Änderungen können von unterschiedlicher Software verarbeitet werden. Eine automatische Analyse der dabei entstehenden Protokolle ist ohne eine explizite Anwendung zur Normalisierung ihrer Unterschiede nicht universell möglich. Die Entwicklung besagter Anwendung bildet den Kern dieser Arbeit. `log-jitsu` ist ein Protokoll-Parser, der die Ausgaben ausgewählter Werkzeuge unabhängig von der genauen Zusammensetzung des Prozesses verarbeiten und aufbereiten soll. Im Vordergrund steht dabei die Erkennung von Semantiken und die kontextuelle Darstellung von Informationen.

Im Folgenden wird eine Zielsetzung konkretisiert, die eine Ausrichtung für diese Arbeit festlegt.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist die Präsentation einer Softwarelösung namens `log-jitsu`, die das zuvor genannte Problem für den ABAP-Kernel lösen soll. Dazu wird zunächst ein Überblick über notwendige Grundkonzepte geboten und eine Analyse der Aufmachung zu verarbeitender Protokolle durchgeführt. Auf der Basis dessen bietet diese Arbeit daraufhin eine Implementierung, die alle für eine grundlegende Lösung des Problems benötigten Funktionen umsetzt. Welche diese sind, wird im Laufe der Arbeit konkret dargestellt. Diese Softwarelösung soll eine eigene Anwendung bilden, die die Normalisie-

rung der Ausgaben bekannter Werkzeuge durchführt und leicht für eine Unterstützung neuer Technologien und Werkzeuge erweitert werden kann. Durch die Abkapselung der Verarbeitungslogik soll ermöglicht werden, dass bei Änderungen des Protokolls durch Nutzung anderer Werkzeuge oder neuer Versionen keine Anpassung der Ausgabe sondern nur eine Aktualisierung von `log-jitsu` vonnöten ist. Diese Eigenschaften sorgen für gute und schnelle Wartbarkeit des Prozesses.

1.3 Methodik

In dieser Arbeit wird eine Systemanalyse durchgeführt, dabei wird die Problemstellung begründet, der Ist-Zustand analysiert, ein Soll-Konzept definiert, eine Umsetzung des Konzepts erläutert und bewertet. Der Ist-Zustand beinhaltet Grundlagen in Kapitel 2 und eine Analyse des Status quo in Kapitel 3. Auf Basis dieser Grundsituation werden in Kapitel 4 Anforderungen an die Lösung gestellt und ein Lösungsvorschlag gegeben, bevor im selben Kapitel eine detaillierte Ausarbeitung der Lösung geschieht. Bei der Definition der Anforderungen an die Softwarelösung orientiert sich diese Arbeit an den Problemstellungen des Ist-Zustands und Richtlinien der ISO 25010 [1], einer internationalen Industrienorm für Sotwarequalität. Die Umsetzung der Softwarelösung folgt in Form einer Prototypisierung, die im Folgenden erläutert wird.

1.4 Prototypisierung

Bei der Prototypisierung von Software werden meist unvollständige Versionen von Softwarelösungen angefertigt, die zum Beispiel zur Demonstration der Wirksamkeit eines Lösungsansatzes dienen. In diesem Fall muss eine Metrik bestimmt werden, die entscheiden soll, ob und wie diese Wirksamkeit erfüllt ist [2, Kap. 1, 9, 10]. Oft werden hierbei Metriken wie Zeiteinsparung, verringerter Rechenaufwand oder eine einfacher zu bedienende Benutzeroberfläche gewählt. Wie genau diese Metriken aussehen und bewertet werden ist von der Problemstellung abhängig. [2, Kap. 9, 10] Prototypen umfassen oft nur einen Teil der zu entwickelnden Softwarelösung, sowohl architektonisch als auch konzeptuell betrachtet. So kann ein Prototyp zum Beispiel eine Applikation aus Backend und Anwendung ohne aufgearbeitetes Frontend, zugehörigen Webserver oder Einbindung in das Datenbanksystem, für das er entwickelt wurde, sein [2, Kap. 3]. Eine häufig verwendete Form der Prototypisierung ist evolutionäre Prototypisierung. Hierbei steht der Prototyp im Herz der später vollständigen Softwarelösung. Er wird

ergänzt und verfeinert und anders als bei anderen Arten der Prototypisierung nicht verworfen sondern weiter genutzt [2, Kap. 3]. Große Vorteile der Prototypisierung sind z.B. die Möglichkeit einer frühen Bewertung des Lösungsansatzes, eine laufende Qualitätssicherung und eine konstante Anpassung von Modell und Features an die (sich möglicherweise ändernden) Anforderungen. [2, Kap. 3]

2 Grundlagen

Bevor ein Lösungskonzept für einen Prototypen vorgestellt wird, müssen zunächst Grundlagen diskutiert werden. Diese umfassen Technologien, die der Entwicklung von `log-jitsu` zugrunde liegen und Konzepte, die für die spätere Analyse und Bewertung relevant sind.

Besonders wichtig für die Implementierung des Protoypen ist ein Grundverständnis der Funktionsweise von Build-Systemen sowie der Entwicklung von Parsern, spezifisch für das Parsen von Build-Protokollen.

2.1 ABAP-Kernel

Der ABAP-Kernel[1] ist das Herzstück eines SAP-Systems. Er bildet eine Schnittstelle zwischen Anwendungen und Betriebssystem und Datenbank im Hintergrund. Essenziell ist er dafür verantwortlich, Programme und Anfragen von Nutzern des Systems zu bearbeiten und auszuführen. Dabei koordiniert er die Kommunikation zwischen verschiedenen Systemkomponenten. [3]

Hauptsächlich besteht er aus plattformabhängigen ausführbaren Dateien im Betriebssystem, die regelmäßig aktualisiert und geändert werden [3]. Der Kernel muss, wie viele andere Programme, aus seiner Form als Programmcode über viele Dateien hinweg nutzbar gemacht werden. Dieser Prozess nennt sich build. Er wird in der Regel von einem Build-System orchestriert und produziert ein Protokoll. Dieses Protokoll enthält unter anderem Umgebungsvariablen des Bauprozesses, Code-Diagnostiken und Ergebnisse integrierter Tests. Der Build-Prozess muss im Regelfall bei jeder neuen Version des Kernels durchlaufen werden, die Diagnostiken sind also relevant für jede neue Änderung am Code. [4]

Somit ist der Kernel Build-Prozess eine essentieller Bestandteil des Kernel Entwicklungsprozesses. Diagnostiken für Aktualisierungen und Änderungen seines Codes spielen in Form von Build-Protokollen also eine zentrale Rolle für Fehlererkennung und Wartung in einem SAP-System. Ein Grundverständnis der Build-Systeme, die die Protokolle generieren, ist vonnöten für das Verständnis dieser Arbeit.

[1] Wird oft und im Kontext dieser Arbeit nur Kernel genannt.

2.2 Build-Systeme

Build-Systeme koordinieren den Build-Prozess einer Anwendung. Sie identifizieren beim Starten des Build-Prozesses, welche Dateien neu kompiliert werden müssen und in welcher Reihenfolge diese voneinander abhängen. Diese Abhängigkeiten sind in sogenannten Makefiles gespeichert, die definieren, welche Dateien für das Bauen einer anderen Datei, eines Ziels vonnöten sind. [5, Kap. 1-2]

Es gibt unzählige Build-Systeme, die den Bauprozess einer Anwendung durchführen können. Für den Bau des Kernels werden je nach Konfiguration `sapmake`[2] und optional Ninja verwendet, weshalb diese beiden für den Inhalt dieser Arbeit relevant sind.

Ihre Funktionsweise, Syntax und ihre Unterschiede werden in späteren Kapiteln dieser Arbeit behandelt.

Die Ausgaben der Build-Systeme während des Bauens müssen gelesen und in ein intern nutzbares Format verarbeitet werden, aus dem log-jitsu eine HTML-Datei generieren kann. Diese Aufgabe übernimmt ein sogenannter Parser. Er übersetzt eine Folge von Zeichen, Zeilen oder Ähnlichem in ein anderes Format, oft eine hierarchische Struktur.

2.3 Parser

Der Parser, der im Rahmen dieser Arbeit entwickelt wird, ist kein Parser als klassischer Teil eines Compile-Prozesses. Es finden keine klaren Compile-Phasen wie Syntaktische und Lexikalische Analyse statt, in die der Prozess geteilt werden kann. Vielmehr handelt es sich im Großteil um das Herauslesen von Inhalt aus unterschiedlich strukturieren Protokollausgaben. So findet der Hauptteil der Normalisierung durch die Arbeit des Parsers statt. Es sollen zwei verschiedene Parserfunktionen aus zwei unterschiedlich formatierten Zeilen die gleiche relevante Information herauslesen.
Code 1 zeigt zwei mögliche Zeilen einer Protokollausgabe. Beide beinhalten die gleiche Information: das Build-System teilt mit, dass es ab hier die Datei „projektarbeit.typ“ kompiliert.

```
1 > Compiling projektarbeit.typ
2 LOG Compiling projektarbeit.typ
```

Code 1: Zwei unterschiedlich formatierte Zeilen mit dem gleichen Inhalt

[2] eine proprietäre Erweiterung von GNU make

Um die Unterschiede im Format der beiden Zeilen - Zeile 1 beginnt mit `>` und Zeile 2 beginnt mit `LOG` - zu normalisieren, muss ein Parser in der Lage sein, diese Präfixe zu ignorieren oder je nach Kontext als Zusatzinformationen zu verarbeiten. Konkrete Informationen über zu normalisierende Unterschiede und damit die Konzeptionierung des Parsers befinden sich in Kapitel 3.

Für die Implementierung der Anwendung mit Parsern für die verschiedenen Build-Protokolle wird in dieser Arbeit die Programmiersprache Rust verwendet. Die dafür entscheidenden Eigenschaften und Vorteile von Rust werden im folgenden Unterkapitel behandelt.

2.4 Rust

Rust ist eine Allzweck-Programmiersprache, die sich durch Typsicherheit, automatische Speicherverwaltung und hohe Nebenläufigkeit auszeichnet. Sie wird von der Open-Source-Community entwickelt und bereitgestellt und zielt darauf ab, Probleme und Sicherheitslücken anderer systemnaher Sprachen wie C zu lösen. [6]

Die Rust-Syntax ist der von C ähnlich aber unterscheidet sich in vielen Details.

2.4.1 Bezeichner und Variablen

In folgendem Codebeispiel Code 2 werden zwei unterschiedliche Arten demonstriert, Bezeichner in Rust zu deklarieren.

```
let bezeichner: Typ = Wert;
let mut variable: Typ = Anfangswert;
```

Code 2: Deklaration von Bezeichnern

Codebeispiel Code 2 demonstriert den Unterschied zwischen normal deklarierten und veränderbaren[3] Bezeichnern. Der Wert von `variable` kann im Laufe des Programms verändert und überschrieben werden, für den Wert von `bezeichner` ist das nicht möglich [6, Kap. 3.1]. Die Angabe eines Typs kann überflüssig sein, wenn Rust den gewünschten Datentypen inferieren kann [6, Kap. 3.2].

[3] engl. mutable, daher das Schlüsselwort mut

2.4.2 Heap und Stack

Ein weiterer wesentlicher Unterschied zu C, der bei der Deklaration von Bezeichnern auftritt, ist die Behandlung von Werten im Speicher. Wie bei C können Werte im Speicher auf den Stack oder Heap gelegt werden. Der Stack speichert Werte in der Reihenfolge, in der er sie erhält und hält nur Werte fester Länge. Werte variabler Länge, wie zum Beispiel eine Zeichenkette, werden auf dem Heap gespeichert. Dafür sucht ein Speicherallokator im Speicherbereich des Heaps nach einem freien Speicherbereich, der im Regelfall der maximal möglichen Länge des Werts (festgelegt durch den Datentypen) im Speicher entspricht, und legt ihn dort ab. Daraufhin gibt er einen Zeiger zurück, der die Adresse des Werts auf dem Heap kennt. Nach dieser Allokation kann der Zeiger auf dem Stack abgelegt werden, da er eine feste Länge hat. Daten auf dem Stack zu speichern ist ein schnellerer Prozess als Speicher auf dem Heap zu allokieren. Analog ist das Abrufen von Daten aus dem Heap ebenso ein langsamerer Prozess als das Auslesen von Werten vom Stack, da der Prozessor zunächst dem Zeiger folgen muss um den Wert zu finden. [6, Kap. 4.1]

Wenn ein Wert einem Bezeichner zugeteilt wird, wird dieser der Besitzer des Werts. Jeder Wert hat genau einen Besitzer und wird automatisch verworfen, wenn der Nutzungsbereich des Besitzers endet. Damit sind Konzepte wie Garbage Collection oder Dekonstruktoren in Rust redundant. Das Beispiel Code 3 illustriert den Nutzunsbereich eines Bezeichners.

```
{ // rust_mascot ist noch nicht deklariert.
  let rust_mascot = "Ferris";
  // Hier ist rust_mascot verwendbar und Besitzer der Zeichenkette.
} // Der Block ist vorbei und der Nutzungsbereich von rust_mascot endet.
  // Damit wird die Zeichenkette "Ferris" aus dem Speicher verworfen.
```

Code 3: Nutzungsbereich eines Bezeichners

Bei der Deklaration des Bezeichners in Code 3 wird für die Zeichenkette `"Ferris"` Speicher auf dem Heap allokiert und der Bezeichner `rust_mascot` als alleiniger Besitzer festgelegt. Wenn der Block, in dem `rust_mascot` gilt, endet, wird der Speicher, in dem `"Ferris"` abgelegt wurde, freigebeben.

