Philipps Universität Marburg

Concepts of Advaced Programming Languages

Dr.-Ing. Christoph Bockisch

{Dice Game Language}

Domänenbeschreibung, Sprachspezifikation und Diskussion

Artur Sterz, Jonas Höchst, Andreas Morgen

19. Juli 2015

Inhaltsverzeichnis

1 Würfelspiele **3**
1.1 Einführung . . . 3
1.2 Wichtige Konzepte . . . 3
1.2.1 Spieler . . . 3
1.2.2 Würfel . . . 3
1.2.3 Spielverlauf . . . 3
1.3 Sprachziele . . . 3

2 Die Sprache **4**
2.1 Grundidee . . . 4
2.1.1 Ein erstes Beispiel . . . 4
2.2 Kontrollierte Sprache . . . 5
2.3 Satzzeichen . . . 5
2.3.1 Punkt (.) . . . 5
2.3.2 Doppelpunkt (:) . . . 5
2.3.3 Komma (,) . . . 5
2.3.4 Semikolon (;) . . . 5
2.3.5 Variablen . . . 5
2.4 Blöcke und Absätze . . . 6
2.5 Groß- und Kleinschreibung . . . 6

3 Designentscheidungen **6**
3.1 Qualitäten der Sprache . . . 6
3.1.1 Modularität . . . 6
3.1.2 Robustheit . . . 7
3.1.3 Datenmodellierung . . . 7
3.1.4 Abstraktionsniveau . . . 7
3.1.5 Verständlichkeit . . . 7
3.2 Alternativen im Sprachdesign . . . 7
3.2.1 Erweiterung bestehender Spiele . . . 7
3.2.2 Mächtigkeit . . . 8
3.2.3 Objektmodellierung . . . 8
3.3 Designspace . . . 8
3.3.1 Werte & Typen . . . 8
3.3.2 Dispatching . . . 8
3.3.3 Ausdrücke . . . 8
3.3.4 Scope . . . 9
3.3.5 Anweisungen . . . 9

4 Prototyp Implementierung **9**
4.1 Python Framework . . . 9
4.2 Interne Datenrepräsentation . . . 10
4.3 Compilerimplementierungen . . . 10
4.4 Erfüllung der geforderten Sprachqualitäten . . . 11
4.5 Mögliche Erweiterungen . . . 11

5 Erfahrungen **12**

6 Anhang **13**
6.1 Codebeispiel 13
6.2 Generierter Code 14
6.3 Abstrakte Syntax 16

Kapitel 1

Würfelspiele

1.1 Einführung

Würfelspiele sind Glücksspiele, die mit Spielsteinen, sogenannten *Würfeln* gespielt werden. Dabei besteht jedes Spiel aus einem oder mehreren Würfeln, die nacheinander oder gleichzeitig geworfen werden, um ein bestimmtes Ergebnis zu erzielen. Hierbei werden von den Spielern kombinatorische Fähigkeiten verlangt.

1.2 Wichtige Konzepte

Für DiGaL sind im Wesentlichen drei Konstrukte wichtig, die im Nachfolgenden erläutert werden.

1.2.1 Spieler

Das erste wichtige Konzept ist der *Spieler*. Den Spieler zeichnet aus, dass er die ausführende Kraft bei einem Spiel ist. Er würfelt mit den Würfeln, bei Entscheidungen muss er diese treffen und auch die Punkteverwaltung hat er inne.

Weiterhin benötigt der Spieler Variablen, in denen aktuelle Werte gespeichert werden. Dies ist zwar sehr technisch ausgedrückt, beschreibt aber das, was "Spieler haben Punkte" bedeutet.

Und schließlich können Spieler aktiv oder inaktiv sein. Beispielsweise kann ein Spieler auf Grund seiner auf 0 reduzierten Punkte aus dem Spiel ausscheiden, er kann aber auch, wie bei dem Spiel *UNO Würfel* bei einer bestimmten Augenzahl eine Runde aussetzen.

1.2.2 Würfel

Ein weiterer Kernbestandteil sind die Würfel. Dabei kann ein Spiel mehrere Würfel haben, jedoch mindestens einen. Desweiteren hat ein Würfel sogenannte *Augen*. Das bedeutet, dass einer bestimmten Seite ein numerischer Wert zugeordnet wird. Dabei hat ein klassischer Würfel sechs Seiten, ist also von 1 bis 6 durchnummeriert. Es gibt jedoch Würfel mit einer beliebigen Anzahl an Seiten, mindestens jedoch zwei.

Außerdem muss erwähnt werden, dass es Spiele gibt, bei denen die Würfel keine numerischen Werte besitzen, sondern mit Piktogrammen bedruckt sind. Diese finden in DiGaL jedoch keine Anwendung, da diese Bilder auch mit einem numerischen Wert kodiert werden können.

1.2.3 Spielverlauf

Das letzte domänenspezifische Konzept, das in der hier diskutierten enthalten sein muss, ist der Spielverlauf. Hauptsächlich sind damit die Regeln gemeint, die beschreiben, was ein Spieler zu tun hat, wenn er an der Reihe ist.

Dabei werden zum einen Aktionen definiert, wie zum Beispiel das Werfen der Würfel. Zum anderen werden aber auch Konditionen geprüft, die erfüllt sein müssen, damit eine Aktion ausgeführt werden kann. So muss beispielsweise bei dem Spiel *Mäxchen* die Augenzahl 21 gewürfelt werden, damit allen anderen Spielern ein Punkt abgezogen werden kann.

1.3 Sprachziele

Das oberste Ziel war es DiGaL so einfach zu gestalten, so dass auch Personen, die keine Programmiererfahrung haben, den vorliegenden Quellcode verstehen können. Die Syntax ist dabei anders als bei herkömmlichen Sprachen und wird hauptsächlich durch Worte ausgedrückt.

Außerdem wurde versucht die Domäne so genau wie möglich in der Sprache abzubilden. Wie diese Ziele erreicht wurden und welche Kompromisse dabei eingegangen werden mussten, wird in den kommenden Kapiteln 2 und 3 diskutiert.

Kapitel 2

Die Sprache

2.1 Grundidee

Bestandteile eines Würfelspiels sind neben den benötigten Materialien, wie den Würfeln, vor Allem das Regelwerk. Dieses enthält alle Informationen, die benötigt werden, um das Spiel korrekt, also im Sinne des Erfinders, zu spielen. Dabei besteht das Regelwerk aus folgenden Bestandteilen:

- Anzahl der Spieler,
- Voraussetzungen für das Spiel,
- Aktionen der Spieler,
- Spielziel und Bedingungen für das Spielende und
- Bewertungsgrundlagen

Unser Ziel war es nun DiGaL so zu gestalten, dass es als Regelwerk eines Spiels erkannt und auch von allen so gelesen werden kann. Damit dies deutlich wird, soll ein Beispiel gegeben werden:

2.1.1 Ein erstes Beispiel

Um einen ersten Eindruck der Sprache zu erhalten, soll hier zunächst ein Beispielprogramm angegeben werden:

```
1  EINS wird so gespielt:
2
3  das spiel ist für 2 bis 10 spieler geeignet.
4  das spiel hat folgende würfel:
5  würfel A hat diese seiten: 1 2 3 4 5 6
6  würfel B hat diese seiten: 1 2 3 4 5 6.
7
8  spieler haben die werte PUNKTE ist 0.
9  spieler haben die werte GEWONNEN ist 0.
10
11 ist ein spieler am zug macht er folgendes:
12 würfelt mit allen würfeln.
13 rechter spieler ist dran, wenn würfel 0 gleich 1 oder würfel 1 gleich 1.
14 aktueller spieler PUNKTE ist aktueller spieler PUNKTE + die summe von allen
      würfeln.
15 wenn aktueller spieler PUNKTE größergleich 100, dann setze aktueller
      spieler GEWONNEN auf 1 und spiel ist zu ende.
16 rechter spieler ist dran.
17
18 gewonnen hat der spieler, bei dem GEWONNEN gleich 1.
```

2.2 Kontrollierte Sprache

Das oben gezeigte Beispiel, gibt einen ersten Eindruck über die Syntax unserer Sprache. Es wurde versucht DiGaL an die natürlichen Sprache anzulehnen. Es ist unmöglich bzw. ein langer Prozess, alle Eigenheiten der deutschen Sprache zu implementieren und abzubilden. Ein Kompromiss aus unserem Design Goal und der Umsetzbarkeit ist eine kontrollierte Sprache: eine Sprache, die auf der einen Seite zwar intuitiv verständlich ist, auf der anderen Seite jedoch die deutsche Sprache so weit einschränkt, dass sie sinnvoll zu analysieren und grammatisch beschreibbar ist.

Das Deklinieren und Konjugieren von Worten ist für das intuitive Verstehen einer Sprache aus unserer Sicht von großer Bedeutung und trägt maßgeblich zum Lesefluss bei. Im obigen Beispiel sei dazu auf auf die Verwendung von `alle würfel` verwiesen, der im Beispiel schon in seiner deklinierten Variante `allen würfeln` verwendet wird.

Auch die Satzstellung ist für das Verständnis einer natürlichen Sprache wichtig. Sätze einer kontrollierten Sprache können schnell sperrig wirken. Unsere Sprache bietet daher verschiedene Optionen, um semantisch das Gleiche auszudrücken. Hier sei als Beispiel das `wenn` Konstrukt von oben angegeben, dass in folgenden beiden Varianten valide ist:

```
1 rechter spieler ist dran, wenn würfel 0 gleich 1 oder würfel 1 gleich 1.
2 wenn würfel 0 gleich 1 oder würfel 1 gleich 1, dann rechter spieler ist
    dran.
```

Designziel der Syntax unserer Sprache ist es zunächst, möglichst viele Varianten eines gegebenen Konstruktes anzubieten. In der Implementierung zeigten sich jedoch Schwierigkeiten, auf die in Kapitel 3 und 4 näher eingegangen wird.

2.3 Satzzeichen

Wie auch in der natürlichen Sprache, werden auch in dieser kontrollierten Sprache Satzzeichen verwendet, um Mehrdeutigkeit für den Interpretierenden zu verhindern. Die Leser unserer Sprache bestehen nicht nur aus Menschen, sondern insbesondere auch vom Parser unserer Implementierung. Die Verwendung von Satzzeichen ist daher obligatorisch.

2.3.1 Punkt (.)

Punkte schließen Sätze der natürlichen Sprache ab. Unsere kontrollierte Sprache übernimmt dieses Paradigma, um Anweisungen in unserer Sprache abzuschließen.

2.3.2 Doppelpunkt (:)

Der Doppelpunkt leitet Aufzählungen ein. Das erste mal tritt nach dem initialen Satz auf und leitet die Aufzählung der Definitionsblöcke ein, danach dient er zum Beispiel noch zur Aufzählung der Würfel, deren Seiten, oder der Aktionen, die ein Spieler vollzieht. Der Doppelpunkt dient eher der Übersichtlichkeit als der Eindeutigkeit.

2.3.3 Komma (,)

Das Komma der natürlichen Sprache trennt Teilsätze ab, die semantisch zusammengehören. In unserer Sprache dienen Kommata dazu mehrere Anweisungen aufzuzählen, und `dann`- und `sonst`-Anweisungen von bedingten Anweisungen abzugrenzen, und die Eindeutigkeit der Programme zu erhalten.

2.3.4 Semikolon (;)

Das Semikolon wird verwendet, um den Schleifenrumpf von weiteren Anweisungen abzutrennen. Dies ist ebenfalls nötig, um die Eindeutigkeit der Sprache zu gewährleisten.

2.3.5 Variablen

Unsere Sprache bietet dem Anwender die domänenspezifischen Objekte *Spieler* und *Würfel* und Variationen davon (*linker/rechter Spieler, Würfel 1, ...*) als Konzept der Sprache an. Darüber hinaus ist es möglich globale Variablen zu definieren, sowie Variablen pro Spieler festzulegen. Spielabhängige Variablen werden in Großbuchstaben geschrieben, um die Trennung von der Syntax zu verdeutlichen (siehe 2.5).

2.4 Blöcke und Absätze

Absätze haben in der natürlichen Sprache den Zweck Abschnitte, die einen eigenen Sinnzusammenhang oder ein eigenes kleines Thema haben, voneinander abzugrenzen. Dieses Paradigma wurde auch in die Syntax von DiGaL übernommen. So haben Programme, die in DiGaL geschrieben sind, vier Absätze, die nach ihrem Sinn unterteilt sind.

Zunächst beginnt ein Programm mit einer Überschrift. Anschließend kommt ein Abschnitt, der die Spielinitialisierung übernimmt. Es werden globale Variablen, Anzahl und Art der Würfel und die Anzahl der Spieler festgelegt. Auch eine Bedingung, wie lange das Spiel läuft, wird hier definiert.

Anschließend folgt ein Abschnitt, in dem die Spieler initialisiert werden, zum Beispiel spielerspezifische Variablen oder eine Bedingung, wann ein Spieler aktiv ist.

Danach folgt der Abschnitt, in dem die Regeln festgelegt werden, was geschehen soll, wenn ein Spieler an der Reihe ist.

Letztlich gibt es eine Art Schlusssatz, der eine Bedingung definiert, wann ein Spieler gewonnen hat.

2.5 Groß- und Kleinschreibung

In diesem Punkt wird bewusst von der natürlichen deutschen Sprache abgewichen. Es wäre möglich auch hier die Regeln der deutschen Sprache einzuführen. Das hätte jedoch negative Auswirkungen auf die Lesbarkeit eines Programms. Da DiGaL eine Programmiersprache ist, bei der es wie bei anderen Programmiersprachen darum geht den Zustand eines Programms zu ändern und das im Wesentlichen über Variablen funktioniert, wurde entschieden Variablenbezeichner komplett groß (z.B. `EINS` oder `PUNKTE` in obigem Beispiel) und den Rest komplett klein zu schreiben. Dies erhöht die Lesbarkeit eines Programms und vereinfacht es so Programmierern ein valides und korrektes Programm zu schreiben. Die Elemente des Abstract Syntax Graph unserer Sprache sind im Anhang zu finden.

