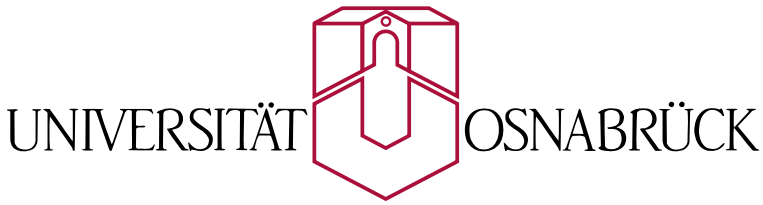


Institut für Informatik

Bachelorarbeit

Entwurf und Implementation eines Toolkits zur Analyse von Java-Quelltexten

Lukas Kalbertodt

10. März 2016

Erstgutachter:	Prof. Dr. Oliver Vornberger
Zweitgutachterin:	Prof. Dr. Elke Pulvermüller

Danksagungen

Ich möchte mich hiermit bei allen Personen bedanken, die mich bei der Erstellung dieser Arbeit unterstützt haben:

- Herrn Prof. Dr. Oliver Vornberger für seine Tätigkeit als Erstgutachter
- Frau Prof. Dr. Elke Pulvermüller für ihre Tätigkeit als Zweitgutachterin
- Herrn Nils Haldenwang für die Betreuung und Beratung
- Herrn Niko Matsakis für die intensive Arbeit an dem Parsergenerator `lalrpop` und die Beantwortung meiner Fragen zum besagten Parsergenerator.
- Meinen Eltern, Andrea und Seven Kalbertodt, sowie Elisa Brauße, für die großartige Unterstützung und hilfreiches Feedback zu dieser Ausarbeitung.

Zusammenfassung

In der vorliegenden Arbeit wird ein Toolkit zur Analyse von Java-Quelltexten und darauf aufbauend eine Applikation zur Assistenz bei Java-Programmierung entwickelt. Diese Applikation ist ein Buildtool, das seine Eingabe beispielhaft auf einige syntaktische, semantische oder stilistische Fehler prüft und diese an den Nutzer meldet. Dazu werden zunächst die nötigen Grundlagen der eingesetzten Methodik, des Compilerbaus und der verwendeten Programmiersprache *Rust* erläutert. Bei der anschließenden Entwicklung des Toolkits wird auf die Herausforderungen bei der syntaktischen Analyse nach den Regeln der Java-Grammatik eingegangen. Außerdem wird besonderer Wert auf eine leicht verständliche Präsentation von Analyseergebnissen und Fehlern gelegt, sodass der Programmierer optimal unterstützt werden kann. Die Erweiterung der Applikation um zusätzliche Features könnte Gegenstand weiterer Arbeiten sein.

Abstract

For this thesis, a toolkit has been developed that analyses source code written in Java as well as an application assisting the programmer while coding in Java. The application is a buildtool checking its input for some syntactic, semantic and stylistic errors which are then being reported to the user. To this end, the necessary essentials of deployed methodology, compiler construction and the utilised programming language *Rust* are elucidated in this thesis. In the subsequent part about developing the toolkit the thesis elaborates on challenges which occur during syntactic analysis performed according to the rules of the Java grammar. Special attention has been payed to the output of effortlessly comprehensible errors and results of made analysis such that the programmer's optimal support can be guaranteed. Extending the application by additional features can be taken as subject for further work.

Inhaltsverzeichnis

1 Einleitung

An der Universität Osnabrück wird in einem Großteil der Veranstaltungen, in denen programmiert wird, die Programmiersprache Java verwendet. Auch in der Einführungsveranstaltung „Informatik A: Algorithmen und Datenstrukturen“ wird den Programmieranfängern diese Sprache beigebracht. Bei der Formulierung von Java-Code kann es schnell zu Fehlern kommen, die entweder vom Java-Compiler gemeldet werden oder zur Laufzeit auffallen. Diese Fehler müssen dann durch Wiederholung der Arbeitsschritte „Testen“ und „Code verbessern“ aufwändig korrigiert werden. Dieser Prozess kann durch unterstützende Werkzeuge enorm beschleunigt und vereinfacht werden.

In dieser Arbeit wird ein Toolkit („`xswag`“) mit mehreren solcher Werkzeuge zur Analyse von Java-Quelltexten entworfen, auf dem mehrere Applikationen (z. B. „`jswag`“) aufsetzen, die beim Programmieren in Java helfen oder Java-Quelltexte auf andere Art auswerten.

1.1 Motivation

In vielen Informatik-Veranstaltungen an der Universität Osnabrück unterstützen Tutoren das Übungssystem, indem sie wöchentlich den Stoff des letzten Arbeitsblattes in einer kleinen Prüfung mit je zwei Studierenden abfragen. In „Informatik A“ helfen die Tutoren den angehenden Programmierern zusätzlich in der sogenannten Beratungszeit beim Lösen ihrer Übungsaufgaben. Dabei fragen die Studierenden oft nach der Bedeutung einer bestimmten Fehlermeldung des Compilers; die Fehlermeldungen des originalen `javac`-Compilers scheinen also eher zu verwirren als zu helfen. Ein passenderes Werkzeug würde den Programmieranfängern mehr Unterstützung bieten.

Doch auch den Tutoren kann durch passende Hilfsmittel eine Menge Arbeit abgenommen werden. Ein bekanntes Problem bei der Korrektur von Übungsaufgaben sind Lösungen, die von anderen Studierenden abgeschrieben oder aus dem letzten Jahr kopiert werden. Ein Tool, welches zwei Quelltexte anhand ihrer Java-Semantik auf Ähnlichkeit überprüft, kann bei der Erkennung solcher Plagiate enorm helfen. Hier ist ein normales, textbasiertes Vergleichstool unterlegen, da es zwei Dateien Zeile für Zeile vergleicht; es gibt aber Teile des Quellcodes, die für die Semantik eines Java-Programms völlig unerheblich sind – beispielsweise Kommentare oder Namen von Variablen. Zumeist unterscheiden sich abgeschriebene Lösungen nur durch eben diese Variablennamen und Kommentare von ihrem Original.

Ein weiterer Faktor, der die Korrektur für die Tutoren verlangsamt, ist die Formatierung und der Stil eines Programms. Idiomatisch geschriebene Java-Programme können von erfahrenen Programmierern viel schneller gelesen und verstanden werden als Programme, die sich nicht an Richtlinien, wie passende Namensgebung, halten [9, S. 608]. Hier könnte eine Applikation, die den Stil eines Java-Quelltextes überprüft, eingesetzt werden, um einen bestimmten Programmierstil von den Studierenden zu verlangen.

1.2 Struktur der Arbeit

Zunächst werden in Kapitel 2 die notwendigen Grundlagen der genutzten Technologien und Verfahren vermittelt. Dabei wird zunächst ein kleiner Einblick in die in dieser Arbeit verwendete Programmiersprache *Rust* gegeben und daraufhin die benötigten Methodiken aus dem Bereich Compilerbau erläutert. Anschließend wird in Kapitel 3 die detaillierte Umsetzung des Toolkits näher beschrieben, bevor im letzten Kapitel ein Fazit formuliert wird.

2 Grundlagen zu eingesetzten Technologien

Um in Java geschriebene Quelltexte analysieren zu können, ist es notwendig, diese in eine Form zu bringen, in der man leicht arbeiten kann – direkt auf Textebene zu arbeiten wäre deutlich aufwändiger und somit unpraktisch. Dieser als *Parsen* bezeichnete Prozess wandelt den Eingabetext – eine Folge von Zeichen – in einen sogenannten AST[1] um. Darauf aufbauend werden weitere Darstellungen des Quelltextes abgeleitet, die als Grundlage für diverse Algorithmen dienen.

Die „Compiler Tree API“ von Java ermöglicht es direkt, eine Java-Datei in eine AST-ähnliche Form zu konvertieren. Allerdings wurde diese Java-Funktionalität in der vorliegenden Arbeit bewusst nicht verwendet, sondern stattdessen ein eigener Parser in der Programmiersprache Rust implementiert. So kann die recht junge Sprache Rust im Kontext des Compilerbaus evaluiert und ein näherer Einblick in die Java-Syntax gewonnen werden.

In diesem Grundlagenkapitel wird Rust zunächst recht ausführlich vorgestellt, da in späteren Abschnitten der Arbeit in Rust formulierte Codeschnipsel verwendet werden, die eine grundlegende Kenntnis der Sprache voraussetzen. Nach der initialen Einführung in Rust werden die Grundlagen des Compiler- und Parserbaus näher erläutert.

2.1 Rust

Rust[2] ist eine von der Organisation Mozilla (bekannt z. B. durch den Browser *Firefox*) gesponserte, moderne, systemnahe und sichere Multiparadigmen-Programmiersprache. Sie wurde am 15. Mai 2015 in der ersten stabilen Version veröffentlicht – die initiale Ankündigung gab es bereits 2010. Rust ist in erster Linie ein Communityprojekt, welches vollständig Open Source und frei auf der Seite GitHub[3] entwickelt wird. Mozilla hilft lediglich, indem es einige Kernentwickler für ihre Tätigkeit bezahlt [1].

Ein auszeichnendes Merkmal an Rust ist die Kombination von Systemnähe (wie in der Programmiersprache C) und Speicher- und Threadsicherheit. Diese Sicherheitsgarantien besagen, dass ein normal geschriebenes Rustprogramm niemals:

[1] *Abstract Syntax Tree*, weitere Erklärung in Kapitel 2.2.2
[2] Offizielle Website: `https://www.rust-lang.org/`
[3] `https://github.com/rust-lang/rust`

- auf uninitialisierten oder ungültigen Speicher zugreift
- auf deallokierten Speicher zugreift (*use after free*)
- deallokierten Speicher nochmals freigibt (*double free*)
- Race Conditions auslöst

Kritische Sicherheitslücken, wie der „Heartbleed"-Bug in 2014[4], sind meist auf eine der genannten Fehlerquellen zurückzuführen. Programme, die in sicherem Rust geschrieben werden, weisen keine dieser Fehler auf und werden daher niemals diese Art von Sicherheitsmängel besitzen.

Anders als in Sprachen wie Python oder Java werden diese Garantien nicht (bzw. kaum) zur Laufzeit, sondern hauptsächlich zur Kompilierzeit sichergestellt – ein Overhead zur Laufzeit wäre für viele Anwendungsfälle von systemnahen Sprachen nicht akzeptabel. Auf die Umsetzung dieses Features wird in Kapitel 2.1.5 näher eingegangen.

Neben dem Sicherheitsaspekt hat Rust allerdings noch weitere Features zu bieten. Zum einen ist die Rust-Syntax sehr modern und erlaubt es, Algorithmen kompakt zu formulieren. Zum anderen weist Rust, u. a. durch *zero-cost abstractions*, eine hervorragende Ausführungsgeschwindigkeit auf, die auf demselben Level wie die von C++ und C steht.

2.1.1 Syntax

Rust verwendet, wie viele andere Sprachen auch, eine C-ähnliche Syntax: Blöcke werden mit geschweiften Klammern umschlossen und es gibt Elemente, wie `if` oder `while`, die den Kontrollfluss beeinflussen können. Allerdings unterscheidet sich die Syntax von Rust durch einige wichtige Eigenschaften von der C-Syntax, so ist z. B. in Rust fast alles eine *Expression*, während in anderen Sprachen strikt zwischen *Expression* und *Statement* unterschieden wird. Das folgende Programm demonstriert einige Elemente der Syntax:

```
1  fn main() {
2      let x = 27;
3      for i in 1 .. x+1 {
4          if x % i == 0 {
5              println!("Teiler {}!", i);
6          }
7      }
8  }
```

Ausgabe des Programms:

```
Teiler 1!
Teiler 3!
Teiler 9!
Teiler 27!
```

[4] weitere Details unter `http://heartbleed.com/`

In Zeile 1 wird die Funktion `main` mit dem Keyword `fn` deklariert; diese Funktion stellt, wie bei vielen anderen Programmiersprachen, den Programmeinstiegspunkt dar. Darin wird dann zunächst ein *Variable Binding* mit dem Namen `x` und einem bestimmten Wert erzeugt. Hier kommt schon das *Type Inference*-Feature der Sprache zum Tragen: Rust ist zwar statisch typisiert, jedoch wurde an keiner Stelle dieses kurzen Codeschnipsels ein Typ annotiert – dies wird in den meisten Fällen vom Compiler erledigt.