Codebeispiele Code 4 und Code 5 demonstrieren, wie sich Werte auf dem Stack und Heap unterschiedlich verhalten.

```rust
let a = 1;
let b = a;
```

Code 4: Wert eines Bezeichners kopieren

Dem Bezeichner `b` in Code 4 wird eine Kopie des Wertes von `a` zugewiesen. Der Wert `1` hat eine feste Länge und wird deshalb auf den Stack gelegt. Die Zuweisung des Werts von `a` zum Bezeichner `b` funktioniert, wie man es in C erwarten würde.

```rust
let z1 = String::from("Ferris");
let z2 = z1;
```

Code 5: Wert eines Bezeichners bewegen

Die Zuweisung des Bezeichners `z2` in Code 5 verhält sich anders, weil `"Ferris"` eine Zeichenkette ist und damit keine festgesetzte Länge besitzt. Bei der Zuweisung eines schon auf dem Heap existierenden Werts zu einem neuen Bezeichner, kopiert Rust nicht den Wert, sondern nur die Referenz (und den Zeiger) darauf. Dies ist insofern sinnvoll, als dass das Kopieren langer Zeichenketten oder Werte ähnlicher Datentypen sehr speicherintensiv sein und die Performanz des Codes stark beeinträchtigen kann. Um zu vermeiden, dass mehrere Variablen am Ende ihres Nutzungsbereichs versuchen, den gleichen Speicher freizugeben - was zur Korruption des Speichers und Sicherheitslücken führen kann, wechselt der Wert auf dem Heap seinen Besitzer und wird von Variable `z1` zu Variable `z2` „bewegt“. `z2` ist nun der Besitzer der Zeichenkette und `z1` ist nicht mehr valide, kann also auch nicht mehr abgerufen werden. [6, Kap. 4.1]

Zusätzlich zum Konzept des Besitzens in Rust ist für diese Arbeit noch das Konzept der Match-Ausdrücke relevant, da sie in der Implementierung der Parser häufig zum Einsatz kommen und ihre Funktionsweise Implikationen für das Design der Komponenten hat.

2.4.3 Match-Ausdrücke

Match-Ausdrücke in Rust sind ein mächtiges Werkzeug zur Erkennung von Mustern. Sie ersetzen die in C gängigen Switch-Case-Anweisungen, die je nach Wert eines Ausdrucks (des sogenannten Prüflings des Match-Ausdrucks) unterschiedlichen Code ausführen.

Sie unterscheiden sich von ihrem Pendant in vier Aspekten:

Ausdruck vs. Anweisung

Match-Ausdrücke in Rust sind Ausdrücke, keine Anweisungen. Das bedeutet, sie geben immer einen Wert zurück, weshalb sie überall verwendet werden können. Beispiel Code 6 demonstriert die Zuweisung eines Rust Match-Ausdrucks zu einer Variablen.

```
binaer_ziffer_als_text =
  match binaer_ziffer {
    0 => "null"
    1 => "eins"
    _ => "keine Binärziffer"
  }
```

Code 6: Eine Match-Anweisung, die Binärziffern einem Text zuordnet

Der Quellcode in Code 6 weist der Variable `binaer_ziffer_als_text` eine Zeichenkette zu, die davon abhängt, welchen Wert `binaer_ziffer` hat. [6, Kap. 6.2]

Mustererkennung

Anders als bei C können Match-Ausdrücke in Rust für alle gängigen Datentypen verwendet werden. Darüber hinaus unterstützen Match-Ausdrücke zusätzliche mächtige Funktionen. [6, Kap. 6.2]

Rust kann Werte des Prüflings ohne zusätzliche Logik in Bereiche unterteilen, wie in Beispiel Code 7.

```
match note {
  1..=4 => println!("bestanden"),
  5..=6 => println!("nicht bestanden"),
  _ => println!("keine Schulnote"),
}
```

Code 7: Eine Match-Anweisung, die entscheidet, ob Noten als bestanden gelten

Liegt der Wert des Bezeichners `note` zwischen 1 und 4, so gibt der Match-Ausdrück in Codebeispiel Code 7 `"bestanden"` aus, liegt er zwischen 5 und 6, so gibt der Ausdruck `"nicht bestanden"` aus. [7]

Weiterhin kann Rust kompliziertere Datenstrukturen destrukturieren und z.B. einzelne Felder evaluieren. Codebeispiel Code 8 illustriert dies an einer Punkt-Variable mit zwei kartesischen Koordinaten.

```
let punkt = (3, 0);

match punkt {
  (0, y) => println!("auf der Y-Achse mit y = {y}"),
  (x, 0) => println!("auf der X-Achse mit x = {x}"),
  (x, y) => println!("bei ({x}, {y})"),
}
```

Code 8: Eine Match-Anweisung, die die Lage eines Punkts im kartesischen Koordinatensystem beschreibt

Der Beispielcode Code 8 destrukturiert die Datenstruktur des Punkts und gibt `"auf der X-Achse mit x = 3"` aus. [6, Kap. 5-6], [8]
Analog ist Rust in der Lage, Enum-Varianten zu destrukturieren [6, Kap. 6.2]. Ähnliche Funktionalität ist in C deutlich aufwändiger.

Zusätzlich können Match-Anweisungen für jeden Arm direkt zusätzliche Boolean-Bedingungen abfragen und Variablenzuweisungen durchführen, so wie in Code 8 der X-Wert des Punkts der Variable `x` für die Ausgabe zugewiesen wird. [9], [10]

Isolierung der Pfade

Die verschiedenen Arme des Ausdrucks sind in Rust voneinander isoliert, d.h. es wird immer genau einer der Arme durchlaufen. Folgende Arme, deren Bedingung auch erfüllt ist, werden übersprungen. Dieses Verhalten muss in C mit einer `break`-Anweisung spezifiziert werden, sodass die Switch-Case-Anweisung aus einem Arm direkt aus der Anweisung springt. Sonst durchläuft C alle Arme mit erfüllten Bedingungen, was zu Bugs oder unerwünschtem Verhalten führen kann. [6, Kap. 6.2]

Sicherheit

Dank der Isolierungs-Eigenschaft von Match-Ausdrücken sowie der Einschränkung, dass Match-Ausdrücke in Rust immer alle möglichen Werte des Prüflings behandeln müssen, ist eine Implementierung in Rust deutlich robuster gegenüber Bugs und unerwünschtem Verhalten. [6, Kap. 6.2]

Eine weitere Funktionalität in Rust, die hier nicht als Unterschied zu C beleuchtet wird, sondern im Detail behandelt wird, weil sie in der Implementierung von hoher Wichtigkeit ist, ist das Nutzen von Iteratoren in Rust.

2.4.4 Iteratoren

Der Trait Iterator[4] wird genutzt, um mit iterierbaren Datentypen umzugehen. Typischerweise sind dies Sammlungen von Objekten eines Datentyps wie z.B. eine Kollektion von Zeilen einer Protokollausgabe. [11, „The three forms of iteration“]

Bei jedem Durchlauf produziert der Iterator Dieser Typ ist das sogenannte Item des Iterators, in Beispiel Code 9 eine Zeile des Protokolls. Eine Implementierung eines Iterators definiert, wie über die Elemente eines Typs iteriert werden soll um Elemente eines anderen zu produzieren, indem eine `next`-Methode implementiert wird. [11, „Implementing Iterator“]

```
impl Iterator for Protokoll {
    type Item = Zeile;

    fn next(&mut self) -> Option<Self::Item> {
//    Hier wird `next` implementiert.
    }
}
```

Code 9: Grundgerüst einer Iterator-Implementierung mit angedeuteter next-Methode

Die `next`-Methode beinhaltet die Anweisungen, die für jedes neue Element der Sammlung ausgeführt werden sollen. [11, „Implementing Iterator“] Bei einer Protokollausgabe könnten diese zum Beispiel das Entfernen eines Zeitstempels am Anfang jeder Zeile sein, woraufhin der Rest der Zeile zurückgegeben wird.

Rust verfügt also über mächtige Funktionalitäten, die für die sichere Implementierung eines Parsers verschiedener Protokollausgaben nützlich sind.

Um die Implementierung der Protokoll-Parser in Rust möglichst erweiterbar und lesbar zu machen, bedient sie sich eines Frameworks namens Nom, das eine Reihe an Grundfunktionalitäten ohne weiteren Entwicklungsaufwand bietet.

2.5 Nom

Nom ist ein Parsergenerator für die Implementierung von Parsern in Rust. Es bietet eine Sammlung an kleinen und einfachen Parserfunktionen in Form einer Rust-Biblio-

[4]Traits in Rust sind Interfaces aus anderen Sprachen sehr ähnlich.

thek [12, „introduction“]. Diese Parserfunktionen sollen kombiniert werden, um einen komplizierteren Parser zu erstellen [12, Kap. 3, 5 - 6].

Nom generiert die Parser in Form von Funktionen, die z.B. Input in Form von Text akzeptieren und Output in Form eines Ergebnis-Formats liefern, das angibt, ob das Parsen des Texts erfolgreich war, welcher Teil geparst wurde und welcher als Rest zurückbleibt [12, Kap. 1].

In folgendem Codebeispiel Code 10 implementiert nom einen simplen Parser, der am Anfang einer Zeichenkette das Zeichen `"F"` erkennen soll.

```
fn f_parsen(input: &str) -> IResult<&str, &str> {
  char('F')(input)
}
fn main() -> Result<(), Box<dyn Error>> {
  let (rest, geparst) = f_parsen("Ferris")?;
  assert_eq!(rest, "erris");
  assert_eq!(geparst, "F");
}
```

Code 10: Simpler Parser für das Zeichen 'F'

Codebeispiel Code 10 implementiert und verwendet eine Funktion `f_parsen`, die sich die Nom-Funktion `char` zunutze macht. Dieses generiert eine Parserfunktion, die den Anfang einer übergebenen Zeichenkette auf ein vorher spezifiziertes Zeichen prüft. Dieses Zeichen ist im Beispiel `"F"`. Die dadurch generierte Parserfunktion wird in `f_parsen` auf dem Input aufgerufen und liefert sein Ergebnis als Rückgabewert. In `main` wird `f_parsen` auf die Zeichenkette `"Ferris"` angewandt und parst `"F"` heraus, es bleibt `"erris"` als Rest zurück. [12, Kap. 2]

Die Kombination von mehreren kleinen Parsern zu einem komplizierteren passiert über weitere Funktion, die eine neue Parserfunktion aus den kleinen Parsern zusammen setzen. Codebeispiel Code 11 demonstriert die Nutzung zwei verschiedener Arten von Kombinatoren.

```
alt((char('F'), char('D')))("Derris")

many1(char('F'))("FFerris")
```

Code 11: Simple Beispiele für Parserkombinatoren

Wie in Codebeispiel Code 11 zu sehen, sind `alt` und `many1` Kombinatoren. `alt` ermöglicht, mehrere Optionen für das Parsen eines Inputs zu geben. Dafür erhält `alt` ein Tupel von Parsern und versucht, alle anzuwenden. Wenn einer von ihnen erfolgreich ist, gibt `alt` einen Erfolg zurück. Im Beispiel wird die übergebene Zeichenkette akzeptiert, weil sie mit einem `"F"` oder `"D"` beginnt. `many1` wendet den ihm übergebenen Parser so oft an, wie er erfolgreich ist. Er muss mindestens ein Mal die Eingabe akzeptieren, damit `many1` einen Erfolg liefert. Im Beispiel wird `many1` zwei mal `char('F')` anwenden.