Kapitel 3

Designentscheidungen

3.1 Qualitäten der Sprache

3.1.1 Modularität

Funktionsdefinitionen

DiGaL bietet keine Konzepte, um Modular Spiele entwickeln zu können. Diese Entscheidung ist bewusst so gewählt worden. Auch hier stand wieder im Vordergrund, die Sprache so einfach wie möglich zu gestalten. Ein Programm in Module aufzuteilen, heißt zum Beispiel eine Möglichkeit zu bieten Funktionen oder Methoden zu definieren, in denen Aktionen gekapselt sind, die wiederholt ausgeführt werden können. Dies würde aber bedeuten, dass man ein Konzept benötigt, dass dem Entwickler einen solchen Mechanismus erlaubt. Und dieses neue Konzept müsste von Anwendern unserer Sprache erlernt werden, wenn sie keine Programmierer sind. Daher wurde entschieden, keine Möglichkeit anzubieten Funktionen zu definieren. Es schien einfacher dem Nutzer die Möglichkeit zu überlassen, Aktionen wiederholt zu definieren.

Desweiteren besteht ein Würfelspiel in DiGaL aus vier Blöcken, wie sie in Abschnitt 2.4 vorgestellt werden. Die Möglichkeit Funktionen zu definieren würde daher bedeuten, dass dieses strikte Konzept gebrochen werden müsste, was wiederum ein Mehr an Komplexität bedeuten würde.

Auslagern in andere Dateien

Da Spiele, die in DiGaL geschrieben sind, nur aus vier Blöcken bestehen, schien es uns nicht sinnvoll, eine Art der Auslagerung von Code in andere Dateien einzuführen. Auch hier müsste mit dem Konzept der strikten Blockbildung gebrochen werden und eine Möglichkeit geschaffen werden diese Dateien einzubinden oder zu importieren. Das könnte unerfahrene Nutzer jedoch zusätzlich verwirren, da es aus unserer Sicht nicht mit unserem Konzept vereinbar ist,

DiGaL eine kontrollierte Sprache zu Grunde zu legen, da es kein sprachliches Konzept gibt, dass paradigmatisch das Gleiche aussagt.

3.1.2 Robustheit

Alle DiGaL Programme haben einen festen Rahmen, dessen freie Flächen durch Variablendefinitionen, Arithmetische Operationen und Kontrollflussanweisungen gefüllt werden. Im Gegensatz zu klassischen Programmiersprachen ist der Programmierer dadurch in seiner Mächtigkeit sehr eingeschränkt. Durch diese Einschränkung ist es ihm wiederum auch weniger leicht möglich, Fehler zu produzieren. Treten Programmierfehler im DiGaL Programm auf, sind diese schon durch die Syntax invalidiert und können nicht in ein Programm übersetzt werden. Als Beispiel sei hier die Iteration über eine Variable genannt, die in unserer Sprache nur als Iteration über eine Gruppe von Spielern oder Würfeln vorgesehen ist.

Funktions- oder Klassendefinitionen sind in DiGaL nicht vorgesehen. Der rein iterative Ansatz reicht aus, den Kontrollfluss und die Komplexität von Würfelspielen zu beschreiben.

Würfelspiele sind in ihren möglichen Ausgängen begrenzt und dadurch gut testbar. Eine kleine Zahl an Würfeln mit seiner geringen Zahl an Würfelflächen, die in einer festen Kontrollstruktur verwendet werden, macht es möglich alle Fälle des Programmes dazu generieren und die Spielausgänge zu testen.

3.1.3 Datenmodellierung

DiGaL bietet keine Möglichkeit um Datenobjekte oder -strukturen anzulegen. Es wird allerdings ein Aspekt der Objektorientiertheit unterstützt, in dem die Möglichkeit angeboten wird, einem Spieler Variablen zuzuordnen. Die einzigen erweiterterbaren Objekte sind also die Spieler. Alle vom Nutzer definierten Variablen, die nicht zu Spielern gehören lassen sich nur global als Variable des Spiels anlegen.

Als weiteres (nicht modifizierbares) Objekt sind Würfel zu nennen, deren Namen und Seiten zwar vom Programmierer definiert, nicht aber neue Felder hinzugefügt werden können.

3.1.4 Abstraktionsniveau

Das Abstraktionsniveau in DiGaL ist bewusst sehr hoch gewählt. Auch hier stand wieder die Einfachheit der Sprache im Vordergrund. Daher werden keinerlei Möglichkeiten angeboten beispielsweise auf Adressen im Arbeitsspeicher zuzugreifen oder Operationen auf Bits oder Bytes auszuführen. Auch typische Probleme in anderen Programmiersprachen wie Initialisierung oder Speicherverwaltung werden vom Compiler übernommen, sodass sich vor Allem unerfahrene Nutzer nicht um derlei Probleme kümmern müssen.

3.1.5 Verständlichkeit

Die Verständlichkeit von DiGaL ist gegeben durch die Verwendung der kontrollierten Sprache. Dadurch ergeben sich keine syntaktisch kryptischen Konstruktionen, die bei späterem Betrachten aufwendig rekonstruiert werden müssen. Auch die Unterstützung von alternativen Formulierungen (siehe Anhang, Abschnitt 6.3) vereinfacht die Verständlichkeit. Falls sich doch komplexe Ausdrücke ergeben sollten, die nicht auf Anhieb verständlich sind, wurde die Möglichkeit geschaffen den Code durch Kommentare zu erklären.

3.2 Alternativen im Sprachdesign

Grundpfeiler unserer Sprache ist, maximal einfache Beschreibungen der domänspezifischen Anwendungen auszudrücken, in Anlehnung an die natürliche Sprache. Würden auf diese Eigenschaft verzichtet, wäre die Begründung für die Einschränkungen nicht mehr gegeben und alle oben genannten Punkte müssten neu diskutiert werden, da die Begründung sich weitestgehend auf das erste Argument stützt.

Zur Problemlösung wäre auch eine Library denkbar, die Paradigmen aus Würfelspielen aufgreift und dem Nutzer zur einfacheren Implementierung anbietet. Durch die sehr auf die Domäne ausgerichtete Sprache, folgen die meisten oben diskutierten Entscheidungen direkt. Einzelne Paradigmen lassen sich jedoch diskutieren und in Frage stellen.

3.2.1 Erweiterung bestehender Spiele

Eine weitere Möglichkeit der Modularisierung ist, bestehende Spiele, also in unserem Fall bestehenden Code zu erweitern. Die aktuelle Version von DiGaL bietet dieses Konstrukt nicht an. Denkbar wäre aber eine Art Vererbung, in dem ein Spiel auf einem anderen basiert und jeder Block um weitere Anweisungen ergänzt werden kann (Hinzufügen von Würfeln, zusätzliche Aktionen bei bestimmten Ergebnissen, ...). Diese Vererbung sollte ein Spiel jedoch nur erweitern, nicht aber bestehende Regeln außer Kraft setzen können, da dies zum einen wieder zu Lasten der Verständlichkeit

geschehen würde. Zum andern würde eine Änderung des zugrundeliegenden Spieles faktisch einer Neuentwicklung gleich kommen und sollte aus diesem Grund auch so behandelt werden.

3.2.2 Mächtigkeit

Um die Ausdrucksmächtigkeit von DiGaL zu erhöhen ist eine wesentliche Änderung Funktionen einzuführen. Der Programmierer könnte Konstrukte definieren, die im Moment von der Sprache zur Verfügung gestellt werden und damit den Code besser strukturieren.

Eine weitere Möglichkeit um die Ausdrucksmächtigkeit zu erhöhen, liegt darin, frei zu definierende `while` Schleifen einzuführen. Im Moment werden in der Sprache lediglich `foreach` und `for` Schleifen angeboten. Die bereits eingeführten Konstrukte zur Iteration könnten verallgemeinert werden, so dass auch Variablen definiert werden können, über die iteriert werden kann. Dies würde allerdings zu Lasten der Robustheit gehen, da der Programmierer dadurch leichter Fehler machen kann, die erst zur Laufzeit auftreten.

