Maturaarbeit

Lizard - Entwurf einer höheren Programmiersprache über CHIP-8

Simon L. Hauswirth

Betreut durch
Stefan Rothe

22. Oktober 2021



Gymnasium Kirchenfeld
Abteilung MN

Inhaltsverzeichnis

1 Einleitung **3**
- 1.1 Fragestellung . . . 3
- 1.2 Einführung . . . 3
  - 1.2.1 Erläuterung zur Darstellung von Zahlen in diesem Dokument . . . 3
  - 1.2.2 Erläuterung zur Darstellung von CHIP-8-Opcodes in diesem Dokument . . . 3
  - 1.2.3 Erläuterung zu den Begriffen interpretierte und kompilierte Programmiersprache 3
  - 1.2.4 Über CHIP-8 . . . 4

2 Hauptteil **7**
- 2.1 Arbeitsprozess . . . 7
- 2.2 Parser . . . 7
- 2.3 Compilerbau . . . 7
  - 2.3.1 Aufbau des Compilers . . . 7
  - 2.3.2 Variablen . . . 8
  - 2.3.3 Ausdrücke . . . 9
  - 2.3.4 Variablendeklarationen . . . 13
  - 2.3.5 Fallunterscheidungen . . . 13
  - 2.3.6 Pixel zeichnen . . . 15
  - 2.3.7 Dezimalzahlen ausgeben . . . 16
  - 2.3.8 Benutzerdefinierte Sprites ausgeben . . . 16
  - 2.3.9 Benutzerdefinierte Sprites bis zu einer Höhe von 10 Pixeln ausgeben . . . 16
  - 2.3.10 Schleifen . . . 17
  - 2.3.11 Eingabe . . . 18
  - 2.3.12 Subroutinen . . . 19
  - 2.3.13 Timer und Sound . . . 20
  - 2.3.14 Kollisionsdetektion, Übertrag und negativer Übertrag . . . 20
  - 2.3.15 Zufall . . . 21
  - 2.3.16 Kommentare . . . 21
- 2.4 Fazit . . . 21
  - 2.4.1 Welche Features von CHIP-8 werden in Lizard nicht abgebildet? . . . 21
  - 2.4.2 Typische Features von anderen Programmiersprachen . . . 22
  - 2.4.3 Das Programmieren von CHIP-8 Spielen mit Lizard . . . 22

1 Einleitung

1.1 Fragestellung

In meiner Maturaarbeit versuche ich zu beantworten, wie höhere Programmiersprachen funktionieren. Dafür entwerfe ich selbst eine höhere Programmiersprache namens ”Lizard” und programmiere in Python 3 einen Compiler. Der Compiler soll Quellcode von meiner eigenen Programmiersprache in ein CHIP-8 Programm übersetzen. Ich werde dabei die nötigen Grundlagen erarbeiten und die Funktionsweise des Compilers ausführlich beschreiben.

1.2 Einführung

1.2.1 Erläuterung zur Darstellung von Zahlen in diesem Dokument

Hexadezimale Zahlen werden mit einem ”0x”’ am Anfang geschrieben, beispielsweise 0x200. Ausnahmen sind CHIP-8-Opcodes. Für die Darstellung von hexadezimalen Ziffern, die grösser als 9 sind werden die Buchstaben A bis F verwendet, beispielsweise 0xA oder 0x23C. Binäre Zahlen werden mit einem ”0b” am Anfang geschrieben, beispielsweise 0b1010.

1.2.2 Erläuterung zur Darstellung von CHIP-8-Opcodes in diesem Dokument

Folgende Variabeln werden in diesem Dokument verwendet, um Opcodes darzustellen: X und Y für die Adresse eines beliebigen CHIP-8-Registers von V0 bis VF, N für eine beliebige hexadezimale Ziffer, NN für eine beliebige 1 Byte lange hexadezimale Zahl und NNN für eine beliebige hexadezimale 12-Bit-Zahl.

1.2.3 Erläuterung zu den Begriffen interpretierte und kompilierte Programmiersprache

Ein Programm, das in einer interpretierte Programmiersprache geschrieben wurde, wird von einem anderen Programm, das als Interpreter bezeichnet wird, gelesen und ausgeführt [3]. Bei einer kompilierten Sprache wird das Quellprogramm von einem Programm, das als Compiler bezeichnet wird, in ein äquivalentes Programm in der Zielsprache übersetzt (Aho, Sethi und Ullman [2, S. 1]). Die

Zielsprache kann eine andere Programmiersprache oder eine Maschinensprache eines beliebigen Systems sein (Aho, Sethi und Ullman [2, S. 1]).

1.2.4 Über CHIP-8

Was ist CHIP-8?

CHIP-8 ist eine einfache interpretierten Programmiersprache [6]. Sie wurde in den 1970er-Jahren von Joseph Weisbecker entwickelt [10]. Weisbecker schrieb in einem Artikel in der Zeitschrift ”Byte”, dass eine solche hexadezimale interpretierte Programmiersprache weniger Hardware benötigen würde als höhere Sprachen wie z. B. BASIC, aber einfacher zu benutzen sei als Maschinensprache [17]. Er präsentierte in dem Artikel ein Beispiel für ein Videospiel, das in CHIP-8 auf einem RCA COSMAC VIP geschrieben wurde [17]. In den späten 1970er-Jahren und den frühen 1980er-Jahren kam CHIP-8 unter anderem auf Computersystemen wie COSMAC VIP, DREAM 6800 und ETI 660 zum Einsatz [6]. Solche Geräte verwendeten typischerweise ein TV-Gerät als Display und hatten 1 bis 4 KB RAM [6].
In den frühen 1990er-Jahren schrieb Andreas Gustafsson einen CHIP-8 Interpreter für den grafischen Taschenrechner HP48, welcher CHIP-48 genannt wird [6]. Damals gab es noch keine andere einfache Möglichkeit, auf dem HP48 ein schnelles Game zu programmieren [6]. CHIP-48 inspirierte zahlreiche CHIP-8 Implementationen für andere Plattformen [6].

Speicher

CHIP-8 verfügt über 4 KB Speicher mit den Adressen von 0x000 bis 0xFFF [6]. Das CHIP-8-Programm selbst wird in diesen Speicher geladen [6]. Ursprünglich wurden die ersten 512 Bytes vom Interpreter selbst verwendet und sollten daher von keinem Programm verwendet werden [6]. Die meisten Programme beginnen bei 0x200 [6].

Register

CHIP-8 hat 16 Register mit jeweils 8 Bit die üblicherweise als VX bezeichnet werden, wobei X eine Hexadezimalziffer ist [6]. Das Register VF wird bei manchen CHIP-8 Instruktionen als Flag verwendet und sollte daher nicht anderweitig verwendet werden [6]. Zudem gibt es einen 16-Bit program counter (PC) und einen 16-Bit Speicherzeiger (I) [6]. Bei I werden üblicherweise nur die unteren 12 Bit verwendet [6].

