Projekt:
Automatisierte Kontrolle der Zugehörigkeit eines Wortes zu einer formalen Grammatik nach der EBNF

von Benjamin Bak und Josua Konsolke

Inhaltsverzeichnis:

1. Unser Projekt

Dieses Projekt wurde von Benjamin Bak und Josua Konsolke entwickelt. Beide sind in der 12. Klasse des Evangelischen Gymnasium Meiningen.
Dieses Projekt entstand im Rahmen der Softwarechallenge von Inverso und wurde ohne fremde Hilfe entwickelt. Zudem hängt dieses Projekt nicht mit anderen Wettbewerben wie “Jugend forscht” oder der Seminarfacharbeit zusammen.
Folgendes ist entstanden: Ein in Python geschriebenes Programm, welches in der Lage ist, .txt-Dateien mit einer beliebigen formalen Grammatik in der “Erweiterten Backus-Naur Form” (EBNF) einzulesen und diese auf Syntax-Fehler überprüfen kann. Anschließend kann der Nutzer ein beliebiges Wort eingeben und dieses auf die Zugehörigkeit zu der Grammatik, welche über die .txt-Datei eingegeben wurde, prüfen. Somit funktioniert dieses Programm also wie eine Art Interpreter, mit dem die formale Grammatik in der EBNF in Python “*verstanden*” wird und diese dann zur Wortüberprüfung nutzen kann.

2. Entwicklung

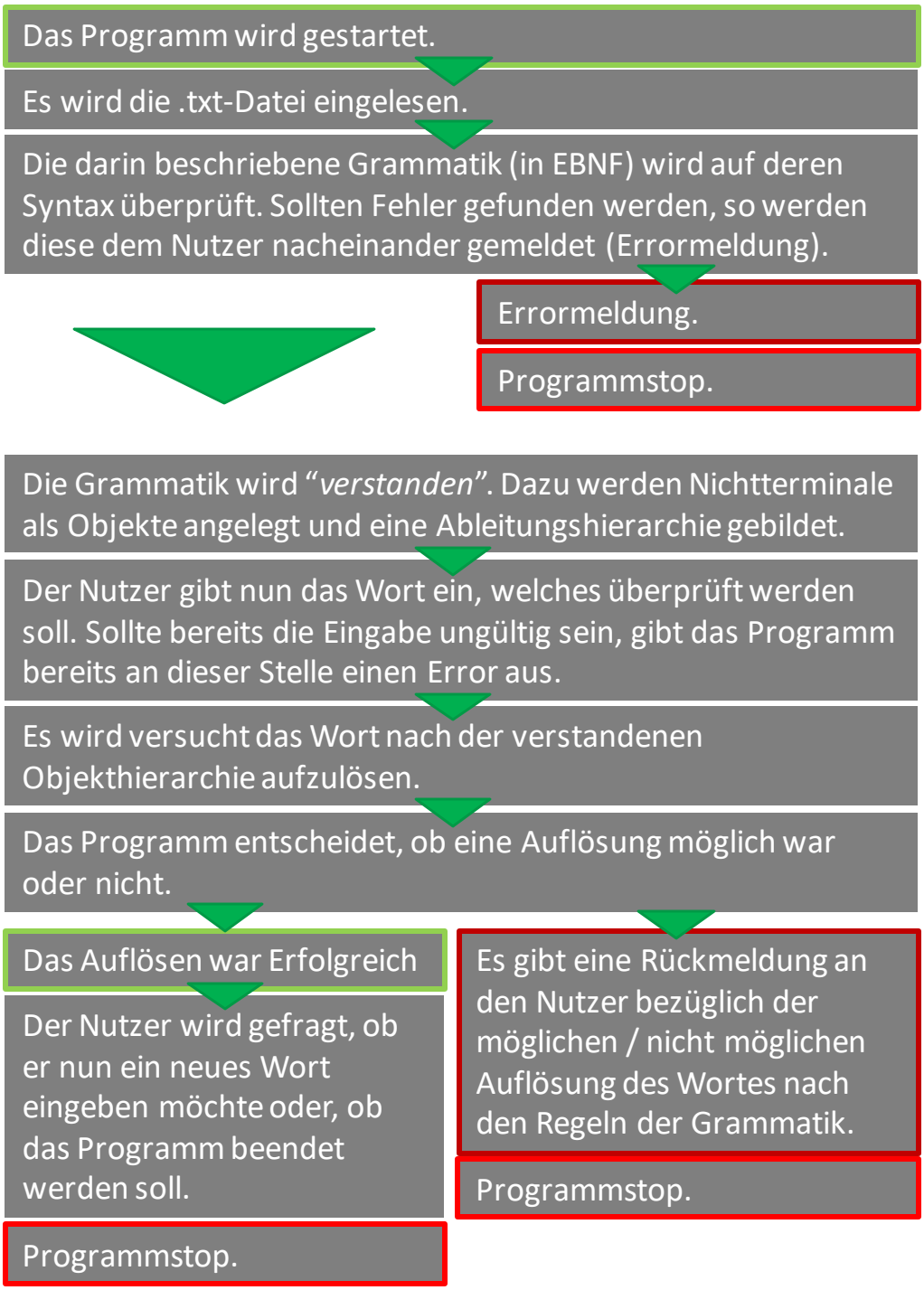