3.2.3 Objektmodellierung

Einige Spiele bestehen nicht nur aus den von uns unterstützten Datenstrukturen `Spieler` und `Würfel`. So hat beispielsweise das Spiel Kniffel neben genannten Objekten auch noch Spielzettel mit Ergebniswerten und Becher. Diese können zwar in DiGaL mit Spielervariablen nachgebildet werden, eine Modellierung als Objekt wäre jedoch auch denkbar und unter Umständen wünschenswert.

Außerdem gibt es in manchen Spielen Spieler, die eine Sonderrolle haben und von uns auch nicht direkt unterstützt werden. Auch hier würde eine Möglichkeit der Modellierung sinnvoll sein.

Schließlich existieren Spiele, die Objekte aus anderen Domänen übernehmen, zum Bespiel Spielkarten oder Spielfelder und Spielfiguren, die sich auf den Spielfeldern bewegen.

Um diese Dinge zu unterstützen könnte das Konzept der Objektorientiertheit eingeführt werden und damit einhergehend auch Möglichkeiten um eigenen Objekte zu erstellen und zu modellieren.

3.3 Designspace

3.3.1 Werte & Typen

Unsere Domäne erfordert keine ausgefallenen Datentypen. Würfelseiten und daraus resultierend auch die Aktionen im Spiel basieren auf Ganzzahl-Werten (`Integer`). Variablen in unserer Sprache werden mit Strings adressiert, die aus Großbuchstaben bestehen. Der Datentyp String besteht in der Sprache selbst nicht und wird bei der Kompilation im Endprodukt hinzugefügt. Denkbar wäre es allerdings, Strings einzuführen, zum Beispiel um eigene textuelle Ausgaben zu definieren, die vom gegebenen Framework abweichen.

Darüber hinaus werden die domänenspezifischen Objekte direkt adressiert. Für Spieler gibt es Ausdrücke, die den jeweils linken oder rechten Spieler ausgehend vom aktuellen Spieler oder einen Spieler auf einem bestimmten Platz adressieren. Ebenso ist es möglich Würfel über deren Nummer oder Bezeichnung zu adressieren. Mengen von Würfeln oder Spielern können über deren verschiedenen Eigenschaften ausgewählt werden: `alle <würfel/spieler>`; `aktive <spieler>`; `würfel 1, würfel 2`.

3.3.2 Dispatching

Man spricht von Dispatching, wenn ein Methodenaufruf zur Laufzeit anhand des tatsächlichen Typs eines Objektes aufgelöst wird. Da DiGaL weder Methoden noch Polymorphie unterstützt, mussten keine Gedanken zu Dispatching gemacht werden.

3.3.3 Ausdrücke

DiGaL unterstützt alle gängigen arithmetischen Ausdrücke außer die Division, da bei Würfelspielen in den meisten Fällen nur mit Ganzzahlen gerechnet wird. Außerdem würde die Division einen neuen Datentypen für Fließkommazahlen voraussetzen, der aber der Einfachheit halber nicht unterstützt wird.

Außerdem sind Vergleichsoperationen wie `größer`, `kleiner`, `gleich` möglich. Auch logische Ausdrücke wie `und` und `oder` sind möglich.

Weiterhin gibt es domänenspezifische Ausdrücke. Hier sei die Operatione `die summe von <würfeln>` genannt, die die Summe aller gewürfelten Augenzahlen ermittelt und die Operation `anzahl <spieler/würfel>`, die die Anzahl von zum Beispiel aktiven Spielern zurück gibt.

3.3.4 Scope

Bei einem Würfelspiel sind zu Beginn alle Variablen global bekannt. Wenn gewürfelt wird, sehen alle Spieler die gewürfelte Augenzahl und auch alle Spieler wissen, wie viele Spieler und Würfel am Spiel beteiligt sind. Daher gibt es im Spiel lediglich einen globalen Scope. In diesem können auch alle Spieler alle Werte lesen und verändern.

Die Spieler hingegen haben eigene lokale Variablen, da diese nicht zum globalen Spiel gehören, sondern jeder Spieler für beispielsweise seine eigenen Punkte verantwortlich ist. Es gibt jedoch die Möglichkeit Werte von anderen Spielern zu ändern. Dies ist bewusst so entschieden worden, da beispielsweise bei dem Spiel Mäxchen bei der Augenzahl 21 allen anderen Spielern ein Punkt abgezogen wird.

3.3.5 Anweisungen

DiGaL bietet dem Nutzer eine eingeschränkte, aber sehr domänenspezifische Sammlung an Anweisungen. Elementare Anweisungen sind Zuweisungen (von ausgewerteten Ausdrücken), Schleifen und bedingte Anweisungen, die so auch in klassischen Programmiersprachen auftreten.

Darüber hinaus sind die elementaren Ausdrücke der zugrundeliegenden Domäne als feste Anweisungen vorgesehen: `würfeln mit <würfeln>`, `sortieren <würfel>`, `nächster <spieler> ist dran`, ... Diese festen Anweisungen können als eine Art Framework für Würfelspiele aufgefasst werden.

Kapitel 4

Prototyp Implementierung

In der Referenzimplementierung von DiGaL wird die ANTLR Syntax zur formalen Definition unserer Sprache verwendet. Darauf aufbauend wird der von ANTLR bereitgestellte Parser und Lexer genutzt, um das in DiGaL geschriebene Spiel zu erfassen.

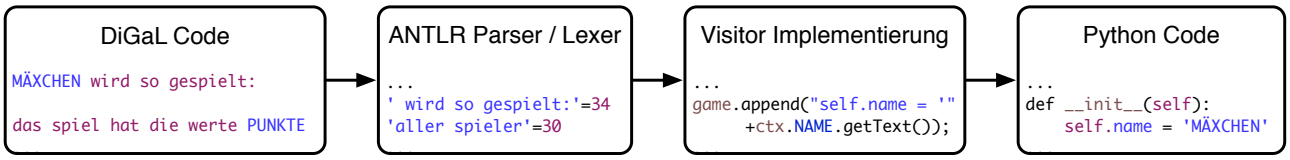


Abbildung 4.1: Ablauf der Kompilation eines DiGaL Programmes

Um sicherzustellen, dass die mit DiGaL geschriebenen Spiele portabel sind, wurde zum Ziel gesetzt möglichst wenige Abhängigkeiten vorauszusetzen. Ein Kriterium war dabei, dass die Ausführung des Codes nur mit der Standardbibliothek einer Zielsprache funktioniert. Daher wurde sich für eine Kompilation der Sprache entschieden, und darauf versichtet für den Endanwender auf die Installation von ANTLR vorauszusetzen. Als Zielsprache dient Python, das Syntaktisch kompakt zu programmieren ist und inhärente Konzepte unserer Sprache unterstützt (Objektorientiertheit, Funktionale Programmierung).

Zur Generierung des Python Codes wird das Visitorpattern entschieden verwendet. Die Entscheidung gründet auf die Simplizität einer Implementierung dieses Patterns, das uns als lernende ANTLR Spielentwickler nur noch abverlangt die vorgegebenen Methoden zu implementieren und uns so auf das wesentliche zu konzentrieren.