Nom liefert eine Vielzahl an anderen Kombinatoren, die zum Beispiel mehrere Kombinatoren nacheinander anwenden oder sicherstellen, dass Beginn und Ende einer Zeichenkette zu einem bestimmten Parser passen. [12, Kap. 3, 5 - 6]

Im Anhang befindet sich Tabelle Tabelle 1, die alle für diese Arbeit genutzten Nom-Funktionen mit einer kurzen Funktionsbeschreibung enthält.

Mithilfe der diskutierten Grundlagen wird im nächsten Kapitel zunächst eine Analyse des Ist-Zustandes durchgeführt, um einen Überblick über die momentane Ausgabe der Protokolle und fehlende Funktionen zu gewinnen.

3 Analyse

Um die Anforderungen an log-jitsu klar definieren zu können, wird in diesem Kapitel die Problemstellung im Detail erläutert. Dazu gehört ein Überblick über die zu normalisierenden Unterschiede, über die Lesbarkeit der Protokollausgaben und die der bisherigen HTML-Ausgabe.

3.1 Grundlegender Aufbau der Protokollausgaben

Um in der Lage zu sein, inhaltlich wichtige Informationen aus einer Protokollausgabe zu lesen, müssen die grundlegende Struktur des Protokolls und die Art und Weise, wie die erwünschten Informationen ausgegeben werden, bekannt sein.[5] Das Protokoll dokumentiert, welche Aktionen von Build-System und verwendeten Compilern durchgeführt werden.

Anmerkung: die in dieser Arbeit untersuchte Implementierung von `log-jitsu` setzt voraus, dass Nutzende für den Aufruf des Build-Systems spezifizieren, dass das Build-Protokoll des Kernels auf der höchsten Nachverfolgungsstufe und mit synchronisierter Ausgabe generiert wird. Dadurch wird garantiert, dass alle für `log-jitsu` benötigten Teile des Protokolls vorhanden und in der richtigen Reihenfolge ausgegeben sind. `log-jitsu` testet bei jedem Protokoll, ob diese Optionen korrekt gesetzt sind.

Sie beginnen mit einer Sektion, in der im Protokoll diverse Vorarbeit geleistet wird. Es werden die vom Benutzer und System gesetzten Variablen für den Bauprozess eingelesen und sich auf ihn vorbereitet. Dabei werden Versionsnummern, Parameter und Umgebungsvariablen in zu Beispiel Code 12 ähnlichen Form ausgegeben.

[5] Syntaxen von Build-Systemen und Compilern in dieser Arbeit entstammen einer Analyse von Build-Protokollen.

```
 1 detected Release '917'
 2 detected SID 'CGK'
 3 Using default P5000 path:
 4     SAP_BUILD_P5000_DIR=/sapmnt/depot
 5 p5000 depot path overwritten with /sapmnt/depot
 6 using image <image-link>
 7 /SAPDevelop/Benutzer/git/sapkernel/sapmake/sapmk.pl -src /SAPDevelop/
Benutzer/git/sapkernel -sid CGK -makecmd /SAPDevelop/Benutzer/git/
sapkernel/sapmake/sapmake -trace 3 --output-sync all
 8 LOG sapmake call: /SAPDevelop/Benutzer/git/sapkernel/sapmake/sapmake
PEDANTIC_DEPENDENCIES=T TRACE_LEVEL=3 SOURCE_ROOT=/SAPDevelop/Benutzer/git/
sapkernel SID=CGK GIT_DESCRIPTION= --output-sync -I /SAPDevelop/Benutzer/
git/sapkernel -f /SAPDevelop/Benutzer/git/sapkernel/sapmake/sapmake.gmk -j
64 -l 64 all
 9 DBG variables from command line:
10 DBG VARIABLE_1 = WERT_1
11 DBG VARIABLE_2 = WERT_2
12 DBG VARIABLE_3 = WERT_3
13 LOG Running on platform linuxx86_64
14 DBG Using pedantic dependencies to /SAPDevelop/Benutzer/git/sapkernel/
sapmake
15 LOG ------------------------------------------------------------------
16 LOG Reading Makefiles
17 LOG ------------------------------------------------------------------
 |
 |
 |
XX LOG ------------------------------------------------------------------
XX LOG Executing Makefiles
XX LOG ------------------------------------------------------------------
 |
 |
```

Code 12: Schematisches Beispiel für den Anfang eines Protokolls

Diese Parameter können wie in Zeilen 10 bis 12 der Beispiel-Protokollausgabe Code 12 einfach aus dem Protokoll gelesen und geparst werden, sollte `log-jitsu` sie benötigen. Der Wert von `VARIABLE_1` wird sich immer in einer Zeile befinden, die mit `DBG VARIABLE_1 =` beginnt.

Ein Großteil der gezeigten Ausgaben wird beim Aufruf beider Build-Systeme generiert, obwohl sie `sapmake`-spezifisch sind. Das liegt daran, dass ein Werkzeug namens `sapmake2ninja`[6] existiert, das von Ninja aufgerufen wird, um neue Ninja-Makefiles zu generieren, wenn dies nötig ist. Dieses Werkzeug ruft unter anderem `sapmake` auf, um wichtige Informationen zum Bauprozess in einem strukturierten Format auszugeben. Diese sind die in Beispiel Code 12 gezeigten Ausgaben, die auch bei einem `sapmake`-Build entstehen.

Zeile 8 des Beispiels zeigt den vollständigen Kommandozeilenaufruf für das hier verwendete `sapmake`.

Der Bereich nach Zeile 17 dokumentiert das Einlesen der Makefiles für den Bau des Kernels. Nachdem diese gelesen wurden und alle Abhängigkeiten vollständig berechnet sind, fängt das System an, Ziele in einer von den Makefiles definierten Reihenfolge zu bauen. Der Bauprozess wird von einem `Executing Makefiles` eingeleitet.

Bauen eines Ziels

Bei dem Bauen eines Ziels müssen das Ziel und alle (noch nicht gebauten) Dateien, von denen das Ziel abhängt, kompiliert werden. Protokollbeispiel Code 13 illustriert schematisch, wie eine Ausgabe für den Bauprozess eines Ziels aussehen könnte.

```
1 DBG BEGIN TARGET=cts/R3trans/gen/datei.o PROJECT=cts/R3trans
2 LOG Compiling datei.c
Hier befinden sich Ausgaben des Compilers von datei.c
|
|
X DBG END TARGET=cts/R3trans/gen/datei.o
```

Code 13: Schematisches Beispiel für das Bauen eines Ziels

Der Beginn des Bauprozesses eines neuen Ziels wird vom Build-System markiert. In Beispiel Code 13 leitet `DBG BEGIN` die Ausgabe ein. Der Pfad des Zielobjekts und der Name des Projekts, zu dem das Ziel gehört, sind in der selben Zeile ausgegeben. Nachdem alle benötigten Dateien kompiliert wurden und das Ziel gebaut ist, beendet die Zeile mit `DBG END` den Abschnitt. In der Protokollausgabe befinden sich also Zeilen, die einem bestimmten Projekt oder Ziel zugeordnet werden können. Die Markierung dieser Zugehörigkeit ist je nach Build-System von Beispiel Code 13 verschieden.

[6] von engl. „sapmake to Ninja“, also dt. „sapmake zu Ninja“

Die Ausgaben des Compilers innerhalb des Abschnitts bestehen aus Diagnostiken und Ähnlichem. `log-jitsu` muss diese Informationen verarbeiten, weshalb sie im nächsten Abschnitt genauer beleuchtet werden.

Compiler-Ausgaben

Compiler produzieren während des Kompilierens unterschiedliche Arten von Protokollausgaben. Diese umfassen allgemeine Informationen und Statusnachrichten - z.B. welche Datei gerade kompiliert wird oder welche Optimierungen der Compiler durchführt - aber auch Diagnostiken: Fehlermeldungen, Warnungen und Hinweise.

Um eine Übersicht über Diagnostiken des Build-Prozesses zu generieren, muss `log-jitsu` z.B. Diagnostiken in ähnlicher Form zu Beispieldiagnostik Code 14 parsen können.

```
 1 /SAPDevelop/Benutzer/git/sapkernel/krn/ma/rit.c:1901:5: warning: passing
NULL to argument 2 of 'void createDiagnosticExampleSnippet(const
SAP_VT_STUDENT, WITTY_IDEA*)' [-Wconversion-null]
 2 1901 |     CREATE_DIAGNOSTIC_EXAMPLE_SNIPPET(author, NULL);
 3      |     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Code 14: Schematisches Beispiel für eine Compiler-Diagnostikausgabe

Die in Beispiel Code 14 zu sehende Diagnostik ist eine Warnung. Hier teilt der Compiler mit, dass die Funktion `createDiagnosticExampleSnippet` in Zeile 1901 der Datei rit.c keinen Wert für ihr zweites Argument erhält.

Die Warnung besteht aus mehreren Komponenten:

- Der Ort, an dem die Warnung beim Kompilieren erstellt wurde; dieser ist in Beispiel Code 14: Spalte 5, Zeile 1901 der Datei mit dem Pfad `/SAPDevelop/Benutzer/git/sapkernel/krn/ma/rit.c`.
- Die Schwere der Diagnostik, in diesem Fall `warning`, also Warnung.
- Eine Nachricht, die beschreibt, was die Warnung ausgelöst hat.
- Der Ausschnitt des Codes, auf den sich die Warnung bezieht, in diesem Fall der Aufruf benannter Funktion mit Zeilenangabe.

Allgemein produzieren die in dieser Arbeit behandelten Compiler immer Diagnostiken in folgender Form: Zunächst werden optional durchlaufene Traces[7] ausgegeben. Danach

[7]Traces sind Nachverfolgungen von Abläufen von Code über Dateien. Damit erkennt man, welche Schritte aufgerufen wurden, wie lange sie dauerten und wo Fehler oder Verzögerungen entstehen.

folgen die Diagnostik - z.B. eine Warnung oder ein Fehler, ein Ausschnitt des Codes, der die Diagnostik verursacht hat und optional zusätzliche Hinweise mit mehr Code als Kontext. Dieses Schema ist in Beispiel Code 15 dargestellt.

