JOHANNES KEPLER
UNIVERSITÄT LINZ

Eingereicht von

Daniel Binder

Angefertigt am

Institut für
Wirtschaftsinformatik -
Information
Engineering

Betreuer / Betreuerin
**o.Univ.-Prof. Dipl.-Ing.
Dr.Dr.h.c. Hanspeter
Mössenböck**

Juni 2021

GRAPHBASED SYNTAX ANALYSIS

Dynamische Syntaxgraphen in tabellengesteuerten Parsern

Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Science (BSc)

im Bachelorstudium

Informatik

**JOHANNES KEPLER
UNIVERSITÄT LINZ**
Altenberger Straße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITÄT LINZ

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Bachelorarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Mauthausen, 30.05.2021

INHALTSVERZEICHNIS

Kurzzusammenfassung

Das Verändern von Grammatiken in Parsern während der Laufzeit erleichtert die Weiterentwicklung und das Zusammenspiel von Programmiersprachen und da es auch mit modernen Werkzeugen oft schwierig ist, Grammatiken während der Laufzeit zu verändern, wurde in dieser Arbeit eine Methode vorgeschlagen, die das erleichtert. Es wird mit einem Lexer eine attributierte Grammatik gelesen aus der ein Graph erzeugt wird. Dieser Graph ist ein dynamisches Objekt und kann somit jederzeit während der Laufzeit verändert werden. In diesem Graph können auch Codeknoten eingebaut sein, welche semantische Aktionen in Form von Java-Code erlauben. Das Abgehen des Graphen passiert in mehreren frei wählbaren Modulen mit jeweils unterschiedlicher Funktionalität u.a. Prüfung auf LL(1)-Konformität, Graphzeichnung (d.h. Zeichnen der Knoten und der Struktur der Grammatik) und Syntaxprüfung mit Ausführung der semantischen Java-Aktionen. Bei der Syntaxprüfung muss ein Lexer mit Programmcode gefüttert werden, der dann zusammen mit dem Graphen an die Syntaxprüfung übergeben wird. Die Modulstruktur ist erweiterbar gestaltet und verwendet grundsätzlich das „Visitor“-Muster um die Komplexität der Erweiterungen zu reduzieren. Als konzeptioneller Beweis wurde die Methode mit der grundlegenden Funktionalität ausprogrammiert, inklusive einiger Tests. Um zu zeigen, dass Turing-Vollständigkeit gegeben ist, wurde ein „Glider“ in Conway’s Game of Life erfolgreich simuliert.

1. Einführung und Motivation

In vielen Bereichen gibt es Dokumente die nach genau definierten Regeln aufgebaut sind (z.B. Telefonbücher, XML/HMTL-Code, Fragebögen, Tests, Behördliche Dokumente, etc.). Diese Dokumente müssen oft auch elektronisch verarbeitet werden. Dazu kann ein Parsergenerator verwendet werden, der die strukturierte Information in Programmiersprachen-Objekte umwandelt, ausgibt, oder nur die Struktur der Information ändert. Der Parsergenerator wird verwendet um den Aufwand und die Komplexität der Implementierung zu reduzieren, indem lediglich die Grammatik (der Aufbau des zu lesenden Dokuments) und der auszuführende Code (was bei Erkennung eines Musters geschehen soll) anzugeben ist.

So können auch beim Lesen von Programmcode in Programmiersprachen Parsergeneratoren eingesetzt werden. Für die Erweiterung der in der Programmiersprache verwendeten Regeln muss oft der ganze Parser neu erzeugt werden und da die Parser oft eine hohe Komplexität aufweisen wäre eine Erweiterung wahrscheinlich umständlich und schwierig. Besonders das Zusammenspiel mehrerer Programmiersprachen ist oft nicht nur für den Ersteller der Sprache(n), sondern auch für einen Programmierer der beide Sprachen kennt, schwierig.

Dabei soll diese Arbeit Abhilfe schaffen, indem der Parser dynamisch (also während der Laufzeit) erweiterbar gestaltet ist. Diese Erweiterung geschieht mit einer Kombination aus EBNF-Grammatikregeln und Java-Code, welcher ausgeführt wird, sobald ein Regel-Muster erkannt wurde. So könnten etwa auch Programmierbibliotheken den bestehenden Parser erweitern, um dem Programmierer aufwendige Programmkonstrukte zu ersparen oder um die natürliche Entwicklung der Programmiersprache voranzutreiben.

2. Hintergrund

Die Erklärungen in diesem Kapitel sind, sofern keine anderen Quellen angegeben sind, **Teile** einer stark vereinfachten und zusammengefassten Version der Vorlesung „Übersetzerbau“ von o.Univ.-Prof. Dipl.-Ing. Dr.Dr.h.c. Hanspeter Mössenböck [1].

2.1. Begriffserklärungen

Java und **Python** sind häufig verwendete Programmiersprachen.

Eine **Methode** ist programmiertechnisch wie auch in der Sprache eine bestimmte Vorgehensweise wie ein Problem gelöst wird. In Programmiersprachen sind Methoden durch einen Namen und einen Körper definiert. Der Körper beschreibt dabei den Lösungsvorgang.

Tokens bestehen aus einem Tokencode und einem Tokenwert. Der Token**code** ist eine Verallgemeinerung der Art der Zeichen. So können Zahlen etwa unter dem Tokencode „number“ zusammengefasst werden oder Pluszeichen unter dem Tokencode „plus“. Der Token**wert** sind die tatsächlichen Zeichen, die im jeweiligen Token zusammengeführt wurden, sofern es mehrere mögliche Zeichen für ein Token gibt. So kann z.B. ein „number“-Token den Wert „7“ oder „25“ haben, ein „plus“-Token kann jedoch nur aus einem „+“-Zeichen bestehen, wodurch eine Speicherung des Wertes unnötig wäre.

Ein **Lexer/Scanner** wandelt einen Zeichenstrom in einen Tokenstrom um. Der Tokenstrom erleichtert die Weiterverarbeitung, da zusammengehörende Zeichen zu Tokens zusammengefasst und kategorisiert werden. So wird z.B. eine Aneinanderreihung von Ziffern eine Zahl, welche, ohne zusätzliche Verarbeitungsschritte, auch als solche verwendet werden kann.

Syntax ist die Anordnung von Elementen z.B. die Anordnung von bestimmten Wortarten in einer Sprache. **Semantik** ist hingegen die Bedeutung der Sprache. Hier ist Semantik die Gesamtheit der Dinge die bei Erkennung eines Regelmusters geschehen soll.

Grammatiken sind Regeln, die den Aufbau einer Sprache beschreiben. In Grammatiken gibt es **Terminalsymbole**, die nicht weiter zerlegt werden können (hier vergleichbar mit Tokencodes), **Nonterminalsymbole**, die in Nonterminal- sowie Terminalsymbole zerlegt werden können, **Produktionen**, die bestimmen wie ein Nonterminalsymbol zerlegt werden kann und ein **Startsymbol**, welches das Äußerste (im Syntaxbaum oberste) Nonterminalsymbol ist.

Attributierte Grammatiken sind Grammatiken, die Zusatzinformationen darüber enthalten, was bei Erkennung eines Regelmusters geschehen soll. Das kann z.B. ausführbarer Programmcode sein, der durch einen Parsergenerator in den fertigen Parser eingefügt wird.

Der **Parser** erstellt aus einem Tokenstrom einen Sytaxbaum. Der **Syntaxbaum**, bestehend aus Syntaxknoten, ist eine hierarchische Struktur und stellt eine weitere Abstraktionsstufe dar, ähnlich wie die Abstrahierung von Zeichen und Zeichenketten zu Tokens. **Syntaxknoten** fassen Tokens oder weitere Syntaxknoten zusammen. Die Art der Zusammenfassung basiert auf der Grammatik, den Regeln der Sprache. So könnten z.B. in einem Telefonbuch ein „plus“-Token gefolgt von einem „number“-Token zu einem „Telefonnummer“-Syntaxknoten zusammengefasst werden.

Als **Programmcode** wird hier ein Text bezeichnet, der bestimmten Grammatikregeln folgt. Die Syntaxprüfung überprüft, **ob** er den Regeln folgt und die in den Regeln definierte Semantik entscheidet, was geschieht, sobald ein bestimmter Regelteil betreten wird.

2.2. Statische und dynamische Compilerstrukturen

Bei Compilern gibt es statische Elemente, die schon beim Erstellen des Compilers bekannt sind, und dynamische, die sich je nach Benutzereingabe verändern. Um den Unterschied zwischen dem vorgeschlagenen Prozess und existierenden Werkzeugen zu verdeutlichen, werden folgend typische statische und dynamische Strukturen erklärt.

Der Übersetzungsvorgang beginnt mit der lexikalischen Analyse (auch Scanning genannt), die einen vom Benutzer zur Verfügung gestellten Zeichenstrom entgegennimmt, und daraus einen Tokenstrom erzeugt. Die Tokencodes sind dabei fix definiert, wohingegen der Tokenwert von der Eingabe abhängt, was auch den Tokenstrom zu einer dynamischen Struktur macht. Beim Parsing wird der Syntaxbaum aufgebaut und es wird außerdem dynamisch eine Symbolliste erstellt, welche Variablen, Objekte und deren Eigenschaften speichert. Der Syntaxbaum kann explizit (es existieren Knotenobjekte im Programm) sowie implizit (durch die Strukturierung des Parsers – d.h. er existiert nur im Arbeitsspeicher und ist auf Objektebene nicht sichtbar) aufgebaut werden. Nun wird der Syntaxbaum abgegangen und ausführbarer Code erzeugt.

