ISA
Institut für
SoftwareArchitektur

THM
TECHNISCHE HOCHSCHULE MITTELHESSEN

Daniel Kirsten, Markus Bader

Vergleich von Parsern in Clojure und Java

12. Februar 2014

Technischer Bericht Nr. 4
des Instituts für SoftwareArchitektur

Zusammenfassung

Im Rahmen des Masterprojekts im Wintersemester 2013/2014 haben die Autoren ein Programm zum Anwenden von Logikfunktionen (z. B. Wahrheitstafeln, Tseitin–Transformation, SAT–Solving) entworfen und mittels der Programmiersprache Clojure implementiert. Unter anderem wird ein Parser zum Umwandeln von logischen Formeln in eine Clojure–Datenstruktur benötigt.

Zu diesem Zweck wurde ein Parser mittels der Bibliothek Instaparse erzeugt, jedoch viel dessen signifikant längere Laufzeit gegenüber einem JavaCC–Vergleichsparser auf.

In diesem technischen Bericht, welcher zugleich die Abschlussdokumentation zum Projekt darstellt, werden die Laufzeiten von fünf Parsern verglichen: Instaparse und Kern, jeweils mit einer vollständigen und einer minimierten Grammatik, sowie der JavaCC–Referenzparser.

Inhaltsverzeichnis

1. Einführung

Bevor auf die eigentlichen Messungen eingegangen wird, sollen zunächst diverse Hintergrundinformationen zu dem Projekt sowie den Parser–Bibliotheken gegeben werden.

1.1. MPA

Der *MNI Proposition Analyzer (MPA)* ist ein im Institut für Softwarearchitektur (ISA) des Fachbereichs Mathematik, Naturwissenschaften und Informatik (MNI) an der Technischen Hochschule Mittelhessen (THM) entwickeltes Programm, um übliche Berechnungen im Bereich der Aussagenlogik anzuwenden. [MPA]

Zu den Features gehören u. a.:

- Parsen von Formeln
- Ausgeben einer Wahrheitstabelle
- Produktion der konjunktiven und disjunktiven Normalform (CNF, DNF)
- Anwenden der Tseitin–Transformation
- Überprüfen der Erfüllbarkeit einer Formel mittels eines externen SAT–Solvers (z. B. SAT4J)
- Pretty–print–Funktionen z. B. für TEX.
- Anwenden eines externen, M4–kompatiblen Makroprozessors

Die Funktionen können über eine GUI angesprochen werden. Das Programm wird in der Lehre an der THM eingesetzt und ist unter der GPL quelloffen verfügbar.

1.2. Logic Workbench

Innerhalb des Masterprojektes der Autoren unter Leitung des bereits für den MPA verantwortlichen Prof. Dr. B. Renz wird ein Programm mit identischen oder ähnlichen Funktionen des MPA entwickelt. Unter dem Projektnamen *Logic Workbench (LWB)* wird im Gegensatz zum MPA (Java) jedoch die Programmiersprache *Clojure* verwendet.

Clojure ist ein Lisp, welches zu Java–Bytecode kompiliert und somit auf der JVM arbeitet. Als Besonderheit ist die einfache Verwendung von Java–Code (etwa Klassen) zu nennen, wodurch auch ein von JavaCC generierter Parser–Code benutzt werden kann (siehe Abschnitt 1.4.3).

Zum Zeitpunkt des Verfassens dieses Textes sind folgende Kernfunktionen implementiert:

- Parsen von Formeln
- Ausgeben einer Wahrheitstabelle
- Produktion der CNF (mittels trivialem Algorithmus oder Tseitin–Transformation)
- Generieren des Dimacs–Formats und Aufrufen von SAT4J

Ein Makroprozessor wurde nicht eingebunden, da Clojure selbst über ein mächtiges Makrosystem verfügt. Im weiteren Verlauf des Projektes sind die Implementierung einer der MPA ähnlichen GUI und ein System zum Anwenden der natürlichen Deduktion geplant. Der Code des Projektes ist unter [LWB] einsehbar.

1.3. Das Problem: Parser–Laufzeiten

Zu der wesentlichen Kernfunktionalität der LWB gehört das Parsen von Formeln, sodass eine Clojure–Liste entsteht, also beispielsweise der String `"A -> !(A & B)"` zu `(impl A (not (and A B)))` geparst wird. Die vollständige Referenzgrammatik des MPA findet sich in Anhang D.2.

Zunächst wurde zur allgemeinen Zufriedenheit die Bibliothek *Instaparse* verwendet. Bei einem Test mit einer großen Formel (Sudoku–Regeln) wurde jedoch eine im Vergleich mit dem Referenzparser sehr viel höhere Laufzeit festgestellt, wenn der mit Instaparse generierte Parser nicht vorher durch einen Stackoverflow abgebrochen wurde. So brauchte dieser in einem Testdurchlauf mehrere Minuten, wogegen der Referenzparser des MPA diese Aufgabe in etwa zwei Sekunden vollbrachte.

Aus diesem Grunde wurden zunächst die Ursachen für die stark wachsende Laufzeit des Instaparse–Parsers eruiert und anschließend eine Vergleichsmessung zwischen verschiedenen Parsern und Grammatiken durchgeführt.

1.4. Die Parser–Bibliotheken

In diesem Abschnitt sollen die für den Vergleich benutzten Parser–Bibliotheken nicht nur vorgestellt werden, auch ihre Anwendung, Konzepte und Eigenheiten werden in Kürze umrissen.

1.4.1. Instaparse

Die von Mark Engelberg entwickelte Bibliothek *Instaparse* gehört zur Gruppe der Parser–Generatoren, d. h. es wird eine Grammatik angegeben, aus der dann ein Programm generiert wird, welches eine Zeichenkette gemäß der Grammatik parsen kann. Dies ist der eigentliche Parser. Da ein Clojure–Programm zur Laufzeit übersetzt werden kann, ist zur Verwendung des generierten Programms kein weiterer Kompilier–Zwischenschritt notwendig, wie etwa bei JavaCC. [INSTA]

Instaparse ist komplett in Clojure geschrieben und wurde deswegen und aufgrund der Simplizität zunächst als Parser für die LWB gewählt. So sehen die Schritte zur Entwicklung eines Parsers beispielhaft wie folgt aus:

1. In einem Clojure–String oder einer externen Text–Datei wird eine Grammatik spezifiziert.
2. Diese Grammatik wird als Argument einer Funktion „parse“ verwendet, welche den Parser zurückgibt, z. B. `(def logic-parser (insta/parser "src/logic/grammar.txt"))`.
3. Mit diesem Parser kann ein String geparst werden, z. B. erzeugt `(logic-parser "A & B")` einen abstrakten Syntaxbaum (AST) `[:and [:atom "A"] [:atom "B"]]`. Als Ausgabeformat stehen dabei hiccup und enlive zur Verfügung.
4. Die Funktion „transform“ nimmt einen AST und eine Hash–Map zur Substitution an und wandelt den Parsebaum um. Aus obigem AST wird somit etwa `(and A B)`.

1.4.2. Kern

Im Gegensatz zu JavaCC und Instaparse gehören mit *Kern* geschriebene Parser zur Gruppe der Combinator–Parser. Dabei gibt es mehrere „kleine“ Parser, also z. B. einen, der eine Konjunktion

parst, und einen, der ein Atom parsen kann. Diese Einzelparser können dann mit sogenannten Kombinatoren miteinander verbunden werden, um einen größeren Parser zu erzeugen. Kern bietet eine Reihe von Funktionen, Makros und Datenstrukturen, um solche Combinator–Parser zu erzeugen, z.B. ChainL–Funktionen.