In Zeile 3 beginnt eine `for`-Schleife, die über das angegebene Intervall iteriert – in anderen Sprachen ist dieser Schleifentyp oft als `foreach` bekannt. Den Rumpf dieser Schleife bildet ein `if`-Ausdruck, der überprüft, ob `i` ein Teiler von `x` ist. In Rust sind die runden Klammern um die Bedingung nicht nötig, dafür sind die geschweiften Klammern um den Block verpflichtend. `println!` in Zeile 5 ist ein Makro zur Ausgabe auf dem Terminal. Makros sind mächtiger als Funktionen und können bereits zur Kompilierzeit gewisse Überprüfungen ausführen – dies wird hier für eine typsichere Stringinterpolation genutzt.

Im folgenden Beispiel werden weitere Teile der Syntax und insbesondere das Feature „Everything is an Expression“ gezeigt:

```
1  fn collatz(mut n: u32) -> u32 {
2      let mut iterations = 0;
3      while n > 1 {
4          n = if n % 2 == 0 { n/2 } else { n*3 + 1};
5          iterations += 1;
6      }
7      iterations
8  }
```

In Zeile 1 wird eine Funktion `collatz` deklariert, welche einen Parameter `n` vom Typ `u32` entgegennimmt und einen Wert vom selben Typ zurückgibt; `u32` ist ein **u**nsigned Integer mit 32 Bits. Sowohl das Variable Binding in Zeile 2 als auch die Parameterdeklaration `n` (welche auch ein Variable Binding ist) enthalten das Keyword `mut`, welches das Binding *mutable* (veränderbar) macht. In Rust sind Variable Bindings standardmäßig immutable – also unveränderlich.

In Zeile 4 wird `n` ein neuer Wert zugewiesen, allerdings mit einem `if-else` Konstrukt. Da der Zuweisungsoperator „`=`“ auf seiner rechten Seite lediglich eine Expression fordert und in Rust quasi alles – also auch ein `if-else` Konstrukt – Expressions sind, tut diese Zeile exakt, was man erwartet. Auch in der letzten Zeile der Funktion wird von diesem Feature Gebrauch gemacht: Eine Funktion gibt automatisch den Wert der letzten

Expression (hier `iterations`) zurück; das `return` Keyword ist nur für sogenannte Early Returns nötig.

2.1.2 Typen und der „Match“-Ausdruck

Rust ist keine strikt objektorientierte Sprache, erlaubt es aber, sie teilweise objektorientiert zu nutzen. Anders als bei Java, gibt es nicht nur Klassen, sondern grundsätzlich zwei Arten von eigenen Typen: Structs und Enums. Structs sind multiplikative Typen, die man aus C kennt und die vergleichbar mit Java-Klassen sind. Im Folgenden wird ein Struct `Point` erstellt, welches zwei Datenfelder besitzt, auf die öffentlich zugegriffen werden kann (`pub` heißt „public“):

```
struct Point {
    pub x: f64,     // f64 = floating point type with 64 bits
    pub y: f64,
}
```

Enums hingegen sind additive Typen und in weniger mächtiger Form aus anderen Sprachen bekannt. Variablen eines Enum-Typen können eine von mehreren Möglichkeiten – in Rust „Variants“ genannt – annehmen, die bei der Definition des Enums aufgelistet werden:

```
enum Ordering {
    Less,
    Equal,
    Greater,
}
```

Dieses Beispiel ist auch mit Java- oder C-Enums umsetzbar, da keine weiteren Daten an den Variants hängen. Dies ist in Rust aber durchaus möglich:

```
enum CssColor {
    None,
    HexCode(String),
    Rgba {
        r: u8, g: u8, b: u8, a: f32,
    },
}
```

Das Enum `CssColor` speichert eine Farbe, wie sie in CSS[5] angeben werden kann, nämlich als hexadezimaler Code, im RGBA-Format oder einfach als wörtlich `none` (andere CSS-Farbformate wurden in diesem Beispiel ignoriert). Hier gehört zu dem Variant `HexCode` ein String, um den originalen hexadezimalen Code zu speichern; `Rgba` speichert sich seine Komponenten einzeln ab. Lediglich `None` enthält keine weiteren Daten. Im folgenden Code kann man sehen, wie ein solches Enum benutzt wird.

```
let color = CssColor::Rgba { r: 100, g: 50, b: 255, a: 0.4 };
match color {
    CssColor::None => println!("Keine Farbe angegeben..."),
    CssColor::HexCode(code) => {
        println!("HexCode: {}", code);
    },
    CssColor::Rgba { a: 0.0, .. } => {
        println!("Komplett durchsichtig!");
    },
    CssColor::Rgba { r, g, b, a } => {
        println!("RGBA Format: {} {} {} {}", r, g, b, a);
    },
}
```

In der ersten Zeile wird eine neue Variable des Typs `CssColor` erstellt und mit einem Wert des Variants `Rgba` gefüllt. Es folgt ein `match` Block – eine viel mächtigere Version des `switch`-Statement, welches in anderen Sprachen zum Einsatz kommt. Ein `match` Block besitzt mehrere sogenannte „Arme“, die mit einem Pattern beginnen, nach dem Fat-Comma-Operator „`=>`“ mit einem Codeblock enden und untereinander durch Kommata getrennt werden. Im Pattern wird die Technik *Pattern Matching* betrieben, wodurch neue Variable Bindings erstellt werden können. Im zweiten Matcharm wird z. B. ein Variable Binding mit dem Namen `code` erstellt, welches den String des `HexCode` Variants an sich bindet. Im dritten Matcharm wird ein Teil der vorhandenen Variablen ignoriert und nicht an einen Namen gebunden; lediglich `a` wird mit der Konstanten `0.0` gleichgesetzt. Das führt dazu, dass der dritte Arm nur betreten wird, wenn der Alphawert Null ist.

Die Begriffe „multiplikativ“ und „additiv“ lassen sich durch eine formale Betrachtung des Typsystems erklären. Typen können als Mengen angesehen werden; eine Variable des Typs hat ein Element dieser Menge als Wert. So ist z. B. `bool` $= \{\texttt{true}, \texttt{false}\}$ und `u8` $= \{0, 1, \ldots, 254, 255\}$. Wenn wir zwei vorhandene Typen A und B mithilfe eines multiplikativen Typen verknüpfen, ist unser Ergebnistyp $C = A \times B$ und dessen

[5] Cascading Style Sheets, Stylesheet-Sprache hauptsächlich fürs Web

Mächtigkeit $|C| = |A| \cdot |B|$. Wenn wir sie jedoch mit einem additiven Typen verknüpfen, erhalten wir $C = A \cup B$ und $|C| = |A| + |B|$. Betrachten wir als Beispiel die folgenden zwei Typen:

```
enum SumType { A(bool), B(Ordering) }
struct ProductType { a: bool, b: Ordering }
```

Variablen dieser Typen können nun die folgenden Werte annehmen:

`SumType` (Fünf Werte)	`ProductType` (Sechs Werte)

`A(true)`
`A(false)`
`B(Less)`
`B(Equal)`
`B(Greater)`
	`{a: false, b: Less}`
	`{a: false, b: Equal}`
	`{a: false, b: Greater}`
	`{a: true,  b: Less}`
	`{a: true,  b: Equal}`
	`{a: true,  b: Greater}`

Additive und multiplikative Typen fallen beide unter den Begriff der *algebraischen Datentypen*. Während man multiplikative Typen in vielen Sprachen findet, gibt es richtige additive Typen nur in wenigen populären Sprachen.

Neben Structs und Enums, gibt es in Rust noch ein paar weitere Verbundtypen: Tupel werden durch `(i32, bool, char)` ausgedrückt und sind anonyme Produkttypen. Tupel-Structs (mit `struct Point(i32, i32)` definiert) sind eine Mischung aus Tupeln und Structs; sie sind selber benannt, ihre Datenfelder sind aber anonym. Zuletzt sind Arrays mit der Syntax `[i16; 10]` homogene, anonyme, multiplikative Typen.

Für alle Typen in Rust, die man selber definiert, kann man auch Methoden mit der `impl` Syntax definieren:

```
impl Point {
    pub fn null() -> Point {
        Point { x: 0.0, y: 0.0 }
    }
    pub fn length(&self) -> f64 {
        (self.x * self.x + self.y * self.y).sqrt()
    }
    pub fn normalize(&mut self) {
        let len = self.length();
        self.x /= len;
        self.y /= len;
    }
}
```

```
14  fn main() {
15      let p1 = Point::null();
16      let mut p2 = Point { x: 3.0, y: 8.0 };
17      println!("length: {}", p2.length());    //  8.544003...
18      p2.normalize();
19      assert!(p2.length() == 1.0);
20  }
```

Wie beispielsweise in Python, gibt man in Rust das Empfängerobjekt explizit als Parameter `self` an. Die erste Methode enthält kein `self` als Parameter und ist daher eine *Associated Function*, was in Java als *statische Methode* bezeichnet wird. Die `null`-Funktion (`null` ist in Rust *kein* Keyword) hat als Rückgabetypen `Point` selber, konstruiert also den Typ, mit dem sie assoziiert ist. Dies ist ein typisches Pattern in Rust, da es keine *Konstruktoren* mit spezieller Syntax gibt. Man erstellt Instanzen von Typen immer mit der Struct-Syntax (wie in Zeile 3 und 16 zu sehen). Falls man jedoch private Datenfelder hat oder spezielle Fälle abdecken möchte, kann man sich ohne Probleme assoziierte Funktionen erstellen, die diese Arbeit übernehmen. Hier wird mit der `null` Funktion ein Nullvektor erstellt.

Die anderen beiden Funktionen sind Methoden, die ein Empfängerobjekt als Parameter bekommen, auf dem operiert werden kann. Diese Methoden können, wie in Java, mit der Punktnotation an Instanzen des Typs aufgerufen werden.

2.1.3 Wichtige Typen der Standardbibliothek

Zwei Typen aus der Rust-Standardbibliothek sind, insbesondere für die Fehlerbehandlung in Rust, enorm wichtig: `Option` und `Result`. Ersteres signalisiert die mögliche Abwesenheit eines sinnvollen Wertes. Hierzu muss man wissen, dass es in Rust keine `null`-Referenz, wie in Java, gibt: Eine Variable des Typs `String` enthält auch immer eine gültige Instanz dieses Typs, somit kann man immer unbedenklich Methoden des `String` Typs an einer solchen Variable aufrufen. Manchmal ist es aber nötig, zu signalisieren, dass ein Typ nur optional vorhanden ist. Dies ist alleine durch das Typsystem möglich und benötigt keine Änderung der Sprache an sich. `Option` ist definiert als:

```
1  enum Option<T> {
2      None,
3      Some(T),
4  }
```

Das `<T>` bedeutet, dass es sich um eine generische Typdefinition handelt, die mit einem beliebigen Typ, der hier als `T` bezeichnet wird, arbeiten kann. `Option<String>` bezeichnet also ein Objekt, was entweder `None` oder `Some(s)` sein kann, wobei `s` ein beliebiger, gültiger `String` ist. Dieser optionale Typ ist z. B. sinnvoll für die `find` Methode einer Hashmap: entweder ein Objekt wurde gefunden oder nicht.

Der Typ `Result` besagt nicht nur einfach „*Wert nicht vorhanden*“, sondern speichert sich weitere Informationen über den Fehlerzustand. Die vollständige Definition ist:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Der letzte, zum Codeverständnis wichtige Typ ist `Vec<T>`. Dieser Standard Containertyp verwaltet intern ein Array und entspricht ziemlich genau dem Typ `std::vector` aus C++ sowie der `java.util.ArrayList` aus Java.