2.3. EBNF-Grammatiken

EBNF-Grammatiken (EBNF = Erweiterte Backus-Naur Form) sind eine Sonderform von Grammatiken, die bestimmte Konventionen einhalten. So werden etwa Nonterminalsymbole mit Groß- und Terminalsymbole mit Kleinbuchstaben geschrieben. Produktionen bestehen immer aus einer linken und einer rechten Seite, welche von einem „=“-Zeichen getrennt wird. Die Produktion wird immer mit einem „.“-Zeichen abgeschlossen. Zeichen oder Zeichenfolgen die direkt in die Regel übernommen werden sollen (ohne Teil der EBNF-Schreibweise zu sein), werden in Anführungszeichen geschrieben. Weiters gibt es einige Zeichen, welche für bestimmte Abarbeitungsmuster verwendet werden:

Der „|“-Operator trennt Alternativen, d.h. je nach Eingabe kann entweder der Linke ODER der Rechte Teil abgearbeitet werden. Die Gruppierung von Alternativen wird durch Einhüllen mit runden Klammern dargestellt z.B. „(„+“ | „-„)“. Optionale Konstrukte werden mit eckigen Klammern eingehüllt und sich wiederholende Konstrukte mit geschwungenen. Ein optionales Konstrukt kann null- oder einmal begangen werden, ein sich wiederholendes Konstrukt null-, ein- oder mehrmals. Hierbei ist anzumerken, dass alle Klammerkonstrukte in Alternativenketten umgewandelt werden können (bei geschwungenen Klammern wären das jedoch unendlich viele).

Ein Beispiel für eine einfache EBNF-Grammatik könnte wie folgt aussehen:

Telefonbuch	= { Telefonnummer }.
Telefonnummer	= („+“ Ländervorwahl | „0“) Nummer.
Ländervorwahl	= ziffer ziffer.
Nummer	= ziffer ziffer ziffer ziffer ziffer ziffer ziffer [ziffer].

Das Telefonbuch besteht aus null oder mehr Telefonnummern.

Eine Telefonnummer besteht aus einem „+“-Zeichen gefolgt von einer Ländervorwahl, welche aus zwei Ziffern besteht, oder einer Null. **Beiden** Optionen folgt eine Nummer, die entweder sieben- oder achtstellig sein kann. Ziffern sind hierbei s.g. Terminalklassen die eine Menge bestimmter Terminalsymbole zusammenfassen (hier: die Ziffern 0-9).

2.4. Übersicht über Syntaxanalyseverfahren

Grob kann zwischen Top-down- und Bottom-up Syntaxanalyse unterschieden werden.

Die **Top-down Syntaxanalyse** beginnt beim äußersten Symbol (das Startsymbol - im Baum das oberste) und verläuft nach folgendem Algorithmus:

1. Das Nonterminalsymbol wird in dessen Produktion zerlegt.
2. Das erste noch nicht verwendete Symbol von links wird betrachtet.
3. Ist das Symbol ein Nonterminalsymbol, gehe zu 1.
4. Ist das Symbol ein Terminalsymbol, wird überprüft, ob die Eingabe mit diesem übereinstimmt.
5. Stimmt es nicht überein und es gibt eine Alternative auf gleicher Ebene, springe zur Alternative und gehe zu 1.
6. Stimmt es nicht überein, wirf einen Fehler.
7. Stimmt es überein, lese das nächste Symbol und gehe zu 1.
8. Gibt es kein Symbol mehr, gehe eine Ebene nach außen zu dem Symbol welches sich nach dem aktuell bearbeiteten Nonterminalsymbol befindet.
9. Gibt es kein äußeres nächstes Symbol, terminiere.

Dies ist eine vereinfachte Form des Algorithmus der nur Grammatiken verarbeiten kann, deren terminale Anfangssymbole bei Alternativenketten paarweise disjunkt sind, d.h. wenn der Algorithmus zu einer Alternative kommt, muss eindeutig klar sein, in welchen Zweig der Alternative er einsteigt. Es ist hierbei jedoch zu beachten, dass es auch bei Wiederholungen und Optionen zu Alternativen kommt, da diese (wie oben erwähnt) in funktionsmäßig gleichwertige Alternativenketten umgewandelt werden können. Weiterhin muss die Grammatik zyklenfrei sein, d.h. es darf sich keine Regel selbst beinhalten, ohne dass vorher mindestens ein Terminalsymbol gelesen wurde, da dies zur unendlichen Ausführung derselben Regel führen kann. Außerdem muss jedes Nonterminalsymbol direkt oder indirekt in eine Kette von Terminalsymbolen ableiten lassen. Sind alle diese Eigenschaften erfüllt, nennt man eine Grammatik „LL(1)-Konform“.

Beispiele für Produktionen die **nicht** LL(1)-Konform sind:

Vorbereitung = „schneide“ „Zwiebeln“ | „schneide“ „Fleisch”.

In beiden Alternativen kommt “schneide“ als erstes. Der Compiler kann sich nicht entscheiden, welche Option gewählt werden soll.

Zweig = „Frucht“ [„Blatt“] „Blatt“. oder mit beliebig vielen Blättern:

Zweig = „Frucht“ { „Blatt“ } „Blatt“.

In beiden Fällen könnte am Zweig nur ein Blatt hängen, aber das weiß der Compiler nach dem obigen Algorithmus nicht, wodurch er sich entscheiden müsste ob er in die Option bzw. in die Wiederholung einsteigt.

Haus = Untergeschoss Obergeschoss.

Untergeschoss = „Raum“ [„Raum“].

Obergeschoss = „Raum“.

Obwohl das Ober- und Untergeschoss einzeln betrachtet LL(1)-Konform sind, ist durch die Zusammenführung in ein Haus ein LL(1)-Konflikt entstanden. Sobald der Compiler den zweiten Raum liest, müsste er sich entscheiden, ob der Raum der optionale Raum des Untergeschosses ist, oder ob der Raum zum Obergeschoss gehört.

Papier = Zeichnung.

Zeichnung = Tisch.

Tisch = Papier | „Stift“.

Das Papier enthält eine Zeichnung, die Zeichnung einen Tisch und der Tisch ein Stück Papier (auf dem wiederum eine Zeichnung zu finden ist). Der oben angegebene Algorithmus läuft ewig im Kreis.

Bild = „Lampe“ „Tisch“ Wand.

Wand = Bild.

Auf dem Bild sind eine Lampe, ein Tisch und eine Wand zu sehen. Diese Wand beinhaltet jedoch ein weiteres Bild auf dem man dasselbe sieht. Ist die Eingabe unendlich lang, terminiert der Algorithmus nicht, obwohl kontinuierlich Tokens gelesen werden.

Wieso würde man so einen Algorithmus verwenden, wenn er mit vielen Grammatiken nicht umgehen kann?

- Durch richtiges Umformen können viele LL(1)-Konflikte entfernt werden ohne die Funktion der Grammatik zu verändern.
- Eine LL(1)-konforme Grammatik kann mit dem rekursiven Abstieg schematisch implementiert werden. Das „1“ in LL(1) steht für ein Vorgriffssymbol, d.h. es kann ein Symbol nach vorne gesehen werden um zu entscheiden in welche Option eingestiegen wird. Ein LL(2)-fähiger Parser könnte zwar das oben angegebene Ober-/Untergeschoss-Problem lösen (da er sieht, ob noch ein dritter Raum existiert oder nicht), doch die Komplexität der Implementierung würde erheblich steigen.

Beim rekursiven Abstieg werden die linken Seiten der Produktionen in Methoden umgewandelt. Diese Methoden arbeiten die rechten Seiten Schritt für Schritt (in der Produktion von links nach rechts) ab. Ist ein Nonterminalsymbol an der Reihe wird die entsprechende bereits existierende Methode aufgerufen. Kommt eine Option, wird mit dem Vorgriffssymbol überprüft, in welchen Zweig der Option eingestiegen werden soll. Ähnlich ist es bei optionalen oder sich widerholenden Konstrukten. Kommt ein Terminalsymbol wird dieses in der Eingabe gelesen. Passt das Terminalsymbol nicht zur Eingabe und es gibt auch keine Option dessen Terminalsymbol zur Eingabe passt, wird ein Fehler geworfen.

Die **Bottom-up Syntaxanalyse** beginnt hingegen mit den einzelnen Tokens und fasst diese zu Nonterminalsymbolen zusammen, wodurch der Syntaxbaum von unten aufgebaut wird.

Bottom-up-Parser können 4 Aktionen ausführen:

- Shift: Lies das nächste Terminalsymbol und speichere es in eine Liste.
- Reduce: Nimm vom Ende der Liste die nötigen Symbole und fasse sie zu einem Nonterminalsymbol zusammen, welches an die Liste angereiht wird.
- Accept: Terminiere.
- Error: Keine weitere Aktion durchführbar. Wirf einen Fehler.

In einer Tabelle werden auf einer Achse alle Zustände und auf der anderen alle Terminal- und Nonterminalsymbole gelistet. Nun kann durch Eintragen der Aktionen in bestimmten

Tabellenteilen die Grammatik verändert werden. Auch für die Erstellung der Tabelle gibt es einen Algorithmus:

Zuerst muss zwischen Kern und Hülle unterschieden werden. Der Kern sind alle Produktionen die gerade abgearbeitet werden. Die Stelle an der sich der Parser in der Produktion befindet, wird hier mit einem „.“-Zeichen dargestellt. Die Hülle sind der Kern und alle Zustände die direkt aus dem Kern abgeleitet werden können. ‚Direkt‘ bezieht sich hierbei auf die aktuelle Position des Parsers – d.h. die rechten Seiten der Produktionen der Nonterminalsymbole nach dem „.“-Zeichen werden der Teil der Hülle.