Im Gegensatz zu Instaparse verfügt es dabei über einen dedizierten Lexer, welcher sich nahtlos in die Parsergrammatik einfügt. Dies sorgt einerseits für kompakten Code, kann andererseits aber auch verwirren. So sind `token*` und `token_` Parser–Funktionen, `token` jedoch eine Funktion des Lexers und verlangt, dass ein solcher korrekt konfiguriert und eingebunden ist. Dafür erhöht der Lexer die Komfortabilität, da z. B. die für einen Identifier erlaubten Zeichen als einfacher regulärer Ausdruck angegeben werden können und sich der Benutzer daraufhin nicht mehr um Leerzeichen kümmern muss.

Das Äquivalent zur benötigten Grammatik kann im Sourcecode unter [LWB] eingesehen werden. Zu beachten ist, dass der Autor von Kern selber aussagt, dass seine Bibliothek nicht performant ist:

> "Kern's design isn't well-suited for achieving very high performance." [KERN]

1.4.3. JavaCC

Wie Instaparse gehört *JavaCC* zur Gruppe der Parser–Generatoren. Es ist jedoch eine Java–Bibliothek und schon weitaus länger in Entwicklung, sodass es ausgereifter ist. [JAVACC]

JavaCC verlangt eine Grammatik, in der nicht nur die Parseregeln, sondern auch die Regeln des zu produzierenden Codes festgelegt werden können. Daraus erzeugt JavaCC dann Java–Code, der kompiliert dann als Parser verwendet werden kann. Da Clojure die Möglichkeit bietet, Java–Code direkt zu verwenden, kann der Quellcode des JavaCC aus dem MPA übernommen werden; es muss jedoch eine andere Ausgabe erzeugt werden. So besteht im MPA der AST aus verschiedenen Klassen des Typs „Knoten" (Node) und kann z. B. unter Einsatz eines Visitor–Patterns transformiert werden. Um die in der LWB verwendeten Clojure–Listen zu produzieren, wird jedoch mittels einer Clojure–Funktion rekursiv durch den AST traversiert und die neue Baumstruktur erzeugt.

Die Bibliothek stellt ebenfalls einen Lexer bereit, jedoch kann dieser nicht optional verwendet werden, sondern es müssen alle Token durch diesen erkannt und an den Parser weitergegeben werden. JavaCC wird produktiv eingesetzt.

2. Versuchsaufbau

Da wie bereits erwähnt der zuerst implementierte Instaparse–Parser bei einem Beispiel nicht mehr praktikabel war, kam die Idee auf, verschiedene Parser zu vergleichen. Neben besagtem Instaparse und der JavaCC–Referenzimplementierung sollte noch ein Parser verglichen werden, der nach einem gänzlich anderem Prinzip arbeitet. Die Wahl fiel auf die Combinator–Parser–Bibliothek Kern.

Um die Ergebnisse der Laufzeitmessung vergleichbar und möglichst aussagekräftig zu halten, müssen Rahmenbedingungen definiert werden. Dazu gehört die Auswahl der Formeln — nach quantitativem und qualitativem Umfang — sowie eine bestimmte Reduktion der Grammatiken, um den Laufzeitunterschied mit einem Anwachsen der Parse–Regeln beurteilen zu können.

2.1. Allgemeine Aspekte

Die Messung wird mit der Methode „time“ gemessen. Es zählt der Aufruf des Parsers und ggf. das Anwenden einer Transformationsfunktion (um aus einem AST Clojure–Listen zu machen). D. h. es wird immer mit der zu parsenden Formel begonnen und letztendlich die selbe Clojure–Datenstruktur erzeugt. Innerhalb der Zeitmessung wird weder etwas auf der Konsole ausgegeben (dies kann zusätzliche Zeit in Anspruch nehmen), noch sind sonstige Seiteneffekte zugelassen.

Sofern möglich, wird jede Formel mit jedem Parser und jeder der für ihn definierten Grammatiken aufgerufen. Ist dies in einem besonderen Fall nicht möglich, so kann ein errechneter Wert angegeben werden, insofern er als solcher markiert wird. Jeder so mögliche Durchlauf wird zehnmal ausgeführt und von allen Ergebnissen das arithmetische Mittel gebildet.

Da einige Parser ihren Zustand des letzten Aufrufes speichern können, ist bei mehreren gleichen Aufrufen hintereinander der erste Aufruf signifikant langsamer als die anderen. Aus diesem Grunde dürfen mit jeweils einem Parser keine zwei identischen Aufrufe direkt aufeinanderfolgend ausgeführt werden. Hierfür wurde für jeden Messdurchlauf die REPL neu gestartet.

Zudem wurde jeder Parser zunächst einmal mit einer für die Messungen nicht herangezogenen Formel verwendet, da der erste Durchlauf auch unabhängig von dem zu parsenden String einen gewissen Initialisierungsaufwand bewältigen muss. So kam es, dass bei einer ersten Messung der Kern–Parser

mit der minimalen Grammatik bei der gleichen Formel länger gebraucht hat, als der entsprechende Parser mit der vollen Grammatik.

2.2. Grammatiken

Als Referenz der zu enthaltenden Regeln dient die Parse–Grammatik des MPA (siehe Anhang D.2). Es wurden ähnliche Definitionen für Instaparse und Kern vorgenommen und sie produzieren zumindest für die Testformeln das gleiche Ergebnis. Diese Grammatiken werden im Folgenden *volle Grammatiken* genannt.

Um jedoch das Wachstum der Laufzeit durch Hinzufügen von weiteren Regeln messen zu können, wurde für Kern und Instaparse jeweils eine weitere Grammatik definiert. Diese kann nur Formeln verarbeiten, welche enthalten:

- Binäroperatoren „and“ und „or“ durch die Zeichen „&“ bzw. „|“
- Unäroperator „not“ durch das Zeichen „!“
- Klammerung mit runden Klammern
- Atome, beginnend mit einem Kleinbuchstaben, fortführend mit Unterstrich, Kleinbuchstaben oder ganze Zahlen

Leerzeichen sind dabei nicht erlaubt. Diese Art von Grammatiken wird ab jetzt *minimale Grammatik* genannt.

2.3. Formeln

Um aussagekräftige Messergebnisse zu erhalten, mussten repräsentative Formeln gefunden werden, welche unterschiedliche Eigenschaften haben:

- verschiedene Größen (gemessen an Anzahl Variablen, Operatoren und Zeichen)
- teilw. Einschränkung auf Basisoperatoren (Nicht, Und, Oder) für die minimalen Grammatiken
- unterschiedliche Strukturen durch verschiedene Problemstellungen

Da einige Formeln durch das MPA–Projekt bereits vorhanden waren, wurden eben solche gewählt. Sie sollen nun im Einzelnen vorgestellt werden und sind unter [LWBFORM] einsehbar. Tabelle 2.1 zeigt eine Übersicht der Formeln unter verschiedenen Metriken.