4.1 Python Framework

Zusammen mit der Definition der Grammatik wurden generische Methoden ausgearbeitet. Die vorhanden Methoden lassen sich in zwei Klassen differenzieren: Generische Methoden, die in vielen Würfelspielen verwendet werden; Methoden die zur Interaktion mit den Mitspielern und zur Repräsentation von Daten gegenüber diesen nötig sind. Die zweite

Gruppe ergibt sich nicht aus der Domäne selbst, sondern folgt aus der nötigen Mensch-Maschine Kommunikation (Ausgaben auf der Kommandozeile).

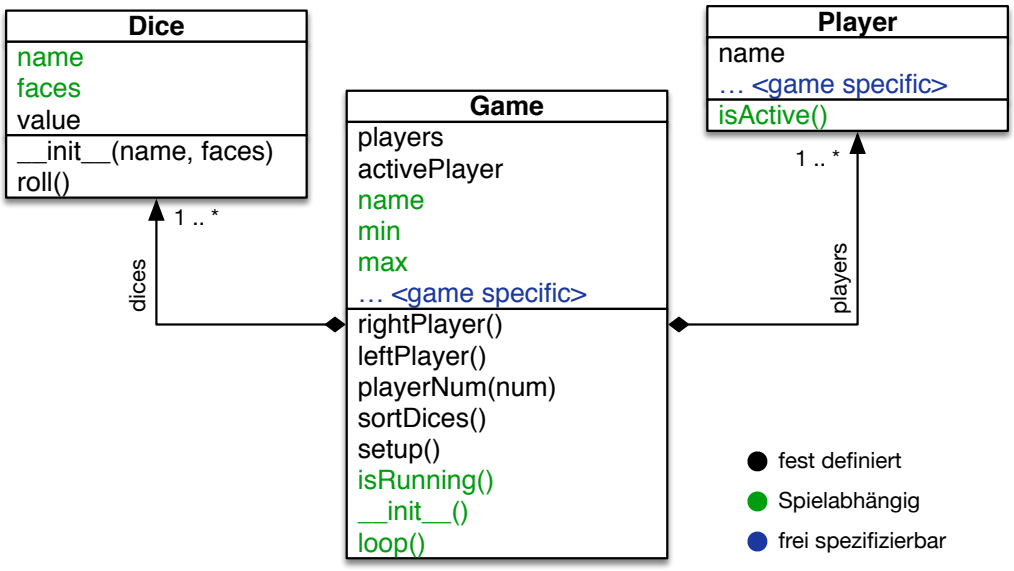


Abbildung 4.2: Klassendesign des Python Frameworks

Die domänenspezifischen Methoden können direkt aus der Sprache verwendet werden und entsprechen den verwendbaren Anweisungen (`würfelt mit <würfeln>`).

4.2 Interne Datenrepräsentation

Wie in den obigen Abschnitten erwähnt, werden für die Implementierung Konstrukte unserer Zielsprache verwendet. Der objektorientierte Ansatz von DiGaL zeigt sich hier in der Umsetzung in Klassen für Spieler, Würfel und das Spiel selbst. Bei der Übersetzung eines Spiels werden Teile der gegebenen Klassen durch die vom Spielentwickler spezifizierten Eigenschaften des Spiels ergänzt. Beispiel dafür sind die Spieler, denen durch den Spielentwickler neuen Felder hinzugefügt werden können.Die Würfelklasse hingegen ist nicht erweiterbar, sondern wird nur durch die Implementierung der Spielklassen verschieden initialisiert.

Die Spielklasse enthält die Initialisierung und den Ablauf des Spiels und ist die zentrale Klasse. Die Klasse enthält einige generische Methoden, die nicht individualisierbar sind und in der Sprache zur Verfügung stehen (`leftPlayer()`, `rightPlayer()`, `sortDices()`, ...). In Kontrast dazu bildet die `loop()`-Methode den Spielablauf ab und wird vom Entwickler frei definiert.

4.3 Compilerimplementierungen

Da die Implementierung recht umfangreich ist kann sie hier nicht vollständig sondern lediglich beispielhaft angegeben werden.

DiGaL wurde so konzipiert, dass alle domänenspezifischen Konzepte, wie sie oben beschrieben sind, direkt abgebildet werden können. Auch in der Pythonumsetzung wurde versucht dies beizubehalten und die Konzepte von DiGaL soweit wie möglich direkt in Python umzusetzen. Grundoperationen wie `+, -, *, größer, kleiner, gleich, ...` wurden direkt in ihren entsprechenden Pythonoperatoren umgesetzt, woraus eine lazy evaluation arithmetischer Operationen folgt. Gleiches gilt für die Zuweisung. Diese kann in DiGaL zwar syntaktisch unterschiedlich ausgedrückt werden (`<Variable> ist <Ausdruck>` oder `setze <Variable> auf <Ausdruck>`), in Python wird daraus aber immer eine einfache Zuweisung.

Desweiteren wurden einige Abstraktionen vorgenommen, um die Paradigmen aus der Domäne umzusetzen. Als Beispiel sei folgender Code gegeben:

```
1 wenn würfel A gleich 2 ...
```

In konventionellen Programmiersprachen wäre diese Operation nicht ohne Weiteres möglich, da ein Vergleich zwischen einem selbst definierten Objekt und einer Zahl zunächst nicht definiert ist. Im Bezug auf die Domäne ist dieser Ausdruck jedoch sinnvoll, da damit implizit gemeint ist, dass der gewürfelte Wert, also eine Zahl mit einer anderen verglichen werden soll. Diesen Umstand nutzen wir, sodass ein logischer Vergleich oder eine arithmetische Operation

mit Würfeln immer als Operation auf dem entsprechenden gewürfelten Wert zu verstehen ist. Im Code sieht das wie folgt aus:

```
1 if [dice for dice in self.dices if dice.name == 'A'].value == 2 ...
```

Diese Idee wurde versucht so weit wie möglich umzusetzen. Sie findet sich beispielsweise in der Referenzierung des nächsten Spielers wieder. Hier genügt es in DiGaL zum Beispiel das Schlüsselwort `linker spieler` zu verwenden. Diese Abstraktion wurde in Python so umgesetzt, dass der Index der Liste, in dem Spieler gespeichert sind, um eins dekrementiert wird.

Weiterhin soll hier auf die Implementierung der Methode `isRunning()` eingegangen werden. Diese repräsentiert die Bedingung, die der Entwickler festlegen kann. Sie bestimmt, ob ein Spiel noch läuft oder nicht. Beispielsweise kann ein Spiel nach n Runden zu Ende sein.

```
1 ODD wird so gespielt:
2
3 das spiel hat den wert RUNDEN ist 5.
4 ...
5 das spiel läuft solange RUNDEN größer 0.
6 ...
```

Im oberen Listing wird beispielsweise die globale Variable `RUNDEN` definiert und als Bedingung festgelegt, dass ein Spiel solange läuft, solange `RUNDEN größer 5` gilt. Das heißt das Spiel läuft fünf Runden. Im kompilierten Pythoncode sieht das wie folgt aus:

```
1 def isRunning(self): return (self.RUNDEN > 0)
```

Diese Methode hat eine weitere Besonderheit. Wird keine Bedingung festgelegt, weil ein Spiel beispielsweise kein deterministisches Ende hat, sondern von Fall zu Fall unterschiedlich ist, hat die Funktion einen Standardfall, bei dem immer `true` zurückgegeben wird. In diesem Fall muss sich der Entwickler aber selbst darum kümmern, das Spiel mit dem Schlüsselwort `das spiel ist zu ende.` zu beenden. Methoden dieser Art, also die gegeben sein müssen, aber vom Nutzer gefüllt werden können, gibt es noch weitere, beispielsweise die Funktion `isActive()`, die nach einer von Entwickler definierten Bedingung entscheidet, ob ein Spieler aktiv ist oder nicht.