Bei der Tabellenerzeugung wird nun für jeden noch nicht abgearbeiteten Zustand die Hülle erzeugt. Ist eine Produktion in der Hülle bereits vollständig abgearbeitet („.“-Zeichen nach der Produktion), müssen das Letzte/die Letzten gelesenen Symbole zu einem Nonterminalsymbol zusammengefasst (reduziert) werden. Daher wird an dieser Stelle in der Tabelle eine „Reduce“-Anweisung eingefügt. Ist die Produktion jedoch noch nicht vollständig abgearbeitet, wird das nächste (Terminal- oder Nonterminal-) Symbol gelesen („Shift“-Anweisung) und in einen anderen Zustand gesprungen, der neu hinzugefügt wird wenn er noch nicht existiert. Dieser Zustand beinhaltet sinngemäß die Produktion mit dem weitergerückten „.“-Zeichen. Sollte der Zustand neu erstellt worden sein, wird wieder die Hülle erstellt und der Algorithmus beginnt von vorne. Wird die Startproduktion akzeptiert, ist die „Accept“-Anweisung statt einem „Reduce“ zu verwenden. Die „Error“-Fälle sind alle in der Tabelle bisher nicht befüllten Zustände, da keine Regel die in der Grammatik bekannt ist zu so einer Kombination aus Zustand und Eingabesymbol führen sollte.

Der Hauptvorteil von Bottom-up Parsern ist, dass Optionen parallel abgearbeitet werden und somit nicht nur LL(1)-konforme Grammatiken verwendet werden können. Weiters bleibt der Parser bei jeder Grammatik gleich - nur die Tabelle verändert sich. Diese Tabellen brauchen jedoch verhältnismäßig viel Speicher und sind händisch schwer zu konstruieren, weshalb zur Konstruktion üblicherweise Werkzeuge verwendet werden.

2.5. Übersicht über Parsergeneratoren

Aus den genannten Algorithmen entstanden automatisierte Werkzeuge, die einen (Scanner und) Parser aus einer Grammatik erzeugen – Parsergeneratoren. Generatoren nehmen allgemein eine Grammatik entgegen und generieren daraus einen Scanner/Parser. Nachfolgend sind nun einige gelistet und erklärt:

Coco/R erzeugt einen Scanner, und einen Parser nach rekursivem Abstieg aus einer Scannerbeschreibung und einer attributierten Grammatik. Die Scannerbeschreibung besteht dabei aus Einstellungen wie z.B. ob zwischen Klein- und Großschreibung unterschieden werden soll, und einer Beschreibung der Tokenklassen (d.h. welche Zeichenfolgen zu einem bestimmten Tokencode gehören). Die attributierte Grammatik besteht aus einer EBNF-Grammatik und ausführbarem Programmcode (typischerweise Java Code) an den entsprechenden Stellen in den Regeln.

Yet Another Compiler-Compiler, kurz **YACC**, ist ein Parsergenerator, der tabellengesteuerte Parser basierend auf einer Grammatik, attributiert mit C Code, erzeugt. Mit speziellen Zeichen kann im C Code auf den einen Stack (ein Stapel an Daten) des Parsers zugegriffen werden, wodurch Information gespeichert und gelesen werden kann. Die Grammatik kann jedoch nicht in der erweiterten Backus-Naur-Form (**E**BNF) geschrieben werden, sondern muss in BNF verfasst sein [2].

Ähnlich wie Coco/R erzeugt auch **JavaCC** einen Parser nach dem rekursiven Abstieg. Der Parser kann an bestimmten Stellen weiter als ein Vorgriffssymbol vorausschauen, wodurch er an diesen Stellen ein LL(k)-Parser ist. In allen anderen Stellen ist es ein LL(1)-Parser aus Performanzgründen. Wie auch Coco/R erlaubt JavaCC EBNF-Grammatiken [3].

ANTLR generiert aus LL(k) Grammatiken Parser-Bäume (verschachtelte Terminal- und Nonterminalknoten) und einen „Walker" der diesen Baum abgeht, um anwendungsspezifischen Programmcode auszuführen [4]. Die Beschreibungen der Tokenklassen sind in ANTLR Teil der EBNF-Grammatik und nicht wie z.B. bei Coco/R Teil einer eigenen Scannerbeschreibung. Außerdem ist es möglich den erzeugten Knotenbaum anzuzeigen [5].

3. Graph-basierte Syntaxanalyse

Wird bei den bestehenden Werkzeugen ein Parser nach rekursivem Abstieg erzeugt, kann die Grammatik ohne Neuübersetzung des Parsers nicht mehr verändert werden und bei tabellengesteuerten Parsern ist die Tabellenerstellung nicht intuitiv, wodurch die Erweiterung meist nur durch Werkzeuge möglich ist.

3.1. Idee

Um diese Probleme zu beheben, wird in dieser Arbeit ein tabellengesteuerter Parser vorgeschlagen, bei dem die Tabelle durch einen Graphen der Grammatik ersetzt wird. Ein Graph ist leichter zu verstehen und daher leichter zu erweitern und bestehende Grammatikregeln können an einem Graphen direkt abgelesen werden.

3.2. Aufbau des Syntaxgraphen

Der Graph besteht aus Knoten die die Information der Regeln (Terminal- und Nonterminalsymbole) speichern. Der Graph wird aufgebaut, indem Symbole zu Knoten auf gleicher logischer Ebene (d.h. die maximale Anzahl von Knoten um die man ein Paar aus runden Klammern setzten könnte, ohne die EBNF-Grammatik zu verändern, wodurch jedes existierende EBNF-Klammerkonstrukt automatisch auf eine logische Ebene gehört) zusammengefasst werden. Aufgrund der Unterschiedlichkeit an Information die zu speichern ist, gibt es mehrere Arten von Knoten. Dies erlaubt außerdem den Aufbau des Projekts in hintereinanderschaltbare Module, was später näher erläutert wird. Die Knotenarten werden überwiegend aus EBNF-Konstrukten wie z.B. Wiederholungen übernommen. Es wird hier zwischen folgenden Knoten unterschieden:

Symbolknoten stellen Terminalsymbole dar. Jeder Symbolknoten enthält ein Terminalsymbol.

Referenzknoten stellen Nonterminalsymbole dar. Sie enthalten den Namen des Nonterminalsymbols d.h. den Namen der Grammatikregel.

Optionsknoten beinhalten alle Optionen auf gleicher logischer Ebene. Es muss in eine der Optionen eingestiegen werden.

Sequenzknoten beinhalten alle Knoten auf gleicher logischer Ebene. Die gespeicherte Information sind nicht nur die Knoten selbst, sondern auch deren Reihenfolge.

Optionalknoten enthalten einen beliebigen anderen Knoten, in den aus Parser-Sicht eingestiegen werden kann, aber nicht muss.

Iterationsknoten enthalten einen beliebigen anderen Knoten in den null-, ein- oder mehrmals eingestiegen werden kann.

Präzedenzknoten sind Zusammenfassungen von Optionen. Sie beinhalten einen beliebigen anderen Knoten.

Codeknoten enthalten Java-Code, der an dieser Stelle im Regelmuster ausgeführt werden soll.

Es könnten auch noch Regelknoten verwendet werden, die Produktionen darstellen und ein Grammatikknoten, der alle Regeln zu einer Grammatik zusammenfasst, doch dies war für die Implementierung nicht nötig, da auf die Regeln sowieso mit dem Namen zugegriffen werden muss (bei Aufruf von Nonterminalsymbolen), wodurch die Verschachtelung in einem eigenen Knoten nur zusätzlichen Aufwand beim Auspacken bedeuten würde.

Die verschiedenen Knoten werden aus der EBNF-Grammatik abgeleitet und ineinander verschachtelt.

3.3. Beispiele

Nachfolgend werden beispielhaft einige EBNF-Produktionen gelistet und wie sie in Knoten umgewandelt werden.

Stall = „Huhn“ „Huhn“ „Kuh“.

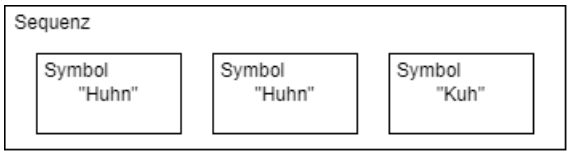

Abbildung 1: Stall-Graph, Sequenz aus Symbolknoten

Die Wörter „Huhn“ und „Kuh“ werden in einen Symbolknoten gepackt. Da alle Symbolknoten auf gleicher Ebene sind, können sie in einen Sequenzknoten zusammengefasst werden.

Auto = „Vordertür“ „Vordertür“ [„Hintertür“ „Hintertür“].

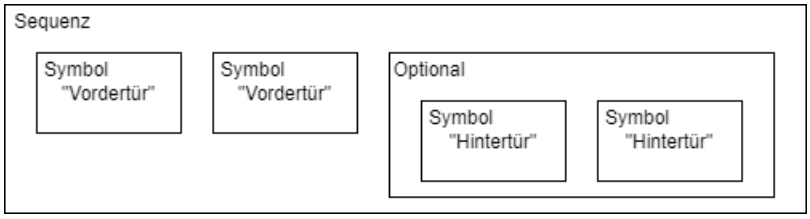

Abbildung 2: Auto-Graph, Optionsknoten in Sequenz

In Abb. 2 sind die Hintertüren in der Grammatik optional, weswegen sie sich in einem Optionalknoten befinden. Der Optionalknoten befindet sich wiederum in dem Sequenzknoten.