2.1.4 Traits

Mit dem neu gewonnenen Wissen über generische Typen könnte man auf die Idee kommen, den `Point` Typ von oben folgendermaßen neu zu definieren, um den Typ der Komponenten nicht auf `f64` zu beschränken:

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    pub fn sum(self) -> T {
        // error: binary operation `+` cannot be applied to type `T`
        self.x + self.y
    }
}
```

Hier zeigt sich wieder einmal, dass Rust ein starkes Typsystem hat: Der Compiler kann nicht wissen, welche Methoden der Typ `T` hat, und somit auch nicht, ob man zwei Variablen vom Typ `T` addieren kann. Dies ist ungefähr vergleichbar mit dem Aufruf von

`compareTo` an einem Java `Object`: Auch hier meldet der Compiler einen Fehler, da er nicht weiß, ob man an dem `Object` wirklich die geforderte Methode aufrufen kann.

Um in Rust bestimmte Funktionalität zu kennzeichnen, nutzt man sogenannte Traits, die in etwa mit Java-Interfaces vergleichbar sind. So kann man mit *Trait Bounds* generische Typparameter auf die Typen einschränken, die ein bestimmtes Trait implementiert haben. In dem Beispiel soll eine Variable vom Typ `T` mit einer Variable desselben Typs addiert werden können; dies wird durch das `std::ops::Add` Trait der Standardbibliothek sichergestellt:

```
1  impl<T: std::ops::Add> Point<T> {
2      pub fn sum(self) -> T::Output {
3          self.x + self.y
4      }
5  }
```

Neben dem *Trait Bound* in der ersten Zeile hat sich auch noch der Rückgabewert der Funktion geändert: Das Ergebnis der Addition muss nicht zwingend wieder vom Typ `T` sein. Daher greifen wir durch `T::Output` auf einen mit dem Trait `Add` assoziierten Typ zu.

2.1.5 Sicherheit durch den Borrowchecker

Rust hält sein Versprechen von Speicher- und Threadsicherheit mit Hilfe des sogenannten Borrowcheckers ein. Dieser arbeitet nach dem „Multi-Read XOR Single-Write“ Prinzip, stellt also zur Kompilierzeit sicher, dass es entweder maximal einen schreibenden Zugriff oder beliebig viele lesende Zugriffe auf ein Objekt gibt. Dies ist durch drei von Rusts Kernkonzepten möglich.

2.1.5.1 Das „Ownership“ Konzept

Jeder Variable in Rust ist ein eindeutiger Besitzer, z. B. eine Funktion oder eine andere Variable, zugeordnet. Der Besitzer ist für die Ressourcenverwaltung zuständig und gibt somit alle Ressourcen frei, sobald sein Scope[6] endet. Außerdem leben alle Variablen in Rust – unabhängig vom Typ – zuerst einmal direkt auf dem Stack und werden nicht, wie z. B. in Java, automatisch im Heap allokiert. Bei einer Zuweisung oder der Übergabe an

[6] Sichtbarkeitsbereich

eine Funktion wird daher nicht nur ein Zeiger kopiert (*Reference Semantics*), sondern die Variable mitsamt ihres Besitzers verschoben (*Move Semantics*). Dies führt zunächst zu unerwarteten Fehlern:

```
fn print_twice(s: String) {
    println!("{} and {}", s, s);
}

let string_a = "move it!".to_string();
println!("{}", string_a);   // Ok

let string_b = string_a;
println!("{}", string_b);   // Ok
println!("{}", string_a);   // "Error: use of moved value!"

print_twice(string_b);
println!("{}", string_b);   // "Error: use of moved value!"
```

Dieses Programm kann nicht erfolgreich kompiliert werden, denn der Compiler gibt die zwei oben genannten Fehlermeldungen aus. Hier ist wieder wichtig zu bemerken, dass der Compiler die Besitzer von Variablen zur Kompilierzeit verfolgt und keinerlei Laufzeitfunktionalität notwendig ist. Um die Fehler im genannten Beispiel zu verbessern, ist es möglich, die Variablen mit der `clone()` Methode komplett zu kopieren. Dies ist aber natürlich in vielen Fällen nicht gewünscht – man bedient sich daher eines anderen Konzepts, nämlich dem des „Borrowing“.

Der Vollständigkeit halber ist zu erwähnen, dass es sogenannte `Copy`-Typen gibt, die beim Zuweisen automatisch mit `clone()` kopiert werden, weil das Kopieren sehr kostengünstig ist. Zu diesen Typen zählen alle primitiven Datentypen – deshalb wurde im Beispiel oben auch der Typ `String` benutzt, welcher *kein* `Copy`-Typ ist.

2.1.5.2 „Borrowing“ und Lifetimes

Wie in vielen anderen Sprachen gibt es in Rust-Referenzen, auch „Borrows“ genannt, welche durch den `&` Operator ausgedrückt werden. Auf Maschinencodeebene sind diese Referenzen nur normale Zeiger; auf höherem Level haben sie jedoch eine wichtige Bedeutung. Es gibt sowohl unveränderliche (`&`) als auch veränderbare (`&mut`) Referenzen. Hier wird der Borrowchecker aktiv und stellt die oben genannte „Multi-Read XOR Single-Write“-Einschränkung sicher.

Die Funktion `print_twice` aus dem Beispiel oben möchte den String nur kurzzeitig

benutzen, also ausleihen, und nicht gleich die Besitzerschaft übernehmen. Korrekt[7] sähe die Funktion folgendermaßen aus:

```
1 fn print_twice(s: &String) { ... }
```

Weiterhin assoziiert der Compiler mit jeder Referenz eine sogenannte *Lifetime*, die ebenfalls zur Kompilierzeit verfolgt wird. So kann der Compiler verhindern, dass auf ungültige Referenzen zugegriffen wird.

Diese simplen, vom Compiler erzwungenen Einschränkungen verhindern viele Probleme, die normalerweise erst zur Laufzeit erkannt werden. Darunter fallen die schon genannten Probleme der Speichersicherheit, aber auch das *Iterator Invalidation*-Problem und Multithreading-Probleme, wie *Data Races*.

2.2 Compilerbau und Parsen

Compiler mit ihrer Aufgabe, einen in einer bestimmten Grammatik formulierten Quellcode in eine Zielsprache umzuwandeln, sind extrem komplizierte Programme – mehrere Abstraktionsstufen sind notwendig, um dieser Komplexität Herr zu werden.

Die Funktionalität eines Compilers wird meist in zwei Bereiche geteilt: Das *Frontend*, welches den Eingabetext analysiert, auf Fehler untersucht und in eine geeignete Zwischendarstellung bringt, und das *Backend*, das aus dieser Zwischendarstellung den Zielcode generiert und dabei optional Optimierungen vornimmt [2, S. 4]. Für diese Arbeit ist lediglich das *Frontend* von Interesse – Codegenerierung, wie im *Backend*, wird in dem entwickelten Toolkit nicht vorgenommen. In den folgenden Unterkapiteln wird näher auf einzelne, wichtige Teile des *Frontends* eingegangen.

2.2.1 Lexing/Tokenization

Der eigentliche Parsing-Schritt wird oft in zwei getrennte Schritte unterteilt: Das Lexing (auch Tokenization oder Scanning genannt) und das Parsing selber [2, S. 5]. Im ersten Schritt unterteilt der sogenannte Lexer (bzw. Tokenizer/Scanner) den Eingabetext in eine Reihe von *Token* (auch *Lexem* genannt), die einem oder mehreren Zeichen im Text entsprechen. Oft hat jeder Operator seinen eigenen Token; Worte und Literale

[7] Korrekt, aber nicht idiomatisch – dient lediglich zur Veranschaulichung des Konzepts

Abbildung 1: Beispielhafte Aufteilung von Java-Code in Token

werden ebenfalls in je einem Token zusammengefasst. In Abbildung 1 sieht man einen Eingabetext, der in Token unterteilt wurde (Whitespace-Token wurden hierbei nicht eingezeichnet).

Der resultierende Tokenstream kann viel einfacher vom Parser verarbeitet werden als der rohe Eingabetext direkt. Hier findet man bereits die erste Stufe von Abstraktion, die zwar insgesamt zu mehr Schritten führt, aber die einzelnen Schritte deutlich einfacher macht.

Die Regeln zum Aufsplitten in Token – *Lexical Grammar* genannt – sind typischerweise regulär, können also z. B. mit einem endlichen Automaten oder einer Regex modelliert werden [2, S. 187]. Die Laufzeit des Lexers liegt somit in $\mathcal{O}(n)$, wobei n die Länge des Eingabetextes ist.

2.2.2 Parsing

Der Parser verarbeitet den durch den Lexer produzierten Tokenstream mit den Regeln der syntaktischen Grammatik weiter und speichert das Ergebnis in einer geeigneten Darstellung. Diese syntaktische Grammatik ist bei den meisten Programmiersprachen nicht regulär: Reguläre Sprachen können nämlich keine balancierten Klammerausdrücke darstellen. Dazu müsste man „zählen“ können, wie viele öffnende Klammern man schon verarbeitet hat; dies ist mit einem endlichen Automaten nicht möglich.

Die Syntax der meisten Sprachen liegt jedoch nur in der nächsthöheren Stufe der Chomsky Hierarchie – den kontextfreien Grammatiken. Die Laufzeit für das Wortproblem[8] für diese Klasse von Grammatiken ist im schlechtesten Fall $\mathcal{O}(n^3)$; für viele Compiler und Parser ist diese kubische Laufzeit zu langsam. Daher bemühen sich Programmiersprachenentwickler meist, eine *deterministisch* kontextfreie syntaktische Grammatik zu entwerfen. Die so erzeugbaren Sprachen sind eine Teilmenge der nicht-deterministisch erzeugbaren Sprachen; das Wortproblem lässt sich aber in einer Laufzeit von nur $\mathcal{O}(n)$ entscheiden.

[8] Entscheiden, ob ein bestimmter String s in der von der Grammatik definierten Sprache liegt

Beim Lösen des Wortproblems kann für erwähnte Grammatiken zusätzlich ein Ableitungsbaum erzeugt werden, der für Compiler wichtiger ist als nur eine boolesche Aussage darüber, ob die Eingabe gültig ist. Ein solcher Ableitungsbaum wird meist *Parsetree* oder auch CST (*Concrete Syntax Tree*) genannt. Jeder Knoten innerhalb dieses Baumes entspricht einem Nichtterminal-Symbol (*non-terminal*) der Grammatik und seine Kinder entsprechen den Symbolen auf der rechten Seite der Ableitungsregel. Ein wichtiges Problem stellen mehrdeutige Grammatiken dar, welche erlauben, zu einer bestimmten Eingabe mehrere unterschiedliche Parsetrees abzuleiten. Dies ist bei Programmiersprachen unerwünscht, da unterschiedliche Parsetrees oft eine verschiedene semantische Bedeutung haben. Ein Beispiel dafür ist die folgende Grammatik, die Addition und Subtraktion von binären Ziffern darstellt:

$$\begin{aligned}
\text{Expr} &\xrightarrow{a} \langle\text{Expr}\rangle + \langle\text{Expr}\rangle \\
\text{Expr} &\xrightarrow{b} \langle\text{Expr}\rangle - \langle\text{Expr}\rangle \\
\text{Expr} &\xrightarrow{c} 0 \\
\text{Expr} &\xrightarrow{d} 1
\end{aligned}$$

Die Eingabe `0 - 1 + 1` kann auf zwei unterschiedliche Arten aus der Grammatik erzeugt werden. Die entsprechenden Parsetrees sind in Abbildung 2 dargestellt.

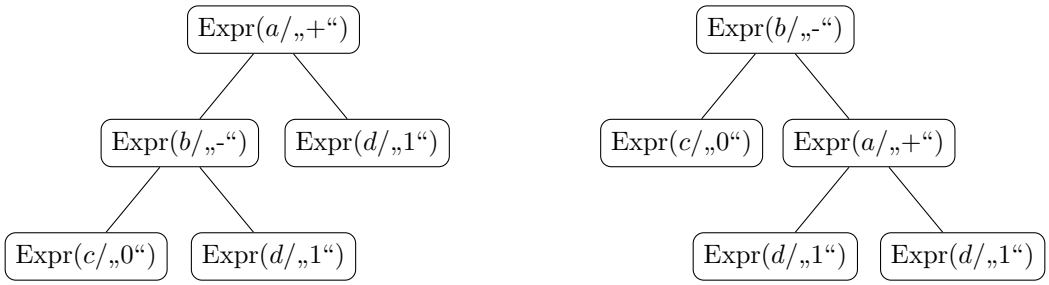


Abbildung 2: Mögliche Parsetrees

Um solche Mehrdeutigkeiten zu eliminieren, muss die Grammatik anders formuliert werden, z. B. so:

$$\begin{aligned}
\text{Expr} &\to \langle\text{Expr}\rangle + \langle\text{Number}\rangle \\
\text{Expr} &\to \langle\text{Expr}\rangle - \langle\text{Number}\rangle \\
\text{Number} &\to 0 \\
\text{Number} &\to 1
\end{aligned}$$

Nach dieser Grammatik ist der rechte Parsetree aus Abbildung 2 nicht gültig und somit ist die Grammatik eindeutig. In Kapitel 3.3.2 werden Probleme dieser Art in der Java-Grammatik näher besprochen.

Die Parser der meisten Compiler produzieren jedoch keinen Parsetree, sondern eine abstraktere Darstellung, die *Abstract Syntax Tree* (AST) genannt wird. Dieser kann zwar auch in einem weiteren Schritt aus dem CST erstellt werden, allerdings ist es üblich, ihn direkt während des Parsens zu erstellen. Oft werden zwei unterschiedliche Parsetrees mit derselben semantischen Bedeutung durch denselben AST dargestellt.

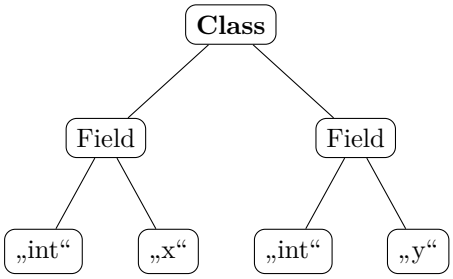
```
class Test {
    int x, y;
}
```