Aufgrund der Spontanität unseres Entschlusses noch an der Softwarechallenge teilzunehmen, fand die Entwicklung innerhalb einer Woche (25.01. bis 29.01.2021) statt. Lediglich die grobe Idee und erste Pläne zur Umsetzung standen bereits fest, während alles andere innerhalb weniger Tage umgesetzt werden musste. Wir begannen also Montag mit der Konkretisierung der Idee und dem Festhalten eines genaueren Planes. Zudem programmierten wir die Einlesefunktion und die Aufteilung der eingegeben EBNF in die einzelnen Zeilen und Zeichen. Des Weiteren begannen wir mit der Programmierung der Überprüfung auf simple Syntax-Fehler innerhalb der EBNF. Dazu legten wir den allgemeinen Aufbau der EBNF im Programmcode fest, sodass beispielsweise ein Gleichheitszeichen oder Semikolon erkannt werden konnte. Außerdem klärten wir die exakte Herangehensweise bezüglich der Kontrolle von Klammersetzung oder dem Prüfen von Inhalten.
Am Dienstag konnten wir dann die Syntax-Fehler-Überprüfung fertigstellen und begannen erste Möglichkeiten für das Übertragen der Grammatik zu diskutieren.
Den Mittwoch nutzten wir dann dazu, unseren Plan für die Wortprüfung und die dafür notwendige Übertragung der Grammatik zu konkretisieren und die Umsetzung festzulegen.
Donnerstag beschäftigten wir uns mit dem Schreiben des Parsers und den Problemen, die durch Rekursionen entstehen können.
Am Freitag konzentrierten wir uns anschließend hauptsächlich auf das Schreiben dieser Arbeit und der Ausformulierung von unseren Gedanken zu diesem Projekt.