Klassenbuch = { Vorname Nachname }.

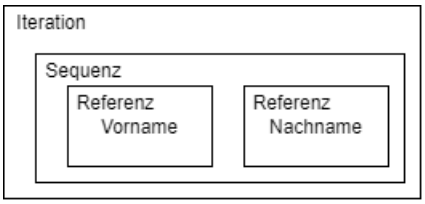

Abbildung 3: Klassenbuch-Graph, Iterations- und Referenzknoten

Das Klassenbuch besteht aus einer beliebigen Anzahl von Vor- und Nachnamen. Vor- und Nachname sind eine zusammengehörende Sequenz, welche beliebig oft vorkommen kann, weshalb sich der Sequenzknoten in einem Iterationsknoten befindet.

Esszimmer = (Digitaluhr | Analoguhr) „Tisch“ „Stuhl“ [Sitzpolster] { „Stuhl“ [Sitzpolster] }.

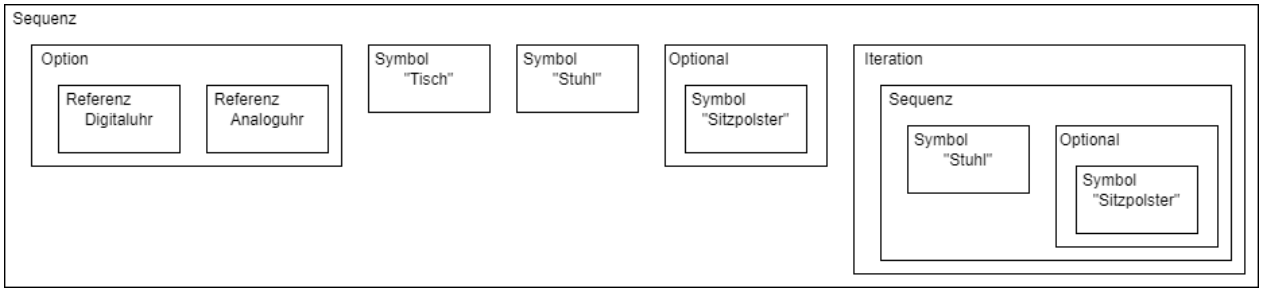

Abbildung 4: Esszimmer-Graph, mehrfache Verschachtelungen

Das „Esszimmer“ in Abb. 4 veranschaulicht die Verschachtelung mehrerer Elemente. Der Raum hat **entweder** eine Analog- **oder** eine Digitaluhr und mindestens einen Tisch mit Stuhl und optionalem Sitzpolster. Weiters können beliebig viele Stühle mit (optional) Sitzpolstern im Raum sein.

3.4. Dynamische Erweiterbarkeit

Der Graph wird vor der Ausführung des Programmcodes aufgebaut und während der Ausführung des Programmcodes abgegangen indem die Terminalsymbole des Graphen mit den Tokens des Programmcodes schrittweise verglichen werden. Er befindet sich als dynamisches Objekt im Programmspeicher. So kann jederzeit eine Regel (oder ein Knoten) hinzugefügt, verändert oder entfernt werden. Dies könnte jedoch zu ungewolltem Verhalten führen, da sich der Leser (der auf die aktuelle Stelle im Graph zeigt) während der Ausführung des Programmcodes an einer beliebigen Stelle im Graph befinden könnte – so auch in dem Teil, der gerade verändert wird. Schlimmstenfalls ist der Leser in einem Graphteil „gefangen“ der nicht mehr zur Grammatik gehören sollte.

4. Implementierung

Um zu zeigen, dass die oben beschriebene Methodik auch implementiert werden kann, wurde als konzeptioneller Beweis ein Programm erstellt, welches die grundlegenden Funktionen beinhaltet.

Alle Einstellungen, die über das gesamte Programm gleich bleiben müssen (z.B. Zeichen für Kommentare, Schlüsselwörter, etc.) werden in einem Konfigurationsobjekt („Config“) gespeichert. Die darin enthaltenen Einstellungen können von außen gelesen und auch überschrieben werden.

4.1. Architektur

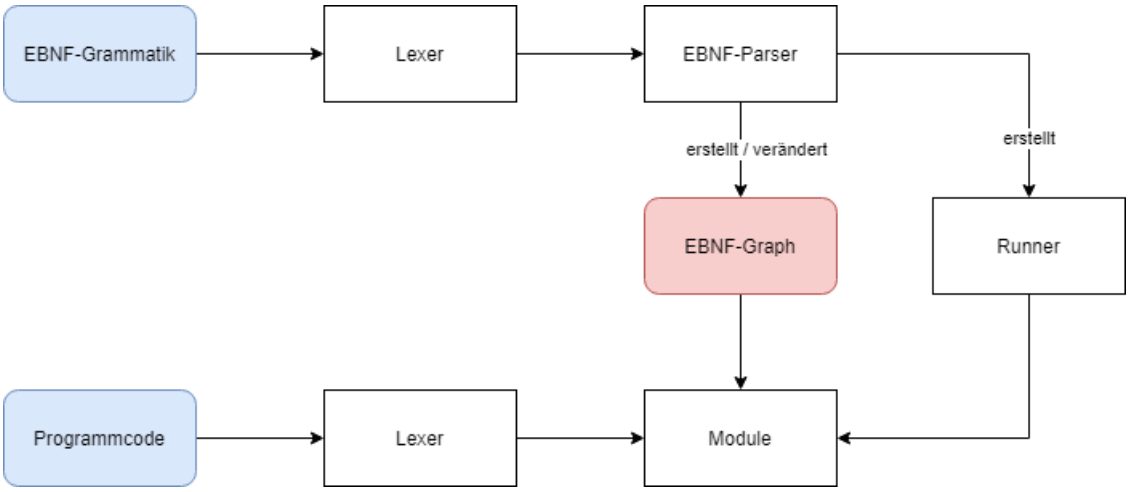

Abbildung 5: Datenfluss

In Abb. 5 ist der Datenfluss zu erkennen. Die EBNF-Grammatik wird in den Lexer gespeist, welcher wiederum vom EBNF-Parser entgegengenommen wird. Der EBNF-Parser verwendet den Lexer um die die Tokens der Grammatik der Reihe nach einzulesen und erstellt daraus den EBNF-Graphen (wie oben dargestellt) und einen „Runner“, der Java-Code, der sich in der Grammatik befindet, ausführt (dazu später mehr). Der Programmcode wird (wie die Grammatik) auch an einen Lexer übergeben. Dieser Lexer wird zusammen mit dem erzeugten Graphen von beliebigen Modulen verwendet, durch die der Graph geschleust werden kann. Module könnten z.B. eine LL(1)-Prüfung, ein Zeichnen des Graphen, oder das Ablaufen des Graphen (inklusive Ausführung des entsprechenden Runner-codes) sein.

4.2. EBNF-Parser

Da die Grammatik von EBNF-Grammatiken, also die Regeln wie eine EBNF-Grammatik auszusehen hat, in LL(1)-konformem EBNF darstellbar ist, wurde der EBNF-Parser im rekursiven Abstieg geschrieben.

```
1   grammar      = {rule} EOF.
2   rule         = IDENTIFIER '=' rhs.
3   rhs          = factor {factor}.
4   factor       = [semantic] primary [semantic]. {'|' [semantic] primary [semantic]}
5   semantic     = '#' code '#'.                  // code = java code that is copied into Runner.java
6   primary      = IDENTIFIER
7                | NUMBER ['>']                  // '>' = load number onto stack at runtime
8                | STRING ['>']                  // '>' = load string onto stack at runtime
9                | SYMBOL ['>']                  // '>' = load symbol onto stack at runtime
10               | '[' rhs ']'
11               | '{' rhs '}'
12               | '(' rhs ')'.
```

Abbildung 6: LL(1)-konforme Grammatik von EBNF-Grammatiken [adaptiert von 6]

In Abbildung 6 ist die EBNF-Grammatik zu erkennen nach der der rekursive Abstieg ausprogrammiert ist. Um die Erweiterung der Grammatik im Programmcode zu ermöglichen wurde ein Sonderzeichen mit einem darauffolgenden Befehl eingeführt, der den EBNF-Parser steuert. So kann z.B. mit „$ READ_EBNF [EBNF-Grammatik]“ eine Grammatik eingelesen werden. Zusätzlich gibt es für Änderungen noch den Befehl „$ ADD_RULE [EBNF-Regel]“ der eine einzelne Regel einliest und zum Graphen hinzufügt, und „$ REMOVE_RULE [Regelname]“

der eine Regel anhand des Namens löscht. Das Ändern einer Regel könnte durch Löschen der originalen und Hinzufügen der geänderten Version der Regel erreicht werden.

4.3. Graphanalyse

Der Graph, der vom EBNF-Parser erzeugt wird, kann nun in einer Reihe von Modulen durchgelaufen werden.