2.3.1. Damenproblem

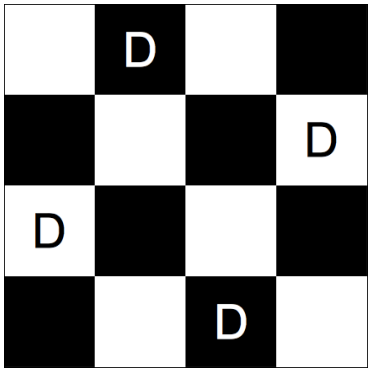

Abbildung 2.1.: Lösung des Damenproblems für $n = 4$: In einem Zug kann keine Dame eine andere schlagen.

Die Forderung lautet auf einem $n \times n$ großen Schachbrett n Damen so zu platzieren, dass keine Dame eine andere in einem Zug schlagen kann. Abbildung 2.1 zeigt eine mögliche Lösung.

Zum Testen wurden zwei verschiedene Formeln verwendet. Die erste ist eine handgeschriebe Lösung für $n = 4$, welche Implikationen enthält und dadurch nicht mit minimalen Grammatiken geparst werden kann. Die zweite Formel ist eine Lösung für $n = 8$ welche mit Hilfe von eigens dafür geschriebenen Funktionen erzeugt wurde. Sie besteht ausschließlich aus den Basisoperatoren Konjunktion, Disjunktion und Negation und ist zudem *konjunktive Normalform (KNF bzw. CNF)*.

2.3.2. Kartenfärbung USA

Man stelle sich eine Landkarte der USA mit eingezeichneten Staatsgrenzen vor. Jeder Staat soll nun eingefärbt werden, wobei keine zwei angrenzenden Staaten dieselbe Farbe erhalten dürfen. Diese Aufgabe soll nun mit möglichst wenigen Farben gelöst werden.

Erstellt man eine Formel, welche sämtliche Bedingungen enthält, so kann man das Problem mit einem SAT–Solver lösen. Zur Konstruktion der Formel sei auf [MPALOG] verwiesen.

So kann z. B. gezeigt werden, dass sich eine Karte der USA nicht mit drei Farben kolorieren lässt, jedoch mit vier Farben. Um eine größere Formel zu erhalten, wurde die für vier Farben verwendet. Sie ist in CNF dargestellt.

2.3.3. Sudoku

Dieses ebenfalls aus dem MPA–Projekt stammende Beispiel befasst sich mit dem Sudoku–Spiel. Drückt man die Regeln logisch aus und gibt einige Feldbelegungen vor, so kann das Rätsel wiederum durch einen SAT–Solver gelöst werden. Da eine gültige Belegung hier jedoch irrelevant ist, wurden lediglich die Regeln für die Messung benutzt. Sie liegen in CNF vor.

Gemäß Tabelle 2.1 wird deutlich, dass die entsprechende Formel signifikant größer als die anderen ist, wodurch die Unzulänglichkeit von Instaparse ursprünglich festgestellt wurde. Um dennoch Messergebnisse eben jenes Parsers erhalten zu können, wurde die Formel zweimal (ungefähr) halbiert. Freilich sind die Regeln dadurch unvollständig geworden, jedoch ist hier lediglich die Parsbarkeit der Formel von Interesse.

Name	**Anz. Var.**	**Anz. Op.**	**Anz. Zeichen**	**min. Grammatik**
4–Damen	200	215	711	nein
8–Damen	1520	2975	7487	ja
Kartenf. USA	1608	3019	16979	ja
$^1/_4$ Sudoku	5832	11015	40103	ja
$^1/_2$ Sudoku	12393	24056	86102	ja
Sudoku	24057	47384	168074	ja

Tabelle 2.1.: Die zum Test verwendeten Formeln mit den Anzahlen der Variablen, Operatoren, Zeichen und ob die Formel mit einer minimalen Grammatik geparst werden kann.

2.4. Messungen

Gemäß oben genannten Vorgaben wurden nun die Messungen durchgeführt. Die gemittelten Ergebnisse sind in Tabelle 2.4 einsehbar; die vollständige Messreihe findet sich unter D.1.

Formel	**Instaparse *m***	**Instaparse *v***	**Kern *m***	**Kern *v***	**JavaCC**
4–Damen	-	111	-	180	10
8–Damen	354	2301	293	590	93
Kartenf. USA	344	2751	300	548	153
$^1/_4$ **Sudoku**	2832	25546	1282	1704	461
$^1/_2$ **Sudoku**	5161	**Fehler**	1643	3586	524
Sudoku	18361	**Fehler**	3035	6894	963

Tabelle 2.2.: Messergebnisse im Mittel in Millisekunden. *m* und *v* stehen für „minimale“ und „vollständige Grammatik“.

Ausgeführt wurden die Tests auf einem Asus Zenbook UX31A mit folgender Spezifikation:

- Intel Core i7–3517U (4 x 1,9 GHz)
- 4 GB RAM
- Windows 7 (64 Bit)
- Java 1.7.0_09 (64 Bit)
- Clojure 1.5.1
- Instaparse 1.2.4
- Kern 0.7.0
- JavaCC 4.0

Das Parsen der halben und der kompletten Sudoku–Regeln mittels der vollständigen Grammatik war mit Instaparse nicht möglich, da es zu einem Stack–Overflow–Fehler (oder ähnlichem) geführt hat.

3. Bewertung

In diesem letzten Abschnitt sollen die erhaltenen Messergebnisse nun analysiert und bewertet werden.

3.1. Beobachtungen

Betrachtet man die Ergebnisse aus Tabelle 2.4, so kann man folgende Beobachtungen machen:

1. JavaCC benötigt immer und deutlich die wenigste Zeit.
2. Die Parser mit den vollständigen Grammatiken sind langsamer als die minimalen Grammatiken.
3. Ab einer bestimmten Größe ist der vollständige Kern–Parser schneller als Instaparse mit der Minimalgrammatik.
4. Die Kern–Parser sind im Allgemeinen schneller als die entsprechenden Instaparse–Produktionen.
5. Die Laufzeiten von Instaparse wachsen signifikant überproportional zur Größe der Formel.
6. Nur bei Instaparse treten bei großen Formeln Fehler auf (Stack Overflow).

Aus 1. folgt, dass der JavaCC–Parser letztendlich in LWB eingesetzt wird. Punkt 2. entspricht den Erwartungen.

Die Beobachtungen 3. und 4. weisen darauf hin, dass die Kern–Bibliothek allgemein performanter arbeitet als Instaparse. Es stellt sich die Frage, ob ein Combinator–Parser grundsätzlich schneller ist als der GLL–Algorithmus, mit dem Instaparse arbeitet [INSTA, GLL]. Diese Untersuchung ist jedoch nicht Gegenstand dieses Berichts.

Die letzten Punkte (5. und 6.) zeigen hinreichend, dass Instaparse nicht produktiv in LWB eingesetzt werden kann. Die generierten Parser sind einfach unzulänglich.

3.2. Analyse des Wachstums

Zwischen den Messergebnissen lassen sich Verhältnisse feststellen, wodurch ein das Wachstum beschreibender Faktor näherungsweise errechnet werden kann. Dies soll in den folgenden Abschnitten geschehen. Für Instaparse gibt es dazu noch eine zusätzliche Überlegung, welche vom Autor Mark Engelberg selber stammt. Diese soll mit den hier gemachten Beobachtungen verglichen werden.

Allgemein ist zu beachten, dass die Garbage Collection der JVM eine Rolle spielt, da sie die Laufzeiten signifikant beeinflussen kann. Da die Bedingungen für alle Parser gleich sind und es stark davon abhängt, wie viel „Garbage“ ein Programm produziert, wird dieser Aspekt angemerkt, nimmt jedoch keinen Einfluss auf die abschließende Bewertung.