Timer und Ton

CHIP-8 hat einen sound timer und einen delay timer, deren Wert jeweils in einem eigenen, speziellen 8-Bit Register gespeichert wird [6]. Solange ein Timer einen Wert hat, der nicht Null ist, zählt er mit

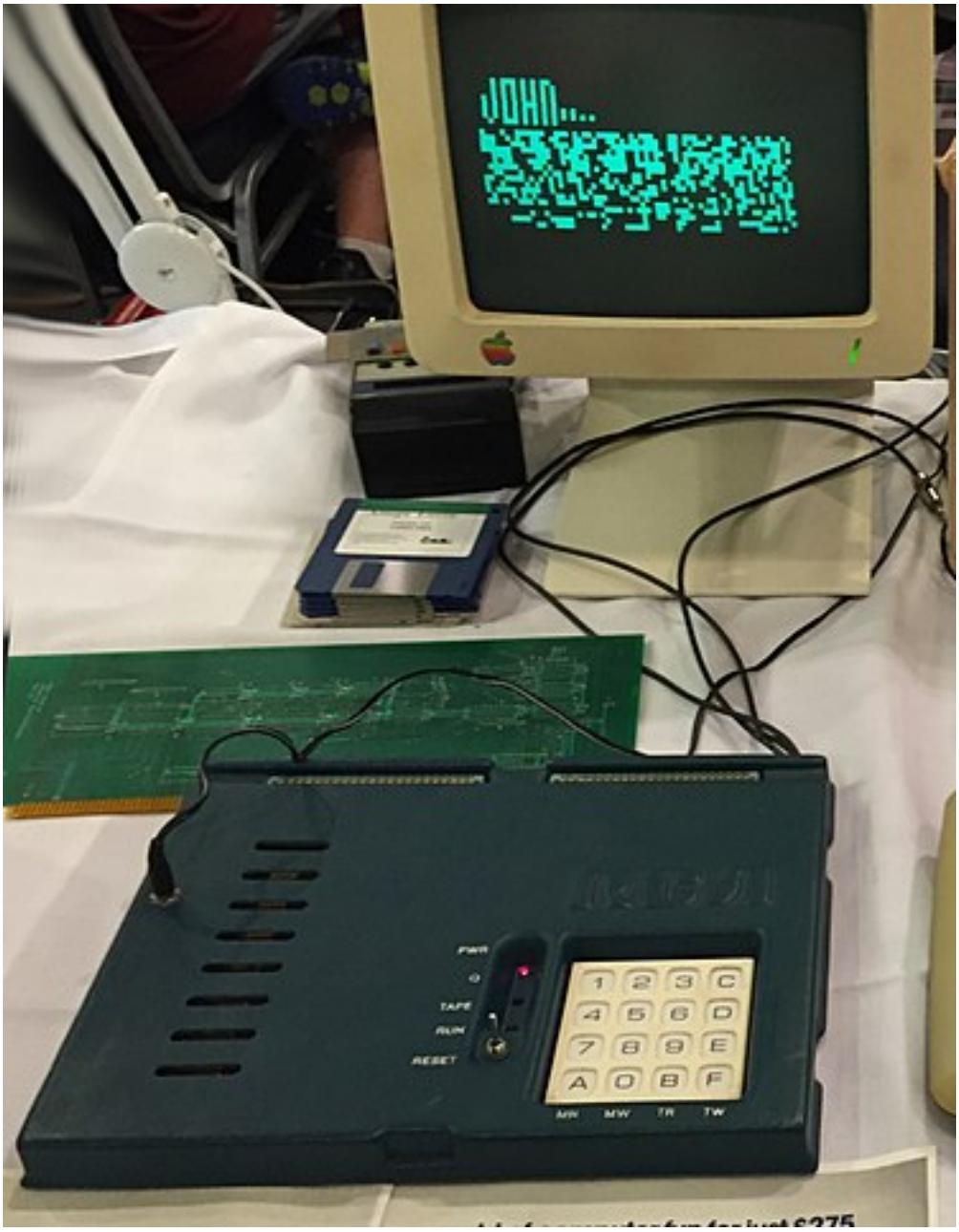


Abbildung 1.1: Ein RCA COSMAC VIP

einer Frequenz von 60Hz hinunter [6]. Der sound timer erzeugt dabei im Gegensatz zum delay timer einen Ton [6].

Eingabe

Die Computer, auf denen CHIP-8 ursprünglich verwendet wurde, hatten eine hexadezimale Tastatur mit 16 Tasten mit folgendem Layout [6]:

1	2	3	C
4	5	6	D
7	8	9	E
A	0	B	F

Abbildung 1.2: Das Layout der hexadezimalen Tastatur

Ausgabe

CHIP-8 Interpreter verwendeten Ursprünglich 64x32-Displays [6]. Auf dieses Display werden mit sogenannten Sprites Grafiken gezeichnet [6]. Ein Sprite ist eine Gruppe von Bytes, welche die binäre Repräsentation vom jeweiligen Bild darstellen [6]. Folgende Grafik wurde beispielsweise mit dem in der unten stehenden Tabelle dargestellten Sprite erzeugt:



(a) Der angezeigte Sprite

Speicheradresse	Gespeichertes Byte
0x300	0b00100100
0x301	0b00100100
0x302	0b00000000
0x303	0b01000010
0x304	0b00111100

(b) Der Sprite im Speicher

Abbildung 1.3: Ein Beispiel eines Sprites

Ein Sprite kann in CHIP-8 bis zu 15 Bytes enthalten, wodurch eine Grafik von maximal 8x15 Pixeln mit einem einzigen Opcode ausgegeben werden kann [6]. In einem CHIP-8 Programm kann auch auf Sprites zurückgegriffen werden, die die hexadezimalen Ziffern repräsentieren [6]. Diese Sprites sind je 5 Bytes lang [6]. Dieses Fontset wird an einer je nach Implementation unterschiedlichen Stelle in den ersten 512 Bytes des Speichers (Adresse 0 bis 0x1FF) gespeichert [9].

2 Hauptteil

2.1 Arbeitsprozess

Die verschiedenen Teile des Compilers und Features von Lizard wurden ungefähr in der Reihenfolge implementiert, in der sie im Folgenden erläutert werden. Der Arbeitsprozess lässt sich grob in drei Phasen aufteilen, die sich ständig wiederholten: Das implementieren eines neuen Features, die Durchführung von Tests und das beheben von Fehlern, die beim Testen entdeckt werden.

2.2 Parser

Der Zweck eines Parsers ist, die Informationen aus dem Quellcode zu analysieren, aufzubereiten und nach einem bestimmten Schema zu strukturieren, damit sie für den Compiler verwertbar sind [1]. Es gibt Tools, die Parser generieren können, welche zusammen mit Python oder anderen Sprachen verwendet werden können [16]. Diese Tools nennt man parser generators [16]. Es gibt auch Python Bibliotheken, um Parser zu bauen [16]. Diese nennt man parser combinators [16]. Der Autor hat entschieden, für dieses Projekt Lark zu verwenden. Lark ist ein parsing-toolkit für Python [5]. Es funktioniert als Python Bibliothek und benötigt die Grammatik der entsprechenden Sprache in der erweiterten Backus-Naur-Form (EBNF) in einer Datei oder einer Zeichenkette als Argument, um den Parser dynamisch zu generieren [16]. Lark stellt auch Tools zur Verfügung, um den Parse Tree zu verarbeiten [5].

2.3 Compilerbau

2.3.1 Aufbau des Compilers

Zuerst wird der Quellcode vom Parser in einen parse tree umgewandelt. Das eigentliche Kompilieren läuft in der `Compiler`-Klasse ab. Diese erbt von Lark's `Interpreter`-Klasse, welche über eine Funktion namens `visit` verfügt. Diese nimmt den parse tree als Argument und geht diesen, beginnend bei der Wurzel, zu den Ästen hin durch [14]. Wenn eine bestimmte Regel, welche in der Grammatik der Sprache definiert wird, darin vorkommt, wird, sofern vorhanden, eine entsprechende benutzerdefinierte Funktion dafür in der `Compiler`-Klasse abgerufen (vgl. [14]). Der Autor konnte also für jede Regel eine entsprechende Funktion definieren. Die CHIP-8 Instruktionen die generiert werden, werden in einer Liste gespeichert.