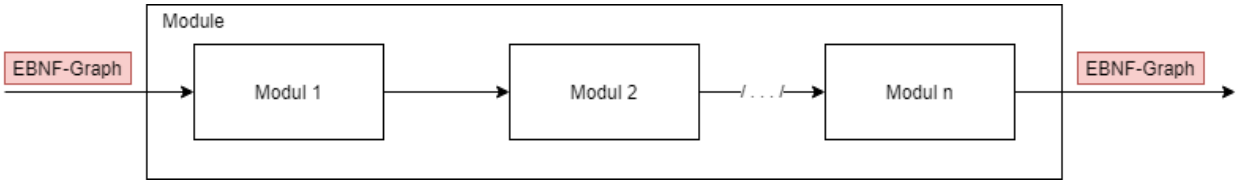

Abbildung 7: Module

In Abb. 7 ist eine Kette von beliebigen Modulen zu sehen, durch die der Graph durchgeschleust wird. Die Auswahl und Reihenfolge der Module ist dem Benutzer überlassen. Implementiert wurden als konzeptioneller Beweis eine LL(1)-Prüfung, eine Syntaxprüfung, und ein Graphzeichner.

Um das Hinzufügen von Modulen so einfach wie möglich zu gestalten, wurde das „Visitor“-Muster angewendet, d.h. ein Programmierer kann in einem Modul den „EBNFNodeVisitor“ implementieren und muss sich keine Gedanken über die Zerlegung des Graphen machen, lediglich darüber, wie jeweils einer der bestimmten Knoten zu behandeln ist um die gewünschte Funktion auszuführen.

```
1   @Override
2   public void visit(EBNFReferenceNode n) {
3       graph.getRules().get(n.getReference()).accept(this);
4   }
```

Abbildung 8: Syntaxprüfung eines Referenzknotens

In Abb. 8 ist beispielhaft eine vereinfachte Implementierung der Syntaxprüfung von Referenzknoten zu sehen. Die Referenz selbst ist nur der Name, den das Nonterminalsymbol hat. Dieser Name kann in allen Regeln gesucht werden.

Kommt das Programm zu einem Referenzknoten wird diese Methode ausgeführt. Es wird von allen Regeln „getRules()“ des Graphen, diejenige ausgesucht, deren Name („n.getReference()“) im Referenzknoten „n“ gespeichert ist. Da in die gefundene Regel auch eingestiegen werden muss, wird „accept(this)“ aufgerufen und das Muster unterscheidet automatisch (je nachdem, welcher Knoten in der Regel enthalten ist) welche Methode als nächstes ausgeführt werden soll.

Die LL(1)-Prüfung erfolgt ohne Implementierung des EBNFNodeVisitors, da die meisten Knoten nur durchgeschleust werden. Stattdessen bietet es sich an, die verschiedenen Knotenebenen mit Rekursion (d.h. die Methode ruft sich selbst auf) zu durchlaufen, um Zyklenfreiheit und die Verschiedenheit von zwei möglicherweise aufeinander folgenden Zeichen zu prüfen.

Die Module geben (unabhängig davon, ob der Graph verändert wurde oder nicht, als Konvention) wieder den Graphen zurück, um die in Abb. 7 dargestellte Kette aufrecht zu erhalten.

4.4. Semantikanschluss

Semantische Aktionen werden direkt in die Grammatik eingearbeitet. Wie in Abb. 7 Zeile 4 zu sehen ist, besteht ein Faktor aus einem „primary“ und optional davor oder danach semantischen Aktionen. Diese Aktionen, deren Beginn und Ende in der Grammatik mit einem „#“-Symbol markiert sind, sind Java-Code der 1:1 in eine Datei „Runner.java“ kopiert wird und an dieser Stelle in der Grammatik (d.h. wenn ein Programmstück diese Stelle im Muster erreicht) ausgeführt wird.

```
1  public class Runner {
2      public static final Stack<String> stack = new Stack<>();
3
4      public static void execute(int number) {
5          switch(number) {
6              case -1:
7                  System.out.println("DUMMY CODE");
8                  break;
9              default:
10                 throw new IllegalArgumentException("Number '" + number + "' has no executable code saved!");
11         }
12     }
13 }
```

Abbildung 9: Runner.java

In Abb. 9 ist die originale Datei „Runner.java“ zu sehen. Sie beinhaltet einen globalen Stack auf den primitive Daten als Text gespeichert werden können. Auf diesen Stack werden auch Nummern, Text und Symbole gespeichert, nach denen in der EBNF-Grammatik ein „>“-Symbol steht (siehe Abb. 7 Zeilen 7, 8, 9). Wird in der EBNF-Grammatik ein „#“-Zeichen erkannt, holt sich der EBNF-Parser eine einzigartige Nummer vom Konfigurationsobjekt. Diese Nummer wird dann in einem Codeknoten gespeichert, der im Graphen an der entsprechenden Stelle angehängt wird. Die selbe Nummer wird auch in der Switch-Anweisung (Abb. 9 Zeilen 5-11) verwendet, um zwischen den Codestücken zu unterscheiden. Im Runner ist angeführt, wie so ein Codestück aussehen kann. Gibt der Benutzer an einer Stelle im Code z.B. die Semantikaktion „System.out.println(„DUMMY CODE“);“ an und der EBNF-Parser bekommt vom Konfigurationsobjekt die Zahl „-1“, würden die Zeilen 6 bis (inkl.) 8 aus Abb. 9 in den Runner kopiert werden. Da der Kopiervorgang jedoch während der Laufzeit statt findet, und die ursprüngliche „Runner.java“-Datei ohne diesen Code kompiliert worden ist, muss der Code erst kompiliert werden. Danach wird ein neues Runner-Objekt (welches den zusätzlichen Code enthält) erstellt und in die Konfigurationsdatei geschrieben damit es auch von anderen Codeteilen gefunden und verwendet werden kann. Sobald ein Modul im Graph nun auf einen Codeknoten stößt, ruft es beim Runner der Konfigurationsdatei die Methode „execute(number)“ (Abb. 9 Zeile 4) mit der Nummer aus dem Codeknoten auf und die semantische Aktion wird ausgeführt.

4.5. Erweiterbarkeit der Grammatiken

Der Graph ist ein dynamisches Objekt im Speicher. So kann jederzeit mit der „add(regelName, graphKnoten)“ eine neue Regel hinzugefügt und mit „remove(regelName)“ eine Regel entfernt werden. Das ändern einer Regel erfolgt über Entfernen der zu ändernden und Hinzufügen der geänderten Version. Die Regeln selbst sind in einer HashMap gespeichert, bei der der Schlüssel der Regelname ist. Der Graph verwaltet nur diese HashMap der Regeln (ein „Wrapper“).

4.6. Limitationen

Die oben vorgeschlagene Architektur bringt auch Nachteile mit sich:

- Das Abgehen des Graphen ist nicht so schnell wie z.B. bei einem Parser im rekursiven Abstieg oder einem tabellengesteuerten Parser.

- Bei jeder semantischen Erweiterung muss die „Runner.java“-Datei erneut kompiliert werden.

Aufgrund von limitierten Ressourcen wurde an folgenden Stellen nur die nötige Funktionalität ausprogrammiert:

- Der Lexer erkennt auch Leerzeichen, Einrückungen und Zeilenenden, damit auch Sprachen wie z.B. Python implementiert werden könnten. Die Erkennung dieser Zeichen kann nicht deaktiviert werden, weswegen diese oftmals nachträglich herausgefiltert werden müssen.
- Der EBNF-Parser kann nur Regeln aus einer Grammatik hinzufügen oder entfernen, nicht ändern. Dazu wäre ein Abgehen des Graphen und Vergleichen mit der neuen Regel nötig. Weiterhin müsste, um dies zu ermöglichen, eine Konvention für Regeländerungen eingeführt werden.
- Das „$“-Zeichen kann an bestimmten Stellen in der Grammatik nicht verwendet werden, da es als Steuerzeichen für den Einstieg in den EBNF-Parser vorbestimmt ist.
- Es gibt kein Modul, welches Syntaxprüfung und EBNF-Parser vereint, wodurch eine Vermischung von Grammatik und Programmcode in einem Dokument nicht möglich ist. Trotzdem kann der Graph über den EBNF-Parser (oder „händisch“ über das Graph-Objekt) vor, nach, oder während der Ausführung eines Modules jederzeit verändert werden.
- Es gibt keine Prüfung, ob sich der Leser eines Modules im aktuellen Graph befindet. Dieser könnte ohne Fehler in einer ausgetauschten Regel ewig weiterlaufen. Das führt auch dazu, dass das Austauschen eines Knotens in dem sich der Leser befindet nicht ratsam ist.
- Für die Attributierte Grammatik, sowie die entstandene Sprache gibt es keine Syntaxhervorhebung.
- Das Programm muss manchmal zweimal ausgeführt werden, damit der Runner alle Zeilen Code beinhaltet.
- Es wird bei fehlerhaftem Programmcode keine Fehlerbehandlungstrategie verfolgt.
- Es gibt für den entstandenen Graphen kein Optimierungsmodul und der generierte Graph ist nicht immer minimal.
- Der relative Pfad zur kompilierten „Runner.java“ wurde ausprogrammiert zu dem Ordner in dem sich die „Runner.class“-Datei befindet.
- Tests wurden auf grundlegende Funktionalität und einige Sonderfälle beschränkt.
- Variablen die über mehrere Codeknoten hinweg verwendet werden, müssen in der Konfigurationsdatei definiert werden. Da die „execute(int number)“-Methode statisch ist, müssen auch diese Variablen statisch sein.

5. Evaluierung

5.1. Implementierung

Insgesamt wurden 26 Klassen, 1 Enum und 1 Interface erstellt. 6 der 26 Klassen sind reine Testklassen, 9 Klassen werden für Knoten (1 davon eine abstrakte Superklasse) verwendet und 3 Klassen für die Beispielimplementierungen der Module.