Zur Übersicht und begleitend zu der nun folgenden Erklärung sei auf die Diagramme D.1 (lineare Darstellung, S. 29) und D.2 (logarithmische Darstellung, S. 30) verwiesen. Die Wachstumsverhältnisse sind hier leicht visuell zu erkennen.

3.2.1. Engelbergs Rechnung zum Instaparse–Wachstum

Zur Geschwindigkeit von Instaparse hat sich dessen Autor *Mark Engelberg* in der zugehörigen Google–Group zu unserer Problemstellung geäußert. Seine Antwort wird im Folgenden erläutert. [INSTGR]

Engelberg hat festgestellt, dass die Grammatik gut konstruiert wurde und die lange Parse–Zeit nicht von Mehrdeutigkeiten in der Grammatik abhinge. Daher bemühte er sich uns zu helfen, dem Problem auf den Grund zu gehen. Alle hier genannten Messergebnisse gehen auf die Angaben und damit auch den Computer — von dem wir keine genauere Spezifikation haben — von Mark Engelberg zurück. Daher können die genauen Laufzeiten auf anderen Rechnern abweichen.

Das angenommene Beispiel waren die vollen Sudokuregeln. Diese sind mit Hilfe von insgesamt 168075 Zeichen (Charactern) beschrieben. Die reine Geschwindigkeit des Lesens wurde durch eine minimale Grammatik geprüft.

```
S = #'a'+
```

Mit dieser Grammatik konnten 168075 Zeichen — in diesem Fall „a“ — innerhalb einer Sekunde geparst werden. Hieraus geht hervor, dass ein starkes Ansteigen der Parse–Zeit eher mit der wachsenden Komplexität der Grammatik zusammenhängt.

Es wurde an dieser Stelle angenommen, dass die Zeit linear zur Komplexität steigt. Wenn also die Grammatik aus 60 Regeln besteht — wie etwa die volle Grammatik des Parsers für die LWB — sollte das Parsen mit dieser Annahme etwa eine Minute dauern.

Um diese Annahme zu überprüfen, hat Herr Engelberg eine Funktion *produce* geschrieben, mit der verschieden große Formeln produziert werden können. Diese haben eine ähnliche Struktur wie die Sudoku–Regeln, wodurch die Laufzeiten des Parsers mit der vollen Grammatik schrittweise im Kleinen überprüft werden können:

```
(defn ^:dynamic produce [n]
  (cond
    (zero? n) "c1"
    :else (str "(" (produce (dec n)) ")" "&" "(" (produce (dec
       n)) ")")))

(def produce (memoize produce))
```

```
=> (produce 1)
"(c1)&(c1)"

=> (produce 2)
"((c1)&(c1))&((c1)&(c1))"
```

Diese Funktion produziert also eine Ausgabe, die immer etwa doppelt so groß wird, wenn man das Argument um eins erhöht:

```
=> (count (produce 14))
114683

=> (count (produce 15))
229371
```

Wir möchten nun einen String parsen, der mit der ungefähren Länge zwischen den Ausgaben von `(produce 14)` und `(produce 15)` liegt.

Nun betrachten wir die Laufzeit, wenn die Zeichenkette auf die doppelte Länge wächst:

```
=> (time (def x (formula-parser (produce 5))))
"Elapsed time: 55.882633 msecs"

=> (time (def x (formula-parser (produce 6))))
"Elapsed time: 104.751713 msecs"

=> (time (def x (formula-parser (produce 7))))
"Elapsed time: 195.774887 msecs"
```

```
=> (time (def x (formula-parser (produce 8))))
"Elapsed time: 558.543622 msecs"

=> (time (def x (formula-parser (produce 9))))
"Elapsed time: 1284.250435 msecs"

=> (time (def x (formula-parser (produce 10))))
"Elapsed time: 2007.042944 msecs"
```

Wie man sehen kann, beansprucht das Parsen einer längeren Zeichenkette mehr Zeit. Wenn die Länge der zu parsenden Zeichenkette verdoppelt wird, benötigt der Parser etwa doppelt so lange. Die Abweichungen können auf das Eingreifen der Garbage–Collection zurückgeführt werden.

Hierbei ist erkennbar, dass Instaparse mit der hier benutzten Grammatik ein akzeptables Verhalten an den Tag legt. Es ist demzufolge nichts verkehrt an der genutzten Grammatik und die Zeit des Parsens wächst linear proportional zur Eingabegröße. Der produzierte Parser hat also kein überproportionales Wachstum.

Um die den Sudoku–Regeln gleichwertige Eingabegröße zu erreichen, müssen wir die hier vorhandene Formel aus `(produce 10)` noch etwa fünfmal verdoppeln (`(produce 15)`). Nach dieser Rechnung sollte der Parser also etwa eine Minute brauchen, um eine Formel der geforderten Eingabegröße zu Parsen.

In der Praxis sieht dies jedoch anders aus. Bei dem Versuch eine Formel von `(produce 13)` fängt die Garbage–Collection an zu arbeiten und stört damit massiv den Ablauf. Durch dieses Verhalten verlängert sich die Zeit, die zum Parsen benötigt wird, ungemein. Wobei es in dieser Größenordnung der Eingabe schon teilweise zu Out-Of-Memory-Errors kommt.

An dieser Stelle können wir also schon einige Punkte festhalten:

1. Im Idealfall braucht Instaparse mit der gegebenen Grammatik etwa eine Minute zum Parsen der gesamten Sudoku–Regeln. Da dies nicht für unseren Anwendungsfall praktikabel war, mussten wir uns nach anderen Möglichkeiten umsehen

2. Die Praxis zeigt, dass der limitierende Faktor der Arbeitsspeicher ist.

Wir wollen uns nun dem Faktor Arbeitsspeicher widmen. Im Versuchsaufbau konnte mit 1 GB allokiertem Speicher bis zu einer Eingabegröße von `(produce 12)` gearbeitet werden, bevor Probleme auftraten. Man kann annehmen, dass Instaparse die Formel für `(produce 15)` innerhalb einer Minute parsen kann, wenn man 8 GB Arbeitsspeicher (dreimalige Verdopplung) dafür allokiert. Diese Theorie wurde von Mark Engelberg nicht getestet, da er seinen Angaben nach keine einfache Möglichkeit hatte, dies auszuprobieren.

Auch mit dieser Annahme hat Engelberg gezeigt, dass der aktuelle Instaparse (v 1.2.4) nicht für unsere Anwendung geeignet ist. Das Programm soll ermöglichen, logische Formeln in akzeptabler Zeit zu Parsen. Zum Parsen einer Eingabegröße der Sudoku–Regeln kann man annehmen, dass etwa 8 GB Arbeitsspeicher ausschließlich für die Verarbeitung zur Verfügung stehen müssten. Diese Größenordnung ist für die aktuellen Rechner (noch) unüblich. Aber selbst wenn man einen Rechner mit genügend Arbeitsspeicher hat, bräuchte dieser immerhin noch etwa eine Minute zum Parsen. Dies ist im Vergleich zu den hier betrachteten Alternativen ein sehr großer Unterschied.

3.2.2. Das Verhältnis der gegebenen Formeln

Zuallererst betrachten wir die Größe der Variablen. Bei dem 4– und 8–Damen–Problem bestehen die Variablen aus zwei Zeichen. Für die Kartenfärbung werden im Schnitt 7,5 Zeichen pro Variable benötigt und für die Sudoku–Regeln vier.