Aufbau der generierten CHIP-8 Zielprogramme

Alle generierten Zielprogramme beginnen, wie bei CHIP-8 üblich, auf Zeile 0x200 mit dem Hauptteil des Programms. Am Ende des Hauptteils wird ein Jump eingefügt, der zur höchsten zugänglichen Speicheradresse (0xFFD) springt um das Programm zu beenden. Danach folgen, sofern vorhanden, die Subroutinen des Programms. Die darauf folgenden Speicherstellen sind bis zu 0xFF9 (einschliesslich 0xFF9 selbst) frei und können zum Speichern von Variablen verwendet werden. Die Speicherstellen 0xFFA bis 0xFFC dürfen nicht für Variablen verwendet werden (vgl. Abschnitt 2.3.7).

2.3.2 Variablen

Der Compiler muss jeder Variable eine Adresse im Speicher zuweisen. Variablennamen helfen beim Programmieren, eine Variable gut wiederzuerkennen. Der Compiler muss also zu jeder Variable, die im Programm zugewiesen wird, den Namen und eine automatisch zugewiesene Adresse speichern. Der Autor schlägt vor, im Progamm des Compilers zwei Klassen zu implementieren, eine `Variable`-Klasse, welche dazu dient, Adresse und Name einer Variable zu speichern und eine `VariableHandler`-Klasse, welche dazu dient, die Variablen zu verwalten.

Die `VariableHandler`-Klasse

Die `VariableHandler`-Klasse kann als Teil des sogenannten Symbol-Table Managers betrachtet werden (vgl. Aho, Sethi und Ullman [2, S. 11]). Sie enthält eine Liste in der alle Variable-Objekte (mit der `Variable`-Klasse erstellt) gespeichert sind, was einem Symbol-Table entspricht (vgl. Aho, Sethi und Ullman [2, S. 11]). Zudem enthält die Klasse die `match`-Funktion, welche einen Variablennamen als Argument nimmt und `True` zurückgibt, wenn eine Variable mit diesem Namen existiert, oder `False` wenn dies nicht der Fall ist.
Bei einer Variablenzuweisung wird die `assign`-Funktion aufgerufen, welche dann ein Variable-Objekt erstellt und dies der Liste hinzufügt nachdem mit der `match`-Funktion überprüft wurde, ob diese Variable nicht bereits existiert. Sollte die Variable tatsächlich schon existieren, wird der Rest der Funktion übersprungen.
Die `store`-Funktion nimmt ebenfalls den Variablennamen als Argument und gibt die CHIP-8 Befehle zurück, die dazu benötigt werden, die Variable zu speichern. Der Wert der Variable muss aber vorher im Register V0 gespeichert werden. Die `recall`-Funktion gibt die Befehle zurück, die benötigt werden, um den Wert abzurufen bzw. wieder in V0 zu speichern. Wenn versucht wird, auf den Wert einer undefinierten Variable zuzugreifen, wird vom Compiler eine Fehlermeldung ausgegeben.
Die Adressen der Variablen müssen so gewählt werden, dass nicht Teile des Programms oder das Fontset des Interpreters überschrieben werden.
Da das Programm bei Adresse 0x200 beginnt und die Adressen vor 0x200 wegen dem Fontset nicht verwendet werden sollten, kommen die Speicherstellen für die Variablen nach dem Programm. Jedoch ist während dem Kompilieren noch nicht bekannt, wie lang das Programm sein wird und somit ist auch unklar, ab wo es im Speicher Platz haben wird. Daher kann den Variablen nicht sofort eine Adresse zugewiesen werden. Der Compiler fügt daher für den entsprechenden Opcode, der den Speicherzeiger I auf die entsprechende Adresse setzen soll, jeweils einen Platzhalter ein. In diesem Platzhalter ist der Name der Variable gespeichert. Sobald das Programm bis auf derartige

Platzhalter kompiliert ist und somit dessen Länge bekannt ist, wird die `generateAdresses`-Funktion des `VariableHandler` aufgerufen, mit der niedrigsten freien Adresse als Argument. Diese Funktion weist jeder Variable aus der Liste eine Adresse zu, welche gleich im entsprechenden Objekt gespeichert wird. Am Schluss müssen die Platzhalter durch die richtigen Opcodes ersetzt werden. Weil darin der jeweilige Name gespeichert ist, kann dieser als Argument verwendet werden, um die `getAdress`-Funktion des `VariableHandler` aufzurufen, welche die zugehörige Adresse zurück gibt.

2.3.3 Ausdrücke

Ein Ausdruck kann beispielsweise ein numerischer Wert, eine Variable oder eine mathematische Operation mit anderen Ausdrücken sein. Diese Arten von Ausdrücken müssen entsprechend vom Compiler auch unterschiedlich verarbeitet werden. Für jeden Ausdruck im Quellprogramm wird ein Objekt erstellt, in dem die relevanten Informationen gespeichert werden. Welche Klasse dazu verwendet wird ist von der Art des Ausdrucks abhängig. Manche Ausdrücke sind mathematische Operationen aus zwei anderen Ausdrücken. In diesem Fall wird ein Objekt erstellt, in dem zwei weitere Objekte gespeichert sind. Wenn nötig kann eines dieser Objekte wiederum zwei weitere Objekte enthalten, beispielsweise bei einer Addition mit drei Summanden. So wird Linksrekursion ermöglicht.
Jedes für einen Ausdruck generierte Objekt enthält eine Funktion namens `translate` welche die entsprechenden CHIP-8 Befehle für den Ausdruck zurück gibt.

Operatorrangfolge

Folgende Tabelle zeigt alle implementierten mathematischen Operatoren nach deren Präzedenz in absteigender Reihenfolge geordnet:

Operator	Beschreibung
`( )`	Operation in Klammern
`* // %`	Multiplikation, ganzzahlige Division und Modulo-Operator
`+ -`	Addition und Subtraktion
`<< >>`	Byte nach links bzw. rechts Verschieben
`&`	Bitweiser UND-Operator
`|`	Bitweiser exklusiver ODER-Operator (XOR)
`^`	Bitweiser ODER-Operator

Tabelle 2.1: Die Operatorrangfolge

Der Autor hat sich hierbei an der Operatorrangfolge von Python (vgl. [15]) orientiert. Die Operatorrangfolge muss in der Grammatik berücksichtigt werden, damit der Parser entsprechende parse trees generiert (vgl. Aho, Sethi und Ullman [2, S. 30-32]). Der Ausdruck `1 + 2 * 3` würde vom Parser zu folgendem parse tree übersetzt:

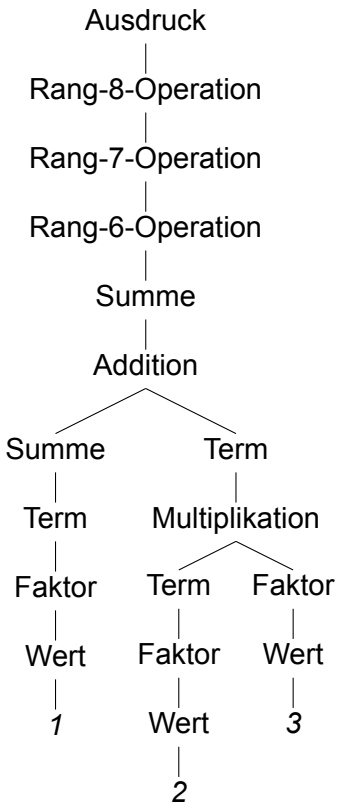


Abbildung 2.1: Der parse tree von 1 + 2 * 3

Ein abstract syntax tree (AST) davon:

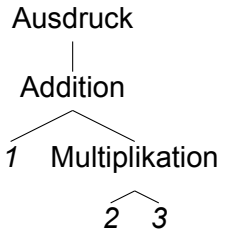


Abbildung 2.2: Der Abstract Syntax Tree von 1 + 2 * 3

Die Addition hat eine höhere Präzedenz als die Multiplikation und ist daher weiter oben im tree.