Klasseninformation

Paket	Klassenname	Typ	Codezeilen
config	Config	Class	104

graph	EBNFGraph	Class	54
graph.node	EBNFGraphNode	abstract Class	17
	EBNFCodeNode	Class	50
	EBNFIterationNode	Class	41
	EBNFOptionalNode	Class	45
	EBNFOptionNode	Class	58
	EBNFPrecedenceNode	Class	41
	EBNFReferenceNode	Class	41
	EBNFSequenceNode	Class	58
	EBNFSymbolNode	Class	58
lexer	Lexer	Class	444
	Token	Class	65
	TokenType	Enum	22
parser	EBNFParser	Class	272
	Runner	Class	18+
modules	EBNFNodeVisitor	Interface	15
	LL1Check	Class	157
	SyntaxVisitor	Class	184
	GraphDrawVisitor	Class	145
utils	Error	Class	22
	GraphPanel	Class	36
test	LexerTest	Class	269
	EBNFParserTest	Class	30
	LL1CheckTest	Class	287
	SyntaxVisitorTest	Class	60
	GameOfLifeTest	Class	100
	GraphPanelTest	Class	62

Tabelle 1: Klasseninformationen

Insgesamt befinden sich 2755 Zeilen in der aktuellen Version des Projektes. Die Anzahl der Zeilen im Programm kann durch die Nutzereingabe steigen, da möglicherweise Code (von semantischen Aktionen) in die „Runner.java“ kopiert wird.

5.2. Beispiele

Im SyntaxVisitorTest wurde eine einfache Grammatik implementiert, die eine Kette von Additionen und Subtraktionen berechnen und ausgeben kann. Die Grammatik sieht wiefolgt aus:

```
1  Program = Calc Print.
2  Calc = number {('+'|'-') number}.
3  Print = "print".
```

Abbildung 10: Grammatik eines einfachen Rechners

Die Grammatik wurde mit semantische Aktionen versehen und wiefolgt in die in Abb. 6 gezeigte Pipeline geschickt:

```
$ READ_EBNF Program          = Calc Print.
Calc                         =
                               number >
                               { ( '+' >
                                 | '-' >
                                 )
                                 number >      #String num2 = stack.pop();
                                                int result = stack.pop().equals("+") ?
                                                Integer.valueOf(stack.pop()) + Integer.valueOf(num2) :
                                                Integer.valueOf(stack.pop()) - Integer.valueOf(num2);
                                                stack.push(result + ""); #
                               }.
Print                        = "print"        #System.out.println(stack.pop()); #.
```

Abbildung 11: Einfacher Rechner mit semantischen Aktionen

Aus der Eingabe in Abb. 11 erstellte der EBNF-Parser wie zu erwarten 3 Regeln mit insgesamt 14 Knoten. Jeweils 3 Knoten für die Regeln „Program“ und „Print“ und 8 Knoten für die Regel „Calc“. Die Erstellung des Graphen (inklusive Durchlaufen des Lexers) dauerte 0,703300ms und die Übersetzung der semantischen Aktionen im Runner 882,478200ms.

Als Beweis, dass das Programm eine Sprache erzeugen kann, die Turing-Vollständig, d.h. fähig ist, einen modernen Computer zu simulieren, wurde eine Grammatik für „Conway‘s Game of Life“ implementiert, welches Turing-Vollständig ist. Um ein Turing-Vollständiges System zu simulieren, muss das simulierende System auch Turing-Vollständig sein.

```
Program = Size ';' Build ';' Run.
Size = number number.      // 2D size
Build = Num { Num }.       // initial board state
Num = number.
Run = number.              // number of iterations
```

Abbildung 12: Grammatik von Conway’s Game of Life

In Abb. 12 ist eine mögliche Grammatik für Conway’s Game of Life zu sehen. Diese Grammatik führte zu insgesamt 23 Knoten mit 6 Knoten für „Program“, 5 Knoten für „Size“, 6 Knoten für „Build“, 3 Knoten für „Num“, und 3 Knoten für „Run“. Die Grapherstellung dauerte 19,917500ms und die Übersetzung des Runner-codes 944,029800ms.

Zu bemerken ist, dass die Grapherstellung bei Conway’s Game of Life um mehr als das 20-fache länger gedauert hat, obwohl nur ungefähr zweimal so viele Knoten erzeugt wurden. Dies könnte auf die unterschiedliche Grammatik-Zeichenlänge zurückzuführen sein, da die Grammatik von Conway’s Game of Life mehr semantische Aktionen beinhaltet. So werden 13 Zeilen Grammatik (siehe Abb. 11) für den einfachen Rechner und 55 Zeilen für Conway’s Game of Life benötigt. Die Übersetzung der „Runner.java“ dauert hingegen kaum länger als die Übersetzungszeit des einfachen Rechners.

6. Erkenntnisse und offene Arbeit

6.1. Eckpunkte und Erkenntnisse

Das Projekt begann mit der Überlegung, dass es in der Programmiersprache Python gängige Praxis ist, Rückgabewerte in ein anonymes Tupel einzuwickeln, wenn es mehrere für eine Methode gibt. In Java hingegen muss eine eigene Klasse (mit dem „new“-Operator) erstellt

werden um die gleiche Funktionalität zu erzielen. Eine Erweiterung der Sprache Java in so eine Richtung würde die Grundphilosophie von Java verletzen, die Einführung eines eigenen Rückgabeobjekts bringt jedoch zusätzliche, möglicherweise unnötige Komplexität mit sich. Weiters gibt es verschiedene Versionen von Programmiersprachen, was zu Kompatibilitätsproblemen oder Einschränkungen im Sprachenentwurf (zum Erhalt einer Abwärtskompatibilität) führen könnte. Man kann auch nicht ohne weiteres die Sprache wechseln, da das Zusammenspiel von verschiedenen Sprachen meist aufwendig und komplex ist. Der Lösungsansatz des Autors war hier, eine universelle Sprache zu erstellen und diese an den nötigen Stellen „einfach zu verändern“ d.h. die Grammatik während der Ausführung anzupassen. Dieses Ziel wurde in dem Projekt eindeutig erreicht, da der Graph, der die Grammatik darstellt ein dynamisches Objekt ist, was heißt es ist jederzeit veränderbar.

Das gesamte Projekt oder Teile, die nicht wie erwartet funktionierten oder eine suboptimale Architektur hatten, wurden im Sinne der „agilen Softwareentwicklung“ mehrmals von Grund auf neu erstellt. Zuerst wurde in 8 Stunden ein Prototyp gebaut, der aus einer EBNF-Grammatik einen Graph erstellen und eine Syntaxprüfung durchführen konnte. Die Architektur des Lexers schien in Ordnung, aber es war nur möglich Sprachen zu erzeugen, die keine Zeilenenden, Leerzeichen oder Tabulatoren in der Grammatik berücksichtigen müssen. Die im EBNF-Parser implementierte Grammatik der EBNF-Grammatik funktionierte bei einigen Grammatikkonstrukten nicht wie erwünscht, da sie nicht LL(1)-konform und somit schwerer zu programmieren war. Ein Beispiel für unerwünschtes Verhalten war die unnötige Erstellung von Sequenzen, da diese sonst an manchen Stellen fehlen würden und es umständlich gewesen wäre alle Sonderfälle auszuprogrammieren um die Knoten richtig anzuordnen. Es gab in dieser Version auch noch Regelknoten, Grammatikknoten und einen EOFKnoten (EOF = End of File = Ende der Datei). Anstatt einem Graphen wurde der Grammatikknoten zurückgegeben. Somit musste sich der Anwender vollständig um die Graphverwaltung kümmern und die Graphklassen auch vollständig verstehen um sie zu erweitern. Auch wenn der Prototyp bis auf den Lexer vollständig verworfen wurde, führte er zu wichtigen Erkenntnissen, die in den nächsten Versionen eingearbeitet wurden.

In der zweiten Version wurde zunächst der Lexer erweitert. Dieser erkannte nun auch Zeilenenden, Leerzeichen und Tabulatoren. Die Anzahl an Einrückungen wurde unter einem neuen Token „indent“ zusammengefasst, um die Weiterverarbeitung zu erleichtern. Der Lexer konnte nun auch verschiedene vordefinierte Nummernsysteme (binär, ternär, oktal, hexadecimal und dozenal) einlesen. Die aktuelle Idee im Lexer war, vordefinierte Symbole zu erlauben, damit der Nutzer diese nur „zusammenstecken“ muss um die Grammatik zu erzeugen. Dies führte jedoch dazu, dass zusammengesetzte Symbole wie „+=“ oder „++“ als zwei verschiedene Symbole gelesen wurden. Dafür sah der Autor zwei Lösungswege. Ein „No-Whitespace“-Symbol, welches zwischen den beiden Symbolen eingefügt werden kann, um bei der Syntaxprüfung auszuschließen, dass dazwischen ein Leerzeichen ist, oder dem Grammatikersteller durch semantische Aktionen die Arbeit der sauberen Prüfung zu überlassen. Beide Ansätze bringen Nachteile mit sich und erhöhen die Komplexität bei Implementierung und Verwendung des Projekts. Der EBNF-Parser wurde angepasst – zwar nicht auf eine LL(1)-konforme Grammatik, aber eine Grammatik, die funktionierte. Es wurden weiterhin verschiedene Befehle zum Steuern des EBNF-Parsers hinzugefügt. Damit die Kette auch bei Fehlern aufrecht erhalten bleibt, wurde ein Errorknoten erstellt und bei Fehlern in den entsprechenden Stellen im Graphen eingefügt. Dies führte jedoch dazu, dass die Fehlerknoten in jedem weiteren Kettenglied behandelt oder in irgend eine andere Form umgewandelt werden mussten. Zu diesem Zeitpunkt wurde auch eine virtuelle Maschine für die Ausführung von einem eigenen „Bytecode“-ähnlichen Code erstellt. Alle Funktionalitäten dieses Codes mussten zuerst einprogrammiert und getestet werden, was die

Erstellung einer mächtigen und gleichzeitig einfach zu benutzenden Sprache praktisch unmöglich machte. Die Syntaxprüfung wurde in dieser Version nach einer nichtdeterministischen Zustandsmaschine (NDFSM) konzeptioniert. Die Idee war, dass ein Leser bei zwei gleich aussehenden Graphzweigen den ersten Zweig betritt und bei einem Fehler zu letzten Abzweigung zurückgeht und den zweiten Zweig betritt, usw.. So ein Leser müsste jedoch auf den aktuellen Graphteil zeigen um ihn effektiv verwenden zu können und da keine Information bezüglich übergeordneter Knoten in den Graphknoten vorhanden war, war ein Zurückgehen ohne Umstrukturierung des EBNF-Parsers nicht möglich, weshalb diese Idee in der nächsten Version verworfen wurde. Es wurden auch erstmals Testklassen für alle Programmteile erstellt.