Operatoren als Symbole (also z. B. „&“ statt „and“) bestehen im Normalfall aus einem Zeichen. Eine Ausnahme besteht hier bei dem 4–Damen–Problem. Dieses enthält auch den Operator ->. Da dieses Zeichen nicht in der Minimalgrammatik vorhanden ist, kann diese Formel auch nicht mit den entsprechenden Parsern übersetzt werden.

Nun wollen wir den Anstieg der Anzahl der jeweiligen Variablen, Operatoren und Zeichen betrachten.

Das 4–Damen–Problem benötigt 200 Variablen, 215 Operatoren und insgesamt 711 Zeichen. Beim 8–Damen–Problem verzehnfacht sich etwa die Anzahl der Zeichen. Die Anzahl der Variablen ist hierbei um das 7,5–fache angestiegen und die Anzahl der Operatoren sogar um das 15–fache.

Für die Kartenfärbung der USA benötigen wir etwa genauso viele Variablen und Operatoren wie für das 8–Damen–Problem. Da in diesem Fall jedoch die Variablen fast viermal so lang sind, steigt die Zeichenanzahl auf mehr als das Doppelte an. Dieses Verhältnis kann zur Beobachtung genutzt werden, ob die Parse–Zeit eher von der Anzahl der Zeichen oder der Anzahl der Variablen und Operatoren abhängt.

Da die Länge der Variablen zwischen dem 8–Damen-Problem und den Sudoku–Regeln weniger schwanken als bei der Kartenfärbung der USA, vergleichen wir die $^1/_4$ Sudoku–Regeln mit dem 8–Damen–Problem. Dabei vervierfacht sich fast die Anzahl der Variablen und Operatoren, wobei die Anzahl der Zeichen sich mehr als verfünffacht. Dieser unterschiedliche Anstieg von Variablen/-Operatoren zu Zeichen kommt daher, dass die Sudokuregeln durch Klammerung stark geschachtelt sind.

Entsprechend der Abstufungen enthalten die $^1/_2$ Sudoku–Regeln doppelt so viele Variablen, Operatoren und Zeichen. Diese Verdopplung geschieht noch einmal beim Schritt auf die vollen Sudoku–Regeln.

3.2.3. Instaparse

Nun wollen wir das Wachstum der Parse–Zeit für Instaparse im Zusammenhang mit der Eingabe betrachten.

Für das 4–Damen–Problem braucht der *Instaparse–Parser mit der vollen Grammatik (IPv)* 111 ms. Das diese Formel mit der Minimalgrammatik nicht zu parsen war, liegt in diesem Fall keine Zeit für den *Instaparse–Parser mit der minimalen Grammatk (IPm)* vor.

Für das 8–Damen–Problem brauchte der IPm 354 ms und der IPv 2301 ms. Für dem IPv ist das mehr als die zwanzigfache Zeit. Dies fällt in etwa in die Größenordnung vom Anstieg der Variablen und Zeichen.

Der IPm braucht 344 ms und der IPv 2751 ms zum Parsen der Kartenfärbung der USA. In beiden Fällen ist das etwa die gleiche Zeit wie beim 8–Damen–Problem. Da die Anzahl der Variablen und Operatoren gleich geblieben ist, wobei sich die Anzahl der Zeichen mehr als verdoppelt, kann man annehmen, dass die Parse–Zeit sich nicht signifikant durch die Länge der Variablen verändert.

Für das Parsen der $^1/_4$ Sudoku–Regeln braucht der IPm achtmal und der IPv etwa elfmal so lange wie für das 8–Damen–Problem. Man kann daher annehmen, dass das Parsen mit dem Instaparse–Parser durch die starke Verschachtelung mittels Klammern die Parse–Zeit steigert. Hierbei kann der Unterschied zwischen der Erhöhung zwischen IPm und IPv daran liegen, dass die Klammer–Regel sehr weit unten in der Grammatik beschrieben ist, wobei die minimale Grammatik weniger Regeln hat, die bis zur Klammerungsregel geprüft werden müssen. Des Weiteren ist anzunehmen, dass beim IPv die Garbage Collection einsetzt und dies die Laufzeit verlängert.

Wie in der Beschreibung von Mark Engelberg (siehe Abschnitt 3.2.1) angenommen und gezeigt wurde, hat sich die Parse–Zeit für IPm von $^1/_4$ zu $^1/_2$ Sudoku–Regeln verdoppelt. Der IPv konnte die $^1/_2$ Sudoku–Regeln nicht mehr parsen, da es dabei immer wieder zu Fehlern (*OutOfMemoryException, StackOverflowError, ...*) kam.

Die benötigte Zeit vervierfacht sich fast zum Parsen der gesamten Sudokuregel. Hier ist zu erkennen, dass sich die Garbage Collection massiv auf die benötigte Zeit auswirkt.

Nun betrachten wir das Zeitverhältnis zwischen IPm und IPv. Zu Anfang lässt sich annehmen, dass der IPv etwa sechsmal länger braucht als der IPm. Beim Parsen der $^1/_4$ Sudoku–Regeln dauert es aber etwa zehnmal so lange. Wir nehmen also an, dass es sich um ein exponentielles Verhältnis (x^n)

handelt. Hierbei konnte ermittelt werden, dass der Exponent (^{n}) bei etwa 1,293 liegt. Mit Hilfe diese Annahme wurden die theoretischen Werte für den IPv und den IPm errechnet.

3.2.4. Kern

Wie beim Instaparse–Parser kann auch der *Kern–Parser mit der minimalen Grammatik (KPm)* nicht das 4–Damen–Problem mit der hier gegebenen Formel lösen. Daher kann diese erste Messung nur mit dem *Kern–Parser mit der vollen Grammatik (KPv)* durchgeführt werden. Dieser brauchte für das Parsen 180 ms.

Für das Parsen des 8–Damen–Problems benötigte der KPm 293 ms und der KPv 590 ms. Für den KPv ist das in etwa eine Verdreifachung der Zeit. Dies steht in keinem einfachen Zusammenhang mit der Anzahl der Variablen, Operatoren und Zeichen. Aufgrund der sonstigen Beobachtungen scheint dies bei kleinen Eingabegrößen der Fall zu sein.

Ähnlich dem Instaparse–Parser brauchen sowohl der KPm (300 ms) und der KPv (548 ms) annähernd die gleiche Zeit zum Parsen der Formel für die Kartenfärbung der USA. An diesem Beispiel kann man ablesen, dass die Dauer des Parsens nicht erkennbar von der Länge der Variablen abhängt.

Zum Parsen der $^1/_4$ Sudoku–Regeln braucht der KPm mehr als viermal so lange und der KPv etwa das Dreifache an Zeit wie für das 8–Damen–Problem. Das ähnelt eher dem Anstieg der Anzahl der Variablen und Operatoren als dem der Zeichen.

Der KPm braucht zum Parsen der $^1/_2$ Sudoku–Regeln nicht merkbar mehr Zeit als zum Parsen der $^1/_4$ Sudoku–Regeln. Der KPv hingegen braucht etwas mehr als die doppelte Zeit. Das Verhalten des KPv kann aus der Verdopplung der Anzahl der Variablen, Operatoren und Zeichen hergeleitet werden, wogegen das zeitliche Verhalten des KPm sich mit den übrigen Beobachtungen nicht deckt.