Division

In CHIP-8 gibt es keinen Opcode für die Division. Man kann aber durch systematisches Subtrahieren eine Ganzzahlendivision durchführen. Seien a und b zwei natürliche Zahlen, wobei b kleiner a sei. Gesucht sei a dividiert durch b. Nun wird b von a subtrahiert. Danach wird überprüft, ob das Resultat davon kleiner oder gleich b ist. Falls nicht, wird erneut b subtrahiert und geprüft, ob das Resultat kleiner oder gleich b ist. Dies wird solange wiederholt bis die Bedingung erfüllt ist. Danach wird die

Anzahl durchlaufener Iterationen ausgegeben. Dies lässt sich in einem Programm folgendermassen umsetzen:

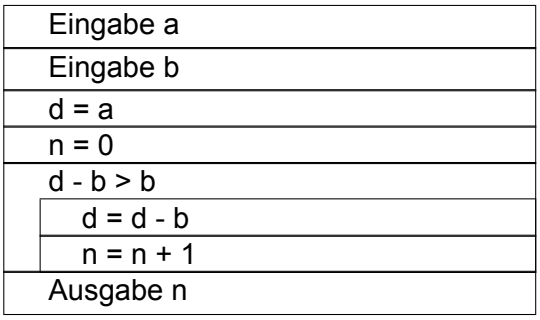


Abbildung 2.3: Ein Nassi-Shneiderman-Diagramm des zur Division verwendeten Algorithmus

Zusätzlich muss sichergestellt werden, dass der Divisor nicht 0 ist, da eine Division durch 0 mathematisch unmöglich ist [7]. Daher wird in der Implementation am Anfang eine Fallunterscheidung hinzugefügt, welche dies prüft. Sollte der Divisor 0 sein, wird ”E0” angezeigt, wobei ”E” für ”Error” steht. Da nur ein hexadezimales Fontset zur Verfügung steht und die nötigen Instruktionen um Buchstaben Pixelweise anzuzeigen sehr lang wären, darf die Fehlermeldung nur die Buchstaben von A bis F enthalten. Der Autor hat keinen Weg gefunden, mit diesen Einschränkungen eine verständlichere Fehlermeldung anzeigen zu lassen.

Modulus

Genau wie die Ganzzahlendivision lässt sich auch eine Modulo-Operation durch systematische Subtraktion durchführen. Der Unterschied ist, das hier anstelle der Anzahl Iterationen der Wert, der nach dem Ausführen von allen Subtraktionen noch übrigbleibt, ausgegeben wird. Dies lässt sich in einem Programm folgendermassen umsetzen:

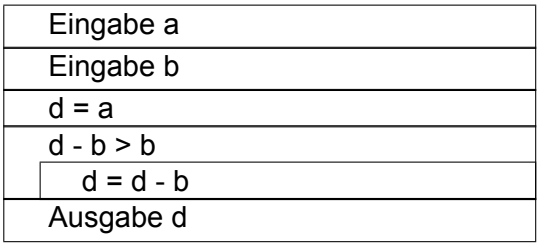


Abbildung 2.4: Ein Nassi-Shneiderman-Diagramm des zur Modulo-Operation verwendeten Algorithmus

Hier wird auf dieselbe Fehlermeldung wie bei der Division (vgl. Abschnitt 2.3.3) ausgegeben, wenn

versucht wird, den Rest einer Division durch 0 zu berechnen.

Multiplikation

In CHIP-8 gibt es keinen Opcode für die Multiplikation. Da in folgender Wahrheitstabelle die dritte und vierte Spalte übereinstimmend sind, kommt eine Multiplikation zweier Bits auf dasselbe Ergebnis wie eine bitweise UND-Operation. Dies gilt jedoch nicht für Zahlen die länger als ein einziges Bit sind. Beispielsweise gilt: $3 \wedge 2 = 2 \neq 3 \times 2$

a	b	$a \wedge b$	$a \times b$
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

Tabelle 2.2: Wahrheitstabelle mit Multiplikation und bitweiser UND-Operation

Der Autor schlägt vor, bei Zahlen die länger als ein Bit sind das Problem in Teilprobleme zu zerlegen, bei denen nur noch Zahlen multipliziert werden müssen, die nicht länger als ein Bit sind, ähnlich einer schriftlichen Multiplikation (vgl. [4]).

a und b seien zwei natürliche Zahlen die miteinander Multipliziert werden sollen.
Dabei gilt: a < 255, b < 255
Folglich haben a und b je maximal eine Länge von 8 Bites. Zuerst beginnt eine Schleife mit 8 Iterationen. Zu Beginn jeder Iteration ist die Variable l gleich Null. a wird nach jeder Iteration um 1 nach rechts geschoben. Die Idee ist, dass nun in jeder Iteration jedes Bit von b separat mit dem niederwertigsten Bit von a multipliziert bzw. bitweise UND-gerechnet wird und alle dabei herauskommenden Produkte zusammenaddiert werden. Diese Summe wird dann in l gespeichert. Dies entspricht dem Errechnen einer einzelnen Zeile in der schriftlichen Multiplikation. n ist die Anzahl bereits abgeschlossener Iterationen. l wird nun um n nach links verschoben. Danach wird die Variable d, welche zu Beginn der Schleife den Wert 0 hatte, um das Resultat davon erhöht. Dies entspricht dem stellengerechten Untereinanderschreiben und anschliessendem Addieren der Zeilen in der schriftlichen Multiplikation. Nachdem alle Iterationen abgeschlossen wurden, ist der Wert der Variable d gleich dem gewünschten Resultat. Als Programm lässt sich das folgendermassen Umsetzten:

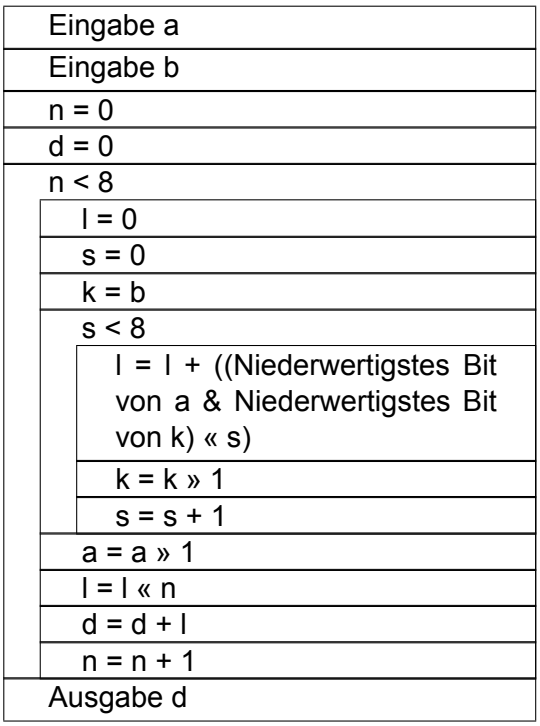


Abbildung 2.5: Ein Nassi-Shneiderman-Diagramm des zur Multiplikation verwendeten Algorithmus

2.3.4 Variablendeklarationen

Bei einer Variablendeklaration wird ein entsprechendes Objekt erstellt. Der zugehörige Ausdruck wird gelesen und ebenfalls ein Objekt daraus generiert, wie unter Abschnitt 2.3.3 beschrieben. Dieses Objekt wird in dem Objekt für die Variablendeklaration gespeichert, welches eine Funktion enthält, die das ganze kompiliert. Beim kompilieren werden, wie unter Abschnitt 2.3.3 erläutert, zuerst die CHIP-8 Instruktionen um den Wert des Ausdrucks in VE zu speichern generiert. Anschliessend werden Instruktionen hinzugefügt, um den Inhalt von VE nach V0 zu kopieren. Mithilfe des `VariableHandler`-Objekts (Symbol-Table Manager) werden schliesslich die Instruktionen generiert um den Inhalt von V0 in der richtigen Speicherstelle abzulegen.