```
class Test {
    int x;
    int y;
}
```

Diese beiden Java-Codes resultieren in unterschiedlichen Parsetrees, könnten aber durchaus durch denselben AST dargestellt werden, da sie semantisch äquivalent sind. Der Parser könnte diesen AST wie in folgender Visualisierung darstellen – der Rust-Code zeigt eine mögliche Implementierung.

```
struct Class {
    fields: Vec<Field>,
}

struct Field {
    type_: String,
    name: String,
}
```



2.2.3 Parsergeneratoren

Zur Umwandlung eines Tokenstreams in einen AST bieten sich mehrere Möglichkeiten an. Natürlich kann man einen Parser komplett manuell implementieren; der Rust-Compiler besitzt z. B. einen solchen handgeschriebenen Parser. Als Alternative gibt es eine große Auswahl an Parsergeneratoren, die eine Beschreibung der Grammatik in den Quellcode eines entsprechenden Parsers umwandeln. Ein bekannter Vertreter ist *Yacc*, welcher den resultierenden Parser in Form von C-Quellcode generiert. Man unterscheidet strikt zwischen den zwei separaten Schritten der Parser-Generierung und des Parsens.

Ein Hauptproblem beim Design eines Parsergenerators ist der Umgang mit mehrdeutigen Grammatiken. Das Ergebnis des Parsingvorgangs soll in den meisten Fällen nur ein einzelner Parsetree bzw. AST sein; entsprechend muss auf die eine oder andere Art mit Mehrdeutigkeit umgegangen werden. Leider ist die Fragestellung, ob eine (deterministische) kontextfreie Grammatik mehrdeutig ist, unentscheidbar [3, S. 202]. Somit kann man die Lösungsansätze in zwei Kategorien teilen:

- Der Parser muss während des Parsens mit Mehrdeutigkeit umgehen. Natürlich könnte man in einem solchen Fall den Parsevorgang abbrechen oder einen zufälligen gültigen Parsetree auswählen; sinnvoller ist oft jedoch, alle möglichen Parsetrees zurückzugeben, aus denen der Verfasser der Grammatik den richtigen aussuchen muss. Dieser Ansatz wird z. B. von GLR-Parsern (*Generalized LR*) verfolgt.
- Zur Zeit der Parsergenerierung versucht der Generator zu beweisen, dass die Grammatik eindeutig ist. Wie oben bereits erwähnt, ist dies nicht für *alle* kontextfreien Sprachen möglich – es wird lediglich versucht, die Eigenschaft für eine möglichst große Untermenge von Sprachen zu beweisen. Wichtige Vertreter dieses Lösungsansatzes sind LL(k)-, LR(k)- und LALR-Parser.

Für spätere Kapitel sind besonders LR(k)- und LALR-Parser von Bedeutung, daher werden andere Techniken hier nicht im Detail behandelt. Der **L**eft to Right **R**ightmost Derivation (LR) Parser wurde schon 1965 von Donald Knuth vorgestellt [6] und stellt eine Grundlage für weitere Parser dar. Wie das „L“ im Namen beschreibt, läuft ein solcher Parser von links nach rechts durch die Eingabe. Dabei verwaltet er einen Stack der zuletzt gelesenen Symbole (terminale sowie nicht-terminale) und versucht in jedem Schritt diese Symbole einer Regel der Grammatik zuzuordnen. Diese Technik nennt sich auch *Bottom-Up-Parsing*, da die Blätter des resultierenden Parsetrees zuerst erzeugt werden.

Das k in den Namen der Parser steht für den Lookahead, also die maximale Anzahl von zukünftigen Symbolen, die der Parser betrachten kann, bevor er eine Entscheidung bezüglich des jetzigen Symbols treffen muss. Im Fall von einem LR(1)-Parser bedeutet dies, dass die Entscheidung darüber, eine Regel anzuwenden, nur aufgrund des Stackinhaltes und eines zusätzlichen Symbols getroffen werden muss. Folgende Beispielgrammatik beschreibt vereinfacht die `import`-Deklaration in Java:

$$
\begin{aligned}
\text{ImportDecl} &\xrightarrow{a} \text{import}\ \ \langle \mathit{Path} \rangle\ \ ; \\
\text{ImportDecl} &\xrightarrow{b} \text{import}\ \ \langle \mathit{Path} \rangle\ \ .\ *\ ; \\
\text{Path} &\xrightarrow{c} \text{Word} \\
\text{Path} &\xrightarrow{d} \text{Word}\ .\ \langle \mathit{Path} \rangle
\end{aligned}
$$

Das in dieser Grammatik verwendete Terminal-Symbol „Word“ ist ein vom Lexer generierter Token, der (hier vereinfacht) aus mehreren Buchstaben besteht. Zulässige `import`-Deklarationen sind z. B.:

- Beispiel A: `import foo.bar;`
- Beispiel B: `import foo.*;`

Die oben beschriebene Grammatik ist zwar eindeutig, allerdings keine gültige LR(1)-Grammatik. Hier tritt ein sogenannter Shift-Reduce-Conflict auf: Nachdem der Parser die ersten beiden Token aus den Beispielen gelesen hat, enthält der Stack nun [`import`, `Word`], wobei in dieser Darstellung „links“ dem obersten Element des Stacks entspricht; der Lookahead-Token ist `.`. In diesem Schritt kann der Parser nicht weitermachen, da er zwei Möglichkeiten hat:

- Er könnte den Lookahead-Token, in Erwartung ihn später verwenden zu können, auf den Stack pushen („Shift“). Dies wäre die richtige Entscheidung für Beispiel A, da der Lookahead-Token mit den nächsten beiden Symbolen einen $\langle \mathit{Path} \rangle$ bildet. Problematisch wird es bei Beispiel B: Angenommen alle weiteren Symbole werden ebenfalls auf den Stack gepushed, so ist dessen Inhalt am Ende [`import`, `Word`, `.`, `*`, `;`]. Dieser Stackinhalt wird aber durch keine Regel von $\langle \mathit{ImportDecl} \rangle$ abgedeckt; die Regeln a und b erwarten beide nach dem `import` Token einen $\langle \mathit{Path} \rangle$ und kein `Word`.
- Er könnte den auf dem Stack sitzenden Token mit der Regel d zu einem $\langle \mathit{Path} \rangle$ reduzieren („Reduce“). Dies wäre in Beispiel B das richtige Vorgehen, da der resultierende Stackinhalt [`import`, $\langle \mathit{Path} \rangle$, `.`, `*`, `;`] mit der Regel b zu einem $\langle \mathit{ImportDecl} \rangle$ weiter reduzierbar ist. Allerdings führt diese Entscheidung wiederum in Beispiel A zu Problemen.

Man kann in diesem Fall eine äquivalente Grammatik formulieren, die auch eine gültige LR(1)-Grammatik ist – z. B. indem man für die Definition von $\langle \mathit{Path} \rangle$ Left- statt Right-Recursion benutzt.

Die andere Parservariante, LALR, ist zwar weniger mächtig als LR(1)-Parser, kann dafür aber deutlich effizienter implementiert werden. In der Praxis gibt es wenige Sprachen, die LR(1), aber nicht LALR sind. LL(1)-Parser sind ebenfalls weniger mächtig als LR(1)-Parser, allerdings fällt dies in der Praxis stärker ins Gewicht. LL-Parser müssen sich nämlich für die zu parsende Regel der Grammatik entscheiden, nachdem sie die k Lookahead Token ausgewertet hat, also bevor sie die restlichen Symbole der Regel gesehen haben. Java ist aufgrund seiner zahlreichen Modifikatoren nicht geeignet, um mit einem LL-Parser mit kleinem k geparsed zu werden. Angenommen der Parser kann innerhalb einer Klassendefinition folgende vier Lookahead Symbole betrachten:

```
1  public static int foo
```

Mit diesen Informationen allein kann nicht entschieden werden, ob es sich beispielsweise um eine Methode (`public static int foo() {}`) oder ein Feld einer Klasse (`public static int foo;`) handelt. Daran sieht man die unterschiedlichen Vor- und Nachteile der zuvor genannten Typen von Parsergeneratoren.

2.2.4 Analyse und Weiterverarbeitung

Aufbauend auf dem AST prüfen Compiler nun die semantische Korrektheit der Eingabe. Dies beinhaltet beispielsweise *Name Resolution*, *Type Checking* und *Accessibility Analysis*. Um die einzelnen Analyseschritte zu vereinfachen, wird der Eingabetext oft in weiteren Formen dargestellt. Einige wichtige Darstellungen sind z. B.:

- **HIR** (High level Intermediate Representation):
  Der AST ist für einige Aufgaben immer noch zu nah an die konkrete Syntax gekoppelt – daher wird mit der HIR eine abstraktere Schnittstelle bereitgestellt.
- **CFG** (Control Flow Graph):
  Der CFG repräsentiert den Programmfluss innerhalb der Eingabe. Da dieser Fluss bekanntlich durch Schleifen und andere Konstrukte Zyklen enthalten kann, reicht hier ein Baum zur Darstellung nicht aus. Auf dem CFG sind besonders Erreichbarkeitsanalysen einfach auszuführen.
- **MIR** (Mid level Intermediate Representation):
  In dieser Darstellung wurden die meisten High-Level Konstrukte der analysierten Sprache in eine primitivere Darstellung gebracht, sodass die MIR oft ein sehr einfach strukturiertes Format ist. Dies hilft bei einigen Analysen, die auf anderen Datenstrukturen sehr aufwändig wären.

Nicht jeder Compiler benutzt alle dieser Darstellungen; es gilt immer zwischen dem zusätzlichen Aufwand zur Konvertierung in eine Darstellung und der Vereinfachung der Analyseschritte abzuwägen. Zusätzlich zu den genannten Repräsentationen verwalten die meisten Compiler eine Symboltabelle, um Namen von Funktionen oder Variablen schnell auflösen zu können.

Im Backend werden diese Zwischendarstellungen dann in Darstellungen konvertiert, die zunehmend „Low Level“ werden. Oft ist dies zuerst eine Low Level Intermediate Representation, wie z. B. die LLVM IR der Compilerinfrastruktur LLVM, die dann in die finale Maschinensprache übersetzt wird.

3 Umsetzung des Toolkits

Dieses Kapitel beschreibt die Implementation der gewünschten Features und geht besonders auf Probleme und Design-Entscheidungen ein.

3.1 Grundstruktur der Applikation

Die komplette Funktionalität wurde aus Gründen der Modularisierung in drei Teile geteilt, die überwiegend unabhängig voneinander arbeiten können. Jeder dieser Teile besteht aus genau einer *Crate* – so wird in Rust eine *Compilation Unit*, also Quellcode, der in einem Schritt vom Compiler übersetzt wird, bezeichnet. Die meisten Bibliotheken in Rust bestehen aus genau einer Crate; eine Aufteilung auf mehrere Crates macht Sinn, falls ein Projekt sehr groß ist oder die einzelnen Crates unabhängig von der restlichen Funktionalität benutzbar sein sollen. Die drei Crates dieser Arbeit sind:

- `xswag-base`: nur Grundfunktionalität, z. B. zum Laden von Dateien und Ausgeben verständlicher Fehlermeldungen. Diese Crate ist komplett unabhängig von einer konkreten Programmiersprache.
  Quellcode: `https://github.com/LukasKalbertodt/xswag-base`

- `xswag-syntax-java`: Java-Parser und weitere Funktionen zur semantischen Analyse von Java-Quellcode. Mit Abstand ist dies die größte Crate – sie entspricht ungefähr einem Compiler-Frontend.
  Quellcode: `https://github.com/LukasKalbertodt/xswag-syntax-java`

- `jswag`: die Endapplikation, die als Programm ausgeführt werden kann. Sie enthält Hilfsfunktionen zum Kompilieren und Prüfen von Quellcode, verwendet aber die fertigen Datenstrukturen von `xswag-syntax-java`.
  Quellcode: `https://github.com/LukasKalbertodt/jswag`

Die Namen sind, nebenbei bemerkt, eine Persiflage an den Modebegriff „swag“, der für eine „coole“ Ausstrahlung und einen „guten“ Stil steht. Das *j* am Anfang des Namens entspricht einem für Java-Software recht populären Namensschema (siehe z. B. JUnit[9]). Initial war geplant, dass `jswag` nur den Stil von Javaprogrammen prüft – erst später wurde der Umfang erweitert. Dabei wurde auch mit der Idee gespielt, die Funktiona-

[9] `http://junit.org/`

lität für weitere Sprachen zu nutzen. Daher wurde bei zwei Crates das *j* durch ein *x* ausgetauscht, da dies nicht einer festen Sprache zugeordnet ist. Die Idee für den Namen entstand teilweise auch durch das Tool *swang*[10]: ein Stylechecker für C++ und eine Mischung aus den Namen *clang* (C++ Compiler) und „swag“.

3.2 Grundfunktionalität

Wie oben schon erwähnt ist die Grundfunktionalität in der Crate `xswag-base` nicht von einer bestimmten Sprache abhängig. Der Hauptzweck besteht darin, sehr einfach verständliche Fehlermeldungen für den Nutzer zu ermöglichen. Ein wichtiger Teil bei Fehlermeldungen ist die Ausgabe des relevanten Quellcodes, sodass direkt ersichtlich ist, wo der Fehler aufgetreten ist – ein indirektes Anzeigen der Position über Zeilennummern ist nur suboptimal.

Um dies bequem zu ermöglichen, wurden zwei wichtige Typen implementiert: `FileMap` und `Span`. Die Definition des Letzteren sieht man (leicht verändert) hier:

```
1  struct Span {
2      pub lo: BytePos,    // inclusive
3      pub hi: BytePos,    // exclusive
4  }
5
6  struct BytePos(pub u32);
```

`Span` ist also lediglich ein Intervall, das einen Bereich im Quelltext repräsentiert und dessen Grenzen vom Typ `BytePos` sind. Die Definition von `BytePos` wird *newtype* Pattern genannt: Auf Maschinenebene unterscheidet sich der Typ nicht von einem `u32`, allerdings wird durch den neu eingeführten Typ im Typsystem verdeutlicht, welche semantische Bedeutung eine Variable dieses Typs hat. Eine Unterscheidung zwischen Zeichen- und Byte-Position ist beim Arbeiten mit UTF-8 Strings äußerst wichtig. Durch Ausnutzung des starken Typsystems von Rust wird zur Kompilierzeit verhindert, dass diese inkompatiblen Positionsangaben vermischt werden.

Ein weiterer Vorteil des *newtype* Patterns lässt sich am Type `LineIdx` zeigen, der genau wie `BytePos` nur einen `u32` umschließt und eine Zeilennummer darstellen soll. Das Problem mit Zeilennummern besteht darin, dass Programmierer gerne mit nullbasierten Indizes arbeiten möchten, während die erste Zeile bei der Ausgabe an den Nutzer mit

[10] `https://github.com/berenm/swang`

„1“ und nicht „0“ bezeichnet werden soll. Dies entspricht der natürlichen Zählweise und der Anzeige in den meisten Texteditoren. Würde man in einer schwach typisierten Sprache direkt einen Integertyp nutzen, müsste man bei jeder Ausgabe manuell darauf achten, eine Eins zu addieren. In Rust lassen sich allerdings für neue Typen selber Traits implementieren. So können wir das Trait `std::fmt::Display` implementieren, welches für die Ausgabe zuständig ist und immer dann bemüht wird, wenn `println!` oder ähnliche Makros einen `{}` Platzhalter ausfüllen müssen.

```
impl fmt::Display for LineIdx {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        // add 1 and call `fmt` implementation of `u32`
        (self.0 + 1).fmt(f)
    }
}

let line = LineIdx(3);
println!("Line {}", line);  // output: "Line 4"
```

Der zweite wichtige Typ, `FileMap`, verwaltet den Inhalt und die Metadaten einer Eingabedatei. Besonders wichtig ist die Speicherung von Positionen der Zeilenanfänge; so kann später zu einer gegebenen Byteposition schnell (in $\mathcal{O}(\log n)$) die zugehörige Zeile gefunden werden. Von außen benutzt man die Methode `get_loc`, um zu einer Position die Zeilen- und Spalteninformationen zu erhalten.

Mit dieser Funktionalität als Grundlage kann nun ein System für nützliche Fehlermeldungen geschaffen werden, welches in `xswag-base` aus den Typen `Report` und `Remark` sowie der Funktion `print` besteht. Ein `Report` besteht aus mehreren `Remark`s und einer Klassifizierung des Fehlertypen – z. B. „Error“ oder „Warning“. Ein `Remark` ist ebenfalls in eine Fehlerkategorie eingeteilt, speichert sich aber außerdem eine Fehlerbeschreibung und einen optionalen Verweis auf einen Codeschnipsel:

```
struct Remark {
    pub kind: RemarkKind,
    pub desc: String,
    pub snippet: Snippet,
}

enum RemarkKind { Error, Warning, Note }

enum Snippet {
    /// No snippet available
    None,
```

```
12      /// Show the original code with this highlighted span
13      Orig(Span),
14      /// Show original code, but replace a part of it with something new and
15      /// highlight the new part.
16      Replace {
17          span: Span,
18          with: String,
19      }
20  }
```

`Report` wird in der kompletten Applikation als Hauptfehlertyp eingesetzt: Der Lexer, Parser und andere Teile geben im Falle eines Fehlers immer diesen Typ zurück. Dies hat den enormen Vorteil, dass man alle Fehler direkt dem Nutzer präsentieren kann, anstatt dass ein weiterer Verarbeitungsschritt erforderlich ist. Die Ausgabe eines `Report`s übernimmt die Funktion `print`, welche neben dem Fehler auch noch eine `FileMap` als Argument erhält, um auf den eigentlichen Code zugreifen zu können.

Hier fällt direkt ein Nachteil des Designs auf: Es kann nicht festgestellt werden, zu welchem `FileMap`-Objekt ein bestimmter `Span` gehört. So muss der Programmierer selber darauf achten, nie Spans unterschiedlicher `FileMap`s miteinander zu vermischen. Dieser Fehler ist zwar während des gesamten Entwicklungszeitraums nicht aufgetreten, trotzdem sollte versucht werden, das Problem in naher Zukunft zu lösen. Vorzugsweise involviert die Lösung keinerlei Checks zur Laufzeit, sondern verifiziert alles zur Kompilierzeit.

In Abbildung 3 sieht man die `print`-Ausgabe eines Fehlers, der vom Parser generiert wird, wenn der Nutzer fälschlicherweise zwei *Access Modifier* bei einer Methode, Klasse oder Ähnlichem angibt. Zuerst wird eine klare Überschrift gedruckt, die ankündigt, dass ein neuer `Report` ausgegeben wird; danach werden alle `Remark`s nacheinander ausgegeben (in diesem Beispiel eins vom Typ `Error`, das andere vom Typ `Note`). Zu

```
+---- ERROR in PlayExample.java : 16 ----+
     =====>  error: duplicate visibility modifier
 16 |     public static private void main(String[] args) throws Peter {
                       ^^^^^^^

     =====>  note: the first visibility modifier is already here
 16 |     public static private void main(String[] args) throws Peter {
            ^^^^^^
```

Abbildung 3: Beispielsausgabe eines `Report`s in `jswag`

```
PlayExample.java:16: error: illegal combination of modifiers: public and private
    public static private void main(String[] args) throws Peter {
                               ^
```

Abbildung 4: Ausgabe von `javac` mit demselben Code wie in Abbildung 3

beiden `Report`s gehört ein Codeschnipsel vom Typ `Snippet::Orig` – das bedeutet, dass der originale Quellcode unverändert ausgegeben wird und ein bestimmter Bereich visuell markiert wird.

Diese Hervorhebung wird durch Einfärben und Unterstreichen des Codes realisiert. Farben werden in der Applikation dieser Arbeit bewusst sehr stark eingesetzt: Sie ermöglichen es dem Nutzer, schnell zu erkennen, wo sich die relevanten Informationen der Ausgabe befinden. Beispielsweise werden Zeilennummern immer in Pink markiert, sodass das Auge des Benutzers zu jeder Zeit weiß, wonach es suchen muss. Die schwarz-graue Ausgabe des originalen `javac`-Compilers kann man in Abbildung 4 sehen. Abgesehen von den fehlenden Farbakzenten fällt auch die niedrige Qualität der Codemarkierung auf: Es wird nur das erste Zeichen eines Tokens unterstrichen und nicht immer der Token markiert, um den es sich wirklich handelt. Das Problem entsteht durch `public` bzw. `private`, daher würde der Nutzer erwarten, dass einer dieser beiden Token markiert ist.

Neben den Codeschnipseln vom Typ `Snippet::Orig`, ist es mit `Snippet::Replace` möglich, veränderten Code auszugeben. Dies eignet sich besonders für Empfehlungen an den Nutzer, sodass dieser direkt sehen kann, wie der funktionierende bzw. korrigierte Code aussieht. In Listing 1 kann man ein Beispiel mit einem sehr bekannten Fehler sehen: Es wurde ein Semikolon am Ende der Zeile vergessen. Zwar ist es für den Parser ziemlich kompliziert zu erkennen, ob ein fehlendes Semikolon für den Fehler verantwortlich ist, aber in diesem Beispiel schafft es `jswag`, einen korrekten Hinweis zu geben (vgl. Abbildung 5). Im Kapitel 3.3.2 wird näher auf die Schwierigkeiten beim Parsen eingegangen.

Die Notiz, welche auf das möglicherweise fehlende Semikolon hinweist, wird zusammen mit einem Code gedruckt, der nicht genau so im Eingabetext existiert. Das fehlende Zeichen wurde eingefügt, um dem Nutzer direkt die mögliche Fehlerursache zu zeigen.