4.4 Erfüllung der geforderten Sprachqualitäten

Zentrale Qualität der Sprache ist die Einfachheit und die daraus folgende Robustheit durch Einschränkung der Mächtigkeit. Ein Beispiel für die Erfüllung dieses Kriteriums liegt in der automatischen Interpretation von Würfelreferenzen. Wird ein Würfel in einer arithmetischen Operation verwendet, so wird stets der aktuell angezeigte Wert referenziert. Dem Spieleentwickler ist es damit nicht möglich einen Würfel mit einer Zahl zu addieren, er adressiert in diesem Fall automatisch das Ergebnis.

Ein Beispiel, in dem die geforderte Robustheit noch fehlt, ist die Verwendung von `für`-Schleifen. Der Nutzer muss hier einen Bezeichner definieren, auf den er im Schleifenrumpf zugreifen kann. Denkbar wäre eine vereinfachte Version, in der Variablen des iterierten Objektes und die des globalen Scopes zusammenliegen und direkt verwendet werden können. Diese Variante würde die Syntax vereinfachen. In dieser Version unserer Sprache wurde zu Gunsten einer einfacheren Implementierung für eine konventionelle For-Schleife entschieden.

4.5 Mögliche Erweiterungen

Im Laufe der Entwicklung von DiGaL wurde Konzepte entwickelt und diskutiert, die es nicht in die Implementierung geschafft haben oder sich im Nachhinein als falsch oder unzulänglich erwiesen haben.

Zunächst wäre da die Implementierung des Frameworks zu nennen. Hier gibt es in der aktuellen Version konkrete Klassen, die dynamisch mit den von Nutzer spezifizierten Elementen gefüllt werden. Da mit Python jedoch eine objektorientierte Sprache verwendet wird, wäre es sinnvoller abstrakte Basisklassen von Spielern, Würfeln und dem Spiel zur Verfügung zu stellen, von denen eine implementierte Klasse angelegt wird, die von der entsprechenden Klasse erbt. Somit hätte man auch die Objektorientiertheit, die in DiGaL teilweise inhärent gegeben ist auch auf die Implementierung angewandt. Eine Möglichkeit der Modularität wäre somit gegeben, die in der aktuellen Version jedoch verloren geht.

Außerdem hat DiGaL einige syntaktische Konstrukte, die von der Entscheidung, die Sprache so intuitiv und einfach wie möglich zu gestalten, abweichen. Zu erwähnen ist die Adressierung und Referenzierung von Würfeln. Im aktuellen Zustand haben Würfel Namen, die der Entwickler vorgeben muss. Will man nun den Wert dieses Würfels haben, kann das auf zwei Arten geschehen. Entweder über den Namen oder über einen Index. Dieser Index wird allerdings erst logisch sinnvoll, wenn eine Sortierung der Würfel stattgefunden hat. Außerdem scheint die Adressierung über den Namen nicht aus der Domäne hervorzugehen, da man Würfeln normalerweise keine Namen gibt. Um das zu

erleichtern oder intuitiver zu gestalten, könnten beispielsweise neue syntatkische Konstrukte wie *Pasch* oder *Straße* eingeführt werden, die die Überprüfung dieser Ereignisse vereinfacht.

Schließlich waren zu Beginn Konstrukte geplant, die es dem Entwickler ermöglichen eigene Nutzerinteraktion zu definieren, beispielsweise Ausgaben auf der Kommandozeile oder die Möglichkeit auf Ereignisse dynamisch zu reagieren und nicht nur nach fest definierten Regeln vorzugehen. Der Fokus lag jedoch zunächst auf den Konzepten, die von der Domäne gegeben sind und die Sprache möglichst einfach zu gestalten. Daher wurde dieser Punkt mit einer niedrigen Priorität hinten angestellt und ist in der aktuellen Referenzimplementierung nicht enthalten.

Kapitel 5

Erfahrungen

Die erste Lektion, die wir gelernt haben, ist, dass das Konzept der natürlichen Sprache Mehrdeutigkeiten mitbringt. Um diese Mehrdeutigkeiten zu verhindern, müssten wir Konstrukte wie Klammern oder erweiterte Interpunktion einführen, was jedoch mit diesem Konzept brechen würde. Als Folge der Verwendung natürlicher Sprache wurde das gleichzeitige Erlernen und Implementieren der Sprache mit Hilfe von ANTLR zu einer Herausforderung, sodass viel Zeit auf diesen Teil verwendet werden musste und einige diskutierten Konzepte nicht umgesetzt werden konnten. Auch für die Implementierung konnte nur ein kleiner Anteil der Zeit investiert werden. Als Lehre daraus ziehen wir, dass wir in künftigen Neuentwicklungen auf eine formellere, aber dennoch leicht zu verstehende Sprache setzen würden. Statt nur auf die Sprache zu setzen, wäre es denkbar den Entwickler durch eine einfache, Baukastenartige Entwicklungsumgebung zu unterstützen.

Weiterhin haben wir versucht, möglichst viele Konzepte, die die Domäne mitbringt, in DiGaL umzusetzen. Im Laufe der Entwicklung haben wir festgestellt, dass eine strikte Umsetzung dieser Konzepte Nachteile mit sich bringt. Für fortgeschrittene Nutzer sind weitergehende Konzepte, die nicht domäneninhärent sind wünschenswert. Hier ist an erste Stelle das Konzept der Vererbung im Sinne der Erweiterbarkeit von Spielen zu nennen.

Außerdem würden wir zunächst die benötigten Technologien lernen, sodass sich die Konzeption der Sprache und die Erlernung dieser nicht gegenseitig behindern. Auch eine iterative Entwicklung, bei der zunächst kleine Teile implementiert und diese anschließend erweitert werden ist wünschenswert. Die Wahl der zugrundeliegenden Technologien wie beispielsweise Parsergeneratoren sollte fundiert getroffen und ausführlich diskutiert werden, da verschiedene Technologien zu unterschiedlichen Zwecken entwickelt wurden.

Schließlich haben wir in der Umsetzung des Compilers einige Konzepte von DiGaL nicht direkt übernommen, was uns an manchen Punkten Schwierigkeiten bereitet hat, da bereits erdachte und diskutierte Lösungen umgeworfen werden mussten. Beispielsweise gibt es in DiGaL nur einen globalen Scope, der in Python jedoch unzulänglich umgesetzt ist. Auch eine bessere Wahl der Zielsprache könnte diskutiert werden. Es sollte eine Sprache gewählt werden, die der eigenen konzeptuell möglichst nah ist. In einer Neuimplementierung würden wir auf Java setzen, um das Framework als Sammlung von abstrakten Superklassen umzusetzen und die Spiele davon erben zu lassen. Eine Erweiterbarkeit von Spielen wäre ebenfalls einfach implementierbar. Denkbar wäre auch eine Implementierung, die nicht als Compiler sondern als Interpreter umgesetzt ist.