```
1 In file included from <Trace>,
2 dateiname.endung:zeile:spalte: <Diagnostik>
3      <Codeausschnitt>
4      ^~~~~~~~~~~~~~~~
5 dateiname.endung:zeile:spalte: <Hinweis>
6      <Codeausschnitt>
7      ^~~
```

Code 15: Volles Schema einer Diagnostik

Alle Komponenten dieser Ausgabe sind der Diagnostik und dem dazugehörigen Code zuzuordnen.

Nachdem in diesem Unterkapitel die grundlegenden zu parsenden Komponenten des Build-Protokolls erläutert wurden, wird in den nächsten zwei Unterkapiteln dargestellt, inwiefern sie je nach Build-System oder Compiler syntaktisch anders aufgebaut sein können und wie diese Unterschiede normalisiert werden können.

3.2 Unterschiedliche Syntaxen von Sapmake und Ninja

Die vom Build-System generierten Teile der Ausgabe hängen in ihrer Form davon ab, welches Build-System für den Bauprozess verwendet wurde. Dieses Unterkapitel diskutiert, inwiefern die Ausgaben von `sapmake` und Ninja unterschiedlich zu parsen sind. Die Unterschiede werden nicht im direkten Vergleich erläutert - die Syntax wird separat für jedes Build-System gezeigt, da für die beiden seperate Parser implementiert werden, die keinen direkten Vergleich nutzen. Gleichzeitig ist die Behandlung der Syntax auf das Nötigste beschränkt, um ausschießlich relevante Informationen zu diskutieren.

Sapmake

`sapmake` ist eine von SAP entwickelte proprietäre Erweiterung für das Open-Source Build-System GNU make. Es erweitert zum Zeitpunkt der Erstellung dieser Arbeit einer veralteten GNU make-Version. Der Großteil der für `sapmake` benötigten Logik ist

Hier wird der englische, im Programmierjargon übliche Begriff *Trace* verwendet, um Verwirrung zu vermeiden. Der männliche Genus ist abgeleitet von dt.: *Ablauf*.

in Makefiles organisiert. Benutzende geben deklarativ an, was gebaut werden soll und `sapmake` generiert die Ziele und Protokollausgabe.

Die im vorausgegangenen Unterkapitel genutzten Codebeispiele Code 12 und Code 13 wurden nach dem Vorbild einer `sapmake`-Ausgabe erstellt. Die Syntax von `sapmake` entspricht also der zuvor gezeigten: Der Compile-Prozess eines Ziels ist im Protokoll durch `DBG BEGIN TARGET` und `DBG END TARGET` begrenzt, Pfade des jeweiligen Ziels und Projekts können anhand der ausgegebenen Variablen erkannt werden.

Ninja

Während sich andere Build-Systeme darauf stützen, dass ihre Makefiles von Menschen geschrieben werden und für diese lesbar sein sollten, ist Ninja ein Build-System, was versucht mithilfe von Optimierungen so schnell wie möglich bauen zu können. Das führt dazu, dass Ninja-Makefiles im Regelfall nicht von Menschen geschrieben werden können, sondern mithilfe von Erweiterungen und oft einem anderen Build-System generiert werden. So müssen diese nur ein Mal zu Beginn und dann bei seltenen, die Makefiles betreffenden Änderungen generiert werden. Alle weiteren Build-Prozesse werden durch Ninja schneller ausgeführt. Die Ninja-Protokoll-Syntax weicht von der in vorausgegangenen Kapiteln dargestellten `sapmake`-Syntax ab.

Der Hauptunterschied, der bei der Zuordnung von Compile-Schritten zu einem Projekt oder Ziel relevant ist, ist, dass Zeilen, die ein neues Projekt einleiten, mit einer Zeichenkette im Format `[<Zahl>/<Zahl>] [<Projektname>]` beginnen. So wäre die Zeile `[6533/6539] [krn/abap/rnd/abp] Linking shared library _out/dw_abp.so` dem Projekt `krn/abap/rnd/abp` zuzuordnen.

Der Anfang des Protokolls, in Beispiel Code 12 zu sehen, ist, wie zuvor erwähnt, größtenteils Build-System-übergreifend. Damit ist er nicht `sapmake`-spezifisch und Bauprozess-Variablen können in einem Ninja-Protokoll genau so herausgelesen werden wie bei `sapmake`.

3.3 Unterschiedliche Compiler

Um im Build-Prozess die veränderten Dateien und alle die, die von ihnen abhängen, zu aktualisieren, müssen sie neu kompiliert werden. Ein Compiler verwertet für Menschen lesbaren Programmcode in ein von der Maschine oder einem anderen Programm

nutzbares und optimiertes Format. Je nach Dateiformat oder Einstellungen des Build-Systems setzt dieses unterschiedliche Compiler für unterschiedliche Dateien ein.

Im Rahmen dieser Arbeit werden nur C-Compiler thematisiert. In Zukunft wird eine Implementierung von `log-jitsu` wahrscheinlich Compiler für andere Sprachen im Bauprozess des Kernels unterstützen, aber zum Zeitpunkt der Erstellung dieser Arbeit werden dort nur C- und C++-Compiler genutzt. `log-jitsu` soll systemübergreifend funktionieren, deshalb muss es sowohl Windows C-Compiler (MSVC) als auch Linux/Unix/Mac Compiler (GCC, Clang) unterstützen. Die oben genannten Compiler produzieren je nach Version unterschiedliche Ausgaben im Compile-Prozess, insbesondere bei Diagnostiken. Die in Beispiel Code 14 gezeigte Warnung ist von GCC generiert, Warnungen von anderen Compilern können anders aussehen.

Diese zweite Normalisierungsstufe, in der `log-jitsu` die Unterschiede zwischen den verwendeten Compilern ausgleichen soll, wird in dieser Arbeit thematisiert aber nicht für jeden möglichen Compiler aufgebrochen. Die Compiler-spezifischen Parserfunktionen leisten keinen Beitrag zum Konzeptbeweis, der sich zwar den Funktionen bedient, aber sie als Black-Box-Funktionen im Kontext nutzen kann. Die Implementierung wird schematisch im Detail behandelt. So werden Funktionen wie `gcc::Parser::parse_diagnostic()` in Beispiel Code 28 genutzt und ihre Funktion erklärt, aber ihre Implementierung nicht gezeigt. Diese ist je nach Compiler unterschiedlich und für das Verständnis von `log-jitsu` nicht vonnöten.

3.4 Ausgabe

Das Ausführen von `sapmake` produziert zusätzlich zum Build-Protokoll je nach Eingabeparametern bereits ohne `log-jitsu` eine HTML-Übersicht der Ergebnisse des Build-Prozesses. Diese Ausgabe zielt darauf ab, die produzierten Ergebnisse in einer Form zu visualisieren, die eine stärkere Struktur bietet als ein Build-Protokoll im Klartext. Die von `sapmake` generierte HTML-Seite weist einige Probleme auf:

Ein großer Teil des Inhalts ist im Klartext vorzufinden, Inhalte sind nicht semantisch gruppiert und es kann nicht direkt nach Logstufen gefiltert werden. Somit besteht keine eingebaute Unterscheidung zwischen Build-System- und Compiler-Ausgaben und Diagnostiken-Objekte werden nicht strukturiert dargestellt.

Die Trennung der Abschnitte des Protokolls in Projekte geschieht in unterschiedlichen HTML-Dokumenten, was projektübergreifendes Suchen auf der Seite erschwert.

Die Implementierung der HTML-Generierung erfüllt aktuelle Standards nicht mehr, was dazu führt, dass die Anzeige und Analyse im Browser nicht performant funktioniert. Zusätzlich sind Patches für die Generierung extrem invasiv und kaum wartbar.

Dadurch, dass diese HTML nur bei Nutzung von `sapmake` generiert wird, für Ninja nicht existiert, und sie einige Probleme aufweist, wird eine eigene HTML-Ausgabe für `log-jitsu` entwickelt. Diese ist kein Teil des Analyse-Kerns der Anwendung und wird deshalb nur am Rande thematisiert und zunächst rudimentär ausgearbeitet. Ziel ist es, die Zuordnung von Inhalten zu Projekten in einer Datei möglich zu machen, nach Logstufen zu unterscheiden und eine moderne, wartbare Architektur zu bieten, die durch Schlankheit möglichst performant ist.

Nach vollständiger Analyse des vorhergehenden Zustands und des Kontexts, unter dem `log-jitsu` entwickelt wird, wird im Folgenden Kapitel ein Rahmen an Anforderungen gesteckt, der als Ziel für die Implementierung der Anwendung gilt.

4 Konzeptionierung und Umsetzung

In diesem Kapitel werden Anforderungen für einen Prototypen definiert, der eine erfolgreiche Aufarbeitung der Build-Protokolle leistet. Sie beschreiben konkret, was die Anwendung erfüllen soll und bieten eine Grundlage für die folgende Vorstellung der Implementierung.

4.1 Anforderungen

Im Folgenden werden die Anforderungen an `log-jitsu` konkret herausgearbeitet. Sie dienen später als Grundlage für die Bewertung und werden in funktionale und nichtfunktionale Anforderungen unterteilt.

4.1.1 Funktionale Anforderungen

Um die Funktionsfähigkeit von `log-jitsu` zu bewerten, muss eine Liste an konkreten Funktionen definiert werden, die es bieten soll.

Aus den im Kapitel 2.2 (Zielsetzung) erläuterten Anforderungen und der in Kapitel 3 durchgeführten Analyse der Problemstellung lässt sich folgende konkrete Liste von funktionalen Anforderungen kompilieren:

- `log-jitsu` läuft automatisch in Konjunktion an den Build, der Output des Builds ist Input für `log-jitsu`.
- Es verarbeitet Diagnostiken, Hinweise, Traces und Code-Ausschnitte im Build-Protokoll und entnimmt Struktur und Informationen.
- Es generiert eine Übersicht über alle Diagnostiken im HTML-Format.
- Diagnostiken werden den Projekten und Zielen zugeordnet.
- `log-jitsu` reproduziert alle Ausgaben pro Projekt in derselben Reihenfolge.
- Es ist in der Lage, trotz unterschiedlicher Compiler und/oder Build-Systeme die gleichen Informationen zu lesen

Zusätzlich zu diesen funktionalen werden nichtfunktionale Anforderungen bestimmt, die `log-jitsu` ebenfalls erfüllen soll.

4.1.2 Nichtfunktionale Anforderungen

Die nichtfunktionalen Anforderungen an `log-jitsu` können in zwei Gesichtspunkte gegliedert werden. Diese sind die Anforderungen von Nutzenden und Entwickelnden, die mit und an der Anwendung arbeiten werden.

Für die Nutzung ist wichtig, dass die von `log-jitsu` generierten Ausgaben gut lesbar sind, die dargestellten Informationen viel semantischen Wert bringen und dass alle relevanten Informationen dargestellt sind. Darüber hinaus soll die Implementierung dafür sorgen, dass `log-jitsu` eine so geringe Laufzeit wie möglich hat. Für die Weiterentwicklung ist wichtig, dass `log-jitsu` so erweiterbar wie möglich ist, damit Unterstützung für andere Build-Systeme, Compiler oder neue Compiler-Versionen einfach zu implementieren ist. Dafür ist zusätzlich zu erweiterbarem Code eine gute Dokumentation von Nutzen. Sollten Änderungen an Build-System- oder Compilerausgaben vollzogen werden, so sollte es möglich sein, `log-jitsu` so anzupassen, dass es auch die geänderten Protokolle verarbeiten kann, ohne eine Anpassung der Ausgabefunktion im Build-System oder Compiler zu benötigen. Dies spart Entwicklungsaufwand und vereinfacht eine zukünftige Annäherung von `sapmake` an neuere GNU make-Versionen. Genauso sollte `log-jitsu` insofern unabhängig sein, als dass für eine Änderung der Verarbeitung durch `log-jitsu` keine Änderung am Build-Prozess nötig ist. Diese Anforderung ist dadurch, dass `log-jitsu` eine alleinstehende Applikation ist, automatisch erfüllt, nur die direkt zuvor beschriebenen Anpassungen sollen unabhängig bleiben.

Konkret ergeben sich also folgende nichtfunktionale Anforderungen:

- Ausgaben sind für Menschen lesbar aufbereitet, die HTML-Oberfläche ist leicht zu navigieren, besteht nur aus einer Seite und erfüllt moderne Standards.
- Semantisch wertvolle Informationen (z.B. zu Diagnostiken gehörende Code-Ausschnitte oder Hinweise) sind im Kontext dargestellt, das bedeutet, dass eine Diagnostik mit Zeitstempel, Code-Ausschnitten, Warnungen und Traces gruppiert wird.
- `log-jitsu` benötigt möglichst wenig Zeit, um das Protokoll zu verarbeiten. `log-jitsu` behandelt unter Umständen mehrere Millionen Zeilen Protokoll, eine Speicherplatz und Zeit schonende Implementierung ist stark erwünscht.
- Um Unterstützung für einen neuen Compiler oder ein neues Build-System zu implementieren, muss nur eine zusätzliche Datei hinzugefügt werden, die die Unterschiede herausparst, und die neue Addition mit in die Liste unterstützter Compiler und Build-Systeme aufgenommen werden.
- Es soll möglich sein, bei Änderungen keine Anpassungen an der Generierung der Ausgabe in z.B. `sapmake` machen zu müssen.

Mit einem klaren Umriss der für eine Lösung zu bewertenden Anforderungen, wird im Folgenden zunächst ein grundlegender Lösungsansatz für eine Parseranwendung umrissen und daraufhin implementiert.

4.2 Konzeption eines Lösungsansatzes

Um in Konjunktion zu einem Kernel-Build laufen zu können, muss die Anwendung in der Lage sein, `stdin`-Input zu akzeptieren und diesen Zeile für Zeile zu bearbeiten. Auf diesem Wege erhält der Parser das Build-Protokoll.

Als grundlegender Ansatz für die Parseranwendung dient Normalisierung an zwei Stellen. Zunächst müssen die Unterschiede der Build-Systeme bei der Generierung des Protokolls ausgeglichen werden, woraufhin diese dann je nach verwendetem Compiler für jede Diagnostik geparst werden können.

Somit ist der Kern des Prototypen in zwei Teile geteilt: einen Teil, der eine Art Vorverarbeitung des Protokolls übernimmt, er akzeptiert die Eingabe des Protokolls, erkennt, welches Build-System das Protokoll erstellt hat und bereitet es in einer vom Build-System unabhängigen Form für den nächsten Teil des Parsers vor indem er die Unterschiede zwischen den Build-Systemen entsprechend individuell behandelt.

Der zweite Teil des Prototypen erhält alle Zeilen des Protokolls in vorbereiteter Form, identifiziert, welcher Compiler für das Schreiben welcher Diagnostiken verantwortlich war und erkennt Warnungen, Fehler, Anmerkungen und Ähnliches für die spätere Verwertung. Diese parst er mithilfe von Nom in ein Datenformat, das die Aufarbeitung der Ergebnisse von Compilern und Tests in einer HTML-Datei ermöglicht.

Diese Struktur ermöglicht den Ausgleich der Unterschiede zwischen Build-Systemen und Compilern und eine leicht wart- und erweiterbare Funktionsweise des Prototypen.

Zusätzlich sind Komponenten vonnöten, die die richtige Aktivierung und Koordination der Teile garantieren und die Generierung der HTML-Ausgabe durchführen.

Im Folgenden wird die Konkrete Umsetzung der in diesem Unterkapitel erstellten Ansätze dargestellt und wichtige Entscheidungen, Einschränkungen und Möglichkeiten diskutiert.

4.3 Umsetzung

Um die Implementierung der Parserapplikation in dieser Arbeit möglichst funktionsnah darzustellen, geschieht dies in einer Struktur, die der Konzeptstruktur von log-jitsu entspricht. Zunächst werden die Vorverarbeitung des Build-Protokolls und die Normalisierung der Unterschiede zwischen Build-Systemen gezeigt.

Der Aufruf für die Verarbeitung des Build-Protokolls hat folgende Form: `kernel-build-aufruf | log-jitsu`. Das `|`-Symbol sorgt dafür, dass der von Build-Prozess produzierte Output durch den `stdin` an `log-jitsu` übergeben wird. Die entstandene Sammlung an Zeilen, die `log-jitsu` erhält, verpackt es in einen Iterator, dank dem später über die Zeilen des Protokolls iteriert werden kann. Zusätzlich wird aus dem Kommandozeilenaufruf herausgelesen, welches Build-System verwendet und welche Output-Art von `log-jitsu` gewünscht wird. Ein Beispielaufruf kann z.B. die Kommandozeilenparameter `-t html -b ninja` enthalten, um zu signalisieren, dass der gewünschte Ausgabetyp (Option `-t`) ein HTML-Dokument und das verwendete Build-Tool für den Bauprozess (Option `-b`) Ninja ist.

Je nach ausgewähltem Build-System wird dieser Iterator an einen unterschiedlichen Parser gegeben, der Zeile für Zeile über das Protokoll iteriert und die Eigenheiten des jeweiligen Systems normalisiert.

4.3.1 Normalisierung der Build-Systeme

Die Parser für `sapmake` und Ninja lesen beide Zeile für Zeile das Protokoll ein. Ihre Aufgabe ist es, für jede Zeile zu identifizieren, zu welchem Projekt sie gehört und wer sie generiert hat. Eine Zeile kann vom Build-System oder einem verwendeten Compiler generiert werden.

Daraufhin wird jede Zeile in ein spezielles Datenformat überführt, `LineWithProject` genannt. Die Definition dieser Struktur ist in Codebeispiel Code 16 gezeigt.