2.3.5 Fallunterscheidungen

Die Grammatik der Fallunterscheidungen in der Erweiterten Backus-Naur-Form (EBNF):

```
comparison: 'if' condition '{' start* '}'
condition: expression '<' expression -> less
| expression '>' expression -> more
| expression '==' expression -> equal
| expression '<=' expression -> less_or_equal
| expression '>=' expression -> more_or_equal
```

```
| expression '!=' expression -> unequal
```

Im folgenden Beispielprogramm wird die Variable a auf die Summe von 1 und 2 gesetzt und anschliessend überprüft, ob a gleich 3 ist. Wenn die Bedingung erfüllt ist, wird die Variable b auf 1 gesetzt:

```
a = 1 + 2
if a == 3 {
          b = 1
}
```

Für jeden Vergleich im Quellprogramm erstellt der Compiler mithilfe der `Comparison`-Klasse ein entsprechendes Objekt. (Dieses entspricht in der Grammatik der `condition`-Regel und nicht der `comparison`-Regel.) Dieses Objekt enthält eine `translate`-Funktion, welche die CHIP-8 Befehle zum ausführen des Vergleichs zurück gibt.

In CHIP-8 gibt es nur Opcodes für die Vergleichsoperatoren gleich oder nicht-gleich (vgl. [6]). Der Autor schlägt daher vor, beispielsweise bei kleiner-oder-gleich den Opcode 8XY5 zu verwenden, der VY von VX subtrahiert und VF auf 0 setzt, wenn es einen negativen Übertrag gibt, und auf 1, wenn es keinen negativen Übertrag gibt. Wenn letzteres der Fall ist, muss der Wert in VY kleiner oder gleich gewesen sein.

Für jede Fallunterscheidung wird ein Objekt mit der `If_statement`-Klasse erstellt. Darin wird das oben genannte Vergleichs-Objekt gespeichert und der bereits kompilierte Abschnitt des Codes, welcher ausgeführt werden soll, falls die Bedingung erfüllt ist. Dieser Abschnitt des Codes wird kompiliert, indem das `Compiler`-Objekt ein weiteres Objekt mit derselben Klasse erstellt, genannt `Subcompiler`, und dieses den entsprechenden Teil des Trees verarbeiten lässt. Das `If_statement`-Objekt hat auch eine `translate`-Funktion, die die CHIP-8 Befehle zurück gibt. Diese beginnen mit den CHIP-8 Instruktionen für den Vergleich, generiert von der `translate`-Funktion des `Comparison`-Objekts. Nach einem Vergleich in CHIP-8 wird jeweils die nächste Zeile übersprungen wenn die Bedingung erfüllt ist [6]. Auf dieser Zeile fügt der Compiler einen Jump ein, der den Abschnitt, der bei erfüllter Bedingung ausgeführt werden soll, überspringt. Wenn die Bedingung also erfüllt ist, wird dieser Jump übersprungen und der Abschnitt wird normal ausgeführt. Anderenfalls wird der Jump nicht übersprungen und das Programm überspringt den Abschnitt.

Jumps

Unter Anderem für Fallunterscheidungen und Schleifen werden in CHIP-8 Jumps benötigt. Allerdings ist es nicht immer möglich zu bestimmen, wo die gewünschte Zeile im fertig kompilierten Programm sein wird, bevor das Programm vollständig kompiliert wurde. Aber es ist möglich, die Zeilennummer relativ zu der Zeilennummer, auf der sich der Jump befindet, zu bestimmen. Der Autor hat daher entschieden, beim Kompilieren anstelle eines Jumps zunächst ein Tupel in die Liste mit den CHIP-8 Instruktionen für das Zielprogramm einzufügen, das die Informationen enthält, in welche Richtung und um wie viele Zeilen gesprungen werden soll und zugleich als Platzhalter dient. Solche Tupel

werden in diesem Abschnitt fortan ”Platzhalter”genannt. Als letzter Schritt geht der Compiler dann nach dem eigentlichen Kompilieren die Liste mit den generierten Instruktionen durch. Wenn dabei ein Platzhalter anstelle eines Opcodes gefunden wird, wird anhand der darin enthaltenen Informationen die absolute Zeilennummer bestimmt (was zu diesem Zeitpunkt problemlos möglich ist) und der Platzhalter durch den entsprechenden Opcode für den Jump ersetzt.

2.3.6 Pixel zeichnen

Der Autor hat entschieden, einen Befehl namens `draw_pixel()` zu implementieren, mit dem auf vorgegebenen Koordinaten ein einzelnes Pixel ausgegeben werden kann.
Die Grammatik des `draw_pixel()`-Befehls in der erweiterten Backus-Naur-Form (EBNF):

```
command: 'draw_pixel' '(' expression ',' expression ')' -> pixel
```

In folgendem Beispielprogramm wird ein Pixel auf den Koordinaten (1|1) gezeichnet:

```
draw_pixel(1, 1)
```

Der erste Ausdruck in den Klammern ist für die x-Koordinate und der zweite für die y-Koordinate. Da CHIP-8 nur Sprites ausgeben kann, muss dafür erst ein Sprite im Speicher sein. Der für ein einzelnes Pixel benötigte Sprite entspricht der Zahl 0b10000000, weil Sprites in CHIP-8 im little-endian Format angegeben werden müssen. Damit der Sprite ausgegeben werden kann, muss er erst im Speicher des Interpreters sein. Dazu hat der Autor eine spezielle Variable eingeführt, auf die der/die Nutzer:in nicht zugreifen kann (dies wird gewärleistet in dem der Variabelname Zeichen enthält, die vom Parser nicht als Teil eines Varialenname akzeptiert würden). Wenn der `draw_sprite`-Befehl kompiliert wird, wird geprüft, ob diese Variable schon im Symbol-Table ist. Falls nicht wird sie hinzugefügt. Danach wird der Wert 0b10000000 in der Variable gespeichert.

Pixel löschen

Es muss berücksichtigt werden, dass dieser Befehl aufgrund der Funktionsweise von CHIP-8 ein Pixel von 1 auf 0 setzt, also löscht, wenn das Pixel auf den entsprechenden Koordinaten bereits auf 1 gesetzt wurde. Daher kann dieser Befehl auch bewusst dazu genutzt werden, ein Pixel wieder zu löschen. Folglich würde folgendes Beispielprogramm zuerst ein Pixel auf den Koordinaten (1|1) zeichnen, dieses dann aber sogleich wieder löschen:

```
draw_pixel(1, 1)
draw_pixel(1, 1)
```

Der `clearscreen`-Befehl

Der Befehl `clearscreen` wird in den CHIP-8 Opcode 00E0 übersetzt, der das Display zurücksetzt.

2.3.7 Dezimalzahlen ausgeben

Folgendes Beispielprogramm gibt ’123’ auf den Koordinaten (10|20) aus:

```
output_decimal(10, 20, 123)
```

Zu beachten ist, dass immer 3 Stellen ausgegeben werden, auch wenn die Zahl eigentlich kürzer wäre. Es würde daher beispielsweise nicht ’2’ sondern ’002’ ausgegeben.

Zum Kompilieren dieses Befehls werden unter anderem die CHIP-8 Opcodes FX33 und FX29 verwendet. Der CHIP-8 Opcode FX33 speichert die Zahl in VX als dualkodierte Dezimalzahl im Speicher [6]. Die erste Ziffer wird an der in I gespeicherten Adresse gespeichert, die zweite an der darauf folgenden Adresse und die dritte wiederum an der nächsten Adresse [6]. Zu diesem Zweck hat der Autor entschieden, dass die vom Compiler generierten CHIP-8 Programme die Adressen 0xFFA bis 0xFFC nicht belegen sollen, damit diese für das Speichern von dualkodierten Dezimalzahlen verwendet werden können. Der Opcode FX29 setzt I auf die Adresse des Sprites welcher das Zeichen repräsentiert, das für die in VX gespeicherte Zahl steht [6]. Bei den verfügbaren Zeichen handelt es sich um die hexadezimalen Ziffern von 0 bis F [6]. Jeder dieser Sprites hat eine Länge von 5 Bytes [6].