Für das Parsen der gesamten Sudoku–Regeln benötigt sowohl der KPm als auch der KPv etwa das Doppelte an Zeit als der jeweilige Parser für die $^1/_2$ Sudoku–Regeln.

Mit Ausnahme des Verhaltens bei kleinen Formeln und des Verhältnisses der benötigten Zeit des KPm zum Parsen der $^1/_4$ Sudoku–Regeln und der $^1/_2$ Sudoku–Regeln, sieht man, dass sich die mit Kern geschriebenen Parser in etwa linear proportional zur Anzahl der Variablen bzw. Operatoren verhalten.

3.2.5. JavaCC

Nun wollen wir uns die Zeiten des *JavaCC-Parsers (JP)* ansehen. Da der JP schon von vornherein vorhanden war und Zeiten mit der vollen Grammatik beachtlich kurz waren, erstellten wir für den JavaCC keine minimale Grammatik. Für das 4–Damen–Problem brauchte der JP etwa 10 ms zum Parsen.

Zum Parsen des 8–Damen–Problems brauchte der JP im Schnitt 93 ms. Das ist etwas mehr als neunmal die für das 4–Damen–Problem benötigte Zeit und entspricht in etwa dem Wachstum der Anzahl von Variablen, Operatoren und Zeichen.

Für die Regeln zum Kartenfärben der USA benötigte der JC 153 ms. Dies ist im Gegensatz zu den anderen Parsern ein Anstieg von etwa zwei Dritteln. Betrachtet man die Messwerte des JC zum Parsen dieser Regeln, fällt auf, dass die meisten Werte bis auf zwei Peeks in etwa bei 90 bis 120 ms liegen. Wenn man den Mittelwert über die Messergebnisse errechnet und dabei die zwei Extreme auslässt, ergibt sich eine durchschnittliche Zeit von etwa 107 ms. Diese Steigerung liegt dann vergleichbar zu den anderen Parsern.

Um die $^1/_4$ Sudoku–Regeln zu Parsen, benötigt der JC etwa 461 ms, also etwa das Fünffache der Zeit für das 8–Damen–Problem. Hierbei lagen drei Messergebnisse über 600 ms und die übrigen sieben etwas unter 400 ms. Würde man ausschließlich diese sieben Messwerte betrachten, wäre das etwa eine Vervierfachung der Zeit ausgehend vom 8–Damen–Problem. Damit lägen wir auch ungefähr beim Wachstum der bisher betrachteten Parser.

Die Messwerte für die $^1/_2$ Sudoku–Regeln liegen nicht so weit auseinander. Hierbei benötigt der JC in Schnitt etwa 524 ms. Dies ist nur eine kleine Steigerung der benötigten Zeit und entspricht nicht dem Doppelten wie bei den übrigen Parsern, sondern eher im Wachstumsverhältnis des KPm, der auch hier bei dem Vergleich der $^1/_4$ Sudoku–Regeln und $^1/_2$ Sudoku–Regeln nur eine relativ kurze Differenz der durchschnittlichen Zeiten hat.

Zum Parsen der vollen Sudokuregeln nimmt sich der JC im Durchschnitt etwa 963 ms Zeit. Hierbei schwanken die Messwerte von knapp unter 700 bis fast 1200 ms, wobei drei Werte sich um 700 mehren und die übrigen sieben über 1000 ms liegen. Unter Betrachtung der Mehrheit kam es hier zu einer Verdopplung der benötigten Zeit ähnlich den anderen Parsern.

Durch die extrem kurze Zeit des Parsens mit dem JC kann es schnell zu messtechnischen Ausreißern kommen, da hier nur eine geringe Zeitverzögerung wie z. B. der Lastwechsel zwischen Prozessoren ausreicht, um eine signifikant längere Zeit zu erreichen. Daher ist es schwerer, für den JP ein sicheres und klares Ergebnis zu erhalten. Jedoch scheint das Wachstumsverhältnis ähnlich linear proportional zur Eingabegröße zu sein, wie bei den übrigen betrachteten Parsern, sofern diese nicht an ihre technischen Grenzen stoßen, wie etwa der IPv ab den $^1/_4$ Sudoku–Regeln und der IPm mit den vollen Sudoku–Regeln.

3.2.6. Fazit

Alle hier betrachteten Parser hatten ein erkennbares Wachstumsmuster (vgl. Abb. 3.1). Diese ähnelten sich mit ein paar Ausnahmen und unterhalb der technischen Grenzen stark. Man konnte erkennen, dass alle Parser in etwa ein linear proportionales Wachstumsverhältnis zur Eingabegröße haben. Wobei der Instaparse–Parser am ehesten an seine technischen Grenzen stoß. Kern war wie Instaparse ein mit Clojure entwickeltes Werkzeug, jedoch Instaparse in zeitlicher Hinsicht klar überlegen. Allerdings benötigte der KPv fast sieben Sekunden zum Parsen der vollen Sudoku–Regeln. Da das erstellte Tool vorzugsweise für die Lehre benutzt werden soll, könnte man in diesem Rahmen einen auf Kern basierten Parser benutzen. Es hat sich jedoch gezeigt, dass ein auf JavaCC basierender Parser mit der gleichen Grammatik um ein Vielfaches schneller ist — in unserer Messreihe für die vollen Sudokuregeln etwa sieben mal so schnell wie der KPv.

Da wir es für sinnvoll halten, unser Projekt nicht unnötig langsam arbeiten zu lassen, haben wir uns entschieden, die schnellste verfügbare und von uns beherrschbare Alternative zu wählen. Damit viel die Wahl auf den JavaCC–Parser.