Kapitel 6

Anhang

6.1 Codebeispiel

```
MAX wird so gespielt:

das spiel hat den wert LETZTES ist 0.
das spiel ist für 2 bis 10 spieler geeignet.
das spiel läuft solange anzahl aktiver spieler größer als 1.
das spiel hat folgende würfel:
würfel A hat diese seiten: 1 2 3 4 5 6
würfel B hat diese seiten: 1 2 3 4 5 6.

spieler haben die werte PUNKTE ist 3.
spieler sind aktiv, solange PUNKTE größergleich 0 gilt.

ist ein spieler am zug macht er folgendes:
wenn PUNKTE von aktueller spieler kleiner 0, dann ist linker spieler dran.
würfelt mit allen würfeln.
sortiert alle würfel absteigend.
ERGEBNIS ist würfel 0 * 10 + würfel 1.
wenn würfel 0 gleich würfel 1, dann setze ERGEBNIS auf ERGEBNIS * 10.
wenn ERGEBNIS gleich 21, dann
     für alle spieler S setze PUNKTE von S auf (PUNKTE von S - 1); und
     setze aktueller spieler PUNKTE auf (PUNKTE von aktueller spieler + 1),
     setze LETZTES auf 0 und
     aktueller spieler ist dran.
wenn ERGEBNIS kleiner als LETZTES, dann
     aktueller spieler PUNKTE ist PUNKTE von aktueller spieler - 1 ,
     setze LETZTES auf 0 und
     linker spieler ist dran.
wenn ERGEBNIS größergleich LETZTES, dann
     setze LETZTES auf ERGEBNIS und
     linker spieler ist dran.

gewonnen hat der spieler, bei dem PUNKTE größergleich 0.
```

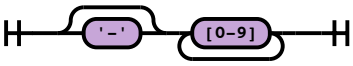
6.2 Generierter Code

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import random

class Player:
    def __str__(self): return self.name + ', ' + ', '.join(map(str, [value for key, value in self.__dict__.items() if key not in ['name']]))
    def __init__(self, name): self.name = name; self.PUNKTE = 3;
    def isActive(self): return self.PUNKTE >= 0;

class Dice:
    def __init__(self, name, faces): self.name = name; self.faces = faces; self.roll()
    def roll(self): self.value = random.choice(self.faces)
    def __str__(self): return self.name+': '+str(self.value)
class Game:
    # Static methods
    def status(self): return 'Status[name, ' + ', '.join([key for key, value in self.players[0].__dict__.items() if key not in ['name']]).lower()+']: '+' - '.join(map(str, self.players))
    def rightPlayer(self): return self.players[(self.players.index(self.activePlayer) - 1) % self.playerCount]
    def leftPlayer(self):  return self.players[(self.players.index(self.activePlayer) + 1) % self.playerCount]
    def playerNum (self, num): return self.players[num]
    def rollDices(self): map(Dice.roll, self.dices)
    def sortDices(self, desc=False): self.dices = sorted(self.dices, key=lambda dice: dice.value, reverse=desc)
    def setup(self):
        self.playerCount = int(raw_input('Enter number of players: '))
        if self.playerCount < self.min: print(self.name+' is made for more than '+str(self.min)+' players. Bring some friends ;-)'); exit()
        if self.playerCount > self.max: print(str(self.playerCount)+' players? Thats too much for '+self.name+'... Maximum: '+str(self.max)); exit()

        for p in range(self.playerCount): name = raw_input('Enter name of player #'+str(p)+': '); self.players.append(Player(name))
        self.activePlayer = self.players[0]
        print('Game initialized '+self.status())

    def isRunning(self): return (len([aktiver spieler, aktiver spieler]) > 1)
    def __init__(self):
        self.players = []
        # Dynamic inits
        self.name = 'MAXCHEN'