2.3.8 Benutzerdefinierte Sprites ausgeben

Der `draw_sprite`-Befehl gibt ein benutzerdefiniertes Byte als Sprite mit einer Höhe von einem Pixel aus. Auch hier muss der Sprite im Speicher des Interpreters sein, damit er ausgegeben werden kann. Der Autor hat daher eine weitere spezielle Variabel eingeführt, auf die der/die Nutzer:in keinen Zugriff hat. Darauf wird der Sprite jeweils gespeichert.
Folgendes Beispielprogramm gibt auf den Koordinaten (10|10) die Binärzahl 0b10100101 als Sprite aus:

```
draw_sprite(10, 10, 0b10100101)
```

Es ist zu beachten, dass der Sprite im little-endian Format angegeben werden muss.

2.3.9 Benutzerdefinierte Sprites bis zu einer Höhe von 10 Pixeln ausgeben

Der `draw_sprite_x10`-Befehl ist dem `draw_sprite`-Befehl sehr ähnlich, nimmt aber 10 Sprites als Argumente anstatt nur einen. Um weniger als 10 Sprites auszugeben, kann man die restlichen Sprites auf 0 setzen. Auch hier werden spezielle Variablen, auf die von dem/der Nutzer:in nicht zugegriffen werden kann, verwendet (vgl. Abschnitt 2.3.8). Die Sprites müssen im Speicher des Interpreters aufeinander folgend sein, damit sie alle mit einem einzigen DXYN-Opcode ausgegeben werden können, aber das ist ohnehin gewährleistet, da der Symbol-Table Manager Variablen in der Reihenfolge speichert wie sie zugewiesen werden (d. h. wenn die erste Variable, die zugewiesen wird, die Adresse n bekommt, bekommt die 10. Variable die Adresse n + 9).
Es ist zu beachten, dass die Sprites im little-endian Format angegeben werden müssen.

2.3.10 Schleifen

While-Schleifen

Die Grammatik der While-Schleifen in der erweiterten Backus-Naur-Form (EBNF):

```
whileloop: 'while' condition '{' start* '}'
```

Beispielprogramm, das mit einer While-Schleife eine Linie auf den Bildschirm zeichnet:

```
i = 0
while i <= 64 {
          draw_pixel(i, 16)
          i = i + 1
}
```

Genau wie Fallunterscheidungen haben While-Schleifen eine Bedingung. Es ist deshalb naheliegend, auch hier die unter Abschnitt 2.3.5 beschriebene `Condition`-Klasse zu verwenden um ein Objekt dafür zu generieren. Für die While-Schleife selbst wird mit der `Whileloop`-Klasse ein entsprechendes Objekt erstellt in dem auch das zugehörige `Condition`-Objekt gespeichert wird. Auch die `Whileloop`-Klasse hat nebst einem `Condition`-Objekt noch den bereits kompilierten Schleifeninhalt als Parameter.

For-Schleifen

Die Grammatik der For-Schleifen in der erweiterten Backus-Naur-Form (EBNF):

```
forloop: 'for' variable ',' expression '{' start* '}'
```

Beispielprogramm, das mit einer For-Schleife eine Linie auf den Bildschirm zeichnet:

```
for i, 64 {
          draw_pixel(i, 16)
{
```

Bei einer For-Schleife wird zu Beginn eine Zählvariabel festgelegt. Deren Name ist benutzerdefiniert, aber der Startwert ist immer 0. Bei jedem Durchlauf wird die Zählvariabel um 1 erhöht. Eine mögliche Erweiterung für Lizard wäre, den/die Benutzer:in den Startwert und die Schrittweite bestimmen zu lassen. Wenn die Zählvariabel den benutzerdefinierten Endwert erreicht hat, wird die Schleife abgebrochen. Dieser Wert wird hier mit dem Ausdruck nach dem Komma angegeben. Der Name der Zählvariabel kommt vor dem Komma. Wie bei einer While-Schleife generiert der Compiler bei einer For-Schleife ein entsprechendes Objekt, aber mit der `Forloop`-Klasse. Zu deren Parameter gehört ebenfalls der Schleifeninhalt, aber nicht in Form von CHIP-8 Instruktionen, sondern noch als Tree. Wenn die Funktion `translate` der `Forloop`-Klasse aufgerufen wird, ruft diese die `assign`-Funktion des `VariableHandler`-Objekts auf, um die Zählvariabel zuzuweisen. Anschliessend werden die Instruktionen generiert, um die Zählvariabel auf 0 zu setzen. Erst danach wird der Schleifeninhalt

übersetzt. Wenn dieser Abschnitt bereits zuvor kompiliert würde, gäbe es eine Fehlermeldung wenn man versuchen würde, in der Schleife auf die Zählvariabel zuzugreifen, da diese beim Kompilieren des Schleifeninhaltes noch nicht im Symbol-Table wäre und somit für nicht definiert gehalten würde. Dem Schleifeninhalt werden noch weitere CHIP-8 Instruktionen hinzugefügt, die jeweils die Zählvariabel um 1 erhöhen. Für die eigentliche Schleife wird dann die `Whileloop`-Klasse verwendet, denn eine For-Schleife lässt sich eigentlich auch mit einer While-Schleife umsetzen.

2.3.11 Eingabe

Zustand einer Taste abfragen

Es soll geprüft werden können, ob eine bestimmte Taste zu einem bestimmten Zeitpunkt gerade gedrückt wird. Es soll auch geprüft werden können, ob eine bestimmte Taste gerade nicht gedrückt wird. Der Autor hat entschieden, dies mithilfe von Fallunterscheidungen zu ermöglichen. Dazu wurde die Grammatik der Fallunterscheidungen folgendermassen erweitert:

```
comparison: 'if' condition '{' start* '}'
condition: expression '<' expression -> less
| expression '>' expression -> more
| expression '==' expression -> equal
| expression '<=' expression -> less_or_equal
| expression '>=' expression -> more_or_equal
| expression '!=' expression -> unequal
| key_status
| not_key_status
```

Neu hinzugekommen sind `key_status` und `not_key_status`. Die Grammatik dafür in der erweiterten Backus-Naur-Form (EBNF) sieht wie folgt aus:

```
key_status: 'is_key' '(' expression ')'
not_key_status: 'is_not_key' '(' expression ')'
```

Im folgenden Beispielprogramm wird ein Pixel auf den Koordinaten (42|0) gezeichnet, falls die Taste F gerade gedrückt wird (0xf entspricht 15 im Dezimalsystem):

```
if is_key(15) {
           draw_pixel(42, 0)
}
```

Im folgenden Beispielprogramm wird ein Pixel auf den Koordinaten (42|0) gezeichnet, falls die Taste E gerade nicht gedrückt wird (0xe entspricht 14 im Dezimalsystem):

```
if is_not_key(14) {
           draw_pixel(42, 0)
}
```

Der Ausdruck in den Klammern entspricht der Zahl die der Taste auf der hexadezimalen Tastatur

entspricht. Vom Compiler werden die `key_status`-Regel und die `not_key_status`-Regel auch wie jede andere `condition` behandelt, aber mit nur einem zugehörigen Ausdruck. Übersetzt wird die `key_status`-Regel mithilfe des Opcodes EX9E. Dieser Opcode überspringt die nächste Zeile, wenn die in VX gespeicherte Taste gedrückt gehalten wird [6]. Die `not_key_status`-Regel wird mithilfe von Opcode EXA1 übersetzt. Dieser Opcode überspringt die nächste Zeile, wenn die in VX gespeicherte Taste nicht gedrückt gehalten wird [6].