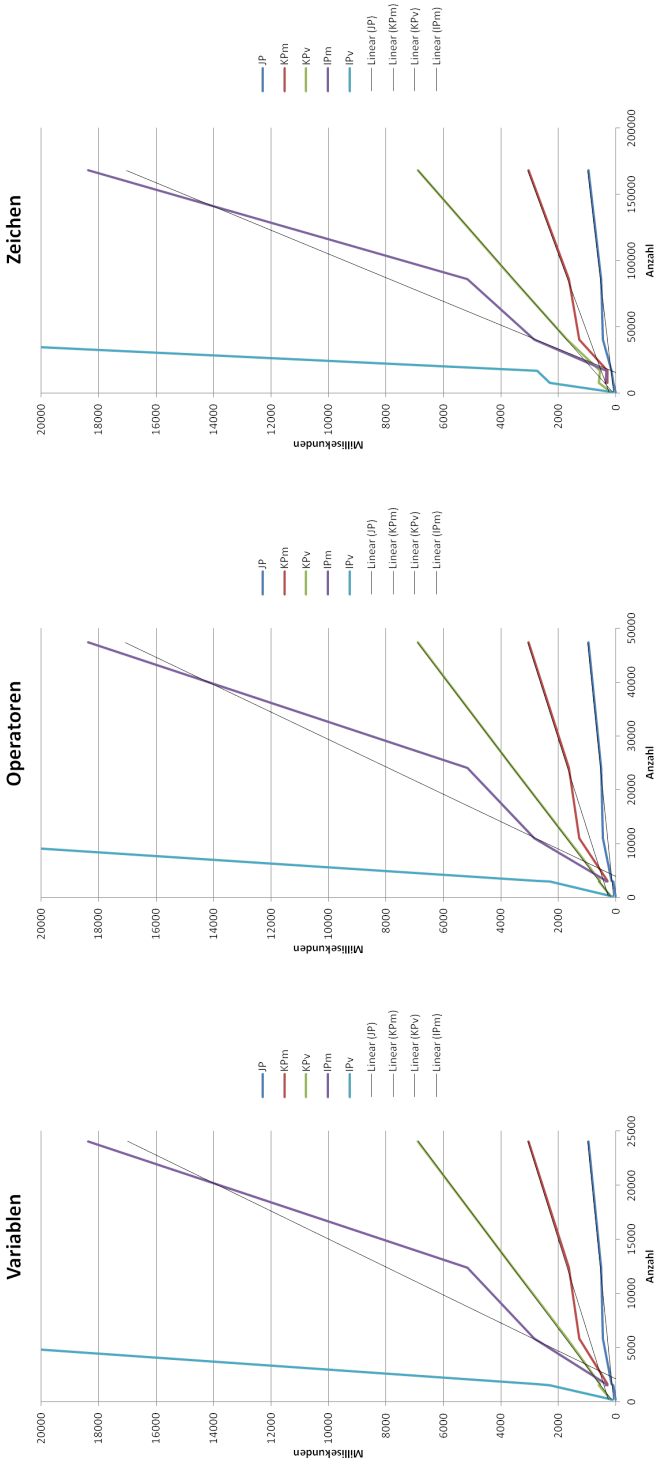

Abbildung 3.1.: Übersicht der Laufzeiten der Parser im Verhältnis zur Anzahl der Variablen, Operatoren und Zeichen

A. Literaturverzeichnis

[LWB] Quellcode des Projektes auf Github,
`https://github.com/moerkb/logical-workbench`

[LWBFORM] Testformeln,
`https://github.com/moerkb/logical-workbench/blob/master/src/parser_measures/formulas.clj`

[KERN] Armando Blancas,
Projektseite der Bibliothek Kern,
`https://github.com/blancas/kern`

[INSTA] Mark Engelberg,
Projektseite der Bibliothek Instaparse,
`https://github.com/Engelberg/instaparse`

[JAVACC] Java Compiler Compiler,
`https://javacc.java.net`

[MPA] Prof. Dr. B. Renz,
Projektseite des MPA,
`http://homepages.thm.de/~hg11260/mpa.html`

[INSTGR] Diskussion über Instaparse–Laufzeiten mit Autor Mark Engelberg,
`https://groups.google.com/forum/#!topic/instaparse/GYoswP-X7O8`

[MPALOG] Prof. Dr. B. Renz,
„Lösungen mit dem MNI Proposition Analyzer MPA — Logeleien, Kartenfärbung, Sudoku, Variabilitätsmodelle“,
`http://homepages.thm.de/~hg11260/mat/mpa-bsp.pdf`

[GLL] Elizabeth Scott, Adrian Johnstone,
„GLL Parsing“,
Department of Computer Science, Royal Holloway, University of London,
`http://www.sciencedirect.com/science/article/pii/S1571066110001209`

B. Abbildungsverzeichnis

C. Tabellenverzeichnis

D. Sonstiges

D.1. Messergebnisse

4 Damen												**Mittel**
javacc	10	9	10	8	7	15	11	9	15	10		10
kern, voll	199	199	191	190	187	186	54	194	201	196		180
instaparse, minimal												(38)
instaparse, voll	118	110	107	110	108	121	108	109	112	108		111
8 Damen												
javacc	92	94	94	72	90	104	93	98	88	109		93
kern, minimal	285	276	292	296	275	279	371	286	287	287		293
kern, voll	645	624	567	649	570	563	468	551	638	626		590
instaparse, minimal	298	369	345	291	366	376	348	355	368	419		354
instaparse, voll	2351	2223	2249	2277	2330	2306	2730	2328	2238	1975		2301
USA												
javacc	101	113	89	336	103	345	119	115	110	105		153
kern, minimal	331	395	324	259	252	227	323	284	330	275		300
kern, voll	483	593	509	488	601	465	539	541	552	709		548
instaparse, minimal	336	442	340	335	352	314	337	355	334	294		344
instaparse, voll	2588	2638	2594	2960	2557	2925	2640	3025	2923	2664		2751
Viertel Sudoku												
javacc	629	379	393	375	380	375	659	651	383	384		461
kern, minimal	1274	1290	1251	1292	1245	1262	1345	1289	1287	1283		1282
kern, voll	1720	1686	1676	1792	1617	1714	1707	1710	1745	1670		1704
instaparse, minimal	2102	2434	3939	2322	3655	2566	2096	2017	3506	3686		2832
instaparse, voll	31108	28401	21306	21629	21233	20991	28230	28365	29472	24725		25546
Halbes Sudoku												
javacc	472	483	580	489	626	552	419	570	443	609		524
kern, minimal	1657	1606	1628	1714	1646	1616	1706	1659	1604	1591		1643
kern, voll	3580	3473	3557	3741	3589	3484	3735	3651	3513	3535		3586
instaparse, minimal	5151	4708	5085	4941	5714	4698	4196	4913	5270	6938		5161
instaparse, voll												(63184)
Sudoku												
javacc	695	1182	1028	1044	1091	670	720	1101	1031	1067		963
kern, minimal	3051	2941	3014	3198	2996	2991	3141	3064	2967	2982		3035
kern, voll	6977	6631	6777	7279	6806	6669	7165	7054	6726	6852		6894
instaparse, minimal	19072	18782	18530	18439	17480	19001	19870	18148	18547	15740		18361
instaparse, voll												(325998)

Tabelle D.1.: Ergebnisse der Messreihen, alle Angaben in Millisekunden. Ergebnisse in Klammern sind errechnet.

D.2. Referenzgrammatik

Das folgende Listing stammt aus der Hilfe–Datei des MPA. [MPA]

```
Proposition ::= Atom | Const
| "(" UnOp Proposition ")"
| "(" Proposition BinOp Proposition ")"
Atom ::= ([letter]|"_"|"\"|"{"|"}")([letter]|[digit]|"_"|"\"|"{"|"}")*
Const ::= ("T"|"1"|"True"|"F"|"0"|"False")
UnOp ::= ("!"|"not")
BinOp ::= see Section below
```

Formeln können mit runden ((...)) oder eckigen Klammern ([...]) zusammengefasst werden. Kommentare sind mit (//...) zeilenweise oder durch (/*...*/) blockweise möglich.