```
1  int x = 0
2  while (x < 10) {
3      x++;
4  }
```

Listing 1: Fehlerhafter Java-Code – in der ersten Zeile fehlt ein Semikolon

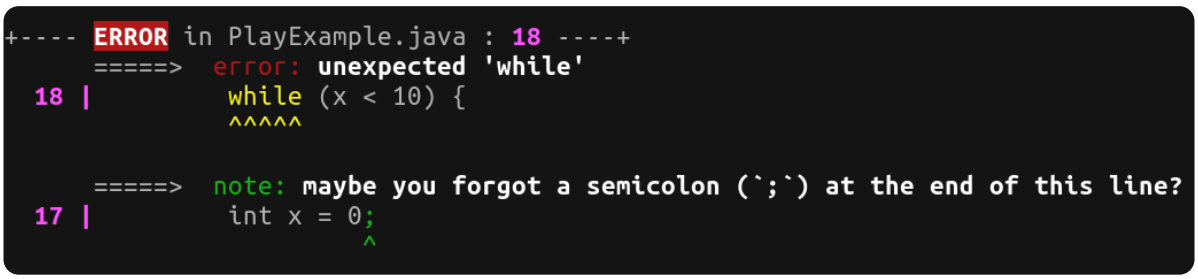


Abbildung 5: Ausgabe von `jswag` beim Kompilieren von Listing 1

Der Anwender kann außerdem anhand des gedruckten Codes gut die betreffende Zeile identifizieren und so schnell in seinem Editor zur Problemstelle springen; der konkrete Code wird Programmierern in den meisten Fällen eher bei der Lokalisierung helfen als nur eine Zeilennummer.

Falls der Bereich im Code, der angezeigt bzw. ersetzt werden soll, mehrere Zeilen umfasst, werden diese ebenfalls korrekt in mehreren Zeilen auf dem Terminal ausgegeben – allerdings ist es dann natürlich unmöglich, die entsprechende Stelle zu unterstreichen; sie wird lediglich farblich markiert. Auch zu lange Fehlerbeschreibungen werden den Wortgrenzen entsprechend auf mehrere Zeilen aufgeteilt.

3.3 Lexer und Parser

Die in `xswag-syntax-java` implementierte Parserfunktionalität nimmt vom Codeumfang den größten Teil dieser Arbeit ein und stellt die Grundlage für weitere Analysen dar. Die Implementierung erfolgt nach dem offiziellen *Java 8*-Standard [4]. Da allerdings eine Realisierung vollständiger Standardkonformität den Umfang dieser Arbeit sprengen würde, ist die Implementation lediglich als Basis für weitere Arbeiten zu sehen.

3.3.1 Lexer

3.3.1.1 Art der Implementierung

Zu Beginn der Entwicklung des Lexers muss entschieden werden, wie genau der Lexer implementiert wird. Neben der Möglichkeit, alles manuell zu programmieren, gibt es auch spezielle Lexer-Generatoren, die aus einer einfachen Beschreibung der Sprache einen

fertigen Lexer generieren. Die bekanntesten Generatoren, wie *JavaCC*[11] oder *Flex*[12], erzeugen den Lexer in Form von C- oder Java-Quellcode. Zwar kann Rust auf die native Funktionsschnittstelle zugreifen und so C-Funktionen aufrufen, nichtsdestotrotz wäre viel Code nötig, um die beiden Sprachschnittstellen zu verbinden. In diesem Fall wäre es besonders nachteilig, da die Konvertierung zwischen den unterschiedlichen Token-Typen viel Aufwand bedeutet. Außerdem sollte die Applikation dieser Arbeit komplett in Rust geschrieben werden, um die Sprache so besser evaluieren zu können.

Trotz des jungen Alters der Sprache gibt es in Rust schon diverse Parser- und Lexer-Generatoren, die einen Lexer in Rust-Code generieren. Zum einen gibt es *Plex*[13], der seit Anfang 2015 in Entwicklung ist. Er nutzt die sogenannte Rust Syntax Extension, um aus einer kurzen Beschreibung einer Grammatik einen DEA[14]-basierten Lexer zu erzeugen. Das bedeutet, dass eine Übergangsmatrix generiert und zur Laufzeit abgearbeitet wird. Für reguläre Grammatiken ist ein endlicher Automat zwar Standardwerkzeug, allerdings ist die Umsetzung von *Plex* vermutlich nicht besonders schnell – der Automat wird intern vergleichsweise so dargestellt:

```
1  // ‘Input‘ is the input type, usually ‘char‘
2  struct Dfa<Input> {
3      states: Vec<State<I>>,
4  }
5  struct State<Input> {
6      transitions: BTreeMap<Input, u32>,
7      default: u32,
8  }
```

Die Zustände werden in einem zusammenhängenden Array gespeichert und jeder Zustand besitzt eine Abbildung Input $\mapsto$ State, wobei der Zustand nur durch seine Position im Array gespeichert wird. Bei jedem Zustandswechsel, in dem nicht in denselben Zustand zurückgewechselt wird, wird sehr wahrscheinlich zweimal auf kalten Speicher[15] zugegriffen: Beim Zugriff auf das Array und beim Abfragen der `BTreeMap` (Annahme `transitions.len()` < B-Parameter der Map). Diese kalten Zugriffe können die Ausführungsgeschwindigkeit stark verringern [5].

Die Entscheidung, auf die Nutzung von *Plex* verzichten, liegt allerdings nicht in der verminderten Ausführungsgeschwindigkeit begründet. Der Grund ist vielmehr, dass die

[11] `https://javacc.java.net/`
[12] `http://flex.sourceforge.net/`
[13] `https://github.com/goffrie/plex`
[14] Deterministischer Endlicher Automat
[15] Speicher, der nicht im CPU Cache ist

erwähnte Syntax Extension API noch nicht stabilisiert wurde und Änderungen der API jederzeit zu Kompilierfehlern führen können. `jswag` soll aber komplett mit stabilisierten Rust-APIs implementiert werden, sodass keine Gefahr besteht, nach einem Update des Compilers nicht mehr kompilierfähig zu sein. Somit kommt auch als Alternative der Lexer-Generator Rustlex in Frage, da dieser ebenfalls die nicht stabilisierte Syntax Extension API nutzt.

Da beide Lexer-Generatoren aus den genannten Gründen nicht eingesetzt werden können und der Rust-Compiler ebenfalls mit einem handgeschriebenen Lexer arbeitet, fiel die Entscheidung darauf, den Lexer manuell zu programmieren.

3.3.1.2 Manuelle Implementierung

Die Struktur und Implementation des Lexers orientieren sich an dem frei verfügbaren Quellcode des (in Rust geschriebenen) offiziellen Rust-Compilers. Zunächst wurde mit dem Typ `Token` die Grundlage geschaffen: Der Lexer produziert eine Sequenz von `Token`, die dann vom Parser weiterverarbeitet werden. Die Anforderungen von Java an den Lexer und `Token` Typ ist zum größten Teil im Kapitel 3. („Lexical Structure“) der Java-Spezifikation beschrieben. Im Abschnitt 3.5. wird beschrieben, welche Art von Token es geben kann – dieser Aufbau wurde nicht direkt übernommen, sondern zur besseren Struktur des Rust-Codes leicht verändert implementiert:

```
enum Token {
    Whitespace,
    Comment,

    // Variants of the Java "Token"
    Ident(String),
    Keyword(Keyword),
    Literal(Literal),

    // Variants of Java "Separator"
    ParenOp,    // "("
    BraceOp,    // "{"
    // ...

    // Variants of Java-*Operator*
    Plus,       // "+"
    Minus,      // "-"
    // ...
}
```

An der Definition kann man einige Designentscheidungen erkennen:

- `Whitespace` und `Comment` speichern keine weiteren Informationen: In der Regel ist die genaue Anzahl von Leerzeichen oder der Inhalt eines Kommentars in späteren Schritten der Analyse unwichtig.
- Statt für Separatoren und Operatoren einen eigenen Typ zu definieren, wurden alle möglichen Variants direkt im `Token` Typ gelistet, um deren Benutzung zu vereinfachen.
- Die Keywords und Literale wurden in einen anderen Typen ausgelagert.

Während der Typ `Keyword` eine einfache Auflistung aller Java-Schlüsselworte darstellt, ist der `Literal`-Typ komplexer und speichert sich zu unterschiedlichen Literaltypen diverse Informationen: Bei einem `boolean`-Literal muss festgehalten werden, ob sein Wert `true` oder `false` ist; bei Fließkomma-Literalen müssen jedoch die Basis, der Exponent, der Radix[16] und der Typ (`float` oder `double`) gespeichert werden.

Um für gute Fehlermeldungen über akkurate Positionsangaben zu verfügen, produziert der Lexer nicht einfache `Token`, sondern `TokenSpan`s, die neben den Token-Informationen auch einen `Span` speichern.

Der eigentliche Lexer wird in einem eigenen Typ umgesetzt, welcher den Trait `Iterator` aus der Standardbibliothek implementiert und somit dem Iterator Pattern entspricht. Dieses in Rust wichtige Pattern ermöglicht die Benutzung von Iterator Adaptoren, wie `map` und `filter`, und erlaubt somit einen sehr funktionalen Programmierstil. In Java 8 ist eine ähnliche Funktionalität unter dem Namen „Streams“ eingeführt worden. Da das `Iterator`-Trait sehr generisch formuliert wurde und alle Operationen *lazy* sind, kann es auch unendliche Iteratoren geben.

Um den nächsten `TokenSpan` liefern zu können, verwaltet der Lexer diverse Informationen zum aktuellen Zustand im Quellcode; dazu gehört auch die Verwaltung eines konstant großen Lookaheads. Zur Analyse regulärer Grammatiken wird zwar kein Lookahead benötigt (da endliche Automaten ebenfalls keinen besitzen), allerdings hat der Zugriff auf das nächste, aktuelle und letzte Zeichen den handgeschriebenen Code stark vereinfacht.

Einen besonderen Fall stellen jedoch die Java-Unicode-Escapes dar (vgl. Kapitel 3.3. des Java-Standards). Diese erlauben, im kompletten Javaprogramm (nicht nur in Stringliteralen) beliebige Unicode-Zeichen durch einen Escapecode darzustellen:

[16] Basis des Stellenwertsystems (im Fall von `float`: dezimal oder hexadezimal)

```
public cl\u0061ss Caf\u00C9 {}
```

Dieser korrekte Java-Code definiert eine Klasse `Café`; der Wert $\texttt{61}_{\text{hex}}$ steht für das Zeichen `a`, $\texttt{C9}_{\text{hex}}$ für das Zeichen `é`. Der Standard erlaubt nach einem Backslash beliebig viele `u`'s – hier wurde, um den Code einfach zu halten, ein beliebig großer Lookahead genutzt.

Der Code zur Behandlung dieser Escapes befindet sich in der Methode `bump`, die ein Zeichen im Eingabetext weiterrückt. Außerdem erkennt `bump` Zeilenumbrüche und meldet deren Positionen an die `FileMap`. Den Kern des Lexer stellt jedoch ein großer Match-Ausdruck dar; hier hilft besonders die Range-Syntax (z. B. `'a' ... 'z'`), um die Länge des Codes zu verringern.