Eingabe abwarten

Es soll die Möglichkeit geben, nicht nur den Zustand einer bestimmten Taste zum jeweiligen Zeitpunkt abzufragen, sondern auch das Programm zu pausieren, bis irgendeine Taste gedrückt wird und danach zu speichern, welche Taste gedrückt wurde. Der Autor hat entschieden, dies mithilfe eines speziellen Ausdruckes zu ermöglichen. Die Grammatik für diesen Ausdruck sieht wie folgt aus:

```
await_key: 'get_key?'
```

Dieser Ausdruck wird mit der CHIP-8 Instruktion FE0A übersetzt. Opcode FX0A warted auf einen Input und speichert anschliessend die gedrückte Taste in Register VX [6].

2.3.12 Subroutinen

Mit der `SubroutineHandler`-Klasse wird zu beginn ein `SubroutineHandler`-Objekt erstellt. Dies kann als Teil des Symbol-Table Managers betrachtet werden (vgl. Aho, Sethi und Ullman [2, S. 11]). Wenn im Code eine Subroutine erkannt wird, wird die `newSub`-Funktion dieses `SubroutineHandler`-Objekts aufgerufen mit dem Namen der Subroutine und dem bereits kompilierten Subroutineninhalt als Argumente. Diese Funktion generiert daraufhin mit der `Subroutine`-Klasse ein neues Objekt für diese Subroutine. Darin wird der Name und der Subroutineninhalt gespeichert. Das Objekt selbst wird im `SubroutineHandler`-Objekt einer Liste mit allen anderen Subroutinen hinzugefügt. Zuvor wird aber geprüft, ob es in dieser Liste bereits eine Subroutine mit diesem Namen hat, denn es dürfen nicht mehrere Subroutinen in einem Programm denselben Namen besitzen. Ist dies dennoch der Fall, wird eine Fehlermeldung erzeugt.
Wird eine Subroutine aufgerufen, wird vorübergehend ein Platzhalter anstelle eines CHIP-8 Opcodes eingesetzt, ähnlich wie bei den Jumps (vgl. Abschnitt 2.3.5), da noch nicht bekannt ist, auf welcher Zeile die Subroutine sein wird.
Nachdem alles andere schon kompiliert wurde, wird die `appendSubroutines`-Funktion des `SubroutineHandler`-Objekts aufgerufen mit dem schon vorhandenen CHIP-8 Code (der noch Platzhalter für Jumps und Subroutinenaufrufe enthält) als Argument. Diese Funktion geht nun die Liste der Subroutinen durch und fügt sie der Reihe nach dem CHIP-8 Code hinten an. Beim Anfügen der Subroutinen wird in einer weiteren Liste gespeichert, welche Subroutine auf welcher Zeile platziert wurde. Beim Ersetzen der Platzhalter für Subroutinenaufrufe durch die entsprechenden CHIP-8 Opcodes wird auf diese Liste zurückgegriffen. Dazu hat das `SubroutineHandler`-Objekt die `getSubAdress`-Funktion, die den Namen als Argument nimmt, in der Liste die zugehörige Adresse sucht und diese zurück gibt. Wenn in der Liste keine Adresse für eine Subroutine mit diesem Namen zu finden ist, dann lässt sich daraus schliessen, dass der/die Nutzer:in versucht hat, eine nicht existierende Subroutine abzurufen. In diesem Fall erzeugt die `getSubAdress`-Funktion eine entsprechende Fehlermeldung.

Im folgenden Beispielprogramm wird eine Subroutine namens ”foo”, welche ein Pixel auf den Koordinaten (x|y) zeichnet, definiert und aufgerufen:

```
x = 0
y = 0

sub foo {
           draw_pixel(x, y)
}

x = 1
y = 1
callsub foo
```

2.3.13 Timer und Sound

Mit dem `set_timer`-Befehl kann der Wert des delay timers auf einen bestimmten Wert gesetzt werden. Im folgenden Beispielprogramm wird der Wert auf 10 gesetzt:

```
set_timer(10)
```

Dieser Befehl wird mithilfe des Opcodes FX15 übersetzt. Opcode FX15 setzt den Wert des delay timers auf den Wert von Register VX [6].

Für den sound timer gibt es den `set_soundtimer`-Befehl. Im folgenden Beispielprogramm wird der Wert des sound timer auf 5 gesetzt:

```
set_soundtimer(5)
```

Dieser Befehl wird mithilfe des Opcodes FX18 übersetzt. Opcode FX18 setzt den Wert des sound timers auf den Wert von Register VX [6].

Der Autor schlägt vor, den Wert des delay timers als speziellen Ausdruck abrufbar zu machen. Im folgenden Beispielprogramm wird die Variabel a auf den Wert des delay timers gesetzt:

```
a = get_delay?
```

Dieser Ausdruck wird mithilfe des Opcodes FX07 übersetzt. Opcode FX07 speichert den aktuellen Wert des delay timers in Register VX [6].

2.3.14 Kollisionsdetektion, Übertrag und negativer Übertrag

Für den Übertrag bzw. den negativen Übertrag wird bei einigen CHIP-8 Instruktionen das Register VF des CHIP-8 Interpreters verwendet. Deshalb führt der Autor einen weiteren speziellen Ausdruck ein, der dem Wert von VF entspricht. Dies ermöglicht dem/der Nutzer:in lesenden Zugriff auf dieses

Register. Im folgenden Beispielprogramm wird der Wert der Variabel a auf den Wert von VF gesetzt:

```
a = get_flag?
```

VF wird in CHIP-8 auch zur Kollisionsdetektion verwendet: Falls beim Anzeigen eines Sprites mit Opcode DXYN ein Pixel auf den Wert 1 gesetzt werden soll, das bereits den Wert 1 hat, bekommt es stattdessen den Wert 0 und VF wird auf 1 gesetzt [6].

2.3.15 Zufall

Der Autor hat noch einen weiteren speziellen Ausdruck implementiert, der einer Zufallszahl zwischen 0 und 255 entspricht. Im folgenden Beispielprogramm wird der Variabel a ein zufälliger Wert zwischen 0 und 255 zugewiesen:

```
a = get_random?
```

Übersetzt wird dieser Ausdruck in die CHIP-8 Instruktion CEFF (Opcode CXNN, vgl. [6]), welche den Wert von Register VE auf das Resultat einer bitweisen UND-Operation von einer Zufallszahl zwischen 0 und 255 und 0xFF, dezimal 255, setzt. Das Resultat einer bitweisen UND-Operation von 255 und einer beliebigen Zahl n zwischen 0 und 255 ergibt wiederum n. Folglich wäre die bitweise UND-Operation in diesem Fall eigentlich nicht nötig und verändert das Resultat in keiner Weise.

2.3.16 Kommentare

In Lizard gibt es Kommentare, die wie die Kommentare in C und C++ (vgl. [13] und [12]) sind. Dazu wurde der Grammatik folgendes hinzugefügt:

```
%import common.CPP_COMMENT
%import common.C_COMMENT
```

Damit werden bei Lark die entsprechenden Kommentare importiert [11].

Ein Beispiel für einen Kommentar in Lizard:

```
//Das ist ein Kommentar
```

2.4 Fazit

2.4.1 Welche Features von CHIP-8 werden in Lizard nicht abgebildet?

In Lizard können Benutzerdefinierte Sprites mit einer Höhe von maximal 10 Pixeln ausgegeben werden, in CHIP-8 mit einer Höhe von bis zu 15 Pixeln. Der Autor wollte aber keinen Befehl mit 15

Argumenten implementieren, da er einen derart langen Befehl für unpraktisch hält.