        self.LETZTES = 0
        self.min = 2
        self.max = 10
        self.dices = [Dice('A',[1, 2, 3, 4, 5, 6, ]), Dice('B',[1, 2, 3, 4, 5, 6, ]), ]
    def loop(self):
        while self.isRunning():
            print(self.status())
            print('\n'+self.activePlayer.name+' ist dran.'),

```

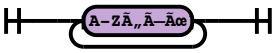
```
46              if self.activePlayer.PUNKTE < 0:
47                  self.activePlayer = self.leftPlayer(); continue
48              raw_input('Enter drücken zum Würfeln...'); map(Dice.roll,
                    self.dices); print('Du hast '+', '.join(map(str, [dice.value
                    for dice in self.dices]))+' gewürfelt!')
49              self.sortDices(desc=True)
50              self.ERGEBNIS = self.dices[0].value * 10 + self.dices[1].value
51              if self.dices[0].value == self.dices[1].value:
52                  self.ERGEBNIS = self.ERGEBNIS * 10
53              if self.ERGEBNIS == 21:
54                  for self.S in self.players:
55                      self.S.PUNKTE = (self.S.PUNKTE - 1)
56
57                  self.activePlayer.PUNKTE = (self.activePlayer.PUNKTE + 1)
58                  self.LETZTES = 0
59                  self.activePlayer = self.activePlayer; continue
60              if self.ERGEBNIS < self.LETZTES:
61                  self.activePlayer.PUNKTE = self.activePlayer.PUNKTE - 1
62                  self.LETZTES = 0
63                  self.activePlayer = self.leftPlayer(); continue
64              if self.ERGEBNIS >= self.LETZTES:
65                  self.LETZTES = self.ERGEBNIS
66                  self.activePlayer = self.leftPlayer(); continue
67
68  if __name__ == '__main__':
69      game = Game()
70      game.setup()
71      game.loop()
72      print('\nSpiel beendet: '+game.status())
73      print([self.name for self in game.players if self.PUNKTE >= 0][0]+' hat
            gewonnen!')
```

6.3 Abstrakte Syntax

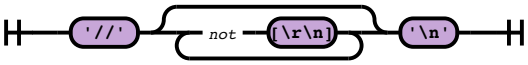
INT



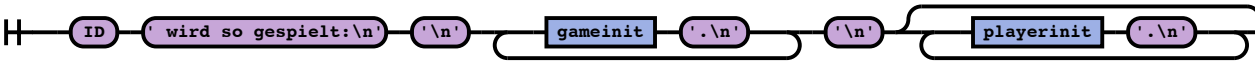
ID



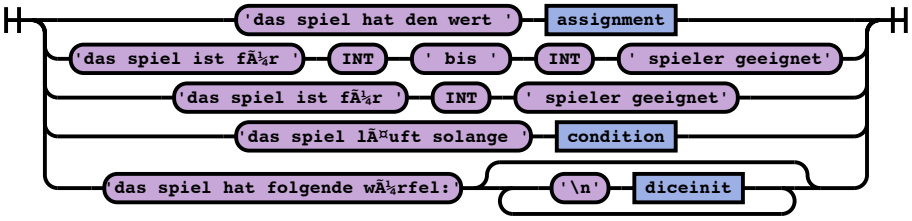
COMMENT



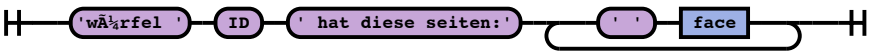
game



gameinit

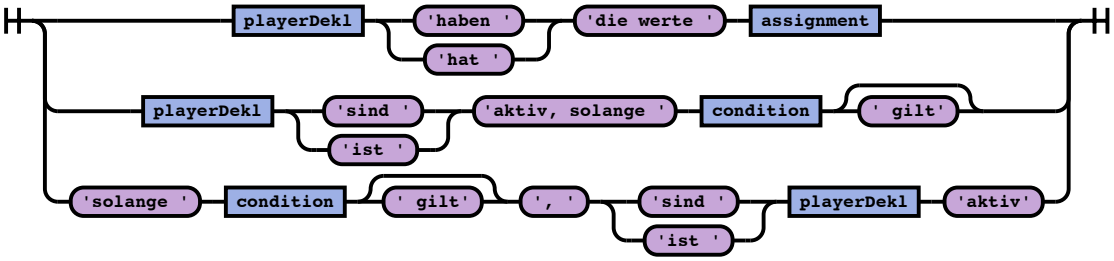


diceinit

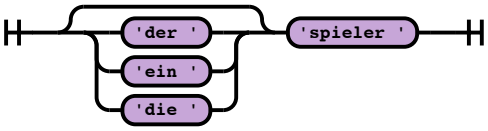


face

playerinit

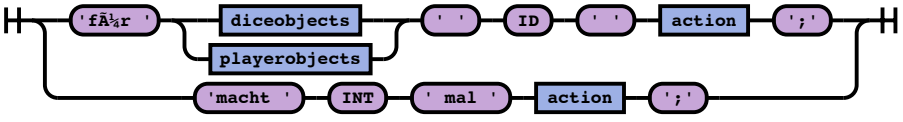


playerDekl

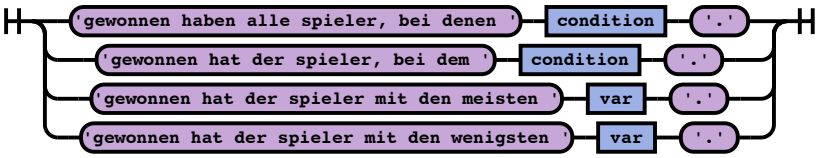


var

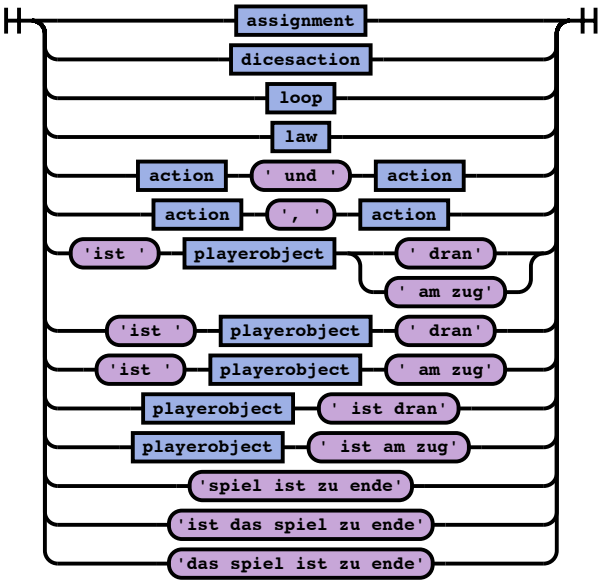
loop



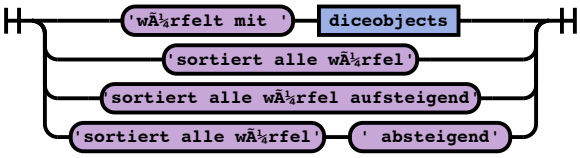
gameend



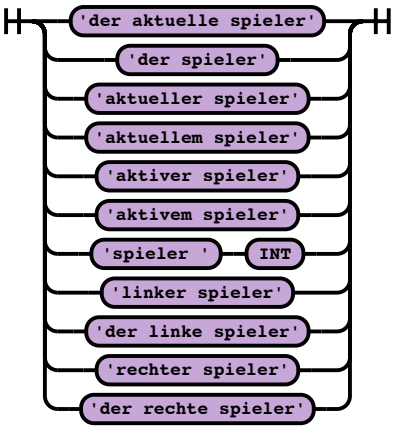
action



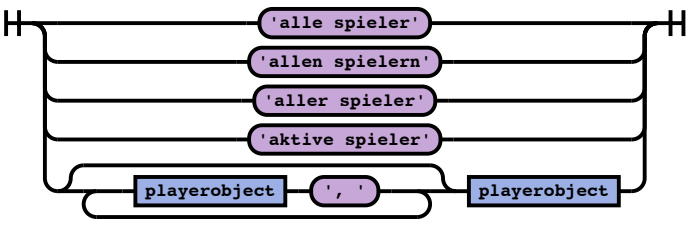
dicesaction



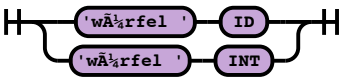
playerobject



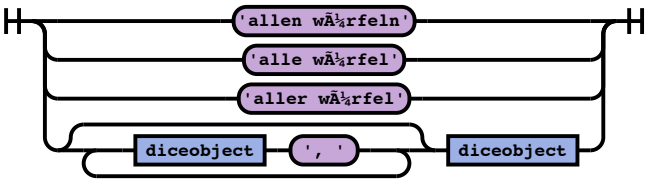
playerobjects



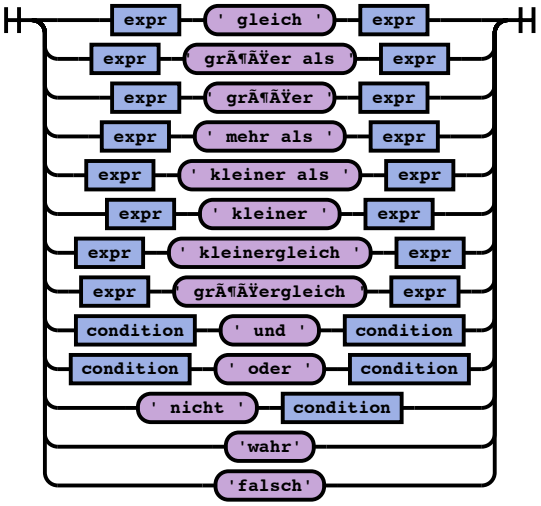
diceobject



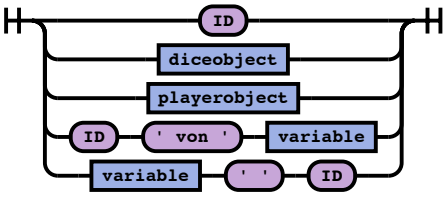
diceobjects



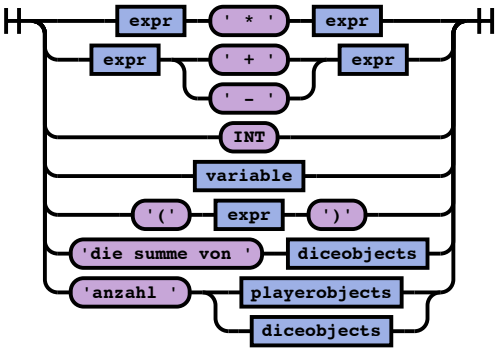
condition



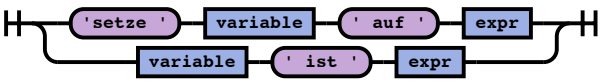
variable



expr



assignment



law