```
pub struct LineWithProject {
    pub project: Project,
    pub log: String,
    pub compiler: LineGeneratedBy,
}
```

Code 16: Definition der Datenstruktur LineWithProject

Instanzen von `LineWithProject` erhalten:

- einen Projekt-Parameter[8], der markiert, für welches Projekt die Zeile generiert wurde
- einen Compiler-Parameter[9], der markiert, von wem die Zeile generiert wurde
- einen Log-Parameter, der den Textinhalt der Zeile speichert

Jede Iteration des Parsers über eine Zeile des Protokolls soll ein `LineWithProject`-Objekt erzeugen.

Um die Funktionsweise dieses Parserschritts zu demonstrieren, wird im Folgenden anhand Beispiel Code 17 die `next`-Methode des Ninja-Parsers gezeigt. Das Parsen von `sapmake`-Ausgaben funktioniert analog, angepasst an die unterschiedliche Syntax.

[8]Bei dem Datentypen `Project` handelt es sich um eine besondere Form einer Zeichenkette, die performanter sein soll.

[9]Bei dem Datentypen `LineGeneratedBy` handelt es sich um einen Aufzählungstypen, der alle bekannten Compiler und Build-Systeme enthält. **Anmerkung:** In dieser Arbeit wird bei jeder Erwähnung des Datentyps `LineGeneratedBy` nur von einem Compiler gesprochen, der die Zeile geschrieben haben soll. Diese Entscheidung wurde absichtlich der Leserlichkeit halber getroffen, eine Zeile kann auch vom Build-System generiert worden sein.

```rust
impl<I: Iterator<Item = String>> Iterator for Parser<I> {
  type Item = LineWithProject;

  fn next(&mut self) -> Option<Self::Item> {
      let line = self.input.next()?;

      let log = if let Ok((rest, name)) = Self::projekt_der_zeile(&line) {
        self.current_project = Project::from(name);
        //  |
        //  Dieser Block ist schematisch in Code 20 ausgelagert.
        //  |
        rest.to_string()
      } else if let Ok((comp_version, _)) =
        tag("DBG _GCC_COMPILER_VERSION")(&line)
      {
        //  |
        //  Dieser Block ist schematisch in Code 22 ausgelagert.
        //  |
      }

      Some(LineWithProject {
          project: self.current_project.clone(),
          compiler: self.current_line_generated_by.clone(),
          log,
      })
  }
}
```

Code 17: Vereinfachte Version des Ninja-Parser-Iterator

Beispiel Code 17 zeigt einen vereinfachten Aufbau der `next`-Methode des Iterators im Ninja-Parser. Zwei Codeblöcke wurden ausgelagert und werden der Lesbarkeit halber einzeln im Detail besprochen. Weitere Arme der Methode wurden zusammen mit der Fehlerbehandlung bewusst ausgelassen, um schematisch identische Funktionen zu vermeiden und das Verständnis des Funktionsprinzips zu vereinfachen. Sie sind für ein Verständnis der `next`-Funktion nicht vonnöten.

Die in Beispiel Code 18 zu sehende Funktion `projekt_der_zeile` identifiziert dort, wo Ninja ein Projekt markiert, welches dieses ist, und parst mithilfe von Nom seinen Namen heraus.

```rust
fn projekt_der_zeile(i: &str) -> IResult<&str, &str> {
    let (rest, _) = delimited(
        char('['),                                  // "["
        separated_pair(digit1, char('/'), digit1),// "Zahl/Zahl"
        char(']'),                                  // "]"
    )(i)?;

    let (rest, _) = multispace1(rest)?;
    let (rest, name) = delimited(
        char('['),                                  // "["
        is_not("]"),                                // Projektname, nicht "]"
        char(']'))(rest)?;                          // "]"
    let (rest, _) = multispace1(rest)?;

    Ok((rest, name))
}
```

Code 18: Eine Funktion, die den Projektnamen für eine Zeile herausparst

Wie in Beispiel Code 18 zu sehen, wird zunächst eine Zeichenkette der Form `[<Zahl>/<Zahl>]` am Anfang der Zeile ignoriert. Danach wird die Angabe des Projekts der Form `[<Projektname>]` identifiziert. `<Projektname>` darf nicht leer sein. `log-jitsu` entfernt die Klammern und gibt ein Ergebnis[10], das den Rest der Zeile und den Namen des Projekts beinhaltet, zurück.

Jede Parserinstanz erhält, wie in folgendem Beispiel Code 19 gezeigt, mehrere Membervariable. Hier sind wichtig: eine Variable je für das momentan laufende Projekt, den momentan schreibenden Compiler und für jeden Compiler, der beim Lesen der Umgebungsvariablen am Anfang des Protokolls identifiziert wurde. Wie Werte der Variablen für C-, Rust- und weitere Compiler, die beim Bauprozess genutzt wurden, gesetzt werden, wird später um Beispiel Code 22 genauer erläutert[11].

[10] im Ergebnisformat `Result(Rest, geparst)`

[11] Der Prozess wird nicht an dieser Stelle erläutert, um die beiden aus der next-Methode entnommenen Codeblöcke in der gleichen Reihenfolge wie in Codebeispiel 16 zu halten. Für in der Zwischenzeit folgende Beispiele wird angemommen, ein solcher Wert sei bereits vorhanden.

```rust
pub struct Parser<I: Iterator<Item = String>> {
    current_project: Project,
    current_compiler: LineGeneratedBy,
    input: I,
    c_compiler: LineGeneratedBy,
    rust_compiler: LineGeneratedBy,
    // ...
}
```

Code 19: Die Membervariablen des Build-System-Parsers

Das für die momentane Zeile identifizierte Projekt wird in der entsprechenden Membervariable `current_project` des Parsers gespeichert und allen weiteren Zeilen als Projekt-Parameter in der `LineWithProject`-Struktur angehängt, bis ein neues Projekt gefunden wird. Daraufhin wird die Variable überschrieben und der Prozess beginnt erneut.

Um zu erkennen, welcher Compiler eine Zeile geschrieben hat, muss der Parser zwischen verschiedenen Arten von Zeilen unterscheiden:

- Zeilen, die ein neues Projekt einleiten (je nach Build-System unterschiedlich)
- Zeilen, die indizieren, dass eine neue Datei kompiliert wird (je nach Build-System unterschiedlich gestaltet, bei Ninja beginnt eine solche Zeile meist mit `Compiling <Dateiname>.<Typendung>`)
- Alle anderen Zeilen

Wenn der Parser erkennt, dass eine neue Datei kompiliert wird, speichert er in seiner `current_compiler`-Variable ab, von welchem Compiler sie bearbeitet wurde. Zeilen die folgen, ohne dass eine neue Datei oder ein neues Projekt beginnt, werden direkt in ihrem `LineWithProject`-Objekt als von diesem Compiler generiert markiert. Beginnt ein neues Projekt, so wird zunächst die Compiler-Variable auf den Namen des Build-Systems zurückgesetzt, da dieses die Zeile und eventuell folgende generiert hat.

```
// ...
// Dieser Codeblock entstammt der next-Methode des Iterators.
// Nom erkennt, ob die Zeile mit "Compiling" beginnt. Wenn ja, wird eine
neue Datei kompiliert.
if let Ok((rest, _)) = take_until("Compiling")(&line[..]) {
  let Ok((rest, _)) = take_until(".")(rest.trim());
  let (rest, _) = char('.')(rest).ok()?;
  // ^
  // Nom liest die Typendung der Datei aus und speichert sie in der
Variable `rest`.
  self.current_compiler = self.compiler_identifizieren(rest);
}
// ...
```

Code 20: Ein Codeblock, der den genutzten Compiler erkennt

Codebeispiel Code 20 - der erste aus Code 17 ausgeschnittene Codeblock - erkennt, ob das Build-System einen Compiler ruft um eine neue Datei zu kompilieren. `log-jitsu` parst mithilfe von Nom den Namen der Datei heraus, identifiziert ihren Typen, fragt den Compiler ab, der im Bauprozess für Dateien des Typen verwendet wird und markiert die Zeile als von diesem Compiler geschrieben.

Dafür benötigt `log-jitsu` zusätzlich folgende Funktion `compiler_identifizieren` in Beispiel Code 21:

```rust
pub fn compiler_identifizieren(typendung: &str) -> LineGeneratedBy {
      match typendung.trim() {
        "c" | "cpp" => {
            // Wenn noch kein C-Compiler gespeichert ist, Fehler melden.
            // Sonst:
            self.c_compiler.clone()
        }
        "rs" => {
            // Wenn noch kein Rust-Compiler gespeichert ist, Fehler melden.
            // Sonst:
            self.rust_compiler.clone()
        }
        // Typ ist unbekannt, also noch nicht unterstützt:
        _ => LineGeneratedBy::UnsupportedCompiler(typendung),
        // Der Typ der Datei wird zur Fehlerbehandlung weitergegeben.
    }
}
```

Code 21: Eine Funktion, die Typendungen einen gespeicherten Compiler zuordnet

Sie fragt je nach Typ der Datei ab, ob ein Compiler für diesen Datentyp beim Parser hinterlegt ist, löst einen Fehler aus, wenn dies nicht der Fall ist und gibt sonst den korrekten Compiler zurück.

Die Funktion `erkenne_typ_der_datei` ist im Detail nicht weiter interessant, in ihr wird die Typendung des Dateinamens herausgeparst.

Unter der Voraussetzung, dass `log-jitsu` einen Compiler für alle verwendeten Dateitypen gespeichert hat, erfüllt der Parser somit seinen Zweck. Um die Voraussetzung erfüllen zu können, muss er jedoch zunächst alle wichtigen Parameter, wie die für den Bau verwendeten Compiler aus dem Anfang des Protokolls lesen. Der in Beispiel Code 22 ausgelagerte Codeblock wird dann ausgelöst, wenn `log-jitsu` am Anfang der Zeile `DBG _GCC_COMPILER_VERSION` liest, also die Variable, die angibt, welcher C-Compiler für den Build genutzt wird, findet.[12]

[12]Buildvariablen befinden sich, wie in Beispiel Code 12 am Anfang des Protokolls.