3. Warum haben wir dieses Projekt gewählt?

Die Idee zu diesem Projekt kam uns ziemlich spontan: Nachdem wir im letzten Halbjahr im Informatikunterricht formale Grammatiken (und besonders die EBNF) behandelt haben und dabei auch Wörter auf ihre Zugehörigkeit zu einer EBNF prüfen sollten, überlegten wir, ob dies auch automatisierbar wäre. Anstatt also mittels eines Ableitungsbaumes die Wörter zu überprüfen, wollten wir versuchen, diese Aufgabe, welche in ähnlicher Form aus informatischer Sichtweise sehr wichtig sein kann, von einem selbstgeschriebenen Programm erledigen zu lassen. Als wir uns genauer mit der Planung des Programms beschäftigt hatten, merkten wir schnell, dass Aufwand und Komplexität ein einfaches Schulprojekt o.ä. übersteigen würden, weshalb wir auf die Softwarechallenge von Inverso gestoßen sind und uns kurzerhand dazu entschlossen haben, noch an diesem Wettbewerb teilzunehmen.

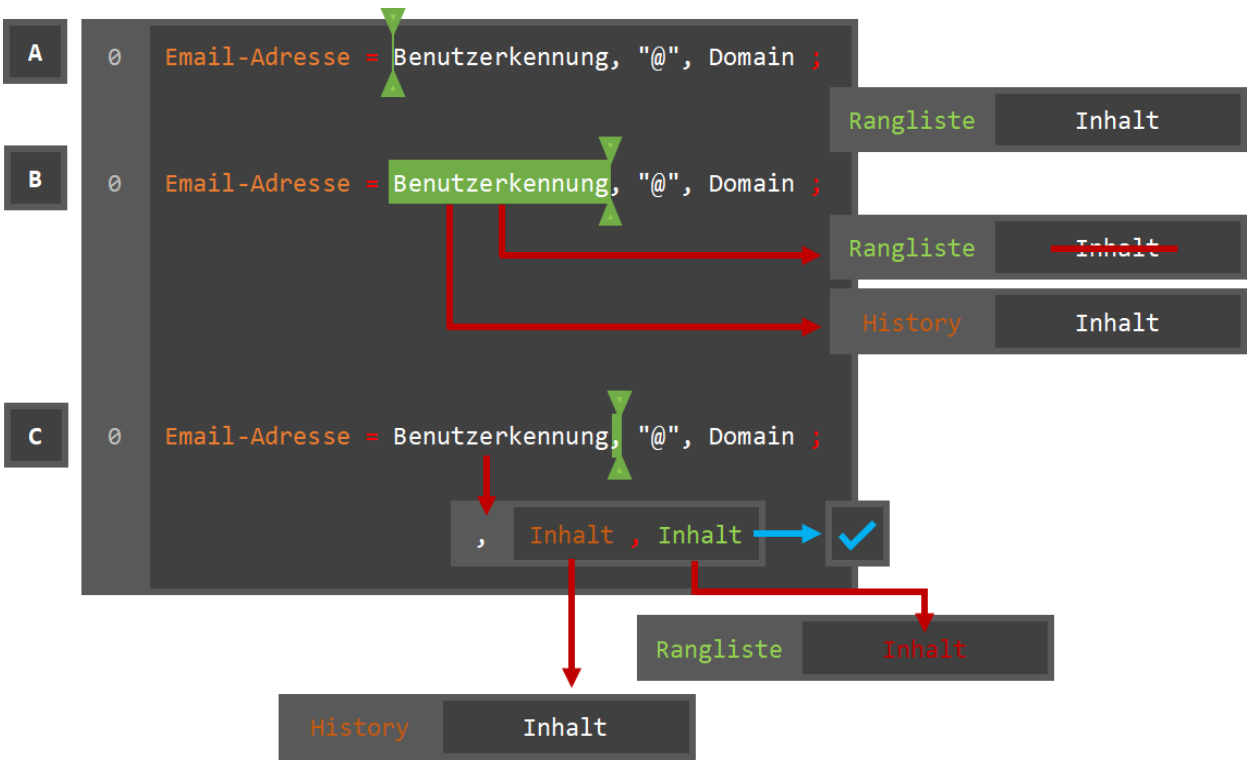
4. Funktionsweise und Funktionsumfang

Die Funktionsweise unseres Programmes lässt sich folgendermaßen beschreiben:



Die Umsetzung der Programmschritte lässt sich dabei im Grunde in drei Hauptprobleme aufteilen:

Erstens, das Schreiben eines Recognizers, der die per .txt-Datei eingegebene formale Grammatik auf die Notationsregeln der EBNF prüft und falls Fehler aufgetreten sind, dem Nutzer sagt, wo dies der Fall ist:
Dazu gibt es die Arrays "Rangliste" und "History". In der Rangliste steht, was das Programm aktuell von der eingegebenen Grammatik erwartet. So sieht man in der untenstehenden Grafik, dass bspw. ein beliebiger Inhalt erwartet wird. Dieser wird mit dem Nichtterminal "Benutzererkennung" erfüllt. Somit wird der erwartete Inhalt aus der Rangliste gestrichen, da nun prinzipiell die Ableitung zu Ende sein könnte (es fehlt natürlich noch ein Semikolon). In der History wird aufgelistet, welche Stücke der Ableitung bereits erkannt wurden, wie z.B. ein Inhalt. Dies ist nützlich, wenn bspw. eine Konnotation geprüft wird, da man so "zweiseitig" prüfen kann, ob auf beiden Seiten des Kommas ein Inhalt steht.