In der dritten Version erhielt der Lexer eine öffentliche Schnittstelle, die dessen Verwendung vereinfachte. Weiters wurde das „Wrapper“-Muster angewandt, um einen Lexer zu erstellen, der im Eingabetext automatisch Zeichen (wie z.B. Tabulatoren) austauscht. Die Grammatik des EBNF-Parsers wurde zu einer LL(1)-konformen Version umgeändert und die Errorknoten wurden entfernt. Die Pakete wurden neu strukturiert. Die einzige Hürde für den Nutzer des Projekts schien das umständliche Zusammenstecken der vorgefertigten Teile zu sein, was zur vierten und vorerst letzten Version des Projekts führte.

In der aktuellen Version wurden die vorgefertigten Tokensymbole entfernt. Stattdessen können beliebige und beliebig lange Symbole und Symbolketten mit Anführungszeichen in der Grammatik verwendet werden. Anstatt einen eigenen Bytecode zu erstellen, wurde Java-Code verwendet, der in einen Runner kopiert wird. Als Abschluss wurde noch Conway’s Game of Life als Grammatik implementiert und ein s.g. „Glider“ hineingeschickt. Der Glider ist ein Muster, welches sich diagonal im Spielfeld fortbewegt und dabei immer wieder die gleiche Form annimmt.

```
1  0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2  0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3  1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
4  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
5  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
6  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
7  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
8  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
9  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
11 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
12 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
13 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
14 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
15 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
16 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
17 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
18 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
19 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
20 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Abbildung 13: Glider nach 0 Generationen (Zeilen 1-3)

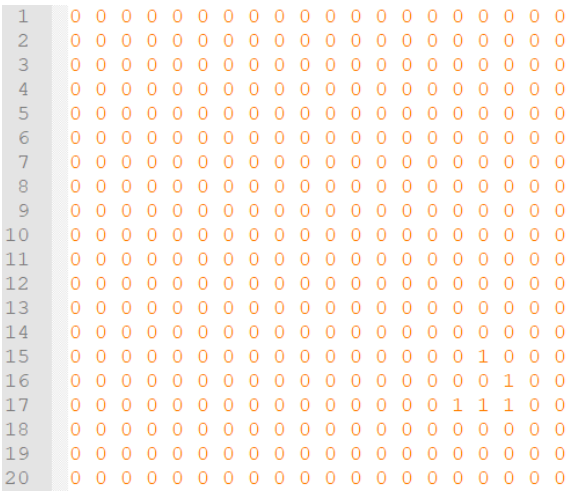

Abbildung 14: Glider nach 58 Generationen (Zeilen 15-17)

In Abb. 13 ist die Eingabe zu sehen und in Abb. 14 die Ausgabe nach 58 Genrationen. Die gesamte Grammatik und die gesamte Testeingabe sind im Anhang in Abb. 15 und Abb. 16 zu sehen.

Der Entwicklungsprozess des Verwerfens und Neuprogrammierens unter Beachtung der gewonnenen Erkenntnisse hat gut funktioniert. So konnte die gesamte Projektarchitektur mehrmals verändert werden, um dem gewünschten Ergebnis immer näher zu kommen. Suboptimale Konstrukte wurden so auch nicht in die nächste Version weitergetragen und umständliches Anpassen wurde durch vereinfachte Funktionalität mit Aussicht auf Lösung des Problems in der nächsten Version ersetzt.

6.2. Offene Erweiterungen

In allen Teilen des Projekts gibt es Möglichkeiten zur Erweiterung und/oder Verbesserung.

Der Lexer erkennt Leerzeichen, Tabulatoren und Zeilenenden, was in vielen Fällen nicht notwendig ist. Entweder der Lexer wird so erweitert, dass er bei bestimmter Konfiguration diese Zeichen ignoriert, oder der Lexer wird in einer eigenen Lexerbeschreibung frei definiert (ähnlich wie bei Coco/R). Eine eigene Lexerbeschreibung würde zwar dem Benutzer mehr Freiheit lassen, jedoch müsste sich dieser die Fähigkeit aneignen so eine Beschreibung zu verfassen.

Ein Ändern von Grammatikregeln ist derzeit nur über Entfernung und Hinzufügen einer geänderten Version der Regel möglich. Ein Ändern könnte implementiert werden, indem entweder eine weitere Option zu einer Regel hinzugefügt werden kann (d.h. ist der oberste Knoten eine Option, wird der neue Teil einfach zur Option hinzugefügt, und ist es keine Option, wird eine Option erstellt, welche der neue Regelknoten ist, zu der der alte Regelknoten und der neue Teil auf gleicher Ebene hinzugefügt werden), oder der neue Teil der Regel wird mit der alten Grammatik verglichen (durch Abgehen des aktuellen Graphen) und wo der neue Teil nicht vorhanden ist, wird er hinzugefügt. Ähnlich könnte so auch das Entfernen von Regelteilen gelöst werden.

Der aktuelle EBNF-Parser kann nur das „$“-Zeichen zur Steuerung verwenden. Eine einfache Erweiterung wäre es, das Steuerzeichen in der Konfigurationsdatei zu definieren, damit es dort bei Bedarf verändert werden kann.

Um den Graphen auch im Programmcode verändern zu können (vereinfacht die Zusammenführung von mehreren Sprachen und die Erweiterung der aktuellen Sprache) kann ein neues Syntaxmodul entwickelt werden, welches durch bestimmte Zeichen an bestimmten Teilen

den EBNF-Parser verwendet um den Graphen zu erweitern. Dafür müsste allerdings nach jeder hinzugefügten Regel mit Codeknoten die Runner.java neu übersetzt und eingebunden werden, damit der hinzugefügte Graphteil auch sofort verwendet werden kann. Dies ist jedoch höchst riskant, da sich der Leser in einem Graphteil befinden könnte, der durch die aktuelle Änderung betroffen ist. Sollten nur Graphteile hinzugefügt werden, könnten diese Probleme mit einer „import“-Sektion am Anfang aller Programmcode-dateien gelöst werden. So müssten nur alle diese Sektionen betrachtet werden und der Graph muss nur einmal erstellt, und die Runner.java nur einmal übersetzt und eingebunden werden. Dies löst auch das Problem des verlorengehenden Lesers. Sollten jedoch Teile des Graphen durch eine Erweiterung entfernt werden, ist dies nicht möglich. Stattdessen könnten diese Regelsätze mit Schlüsselwörtern ausgetauscht werden. Die Verwendung von Schlüsselwörtern muss dafür nicht unbedingt in ein Modul einprogrammiert werden, da diese Schlüsselwörter in manchen Fällen (wenn nur der dem Schlüsselwort untergeordnete Graphteil ausgetauscht wird) auch Teil der Grammatik sein könnten.

Sollte der Leser des Graphen verloren gehen, oder es besteht Gefahr dass er verloren geht, sollte es zumindest eine Warnung geben. Dies könnte durch eine Liste der besuchten Regeln gelöst werden. Beim Eintritt in eine Regel wird der Regelname an die Liste angehängt und beim Verlassen einer Regel wird das letzte Listenelement gelöscht. Eine Regel, die schon in der Liste ist, muss trotzdem hinzugefügt werden, da sonst beim Verlassen der Regel unklar ist, ob das Element gelöscht werden kann oder nicht. Der hierarchische Aufbau der Regeln erlaubt so, dass bei Änderungen nur geprüft werden muss, ob sich der Regelname der zu ändernden Regel in der Liste befindet, und wenn ja, ist die Änderung unsicher.

Um die Programmierung der EBNF-Grammatik zu vereinfachen wäre eine Syntaxhervorhebung von Vorteil. Ein Plugin der Programmierumgebung könnte eine Java-Syntaxhervorhebung dahingehend verändern, dass es auch die EBNF-Grammatik außerhalb der Codeknoten erkennt und entsprechend farblich kennzeichnet. Diese Kennzeichnung könnte auch zusammen mit einer Fehlerbehandlungsstrategie für die EBNF-Grammatik und den Programmcode eingeführt werden. Damit zusammenhängend wird auch aktuelle die Runner.class bei jedem Neuübersetzen der Runner.java gelöscht, ob diese kompilierbar ist oder nicht. Dies führt dazu, dass die Runner.class bei nicht übersetzbarem Code nicht existiert und das laufende Programm die benötigte Datei nicht findet, wodurch ein Fehler entsteht und die Runner.java manuell neu übersetzt werden muss. Mit einer Prüfung, ob der aktuelle Code übersetzbar ist, kann auch verhindert werden, dass die alte Runner.class-Datei gelöscht wird und dass das laufende Programm die Ausführung fortsetzt. Um die Programmierung weiter zu vereinfachen wurde der Pfad der kompilierten Runner.class ausprogrammiert und die Datei wird ersetzt (gelöscht und neu kompiliert).

Es gibt derzeit kein Werkzeug zur Optimierung des Graphen. So könnten besonders nach Änderungen ineffiziente Graphteile bestehen bleiben. Dieses Problem könnte in Form eines Moduls gelöst werden. So könnte z.B., wenn ein Sequenzknoten direkt einen anderen Sequenzknoten beinhalten, der Innere ausgepackt werden:

Sequenzknoten(A B Seqeuenzknoten(C D) E) wird zu Sequenzknoten(A B C D E).

Die vorhandenen Testklassen sind zwar mit Hilfsmethoden ausgestattet, die es erlauben, effizienter Tests zu schreiben, es existieren jedoch nicht genügend Tests um die fehlerfreie Funktion des Programms zu gewährleisten. Diese Tests könnten nach den vorhandenen Mustern um zusätzliche Testfälle erweitert werden.

Um Grammatiken wie Python zu verwenden, ist es nötig ein neues Syntaxanalysemodul zu schreiben, welches auch mit Einrückungen, Leerzeichen und Zeilenenden umgehen kann. Der

Lexer liefert diese Tokens bereits, aber sie werden im aktuellen Modul bewusst ignoriert um die Implementierung zu vereinfachen. Bei so einer Erweiterung wäre es möglicherweise sinnvoll die Berücksichtigung von solchen „leeren Zeichen“ ein bzw. ausschaltbar zu machen, damit kein zusätzlicher Aufwand entsteht, falls diese Funktionalität nicht benötigt wird.

Derzeit müssen Variablen, die über mehrere Codeknoten definiert werden in der Konfigurationsdatei manuell in einen String gespeichert werden, der global in den Runner kopiert wird, da sonst vom Java-Compiler der Fehler geworfen wird, dass die Variablen möglicherweise verwendet werden, bevor sie initialisiert worden sind (d.h. eine Variable wird verwendet, bevor sie einen Wert zugewiesen bekommen hat). Diese globalen Variablen könnten automatisch aus dem Java-Code in den Codeknoten entnommen werden.

Um das Projekt an die individuellen Bedürfnisse der Nutzer anzupassen, können verschiedene Module entwickelt werden. Ein Modul könnte zwischen verschiedenen Grammatiken übersetzten (z.B. Java und Python oder eine primitive Übersetzung von Deutsch auf Englisch), ein anderes könnte die grafische Bearbeitung und Programmierung des Graphen erlauben, ein weiteres die erzeugte Struktur (Knoten, Codezeilen, etc.) messen.

Der nächste Schritt wäre die vorsichtige Erstellung einer Programmiersprache, bei der die Möglichkeit von Grammatikerweiterungen im Design-Prozess berücksichtigt wird. In so einer Programmiersprache könnten Schlüsselwörter verwendet werden um frühere Versionen der Sprache sowie beliebige Versionen anderer Sprachen zu verwenden. Es könnten alle Programmiersprachen unter einer Sprache vereint werden und es würde dadurch keine Hürden bei der Verwendung von mehreren Sprachen in einem Projekt geben. Darüber hinaus entwickelt sich so eine Sprache selbst weiter, da Konstrukte nicht von den Sprachentwicklern einprogrammiert werden müssen, sondern von den Nutzern der Sprache geteilt werden können.

Die dynamische Erweiterbarkeit könnte auch in bestehenden Werkzeugen (zumindest in tabellengesteuerten Parsern) eingeführt werden.

6.3. Zusammenfassung der Ergebnisse

Die grundlegende Funktion der vorgestellten Methode wurde in ungefähr 2750 Zeilen gelöst, wobei die größten Teile der Lexer, EBNF-Parser und die drei implementierten Module sind. Diese Hauptklassen wurden in mehreren Phasen mehrmals komplett neu geschrieben, was dem Autor erlaubte, sich der genauen Funktionalität der einzelnen Klassen und dem Aussehen der nötigen Schnittstellen zwischen den Klassen heranzutasten. Manche Funktionen konnten auch auf spätere Versionen des Projekts verschoben werden, in denen diese vollständig benötigt werden bzw. in denen klarer ist, wie die Funktion möglichst leicht realisiert werden kann. Da das Ziel des Projekts ein konzeptioneller Beweis war und da die Ressourcen stark limitiert waren, gibt es auch viele offene Erweiterungen. Für die meisten Erweiterungen wurde ein Lösungsansatz bereitgestellt. Am wichtigsten wäre (nach Beurteilung des Autors) die Erweiterung der Testfälle, da diese noch nicht den gesamten Funktionsumfang ausreichend abdecken. Einer dieser Testfälle, Conway’s Game of Life (zu sehen im Anhang), stellt die Turing-Vollständigkeit der Sprache erfolgreich dar.

Zusätzliche Module sind eine einfache Möglichkeit die Funktionalität zu erweitern und können dank des „Visitor“-Musters auf Knotenebene programmiert werden anstatt auf Graphebene. Die natürliche Weiterentwicklung der Methode könnte zu einer eigens für die Erweiterbarkeit der Grammatik konzipierten Programmiersprache führen.

7. Literaturverzeichnis

[1] Hanspeter Mössenböck. (2020). *Vorlesung Übersetzerbau*. Persönliches Vorlesungsskriptum von H. Mössenböck, JKU, Linz, Österreich.

[2] Yacc. (2021, Juni 3). Von *Wikipedia*. http://en.wikipedia.org/wiki/Yacc.

[3] JavaCC | The most popular parser generator for use with Java applications. (2021, Juni 3). In *Github*. http://javacc.github.io/javacc.

[4] About the ANTLR Parser Generator. (2021, Juni 3). Von *Antlr*. http://antlr.org/about.html.

[5] Getting Started with ANTLR v4. (2021, Juni 3). In *Github*. http://github.com/antlr/antlr4/blob/master/doc/getting-started.md.

[6] rici (stackoverflow username). (2015, Oktober 27). Antwort auf “Ebnf – Is this an LL(1) grammar?“. In stackoverflow. http://stackoverflow.com/questions/20175248/ebnf-is-this-an-ll1-grammar.

8. Tabellenverzeichnis

9. Abbildungsverzeichnis

10. Anhang