```
// ...
// Dieser Codeblock entstammt der next-Methode des Iterators.
match comp_version.trim() {
  "clang" => self.c_compiler = LineGeneratedBy::SomeClangVersion,
  _ => {
    if let Ok(n) = comp_version.trim().parse::<i32>() {
      match n {
        i if i < 6 => self.c_compiler = LineGeneratedBy::Gcc5AndEarlier,
        6 => self.c_compiler = LineGeneratedBy::Gcc6,
        7..=9 => self.c_compiler = LineGeneratedBy::Gcc7To10,
        i if i >= 10 => self.c_compiler = LineGeneratedBy::Gcc10AndLater,
        _ => {
          self.c_compiler = LineGeneratedBy::UnsupportedCompiler(
            comp_version.trim().to_string(),
          )
        }
      }
    } else {
      self.c_compiler = LineGeneratedBy::UnsupportedCompiler(
        comp_version.trim().to_string(),
      );
    }
  }
}
// ...
```

Code 22: Auslesen des für einen Build benutzten C-Compiler

Code 22 - der zweite aus Code 17 ausgeschnittene Codeblock - liest den Inhalt der Variablen aus. Ist die Variable eine Zahl - hier geprüft durch `if let Ok(n) = comp_version.trim().parse::<i32>()`, so wird sie in ein 32-bit Integer-Format geparst und einer GCC-Version zugeordnet. Eine Zahl korrespondiert hier direkt zu einer GCC-Version, da die Variable im Protokoll grundsätzlich für die GCC-Version steht. Einen Ausnahmefall gibt es, wenn sie `"clang"` ist, so handelt es sich bei dem C-Compiler um eine (noch unbekannte) Clang-Version. Ein folgender, hier nicht gezeigter Codeblock identifiziert anhand einer anderen Buildvariable die genaue Clang-Version. In beiden Fällen speichert der Parser die Compilerversion ab, um sie später verwenden zu können. Scheitern beide Zuordnungen, so ist der Compiler unbekannt und wird als nicht unterstützer Compiler gespeichert.

Analog funktioniert die Erkennung von Compilern anderer Sprachen wie Rust. Andere Variablen wie die Nachverfolgungsstufe werden auf die gleiche Art und Weise ausgelesen.

Da `log-jitsu` die herausgeschriebenen Variablen am Anfang des Protokolls findet, ist zu erwarten, dass zum Zeitpunkt, wo Ziele gebaut werden und `log-jitsu` versucht einen gespeicherten Compiler abzurufen, bereits einer vorhanden ist.

Die entstandene Sammlung von mit Projekt und Compiler markierten Zeilen werden an den zweiten Parser weitergegeben. Dieser implementiert ebenfalls einen Iterator, über `LineWithProject`.

4.3.2 Normalisierung des Compiler-Outputs und Informationsverarbeitung

Um die vom Build-System-Parser generierten, mit Projekt und Compiler markierten Zeilen zu verarbeiten, wird ihr Inhalt iterativ überprüft.

Der Parser für die weitere Verarbeitung der Zeilen besteht aus zwei Hauptkomponenten: einem großen Protokollparser - dem Iterator der die Zeilen verarbeitet und Ergebnisse ausgibt, und kleine, Compiler-spezifische Parser, die der Protokollparser nutzen kann, um unterschiedliche Zeilen zu verarbeiten.

Jede Instanz eines Compiler-spezifischen Parsers implementiert ein Trait namens `CompilerDiagnosticParser`, gezeigt in Code 23.

```
pub trait CompilerDiagnosticParser {
    fn parse_diagnostic<'a>(&self, i: &'a str) -> IResult<&'a str,
Diagnostic>;
    fn parse_trace<'a>(&self, i: &'a str) -> IResult<&'a str, Trace>;
    fn parse_code_snippet<'a>(&self, i: &'a str) -> IResult<&'a str, &'a
str>;
    fn parse_note<'a>(&self, i: &'a str) -> IResult<&'a str, Note>;
}
```

Code 23: Die für CompilerDiagnosticParser zu implementierenden Funktionen

Dieser Trait erwartet vier Funktionen, die je eine Diagnostik, ein Trace, einen Codeausschnitt oder einen Hinweis erkennen und verarbeiten können.

Für jede Zeile wird ein Parser instanziiert, der Methoden besitzt, die auf den jeweiligen Compiler angepasst sind, der die Zeile generiert hat.

Dies geschieht wie in Code 24, indem anhand des im `compiler`-Attributs der `LineWithProject` markierten Compiler entschieden wird, welcher Parser vonnöten ist.

```
let parser = if compiler.war_clang() { // Diese Funktion testet auf Clang-
Versionen
  clang::Parser::neuer_diagnostic_parser()
} else if compiler.war_gcc() { // Diese Funktion testet auf GCC-Versionen
  gcc::Parser::neuer_diagnostic_parser()
} else { // Auf andere Parser testen
  // ...
};
```

Code 24: Instanziierung des richtigen Parsers für die Zeile

Hat eine Clang-Version die Zeile generiert, so wird ein Parser für Clang-Output instanziiert; für andere Compiler geschieht dies analog. Mithilfe dieser Instanz können trotz Compiler-spezifischer Unterschiede die gleichen Informationen aus Zeilen gelesen werden. Damit ist die zweite Normalisierung erreicht.

Der Protokollparser testet der Reihe nach, ob eine der Parserfunktionen für Diagnostiken, Traces, Codeausschnitte oder Hinweise erfolgreich die Zeile parsen kann. Da die Mengen der Zeilen aller dieser möglichen Formate in ihrer Form disjunkt sind, sich also gegenseitig ausschließen, kann der Protokollparser so sicher erkennen, um welche Art es sich handelt. Sollte eines der Formate auf die Zeile passen, so wird sie für dieses geparst und verarbeitet.

Da die Zeilen eines Protokolls semantisch voneinander abhängen, Code-Ausschnitte und Notizen zum Beispiel immer zu einer Diagnostik gehören, reicht ein einfaches Parsen der Zeilen nicht, um bedeutungsvolle Ergebnisse zu generieren.

Die Ergebnisse werden dafür zusammen in Instanzen einer Diagnostik-Struktur gruppiert. Diese Struktur ist in Code 25 implementiert.

```
impl Diagnostic {
    fn new(message: String, severity: Severity, source: SourceLocation) ->
Self {
        Diagnostic {
            message,
            severity,
            location: source,
            code: None,
            trace: vec![],
            notes: vec![],
        }
    }
}
```

Code 25: Aufbau der Diagnostik-Struktur

Eine Diagnostik beinhaltet eine Schwere - z.B. Warnung oder Fehler, eine Nachricht, einen Ort, an dem z.B. der Fehler aufgetreten ist, angehängte Codeausschnitte, Notizen und Traces.

Ähnlich wie beim Build-System-Parser erhält der Protokollparser eine Laufvariable `current_project_and_diagnostic`, deklariert als `Option<(Project, Diagnostic)>`[13]. Diese Variable speichert ab, zu welchem Projekt und welcher Diagnostik die momentane Zeile gehört.

In Reihenfolge werden folgende Schritte durchlaufen:

Beispielschritt Code 26: `log-jitsu` setzt in jeder Iteration der `next`-Funktion die Variablen `current_project_and_diagnostic` und `buffered_traces` zurück . Der Rest jeder Iteration befindet sich in einer Schleife, die dann endet, wenn eine Diagnostik abgeschlossen ist oder keine Zeilen mehr vorhanden sind. Eine Iteration des Protokollparsers kann mehrere Zeilen des Protokolls einlesen.

[13]Eine `Option` ist ein Datentyp, der explizit `None`-Werte, also leere Werte erlaubt und besondere Funktionalität bereitstellt.

```
fn next(&mut self) -> Option<Self::Item> {
  let mut buffered_traces = Vec::new();
  let mut current_project_and_diagnostic: Option<(Project, Diagnostic)> =
None;
  while let Some(LineWithProject {
        project: next_project,
        compiler,
        log: next_line,
  }) = self.input.peek()
  {
    // ...
  }
}
```

Code 26: Grundgerüst und Anfang der next-Funktion

Beispielschritt Code 27: Zunächst versucht `log-jitsu` mithilfe von Nom, einen Trace zu parsen. Dieser hat das Format eines Dateinpfades und gehört kontextuell zu einer folgenden Diagnostik. Um nach dem Einlesen aller zu dieser Diagnostik gehörenden Traces eine Zuordnung herstellen zu können, werden sie in dem Vektor `buffered_traces` zwischengespeichert.

```
if let Ok((_rest, trace)) = parser.parse_trace(next_line) {
    buffered_traces.push(trace);
    self.line_for_debugging += 1;
    drop(self.input.next());
    continue;
  }
```

Code 27: Parsen eines Trace

Falls ein Trace erkannt wird, speichert ihn der Codeblock Code 27 ab, konsumiert die Zeile und löst den nächsten Durchlauf der Schleife für die nächste Zeile aus.

Beispielschritt Code 28: Wenn `log-jitsu` eine Diagnostik erkennt und eine bereits offene Diagnostik in `current_project_and_diagnostic` vorliegt, so wird diese beendet, da nun eine neue beginnt. Die Schleife wird beendet und eine neue Iteration der `next` -Funktion wird eingeleitet, ohne die Zeile zu konsumieren.

```
if let Ok((_rest, mut diagnostic)) = parser.parse_diagnostic(next_line) {
  if let Some((project, previous_diagnostic)) =
current_project_and_diagnostic {
    return Some(IterResult::Diagnostic(project, previous_diagnostic));
  } else {
    diagnostic.trace.append(&mut buffered_traces);
    current_project_and_diagnostic =
      Some((next_project.clone(), diagnostic.clone()));
    drop(self.input.next());
    continue;
  }
};
```

Code 28: Parsen einer Diagnostik

Sollte keine laufende Diagnostik offen sein, so erhält das Diagnostik-Objekt alle zuvor eingelesenen Traces. Die Variable `current_project_and_diagnostic` wird auf die momentane Diagnostik aktualisiert und die Zeile konsumiert. Daraufhin wird der nächste Durchlauf der Schleife gestartet.

Anschließend werden folgende Codeausschnitte und Hinweise mit Nom in ähnlichem Schema geparst und an die Diagnostik angehängt. Es existiert zusätzliche Logik zur Erkennung und Verarbeitung besonderer Formate von Hinweisen, die z.B. über mehrere Zeilen gehen, diese sind jedoch zum Verständnis des Prinzips von `log-jitsu` nicht von Relevanz.

Sollte die getestete Zeile zu keinem der Formate passen, so wird sie als `NormalLine` - eine Klartextzeile - weitergegeben und als nicht vom Compiler generiert behandelt. Da alle vom Build-System generierten Zeilen in diese Kategorie fallen, an dieser Stelle semantisch nicht mehr relevant sind und so verarbeitet werden können, gibt es keinen spezifisch für Build-System-Ausgaben implementierten `CompilerDiagnosticParser`.

Nach Ende des Parse-Prozesse werden die vom Parser generierten `Diagnostic`- und `NormalLine`-Objekte weitergegeben um die HTML-Ausgabe zu produzieren. Dies geschieht in einer eigenen Komponente der Anwendung.

4.3.3 HTML-Ausgabe

Die Implementierung einer HTML-Ausgabe zur Kontextualisierung und Visualisierung der Diagnostiken wird in diesem Unterkapitel dargestellt. Diese Darstellung passiert anhand des Code-Ausschnitts Code 29, der die grundlegende Funktionsweise dieser

HTML-Generierung zeigt. Es wird nur eingeschränkt auf die genaue Umsetzung aller Komponenten eingegangen, da die tatsächliche Aufgabe von `log-jitsu`, die Normalisierung der Compiler- und Build-System-Unterschiede und die Verarbeitung der Informationen im Build-Protokoll, bereits umgesetzt ist.

```
pub fn complete_html(log_parser_results: impl Iterator<Item = IterResult>)
-> String {
  //Sammelt alle Zeile pro Projekt:
    let log_entries_by_project =
IterResult::collect_results_by_project(log_parser_results);
  //Schreibt den Kopf des HTML-Dokuments:
    let mut builder = String::new();
    builder.push_str(include_str!("html_head.html"));
  //Fügt ein Navigationselement für jedes Projekt hinzu:
    push_nav_element(&mut builder, &log_entries_by_project);
  //Fügt für jedes Projekt den Inhalt hinzu:
    push_main_element(&mut builder, log_entries_by_project);
  //Stellt das Dokument fertig und gibt es zurück:
    builder.push_str("</body></html>");
    builder
}
```

Code 29: Generieren einer HTML-Seite

Wie in Code-Ausschnitt Code 29 zu sehen, werden alle Zeilen pro Projekt gesammelt und es wird eine Navigation erstellt, die die Auswahl der anzuzeigenden Projekte ermöglicht. Um die erwartete HTML-Seite zu generieren, liest `log-jitsu` alle `Diagnostic` - und `NormalLine`-Objekte ein und gruppiert sie strukturgerecht unter dem jeweiligen Projekt, dem sie zugeordnet wurden. Die Seite wird in der HTML-Syntax fertiggestellt und zurückgegeben und von `log-jitsu` in eine Datei namens `out.html` geschrieben. Verantwortlich dafür ist der originale Aufruf der Parser, der die zurückgegebene Liste von Ergebnissen direkt an den HTML-Generator weitergibt, vgl. Code 30.