2.4.2 Typische Features von anderen Programmiersprachen

In Lizard sind sowohl Schleifen und Fallunterscheidungen als auch Subroutinen vorhanden. Alle Variablen sind global, Gültigkeitsbereiche gibt es also nicht. Daher gibt es auch keine Argumente und Rückgabewerte bei Subroutinen. In manchen Sprachen sind Gültigkeitsbereiche sehr wichtig wenn beispielsweise ein grösseres Projekt durchgeführt wird an dem mehrere Personen arbeiten und man vermeiden will, dass Probleme entstehen weil Beispielsweise zwei Personen denselben Namen für jeweils unterschiedliche Variablen verwenden. Allerdings ist Lizard für derartige Projekte ohnehin nicht geeignet. Vergleiche werden in Lizard nicht als Ausdrücke die entweder `True` oder `False` zurückgeben behandelt. So würde Beispielsweise der Code `a = 1 < 2` einen Fehler beim Parsen verursachen, weil so etwas in der Grammatik nicht vorgesehen ist.
Nach einem if-statement können in Lizard keine else-if-statements oder else-statements hinzugefügt werden. Dieses Feature wäre auch eine mögliche erweiterung von Lizard.

Typen

Es gibt in Lizard keine Typen abgesehen von Integer. Auch Listen gibt es keine. Der Autor würde es aber für möglich halten, dies noch zu Implementieren, wenn mehr Zeit zur Verfügung stehen würde. Auch Zahlen mit Vorzeichen hätten wahrscheinlich mit mehr Zeit implementiert werden können. Der Autor hielt das Implementieren von Zeichenketten nicht für sinnvoll. Zwar könnte man einen ASCII-Charakter in einem Byte speichern [8], aber eine Zeichenkette auch auszugeben wäre recht aufwendig, da das Fontset eines CHIP-8 Interpreters nur hexadezimale Ziffern, also dezimale Ziffern von 1 bis 9 und Buchstaben von A bis Z, enthält und das Hinzufügen von weiteren 79 druckbaren ASCII-Zeichen (Es gibt 95 druckbare ASCII-Zeichen [8] und 16 sind bereits im Fontset des Interpreters vorhanden) als Sprites mit einer Grösse von 5x4 Pixel im Speicher des Interpreters 395 Bytes alleine nur für die Sprites brauchen würde. Hinzu kämen noch weitere Schwierigkeiten, die hier nicht alle erläutert werden.

2.4.3 Das Programmieren von CHIP-8 Spielen mit Lizard

Lizard ist zweifellos für Menschen besser lesbar als CHIP-8, da CHIP-8 aus hexadezimalen Opcodes besteht. Allerdings übersetzt der Compiler den Quellcode nicht immer auf die effizienteste Art. Wird beispielsweise eine Variabel geändert und gleich wieder abgerufen, wird der Wert zuerst nach V0 kopiert, gespeichert und dann vom Speicher wieder in V0 geladen, was unnötig wäre, da der Wert ohnehin noch in V0 ist. Daher wäre ein Programm, das direkt in CHIP-8 geschrieben worden wäre, wahrscheinlich effizienter. Der Autor hat selbst eine Implementation von Pong in Lizard geschrieben.

Abbildungsverzeichnis

1.1 Dave Ruske from flickr, CC BY 2.0 <https://creativecommons.org/licenses/by/2.0>, via Wikimedia Commons 5
1.2 Eigene Darstellung 6
1.3 Eigene Darstellung 6

2.1 Eigene Darstellung 10
2.2 Eigene Darstellung 10
2.3 Eigene Darstellung 11
2.4 Eigene Darstellung 11
2.5 Eigene Darstellung 13

Tabellenverzeichnis

2.1 Eigene Darstellung 9
2.2 Eigene Darstellung 12

Literatur

[1] chrissikraus / Stephan Augsten. *Was ist ein Parser?* URL: https://www.dev-insider.de/was-ist-ein-parser-a-756662/ (besucht am 03. 04. 2021).

[2] Alfred V. Aho, Ravi Sethi und Jeffrey D. Ullman. *Compilers - Principles, Techniques and Tools*. Mark S. Dalton, 1986.

[3] Albatross. *A Brief Description*. URL: http://cplusplus.com/info/description/ (besucht am 27. 09. 2021).

[4] Mathe by Daniel Jung. *Binärsystem, Dualsystem, Zweiersystem, Multiplikation | Mathe by Daniel Jung*. YouTube. URL: https://www.youtube.com/watch?v=oFseqcdcZFU (besucht am 19. 09. 2021).

[5] erezsh. *Lark - a parsing toolkit for Python*. URL: https://github.com/lark-parser/lark (besucht am 03. 04. 2021).

[6] Thomas P. Greene. *Cowgod's Chip-8 Technical Reference*. URL: http://devernay.free.fr/hacks/chip8/C8TECH10.HTM#2.2 (besucht am 02. 04. 2021).

[7] W.A. Hemmerich. *Division durch 0*. URL: https://matheguru.com/allgemein/division-durch-0.html (besucht am 19. 09. 2021).

[8] Torsten Horn. *Text-Encoding, Codepage, Charset*. URL: https://www.torsten-horn.de/techdocs/encoding.htm (besucht am 02. 04. 2021).

[9] Tobias V. Langhoff. *Guide to making a CHIP-8 emulator*. URL: https://tobiasvl.github.io/blog/write-a-chip-8-emulator/ (besucht am 15. 10. 2021).

[10] Laurence Muller. *How to write an emulator (CHIP-8 interpreter)*. URL: https://multigesture.net/articles/how-to-write-an-emulator-chip-8-interpreter/ (besucht am 19. 07. 2021).

[11] Réne Nyffenegger. *Python library: Lark*. URL: https://renenyffenegger.ch/notes/development/languages/Python/libraries/Lark/index (besucht am 20. 10. 2021).

[12] Stefan Ram. *Kommentare in C++*. URL: http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/c++_formatierung_de (besucht am 20. 10. 2021).

[13] Sebastian Rothe. *Kommentierung in C*. URL: https://hps.vi4io.org/_media/teaching/wintersemester_2012_2013/epc-1213-rothe-kommentare-ausarbeitung.pdf (besucht am 20. 10. 2021).

[14] Erez Shinan. *Transformers & Visitors*. URL: https://lark-parser.readthedocs.io/en/latest/visitors.html#interpreter (besucht am 09. 08. 2021).

[15] *The Python Language Reference*. Python Software Foundation. URL: https://docs.python.org/3/reference/expressions.html#operator-precedence (besucht am 04. 10. 2021).

[16] Gabriele Tomassetti. *Parsing In Python: Tools And Libraries*. URL: https://tomassetti.me/parsing-in-python/ (besucht am 03. 04. 2021).

[17] Joseph Weisbecker. „An Easy Programming System“. In: *Byte* 3.12 (1978), S. 108–122.

Danksagung

Bei meiner Maturaarbeit wurde ich von vielen Personen unterstützt. Daher möchte ich mich im folgenden bei diesen Personen bedanken:

Ondrej Bačo, der mir seinen selbst programmierten CHIP-8 Interpreter zu Testzwecken zur Verfügung stellte und mir nützliche Feedbacks zu frühen Entwürfen von Teilen dieser Arbeit gab.

Stefan Rothe, der diese Arbeit betreut hat und stets gute Tipps gegeben hat.

Alle Kolleg:innen die mir geholfen haben, meine Liste mit Ideen für möglichen Namen für diese Programmiersprache etwas einzugrenzen.

Meine Eltern, die meine Rechtschreibung kontrolliert haben.

Eigenständigkeitserklärung

Hiermit erkläre ich meine Maturaarbeit selbstständig verfasst zu haben. Alle verwendeten Quellen wurden angegeben.

Simon L. Hauswirth