```
Program          = Size ';' Build ';' Run.

Size             = number >
                   number >        #ySize = Integer.valueOf(stack.pop());
                                    xSize = Integer.valueOf(stack.pop());
                                    board = new String[xSize][ySize]; #
                   .

Build            =                 #x = 0; y = 0; #
                   Num { Num }.

Num              = number >        #board[x][y] = stack.pop();
                                    x = x + 1 >= xSize ? 0 : x + 1;
                                    y += x == 0 ? 1 : 0; #
                                 .

Run              = number >        #int iterations = Integer.valueOf(stack.pop());
                                    String[][] temp = new String[xSize][ySize];
                                    int nrOfNeighbours;
                                    while(iterations-- > 0) {
                                        for(x = 0; x < xSize; x++) {
                                            for(y = 0; y < ySize; y++) {
                                                nrOfNeighbours = 0;
                                                if(x - 1 >= 0 && board[x - 1][y].equals("1")) nrOfNeighbours++;
                                                if(y - 1 >= 0 && board[x][y - 1].equals("1")) nrOfNeighbours++;
                                                if(x + 1 < xSize && board[x + 1][y].equals("1")) nrOfNeighbours++;
                                                if(y + 1 < ySize && board[x][y + 1].equals("1")) nrOfNeighbours++;
                                                if(x - 1 >= 0 && y - 1 >= 0 && board[x - 1][y - 1].equals("1")) nrOfNeighbours++;
                                                if(x - 1 >= 0 && y + 1 < ySize && board[x - 1][y + 1].equals("1")) nrOfNeighbours++;
                                                if(x + 1 < xSize && y - 1 >= 0 && board[x + 1][y - 1].equals("1")) nrOfNeighbours++;
                                                if(x + 1 < xSize && y + 1 < ySize && board[x + 1][y + 1].equals("1")) nrOfNeighbours++;

                                                if(nrOfNeighbours < 2 || nrOfNeighbours > 3) temp[x][y] = "0";
                                                else if(nrOfNeighbours == 3) temp[x][y] = "1";
                                                else temp[x][y] = board[x][y];
                                            }
                                        }

                                        //copy temp into board
                                        for(int i = 0; i < temp.length; i++) {
                                            System.arraycopy(temp[i], 0, board[i], 0, temp[0].length);
                                        }

                                        //print board
                                        for(int i = 0; i < board.length; i++) {
                                            for(int j = 0; j < board[0].length; j++) {
                                                System.out.print(board[i][j] + " ");
                                            }

                                            System.out.println("");
                                        }

                                        System.out.println("---");
                                    } #
                   .
```

Abbildung 15: Attributierte Grammatik von Conway's Game of Life

```
20 20 ;
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
; 60
```

Abbildung 16: Testeingabe für "Glider" in Conway's Game of Life