Die binären Operatoren „BinOp“ sind wie folgt organisiert:

Beschreibung	**Syntax**	**Assoziativität**
Und	&, and	links
Nicht–Und	!&, nand	links
Oder	|, or	links
Nicht–Oder	!|, nor	links
Rechtsimplikation	<-, if	rechts
negierte Rechtsimplikation	nif	rechts
Implikation	->, impl	rechts
negierte Implikation	nimpl	rechts
Äquivalenz	<->, iff	links
exklusives Oder	^, xor	links

Tabelle D.2.: Binäroperatoren mit Syntax und Assoziativität, absteigend sortiert nach Präzedenz

D.3. Diagramme

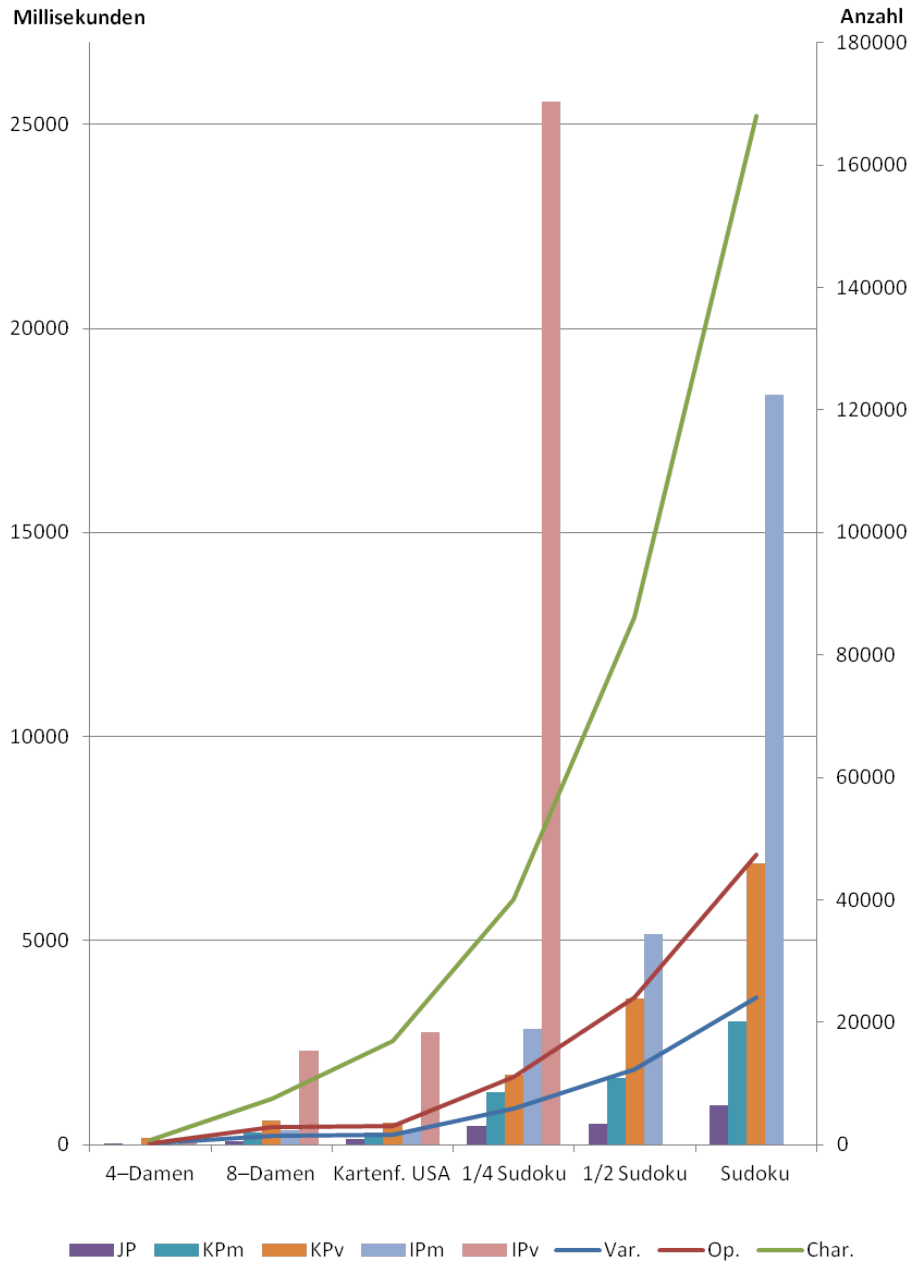

Abbildung D.1.: Formeln und Parsezeiten, linear

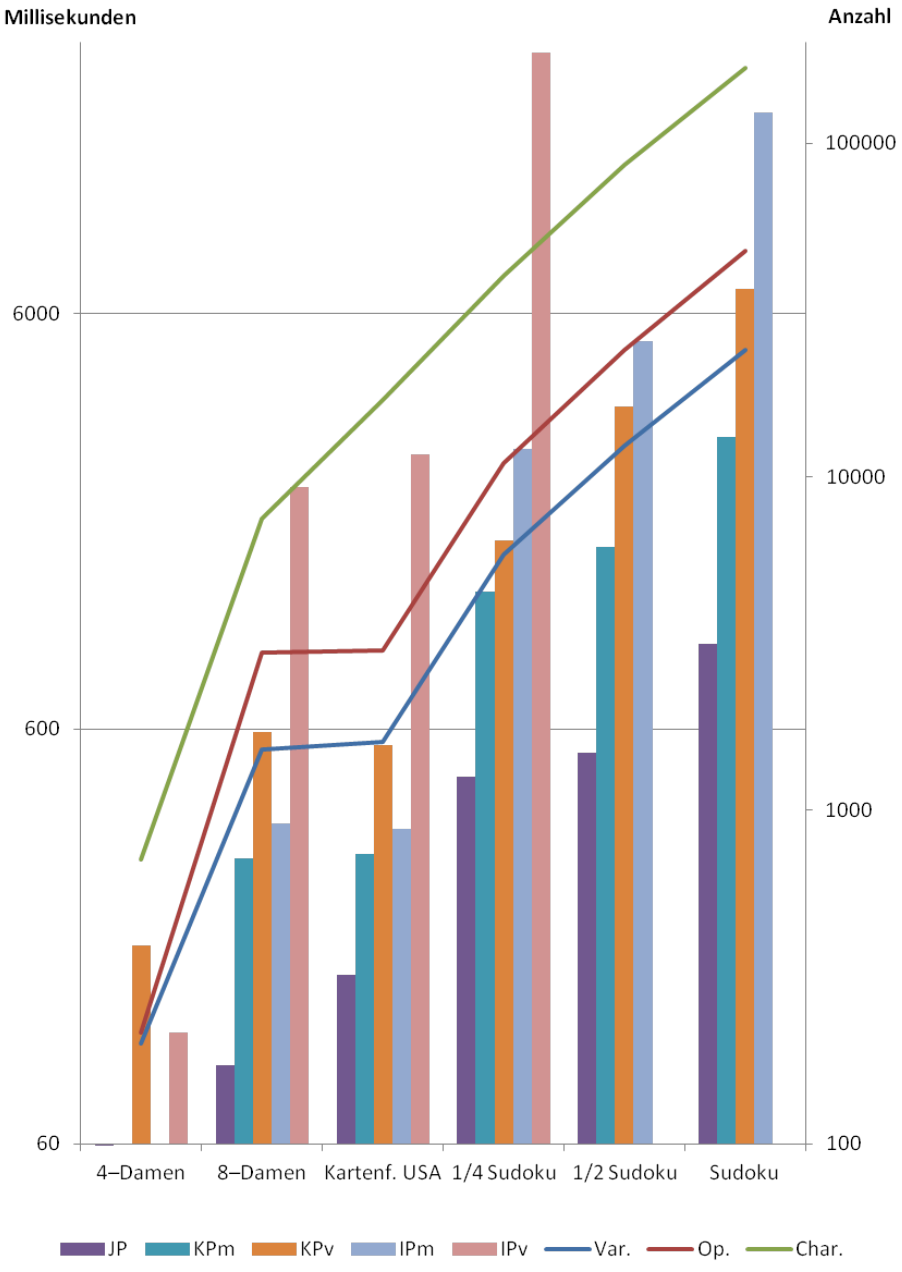

Abbildung D.2.: Formeln und Parsezeiten, logarithmisch