```
let log_parser_result: Box<dyn Iterator<Item = IterResult>> =
Box::new(LogParser::parse_iter(Parser::new(stdin_lines)));
let output = output::html::complete_html(log_parser_result);
fs::write("out.html", output)?;
```

Code 30: Schreiben des HTML-Dokuments

Ein Beispiel einer von `log-jitsu` generierten HTML-Seite ist in Abbildung 1 zu sehen. Nutzende können auf der linken Seite das anzusehende Projekt auswählen und erhalten rechts eine Übersicht über alle Zeilen und Diagnostiken des Projekts, in Reihenfolge des Protokolls. Die Anzeige der Projekte auf der linken Seite ist alphabetisch sortiert.

Mit einem Überblick über die Hauptkomponenten von `log-jitsu` kann eine Bewertung der Funktionalität im Folgenden Kapitel durchgeführt werden. Dieses entscheidet, ob ein Produktiveinsatz von `log-jitsu` möglich ist und welche Bedingungen dafür erfüllt werden müssen.

5 Bewertung

Dieses Kapitel bietet eine Bewertung der vorgestellten Implementierung von `log-jitsu` anhand der zuvor definierten Anforderungen.

Die zu untersuchenden Anforderungen sind:

Funktionale Anforderungen: `log-jitsu`

1. läuft in Konjunktion an den Build.
2. verarbeitet alle Inhalte des Build-Protokolls.
3. generiert eine Übersicht über alle Diagnostiken im HTML-Format.
4. ordnet Diagnostiken ihren Projekten und Zielen zu.
5. reproduziert alle Ausgaben pro Projekt in derselben Reihenfolge.
6. ist in der Lage, trotz unterschiedlicher Syntax die gleichen Informationen zu erkennen.

Nichtfunktionale Anforderungen:

1. Ausgaben sind für Menschen lesbar aufbereitet, die HTML-Oberfläche ist leicht zu navigieren.
2. Semantisch wertvolle Informationen sind im Kontext dargestellt.
3. `log-jitsu` ist möglichst schnell.
4. Unterstützung für einen neuen Compiler oder ein neues Build-System ist einfach zu implementieren.
5. Für Änderungen muss nur `log-jitsu` angepasst werden.

Diese sind dem Unterkapitel 4.1 entnommen und zur Erinnerung hier abgekürzt und paraphrasiert gelistet.

Da das Ziel dieser Arbeit die Implementierung eines Prototypen war, der diese Anforderungen erfüllt, sind einige der Anforderungen per Design erfüllt. Deshalb liegt der Fokus dieser Arbeit auf der Implementierung und die Analyse der Anforderungen wird knapp und so präzise wie möglich durchgeführt. Die trivial erfüllten Designaspekte werden in Zusammenhang mit den zugehörigen Anforderungen beleuchtet. Zusätzliche Anforderungen werden anhand zweier Beispiele geprüft:

Zunächst wird `log-jitsu` anhand eines Aufrufs zusammen mit einem Build-Prozess getestet. Dies testet den Produktiveinsatz der Anwendung und ermöglicht das Überprüfen einiger wichtiger Anforderungen. Ein Kommandozeilenaufruf für diesen Test sieht etwa wie folgt aus: `/SAPDevelop/Benutzer/git/sapkernel/maketools/container/unix/run_sapmake.sh --verbose -makecmd /SAPDevelop/Benutzer/git/sapkernel/sapmake/sapmake -trace 3 --output-sync all | log-jitsu`.

Der Aufruf produziert durch die in Unterkapitel 4.3.3 demonstrierte Implementierung eine HTML-Datei. Screenshots dieser Datei sind im Anhang zu finden und werden im Laufe der nächsten Unterkapitel behandelt. Es werden nicht alle Anforderungen anhand dieses Produktiveinsatzes demonstriert, da die korrekte Verarbeitung einer Diagnostik

im Rahmen dieser Arbeit anhand eines konkreten Beispiels anschaulicher zu diskutieren ist.

Es folgt also als zweiter Test die Eingabe eines Protokolls mit einer einzigen Testdiagnostik, die das in Code 25 gezeigte Grundschema einer Diagnostik voll abbildet.