```
match self.curr {   // current character
    '(' => { self.bump(); Token::ParenOp },
    '0' ... '9' => self.scan_number_literal(),
    c if is_java_ident_start(c) => self.scan_word(),
    // ...
}
```

Einfache Operatoren werden direkt in dem Match-Block behandelt; komplexere Token, wie Literale oder Kommentare, werden in gesonderten Methoden verarbeitet.

Neben der Implementation von Unicode-Escapes hat die Umsetzung des Lexings für Ganzzahl- und Fließkomma-Literal viel Aufwand bedeutet. In den Kapiteln 3.10.1. und 3.10.2. beschreibt der Standard die komplexe Syntax der beiden Literaltypen. Beispielsweise ist `0xC_a.p-0___3f` ein gültiges Fließkomma-Literal mit dem Wert 25,25 vom Typ `float`, welches in Hexadezimal und mit Exponenten angegeben wurde. Ganzzahl-Literale kann man, zusätzlich zu hexadezimaler und dezimaler, auch noch in oktaler und binärer Schreibweise angeben.

Als besonders schwierig erwies sich die manuelle Implementierung dieses Teils des Lexers, bei der darauf geachtet wurde, doppelte Codes zu vermeiden. Hier wäre die Nutzung von Regex eine Vereinfachung gewesen, auf die allerdings verzichtet werden musste, da zur Zeit keine Regex-Implementation existiert, die zusammen mit dem vorhandenen Lexer-System funktionieren würde.

3.3.2 Parser

Bei der Implementation des Parsers stellte sich dieselbe Frage, wie beim Lexer: Ist eine manuelle Implementierung zu bevorzugen oder ist die Benutzung eines Parsergenerators sinnvoll? Während der handgeschriebene Lexer des Rust-Compilers ungefähr 1500 Zeilen umfasst, ist der ebenso handgeschriebene Parser des Compilers ungefähr 6000 Zeilen lang – die Prüfung der syntaktischen Grammatik ist in der Regel nämlich deutlich komplexer als die einer *Lexical Grammar*. Die Nutzung eines Generators schien in diesem Fall also deutlich sinnvoller als noch beim Lexer.

Auch im Bereich Parsergeneratoren und Parserkombinatoren überrascht die junge Infrastruktur mit einer Vielzahl an Bibliotheken und Frameworks, wie z. B. `nom`[17], `oak`[18] oder `rust-peg`[19]. Aus allen verfügbaren Generatoren stach jedoch besonders `lalrpop` heraus [7]. Sein Autor, Dr. Niko Matsakis, ist einer der Kernentwickler von Rust, arbeitet seit 2011 bei Mozilla Research und hat durch seine Lehre im Bereich Compilerdesign einige Erfahrung mit Parsern gesammelt.

Der Vorteil von `lalrpop` gegenüber den anderen verfügbaren Bibliotheken ist zum einen, dass es auf größere Grammatiken (wie die einer Programmiersprache) ausgelegt ist, und zum anderen, dass die Syntax für die Beschreibung der Grammatik sehr an die von Rust angelehnt ist. Zur Zeit der Entwicklung unterstützte `lalrpop` die Erzeugung von sowohl LR(1)- als auch LALR-Parsern; die Form der Grammatik ist in beiden Fällen dieselbe. Mit Hilfe von Build-Scripts, die von dem Rust-Buildtool `cargo` automatisch ausgeführt werden, generiert `lalrpop` aus der Definition der Grammatik einen in Rust geschriebenen Parser. Dieser ist in einem eigenen Modul untergebracht, welches man auf normale Weise in sein Projekt einbindet. So kann man dann die vom Parser exportierten Funktionen aufrufen, die den Parsingvorgang anstoßen und das Ergebnis zurückliefern.

Da das Ergebnis ein AST und kein Parsetree sein soll, reicht es nicht, nur die Grammatik anzugeben. Stattdessen muss zusätzlich die Transformation zum AST spezifiziert werden. Dies wird mit `lalrpop` in den Regeln der Grammatik in dem sogenannten Action-Code erledigt. Jedes non-terminal liefert den Wert eines bestimmten Typs zurück – in der Regel einen Knoten aus dem AST. In Abbildung 3.3.2 sieht man die beispielhafte Umsetzung der ⟨ImportDecl⟩-Grammatik aus dem Kapitel 2.2.3. Eine Besonderheit ist der Support von Regex-artigen Annotationen, wie dem Kleene-Stern in Zeile 9, um die Grammatik zu vereinfachen.

[17] `https://github.com/Geal/nom`
[18] `https://github.com/ptal/oak`
[19] `https://github.com/kevinmehall/rust-peg`

```rust
// in module `ast`

enum Import {
    Single(Path),
    Wildcard(Path),
}

struct Path {
    segments: Vec<String>,
}
```

AST Definition in Rust

```
pub ImportDecl: ast::Import = {
    "import" <Path> ";" =>
        ast::Import::Single(<>),
    "import" <Path> "." "*" ";" =>
        ast::Import::Wildcard(<>),
};

Path: ast::Path = {
    Word ("." <Word>)* => /* ... */
};
```

Grammatik in `lalrpop`-Syntax

Falls bei der Generierung des Parsers Fehler auftreten, z. B. aufgrund einer mehrdeutigen Grammatik, werden diese auf dem Terminal ausgegeben. Wenige Wochen vor der Abgabe dieser Arbeit wurden die Fehlermeldungen in einem Update von `lalrpop` deutlich menschenlesbarer gemacht [8]; so werden komplizierte Shift-Reduce Konflikte einfach dargestellt und erklärt. So führt etwa die folgende Grammatik (in vereinfachter `lalrpop`-Syntax) zu dem Fehler aus Abbildung 6.

```
pub Expr = {
    Expr "+" Expr,
    Expr "-" Expr,
    "0",
    "1",
};
```

```
test.lalrpop:6:5: 6:23: Ambiguous grammar detected

  The following symbols can be reduced in two ways:
    Expr "+" Expr "-" Expr

  They could be reduced like so:
    Expr "+" Expr "-" Expr
    ├─Expr─────┘          │
    └─Expr────────────────┘

  Alternatively, they could be reduced like so:
    Expr "+" Expr "-" Expr
    │          └─Expr────┐
    └─Expr──────────────┘

  LALRPOP does not yet support ambiguous grammars. See the LALRPOP manual for
  advice on making your grammar unambiguous.
```

Abbildung 6: Ausgabe von `lalrpop`

So fiel die Wahl schließlich auf `lalrpop` als Parsergenerator, was sich als positiv herausstellte, insbesondere, weil Herr Matsakis die Entwicklung durch ausführliche Erklärungen zum Thema Parserbau unterstützt hat.

Die Implementierung der Java-Grammatik hat sich an einigen Stellen als problematisch herausgestellt. Die Vorrangbeziehungen zwischen unterschiedlichen Operatoren in Java wurde z. B. durch einen typischen Trick bei der Formulierung der Grammatik erreicht: Jede Prioritätsstufe wird durch eine eigene Regel dargestellt, die nur entweder links oder rechts-rekursiv ist. Ein Beispiel für die arithmetischen Operatoren `+ - * /` sieht man hier:

```
pub Expr = SumExpr;
SumExpr = {
    SumExpr "+" ProductExpr,
    SumExpr "-" ProductExpr,
    ProductExpr,
};
ProductExpr = {
    ProductExpr "*" Num,
    ProductExpr "/" Num,
    Num,
};
Num = r"[0-9]+";      // regex syntax
```

Durch diese spezielle Formulierung der Grammatik wird sichergestellt, dass der Parsetree eindeutig ist. Glücklicherweise sind viele dieser grammatikalischen Regeln, die zu Mehrdeutigkeit führen können, schon in der Java-Spezifikation in eine solche eindeutige Form überführt worden. Weiterhin unterstützt `lalrpop` die Definition von Makros, mit denen bestimmte Konzepte abstrahiert formuliert werden können, um Code zu sparen. Diese Regeln für links-assoziative, binäre Operatoren ließen sich so mit deutlich weniger doppeltem Code formulieren.

Ein weiteres Problem in der Java-Syntax stellt das sogenannte *dangling else*-Problem dar, welches auch in anderen Sprachen wie C auftritt. Veranschaulicht wird das Problem durch diesen Java-Code:

```
if (true)
    if (true)
        doSomething();
else
    somethingElse();
```

Die Formatierung und Einrückung dieses Codes suggeriert, dass das `else` zum äußeren `if` gehört. Das ist aber (zumindest in Java) nicht der Fall. In der Spezifikation wurde nämlich diese Mehrdeutigkeit der Grammatik mit der arbiträren Regel, dass ein `else` immer zu dem letzten (innersten) `if` gehört, aufgelöst. Diese Entscheidung in der Grammatik zu formulieren, um Mehrdeutigkeit zu eliminieren, stellt sich jedoch als schwieriger heraus, als es ist: Zusätzlich zu dem `if`-Statement müssen noch andere Statements, wie `for` und `while`, eine Sonderbehandlung bekommen, da sie alle rechts-rekursiv sind. Das heißt in diesem Fall, dass sie alle als letztes „Symbol“ wiederum ein Statement erwarten, was nicht zwangsweise von geschweiften Klammern umschlossen werden muss.

Insgesamt ist die syntaktische Java-Grammatik eher komplex, da es sehr viele Sonderregeln gibt und eine Trennung von Statement und Expression vorgenommen wurde. Außerdem ist die Grammatik an einigen Stellen so formuliert, um gewisse Logikfehler abzufangen; das klingt zwar grundsätzlich gut, erschwert aber das Erzeugen von guten Fehlermeldungen. Beispielsweise ist folgende Methodendefinition in Java ungültig:

```
1 void foo(int x) {
2     x + 1;
3 }
```

Der Rumpf einer Funktion besteht aus einer Liste von *StatementExpressions*. Eine *StatementExpression* kann eine Zuweisung, ein Methodenaufruf oder eine Inkrement- oder Dekrementoperation sein. Andere Arten von Ausdrücken an Stelle eines Statements zu benutzen macht, wie im Beispiel oben, im Endeffekt keinen Sinn, da es keinerlei Seiteneffekte gibt. Doch das Ablehnen einer solchen Eingabe bereits zur Zeit der Syntaxprüfung hat den Nachteil, dass man nur erschwert eine sinnvolle Fehlermeldung ausgeben kann. Falls man solche Ausdrücke in der syntaktischen Grammatik zuließe, könnte man im darauffolgenden Schritt den AST recht trivial auf solche Seiteneffekt-freien Statements untersuchen und so sinnvollere Fehlermeldungen produzieren. Die Analyse auf Seiteneffekte-Freiheit ist in Java zudem noch stark vereinfacht, da Java keine Operatorüberladung zulässt und somit für alle arithmetischen Operatoren klar ist, dass sie Seiteneffekt-frei sind.

3.3.3 Fehlererkennung beim Parsen

Besonders Programmieranfänger scheitern beim Schreiben eines Programms oft an Syntaxfehlern und passende Fehlermeldungen würden hier enorm helfen. Leider unterschei-

det sich die Art, wie Menschen über Syntax nachdenken, stark von der algorithmischen Umsetzung der meisten Parser. Letztere lesen in jedem Schritt einen Token und überprüfen, ob dieser zu der Grammatik passt. Menschen hingegen sehen bestimmte Muster und analysieren so den Quellcode. Der folgende Code veranschaulicht das Problem:

```
1 wihle (x == 3) {
2     IO.println("x ist ungefähr Pi");
3 }
```

Ein Mensch würde hier sofort sehen, dass `wihle` ein falsch geschriebenes `while` ist. Für einen Parser und Lexer sieht das jedoch ganz anders aus: Der Lexer erkennt den ersten Token nicht als Schlüsselwort, sondern einfach als `Word`. Der Parser erhält diesen und die nachfolgenden Token und kommt zu dem Schluss, dass die ersten sechs Token (`wihle(x == 3)`) einen Methodenaufruf darstellen. Dann wird ein Semikolon erwartet, allerdings eine öffnende geschweifte Klammer gefunden. Diese könnte, falls vorher kein Fehler aufgetreten ist, ein gültiges *BlockStatement* einleiten. Der Parser würde die ganze Eingabe also akzeptieren, falls sich vor der öffnenden geschweiften Klammer ein Semikolon befindet. Das Ergebnis, wie es der Parser sehen würde, wird im Folgenden mit passender Formatierung gezeigt:

```
1 wihle(x == 3);
2 {
3     IO.println("x ist ungefähr Pi");
4 }
```

Eine optimale Fehlermeldung in diesem Fall würde auf die Möglichkeit hinweisen, dass der Programmierer ein `while` falsch geschrieben hat, und bemerken, dass eine Methode mit dem Namen `wihle` nicht gefunden wurde. Dies allerdings umzusetzen ist sehr kompliziert. Zum einen muss der Parser entscheiden können, wann in einer bestimmten Situation ein Schlüsselwort gemeint sein könnte und wann nicht; lediglich Vorschläge aufgrund der Stringdistanz zu machen, wird vermutlich zu viele *false positives* liefern. Weiterhin besteht ja die Möglichkeit, dass eine Methode mit dem Namen `wihle` später im Programm definiert ist. Um dies herauszufinden, müsste der Parser nach diesem Syntaxfehler eine Art Error Recovery durchführen, um weiter parsen zu können. Auch das ist beim Parserbau nur schwer zu implementieren.

Ein weiteres Beispiel, welches schon in Kapitel 3.2 vorgestellt wurde, ist das fehlende Semikolon am Ende einer Zeile bzw. Anweisung. Hier ist es für einen erfahrenen menschlichen Leser wieder sofort ersichtlich, wo der Fehler liegt. Für den Parser ist es allerdings

wieder eine Herausforderung, festzustellen, was genau der Fehler ist – auch weil er auf einem Token-Stream arbeitet, der keine Informationen über Zeilenumbrüche enthält (da diese für die syntaktische Korrektheit irrelevant sind). Der originale `javac`-Compiler meldet in einem solchen Fall Fehler in den darauf folgenden Zeilen, die für die meisten Anfänger keinen Sinn ergeben.

Um trotzdem halbwegs verständliche Fehlermeldungen zu produzieren, kann man sich nur einiger spezieller Tricks bedienen. In `jswag` wird z. B. bei einem Syntaxfehler Folgendes überprüft:

- Befindet sich der unerwartete Token am Anfang seiner Zeile?
- Ist in der ersten nichtleeren, darüberliegenden Zeile am Ende *kein* Semikolon?
- Kann nach dem letzten Token dieser vorherigen Zeile überhaupt ein Semikolon stehen? Bei öffnenden Klammern ist das beispielsweise verboten.

Wenn diese Bedingungen zutreffen, weist `jswag` darauf hin, dass ein fehlendes Semikolon eventuell der Fehler ist, was allerdings nur eine sehr grobe Abschätzung darstellt. `jswag` findet somit nicht alle Fehler und produziert in vielen Fällen ein *false positive*.

Zusammenfassend lässt sich sagen, dass gute Beschreibungen gerade für Syntaxfehler nur schwer zu erzeugen sind. Während beim Compiler meist Lexer, Parser und semantische Analyse getrennt sind, verarbeiten Menschen alle drei Ebenen gleichzeitig. Zusätzlich werden dabei Informationen der unterschiedlichen Schichten verknüpft und so Rückschlüsse gezogen, Verbesserungen vorgenommen und neu interpretiert. So sind für den Parser das Schlüsselwort `while` und das Wort `wihle` zwei komplett unterschiedliche Token, wohingegen ein Mensch nach weiterer syntaktischer und semantischer Analyse deduziert, dass mit `wihle` eigentlich das Schlüsselwort `while` gemeint ist, aber falsch geschrieben wurde.

3.4 Endanwendung `jswag`

Die in den vorherigen Kapiteln vorgestellte Funktionalität wurde lediglich in einer der beiden `xswag`-Bibliotheken implementiert. Um als Nutzer nun bequem darauf zugreifen zu können, wird die Endapplikation `jswag` genutzt. Diese bietet ein Commandline-Interface, um genau konfigurieren zu können, welche Arbeitsschritte und Analysen ausgeführt werden sollen. Weiterhin soll `jswag` aus Nutzerperspektive ein vollwertiger Ersatz für `javac` sein; `jswag` unterstützt also auch eine Weiterleitung von bestimmten

Aufgaben an den offiziellen Java-Compiler. Es ist daher nicht nur ein *Static Analyzer*, sondern auch ein Buildtool, welches besonders Programmieranfänger unterstützen soll.

Als Eingabe erhält `jswag` grundlegend drei Argumente:

- Eine Liste von Dateien und Ordnern, auf denen operiert werden soll
- Eine Liste von `Job`s, die für diese angegebenen Dateien ausgeführt werden sollen
- Weitere, optionale Parameter, um spezielles Verhalten der Applikation zu steuern

Zur Verarbeitung der Commandline-Parameter wird die Bibliothek `docopt` genutzt. Dieses ursprünglich in Python geschriebene Tool erlaubt es, die Konfiguration erlaubter Parameter sehr deklarativ, in einem Usage-String, zu spezifizieren. Dieser Usage-String beschreibt in englischer Sprache, wie die Applikation zu bedienen ist; daraus generiert `docopt` dann Code zur Prüfung der Eingabe. `jswag` besitzt folgendes Commandline-Interface (Beschreibung gekürzt):

```
Usage: jswag build [options] [<file>...]
       jswag run [options] [<file>...]
       jswag [options] <file>...
       jswag raw [<file>...]
       jswag (--help | --version)

Commands:
    build       Compiles all files and runs simple analysis checks on them.
                Automatically adds these parameters:
                    $ --check --pass-through --analyze style
    run         Works like ‘build‘, but runs the file afterwards. Automatically
                adds these parameters to the already added parameters of
                ‘build‘:
                    $ --run
    <none>      For compatibility this works similar to the original ‘javac‘
                command. Right now it’s exactly the same as ’build’, except
                that the file list mustn’t be empty.
    raw         Does nothing automatically. Every task has to be stated
                explicitly with command line parameters.

Actions:
    -a <check>, --analyze <check>   Run the given check. Implies ‘-c‘.
    -c, --check                     Check files for language errors with
                                    internal tools.
    -p, --pass-through              Call ‘javac‘ to compile the files.
    -r, --run                       Tries to execute the compiled classes in
                                    the order they were given. Requires ‘-p‘.
```

Die Gestaltung dieses Interfaces lehnt sich an das Interface von `cargo`, dem Rust-Buildtool, an. Falls ein übergebener Pfad ein Verzeichnis und keine Datei ist, werden alle Dateien direkt in diesem Verzeichnis in die Liste der zu bearbeitenden Dateien eingefügt; es wird jedoch nicht rekursiv in Unterverzeichnisse abgestiegen. Die so entstandene Dateiliste wird allerdings nochmals mit Hinsicht auf die passende Dateiendung gefiltert. So wird vermieden, dass bei einem Aufruf wie `$ jswag Baum*` fälschlicherweise auch versucht wird, die `.class` Dateien zu kompilieren. Diese Filterung soll in späteren Versionen von `jswag` deaktivierbar sein. Zuletzt gibt es eine Sonderregelung für den Fall, dass keine Pfade angegeben werden: Dies ist äquivalent zum Aufruf mit dem Pfad des jetzigen Ordners (z. B. `.`). Somit werden also alle Java-Dateien im aktuellen Verzeichnis kompiliert.

Wie der oben gezeigte Usage-String weiterhin verrät, kann man mit dem Parameter `--analyze` bestimmte Analysen anstoßen. Derzeit existiert nur der Proof-of-Concept Stylechecker, der z. B. testet, ob Namen im korrekten Stil geschrieben wurden. Im Folgenden ist eine Beispielausgabe dieser Analyse zu sehen:

```
   Checking | 'hello_world.java'
+---- WARNING in hello_world.java : 1 ----+
      =====>  warning: This type name should be written in 'UpperCamelCase'
   1 | public class hello_world {
                   ^^^^^^^^^^^

      =====>  note: Consider changing the name as shown here
   1 | public class HelloWorld {
                   ^^^^^^^^^^


+---- WARNING in hello_world.java : 7 ----+
      =====>  warning: This method name should be written in 'lowerCamelCase'
   7 |     private static float ReturnPi_squared() {
                               ^^^^^^^^^^^^^^^^

      =====>  note: Consider changing the name as shown here
   7 |     private static float returnPiSquared() {
                               ^^^^^^^^^^^^^^^
```

Ursprünglich war geplant, eine weitere Funktionalität in `jswag` einzubauen, die Dateien bei jeder Änderung auf der Festplatte neu kompiliert. So müsste der Benutzer nicht manuell den Kompiliervorgang starten. Da dieses Feature aufgrund des begrenzten Zeitrahmens der vorliegenden Arbeit nicht mehr implementiert werden konnte, bietet es sich als Gegenstand zukünftiger Arbeiten an.

4 Reflexion und Fazit

Das in dieser Arbeit entstandene Toolkit ist in der Lage, eine benutzbare Teilmenge der Sprache Java zu parsen und so grundlegende Datenstrukturen zur Analyse eines Quelltextes zur Verfügung zu stellen. Die darauf aufbauende Applikation nutzt die bereitgestellte Funktionalität, um Programmierern die Möglichkeit zu geben, über ein bequemes Interface Java-Quelltexte untersuchen zu können. Obwohl man diese Funktionalität bereits sinnvoll einsetzen kann, ist sie für den alltäglichen Gebrauch noch nicht geeignet, da sich die Implementationen der sinnvollen, auf dem Toolkit aufbauenden Analysen noch in der *Proof of Concept*-Phase befinden.

Jedoch wurde mit dem `xswag`-System eine nützliche Grundlage für weitere Arbeiten gelegt. Zum einen können auf ihm basierende Applikationen, wie das eingangs erwähnte Vergleichstool zur Erkennung von Plagiaten, geschrieben werden. Zum anderen bietet das Toolkit selber Möglichkeiten für Optimierungen und Forschungen: Interessant wäre z. B. eine programmiersprachenunabhängige Konzeption wichtiger Datenstrukturen, wie der HIR (*high level intermediate representation*). Nachdem Parser für andere Sprachen geschrieben wurden, müssten so gewisse Algorithmen nur einmal implementiert werden und würden trotzdem für alle Sprachen nutzbar sein. Auch in Bezug auf das Design von hilfreichen Fehlermeldungen könnten viele weitere Ideen einfließen.

Bezüglich des Einsatzes von Rust beim Compilerbau kann konstatiert werden, dass sich die Programmiersprache gut für diese Art von Software eignet. Zum einen ist Rust durch das Typsystem sehr sicher und weist außerdem eine hohe Ausführungsgeschwindigkeit auf. Zum anderen eignen sich besonders seine additiven Typen (`enum`) für die Darstellung vieler Elemente, wie den Token und Knoten im AST.

Ziel ist es letztlich, auf dem Toolkit aufbauende Applikationen, wie `jswag`, im Übungsbetrieb der Informatik-Fakultät der Universität Osnabrück einzusetzen. Befragte Informatik-Tutoren sind der Meinung, dass ein solches Tool sowohl den Studenten als auch den Mitarbeitern helfen wird. Durch den rasant steigenden Bedarf an qualifizierten Programmierern auf dem Arbeitsmarkt sollen insbesondere junge Menschen für die Informatik begeistert werden. Ein intelligentes Tool, welches leicht verständliche Hinweise gibt, wäre zumindest teilweise ein Ersatz für eine persönliche Betreuung. Momente der Frustration können so bei Programmieranfängern partiell verhindert werden – entsprechend kann die Begeisterung für das Fach steigen.

Literatur

[1] *Frequently Asked Questions - The Rust Programming Language*. `https://www.rust-lang.org/faq.html#is-this-project-controlled-by-mozilla`, . – Aufruf am 09.03.2016

[2] Aho, A. V. ; Sethi, R. ; Ullman, J. D.: *Compilers, Principles, Techniques*. Addison wesley, 1986

[3] Aho, A. V. ; Ullman, J. D.: *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., 1972

[4] Gosling, J. ; Joy, B. ; Steele, G. ; Bracha, G. ; Buckley, A. : *The Java Language Specification*. 2015

[5] Grunwald, D. ; Zorn, B. ; Henderson, R. : Improving the cache locality of memory allocation. In: *ACM SIGPLAN Notices* Bd. 28 ACM, 1993, S. 177–186

[6] Knuth, D. E.: On the translation of languages from left to right. In: *Information and control* 8 (1965), Nr. 6, S. 607–639

[7] Matsakis, N. : *LALRPOP: LR(1) parser generator for Rust*. `https://github.com/nikomatsakis/lalrpop`, 2016. – Aufruf am 08.03.2016

[8] Matsakis, N. : *Nice Errors in LALRPOP*. `http://smallcultfollowing.com/babysteps/blog/2016/03/02/nice-errors-in-lalrpop/`, 2016. – Aufruf am 08.03.2016

[9] Soloway, E. ; Ehrlich, K. : Empirical studies of programming knowledge. In: *Software Engineering, IEEE Transactions on* (1984), Nr. 5, S. 595–609

Erklärung

Ich versichere, dass ich die eingereichte Bachelor-Arbeit selbstständig und ohne unerlaubte Hilfe verfasst habe. Anderer als der von mir angegebenen Hilfsmittel und Schriften habe ich mich nicht bedient. Alle wörtlich oder sinngemäß den Schriften anderer Autoren entnommenen Stellen habe ich kenntlich gemacht.

Osnabrück, den 10. März 2016

(Lukas Kalbertodt)