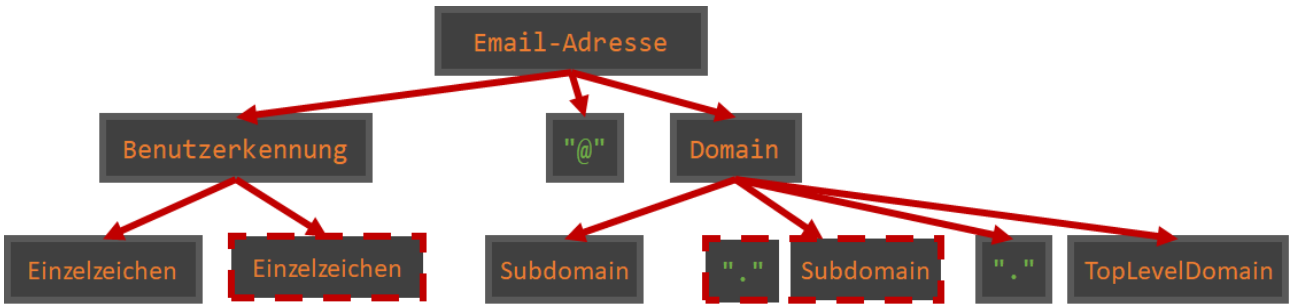


Zweitens, das Schreiben eines Parsers, der die nun überprüfte Grammatik in ein weiter verarbeitbares Format bringt. Also wird für jedes Nichtterminal ein Objekt der Klasse Nichtterminal angelegt und dieses in die Ableitungshierarchie einfügt. Ein besonderes Problem stellt hier bereits der Umgang mit Rekursionen dar, da diese die Hierarchie theoretisch ins Unendliche verlängern würden.
Drittens, das Überprüfen des vom Nutzer eingegebenen Wortes, also ob dieses nach den Regeln der Grammatik gültig ist.:

Hierbei legten wir zuerst, für jedes Nichtterminal ein Objekt an, welches danach um seine Ableitungen, z.B. andere Nichtterminalobjekte ergänzt wurde. Aus diesen erstellten wir dann eine Hierarchie in Form eines multidimensionalen Arrays, welches dann für jedes Nichtterminal ein Subarray änthält, bis nur noch Terminale in den Subarrays enthalten sind oder die notwendigen Nichtterminalobjekte, welche auf Rekursionen hindeuten, mit welchen dann im Späteren Programmverlauf umgegangen wird.

Grammatik.txt

```
Email-Adresse = Benutzerkennung, "@", Domain ;
Benutzerkennung = Einzelzeichen, { Einzelzeichen } ;
Einzelzeichen = Buchstabe | Ziffer | "-" | "_" | "." | "!" ;
Ziffer = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
Buchstabe = "a" | ... | "z" ;
Domain = Subdomain, { ".", Subdomain }, ".", TopLevelDomain ;
TopLevelDomain= Buchstabe, Buchstabe, [Buchstabe], [Buchstabe] ;
Subdomain = (Buchstabe | Ziffer), { Buchstabe | Ziffer | "-" },
(Buchstabe | Ziffer) ;,
```



Drittens, das Überprüfen des vom Nutzer eingegebenen Wortes, also ob dieses nach den Regeln der Grammatik gültig ist:

> Hierfür nutzen wir die soeben beschriebene Hierarche und schauten uns die einzelnen "Endzweige" der Hierarche an, ob die entsprechenden Zeichen des Wortes, mit den Terminalen übereinstimmen. War dies der Fall so gingen wir zum Nächsten über, war dies nicht der Fall, so war das Wort nicht zulässig.

5. Einsatz-Szenarien und Nutzen

Genutzt werden kann dieses Programm prinzipiell immer dann, wenn man eine formale Grammatik überprüfen möchte. Besonders um lästige Syntax-Fehler zu erkennen und zu verbessern. Zudem ist vor allem die Wortprüfung an vielen Stellen sinnvoll, wenn man beispielsweise ein Passwort auf seine Gültigkeit überprüfen möchte. Da dieses Programm eigentlich universell auf beliebige formale Grammatiken anwendbar ist, sind hier keine Grenzen gesetzt. Auch wenn es natürlich noch Fehlerquellen gibt, hat das Projekt definituiv Einsatzmöglichkeiten.

6. Ausblick

Unser Projekt bietet grundsätzlich noch viele Möglichkeiten zur Verbesserung. Besonders die Nutzeroberfläche ist definitiv noch ausbaufähig und könnte noch "ansehnlicher" und einfacher gestaltet sein. Zudem könnte durch Erweiterungen dem Nutzer die Eingabe erleichtert werden, indem man z.B. festlegt, dass mit "A"|"..."|"Z" das gesamte Alphabet gemeint ist. Dadurch müsste der Nutzer nicht alle 26 Buchstaben einzeln in die Grammatik eintippen und würde eine Menge Zeit sparen. Auch die Fehlermeldungen könnten noch verbessert werden, indem man genauer erklärt, was falsch ist und eventuell schon Verbesserungsvorschläge anbietet. Das Programm steht somit also noch nicht am Ende seiner Entwicklung und bietet uns noch Möglichkeiten zu Verbesserungen (z.B. bis zur Stufe 2 des Wettbewerbs).

https://github.com/BenjaminBak/EBNF-Check