```
1 /SAPDevelop/Benutzer/git/sapkernel/flat/nlsui3.c: In function 'int
readdir_rU16(DIR*, direntU16*, direntU16**)':
2 /SAPDevelop/Benutzer/git/sapkernel/flat/nlsui3.c:577:61: warning: 'int
readdir_r(DIR*, dirent*, dirent**)' is deprecated [-Wdeprecated-
declarations]
3   577 |   rc = readdir_r( dirp, &(c_entryBuffer.c_entry), &c_result );
4       |                                                             ^
5 /sapmnt/appl_sw/sysroot-sles15/usr/include/dirent.h:189:12: note:
declared here
6   189 | extern int __REDIRECT (readdir_r,
7       |            ^~~~~~~~~~
```

Code 31: Testeingabe

Das Protokoll in Testbeispiel Code 31 ist eine Diagnostik, der ein Trace, ein Codeausschnitt und eine Notiz zugeordnet werden können. Eine Analyse der Testeingabe durch `log-jitsu` gibt das Diagnostik-Objekt Code 32 im Anhang zurück.

Produktiveinsatz, Testeingabe und Diagnostik-Objekt werden im Folgenden für die Prüfung der Anforderungen näher betrachtet.

5.1 Funktionale Anforderungen

Um die Erfüllung der funktionalen Anforderungen für `log-jitsu` zu beurteilen, wird jede Anforderung einzeln geprüft.

Funktionale Anforderung 1 lässt sich leicht anhand des Kommandozeilenaufrufs von `log-jitsu` überprüfen. Durch die Form `kernel-build-aufruf | log-jitsu`, wie in Kapitel 4.3 erklärt, wird der Build-Prozess durchlaufen und seine Ausgabe an `log-jitsu` weitergegeben. Zur Überprüfung, ob dieser Prozess erfolgreich war, reicht also eine Überprüfung der anderen Anforderungen. Sollten diese erfüllt sein, so hat `log-jitsu` die korrekte Eingabe entgegengenommen und Anforderung 1 ist erfüllt.

Funktionale Anforderung 2 ist durch die Implementierung des Protokoll-Parsers (vgl. Code 26, Code 27, Code 28) erfüllt. Diese garantiert, dass alle einer Diagnostik

angehörenden Komponenten dieser zugeordnet werden und Ausgaben immer nach Diagnostiken gegliedert geschehen. Die Ausgabe Code 32 zeigt ein Diagnostik-Objekt, das die Schwere und Nachricht der Diagnostik und ihren Pfad beinhaltet. Zusätzlich wurden der Trace zuvor, der Code-Ausschnitt danach und der Hinweis korrekt dem Diagnostik-Objekt zugeordnet. Es werden alle Teile des Testeingabe korrekt verarbeitet, in die jeweilig richtigen Datentypen geparst und zusammen in ein Diagnostik-Objekt gruppiert, das alle Informationen in Kontext zueinander stellt.

Funktionale Anforderung 3 ist durch die Produktion der in Abbildung 1 zu sehenden HTML-Datei erfüllt. Sie enthält alle Diagnostiken der Protokollausgabe.

Funktionale Anforderung 4 ist durch die im Build-System-Parser implementierte Zuordnung jeder Zeile zu einem Projekt (vgl. Code 17) erfüllt. Die Anzeige in Abbildung 1 ist nach Projekten (alphabetisch sortiert) gegliedert.

Funktionale Anforderung 5 kann durch die Protokollausgabe in Abbildung 1 als erfüllt bestätigt werden. Alle Zeilen der HTML-Ansicht befinden sich in derselben Reihenfolge, in der sie im Protokoll vorzufinden waren.

Funktionale Anforderung 6 Ist per Design und durch Implementierung von Parsern für verschiedene Build-Systeme und Compiler erfüllt.

Zusätzlich zu den funktionalen wurden nichtfunktionale Anforderungen definiert. Diese werden folgend geprüft.

5.2 Nichtfunktionale Anforderungen

Die nichtfunktionalen Anforderungen an `log-jitsu` wurden getrennt in zwei Aspekte: Eine Beurteilung der Nutzung und der Weiterentwicklung.

Die Analyse der Nutzendenperspektive geschieht in Form einer Beurteilung der sich ergebenden HTML-Seite:
Nichtfunktionale Anforderung 1: Um die Lesbarkeit der Ausgabe-HTML für einen Menschen zu beurteilen, wird ein Blick auf die vom Testaufruf generierte HTML geworfen. Wie in Abbildung 1 zu sehen, ist die Nutzeroberfläche und damit die Navigierung intuitiv und übersichtlich, es ist eindeutig, welche Projekte angezeigt werden und welche nicht. Informationen sind protokolltreu vorhanden und die Gruppierung von Informationen unter z.B. Projekten trägt erheblich zum Verständnis bei. Es ist möglich, nach Diagnostiken oder restlichen Zeilen zu filtern.

Nichtfunktionale Anforderung 2: Durch die Architektur von `log-jitsu` sind alle semantisch zusammenhängenden Daten (Zeitstempel, Traces, Diagnostiken, Code-Ausschnitte etc. und wie sie zusammenhängen) gemeinsam in Diagnostik-Objekte oder unter Projekte gegliedert. `log-jitsu` liefert also eine vollständige Erkennung und Aufarbeitung semantischer Daten. Diagnostiken in der HTML-Ausgabe werden voneinander getrennt, farblich markiert und mit Hinweisen und Code-Ausschnitten dargestellt. Es fehlen zusätzliche Informationen wie der Zeitstempel einer Diagnostik im Protokoll, diese können aus den von `log-jitsu` generierten Objekten hinzugefügt werden.

Nichtfunktionale Anforderung 3: Beim Design von `log-jitsu` wurde an vielen Stellen auf Geschwindigkeit gesetzt. Beispiele sind die Nutzung von Nom, einem hoch optimierten Parsergenerator, die Unterscheidung zwischen Compilern in Form von Aufzählungstypen. Getestet wird diese Geschwindigkeit anhand des Durchlaufs von `log-jitsu` in Konjunktion an einen echten Build-Prozess. Dieser produzierte 191.694 Zeilen Protokoll und wurde bei zehn Tests von `log-jitsu` in durchschnittlich 3,8 Sekunden (mit einer Varianz von unter 0,05 Sekunden) zu einer HTML-Datei verarbeitet. In gegebenem Beispiel verarbeitet `log-jitsu` also etwa 50.500 Zeilen pro Sekunde. Einfache und stark optimierte Parser in Rust setzen ungefähr 500.000 Zeilen pro Sekunde um [13]. Dadurch, dass `log-jitsu` einen komplizierteren Prozess mit Einsatz mehrerer verschiedener Parser in zwei Parserstufen und der Generierung einer HTML-Datei durchläuft, kann mit einem deutlich höheren Aufwand pro Zeile gerechnet werden. Jeder Schritt produziert overhead, jede Zeile wird mehrere Male verarbeitet und in neuen Datenobjekten gespeichert. Unter der Annahme, dass ein 10-15-facher Aufwand zu erwarten ist, schneidet `log-jitsu` im Vergleich mit stark optimierten Rust-Parsern gut ab. Die generierte HTML-Seite erfüllt diese Anforderung nicht vollständig. Als einzelnes Performanzproblem bleibt das Wechseln zwischen einer Ansicht mit allen Zeilen zu einer Ansicht mit nur Diagnostiken. Dieses ist so unperformant, dass dem Nutzenden empfohlen wird, die Seite stattdessen zu aktualisieren. Eine Überarbeitung der HTML-Ausgabe ist vonnöten.

Ein Schwerpunkt der Beurteilung einer Entwickelndenperspektive auf `log-jitsu` betrachtet die Wartbarkeit der Implementierung:

Nichtfunktionale Anforderung 4: Dadurch, dass die Normalisierungen für Build-System- und Compiler-Unterschiede in eigene Parser ausgelagert sind, können jederzeit zusätzliche Parser hinzugefügt werden, die eine Unterstützung der Neuerungen ermög-

lichen. Diese Parser müssen lediglich eine gewisse Form erfüllen, die Eigenheiten der jeweiligen Ausgaben normalisieren und richtig aktiviert werden, vgl. Beispiel Code 24.

Nichtfunktionale Anforderung 5: Analog ist die Verarbeitung von Protokollen, die durch eine Anpassung oder Änderung einer Komponente des Build-Prozesses eine neue Struktur erhalten, durch Anpassungen an `log-jitsu` zu lösen. In den meisten Fällen genügt es, einen zusätzlichen Parser zu implementieren, der Protokolle der geänderten Form verarbeiten kann. Bei tiefergreifenden Unterschieden ist ein Eingriff in die Grundfunktionalität von `log-jitsu` denkbar, um die Änderungen in der Protokollausgabe auszugleichen. In beiden Fällen ist keine weitere Anpassung am Build-System vonnöten.

Nach dem Betrachten aller zuvor definierten Anforderungen wird in folgendem Unterkapitel ein Ausblick auf mögliche Erweiterungen durchgeführt.

5.3 Ausblick

Zusätzlich zu den in dieser Arbeit vorgestellten Implementierungen der Kern-Logik von `log-jitsu` und seiner HTML-Ausgabe sind Ergänzungen denkbar, die teils notwendig sind, um `log-jitsu` für einen Produktiveinsatz nutzbar zu machen oder halten. Diese werden in diesem Abschnitt kurz dargestellt. Anschließend folgt eine Bewertung des Protoypen.

5.3.1 Ergänzungen für `log-jitsu`

In Zukunft werden Komponenten des Kernels Code enthalten, der nicht nur in den Sprachen C und C++ geschrieben ist. Dementsprechend wird die ermöglichte Erweiterung von `log-jitsu` auf andere Compiler, zum Beispiel Rust-Compiler, vonnöten sein. Genauso sind Ausgaben von Werkzeugen, die z.B. Code auf Unsauberkeiten und Probleme im Format untersuchen, etwa `clippy`[14], von Interesse. Eine Aufarbeitung dieser Ausgaben ist möglich.

Dadurch, dass viele Entwickelnde über eine Kommandozeile arbeiten, ist die Ausgabe der Diagnostiken in ihrem Ergebnisformat nicht für alle Anwendungen geeignet. Eine Implementierung einer Ausgabefunktion, die die Diagnostiken in einer semantisch lesbaren Form für die Kommandozeile ausgeben kann, ist sinnvoll. Erweiterungen für andere Ausgabeformate sind denkbar, so zum Beispiel die automatische Generierung von GitHub-Code-Annotationen für aktualisierten Code.

[14] `clippy` ist ein Analyse-Werkzeug für Rust-Code

5.3.2 Ergänzungen für die HTML-Ausgabe

Die von `log-jitsu` generierten Ergebnis-Objekte beinhalten allen semantischen Kontext, der für eine Zeile oder Diagnostik zu identifizieren ist. Diese Informationen werden in der HTML-Ausgabe nicht zu Genüge dargestellt.
Es fehlt eine Anzeige der Zeitstempel von Zeilen im Protokoll und eine optisch ansprechende Aufarbeitung der Daten. Gleichzeitig muss die Performanz der HTML-Seite verbessert werden. Eine Anpassung an die SAP-Design-Richtlinien von Fiori und die Nutzung bereitgestellter Funktionselemente ist sinnvoll.

5.4 Fazit

Dieses Unterkapitel bietet eine Beurteilung der in dieser Arbeit dargestellten Implementierung von `log-jitsu`. Diese Beurteilung umfasst zusätzlich zu einer Diskussion der für die Implementierung gesetzten Anforderungen und Ziele ein Urteil dazu, inwiefern `log-jitsu` für einen Produktiveinsatz geeignet ist.

Das Design von `log-jitsu` ermöglicht eine triviale Erfüllung der meisten Anforderungen. Die Analyse der funktionalen Anforderungen zeigt, dass der Prototyp alle erwünschten Funktionen bietet. Der Nutzen von `log-jitsu` in der Durchführung dieser Aufgaben ist signifikant, eine automatische Analyse und Aufarbeitung des Build-Protokolls wird möglich gemacht.

Die Beurteilung der nichtfunktionalen Anforderungen zeigt eine klare Verbesserung der Aufbereitung von Informationen für Nutzende und eine den Industriestandard erfüllende Performanz. Gleichzeitig steht eine Verbesserung der HTML-Generierung aus. Diese erfüllt Erwartungen an Funktionalität und Geschwindigkeit noch nicht. Da die HTML-Ausgabe von `log-jitsu` nicht der Fokus der Implementierung war und sie vollkommen unabhängig vom Rest der Verarbeitung ist – die von `log-jitsu` generierten Ergebnis-Objekte beinhalten alle Informationen, die nötig sind, um eine zufriedenstellende HTML-Ausgabe zu implementieren – steht dies einer positiven Bewertung von `log-jitsu` nicht im Wege.

`log-jitsu` weist durch die Auslagerung von Compiler- und Build-System-spezifischer Parserlogik eine hohe Erweiterbarkeit auf. Aufzählungstypen und die Instanziierung der korrekten Parserinstanzen können um weitere Einträge erweitert werden.

Insgesamt ist der Kern der `log-jitsu`-Implementierung stark für einen produktiven Einsatz zu empfehlen. Er ist in der Lage, verschiedene Build-Protokolle zu analysieren und in eine Datenstruktur aufzubereiten, die semantische wichtige Kontexte erhält.

Literaturverzeichnis

[1] International Organization for Standardization und International Electrotechnical Commission, *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. ISO/IEC, 2011. [Online]. Verfügbar unter: https://www.iso.org/standard/35733.html

[2] T. Spitta, *Software Engineering und Prototyping: Eine Konstruktionslehre für administrative Softwaresysteme*, 1. Aufl. Springer, 2000. [Online]. Verfügbar unter: https://link.springer.com/book/10.1007/978-3-642-95519-8

[3] H. Neuendorf, *SAP-Architektur: Evolution und Technologie des SAP-Systems*. Springer Vieweg. [Online]. Verfügbar unter: https://www.springer.com/de/book/9783658324567

[4] SAP SE, „Preparing SAP Kernel Update / Rolling Kernel Switch Media“. [Online]. Verfügbar unter: https://help.sap.com/docs/SAP_LANDSCAPE_MANAGEMENT_ENTERPRISE/e7dead4286c545808b3bd24feee7448c/524bf2f96b414eb28c703714128a1935.html

[5] Free Software Foundation, „GNU Make Manual“. [Online]. Verfügbar unter: https://www.gnu.org/software/make/manual/make.html

[6] S. Klabnik und C. Nichols, *The Rust Programming Language*. No Starch Press, 2022. [Online]. Verfügbar unter: https://nostarch.com/rust-programming-language-2nd-edition

[7] The Rust Project, „Match expressions“.

[8] The Rust Project, „Destructuring“.

[9] The Rust Project, „Guards“.

[10] The Rust Project, „Binding“.

[11] The Rust Project, „Iterator“.

[12] The Rust Project Developers, *The Nominomicon*. 2023. [Online]. Verfügbar unter: https://tfpk.github.io/nominomicon/

[13] J. Zhu *u. a.*, „Tools and Benchmarks for Automated Log Parsing“, 2019, *IEEE / ACM*. [Online]. Verfügbar unter: https://dl.acm.org/doi/10.1109/ICSE-SEIP.2019.00021

Anhangsverzeichnis

1 Anhang: Verwendete Nom-Parserfunktionen

Name	Beschreibung
char(a)	Akzeptiert das Zeichen `a` am Anfang der Kette
alt((parser1 , parser2, …))	Testet mehrere Parserfunktionen, nur eine muss erfolgreich sein
many1(parser)	Wendet einen Parser so oft an, wie er erfolgreich ist; mindestens 1-mal
tag(A)	Akzeptiert die Zeichenkette `A` am Anfang der Kette
delimited(parser1, parser2, parser3)	Akzeptiert eine von parser2 akzeptierte Zeichenkette, die von zwei anderen akzeptierten Zeichenketten umgeben ist
separated_pair(parser1, trennzeichen, parser1)	Akzeptiert zwei von einer bestimmten Zeichenkette getrennte Zeichenketten
digit1	Akzeptiert eine (mindestens ein Zeichen lange) Kette an Ziffern
multispace1	Ignoriert (mindestens ein) Leerzeichen am Anfang der Kette
is_not(a)	Akzeptiert ein Zeichen, das nicht `a` ist
take_until(A)	Falls Nom die Zeichenkette `A` im Text findet, konsumiert es alles bis inklusive `A`

2 Anhang: Testausgabe

```
Diagnostic {
  message: "'int readdir_r(DIR*, dirent*, dirent**)' is deprecated [-
Wdeprecated-declarations]",
  severity: Warning,
  location: SourceLocation {
    path: "/SAPDevelop/I587738/git/sapkernel/flat/nlsui3.c",
    row: Some(577),
    column: Some(61)
  },
  code: Some("   rc = readdir_r( dirp, &(c_entryBuffer.c_entry),
&c_result );\n
^\n"),
  trace: [
    In {
      context: "function 'int readdir_rU16(DIR*, direntU16*,
direntU16**)'",
      location: SourceLocation {
        path: "/SAPDevelop/I587738/git/sapkernel/flat/nlsui3.c",
        row: None,
        column: None
      }
    }
  ],
  notes: [
    Note {
      message: "declared here",
      location: Some(SourceLocation {
        path: "/sapmnt/appl_sw/sysroot-sles15/usr/include/dirent.h",
        row: Some(189),
        column: Some(12)
      }),
    code_snippet: Some(" extern int __REDIRECT (readdir_r,\n
^~~~~~~~~~\"\n"),
    trace: []
    }
  ]
}
```

Code 32: Ausgabe von log-jitsu nach Testeingabe

3 Anhang: Screenshots der generierten HTML-Datei

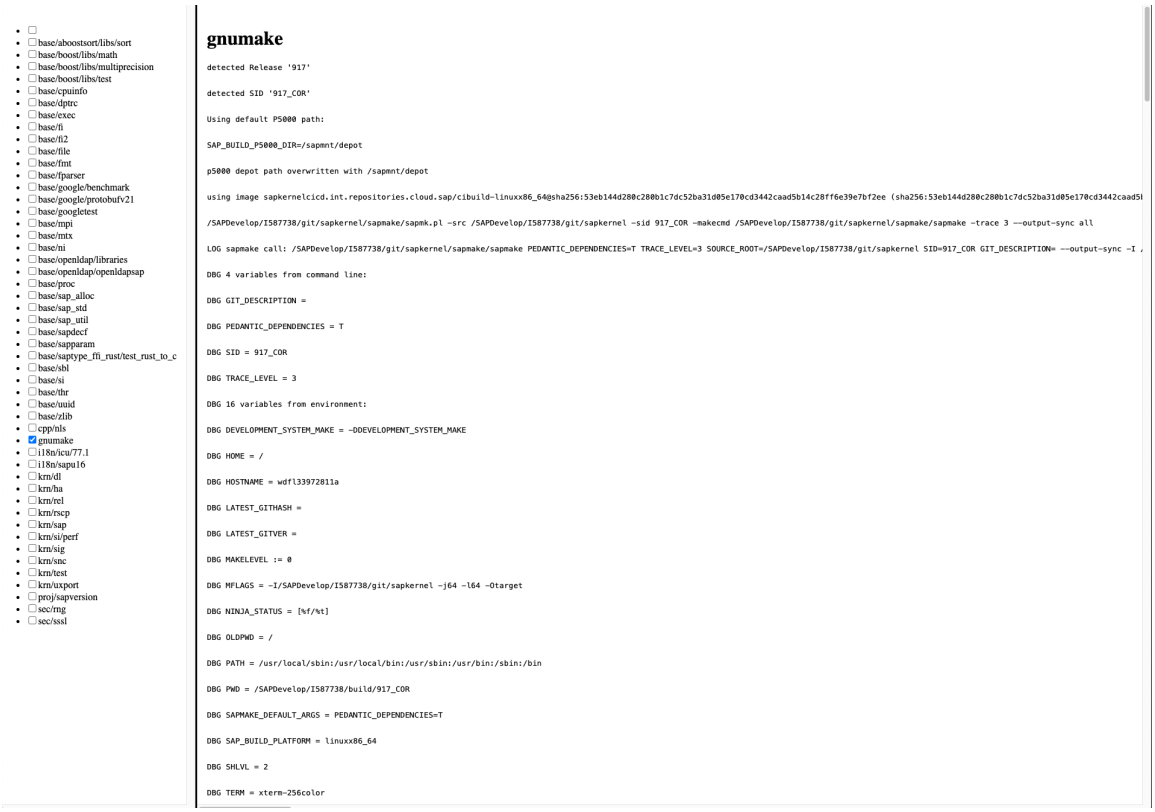

Abbildung 1: Vom Build-System generierter Output