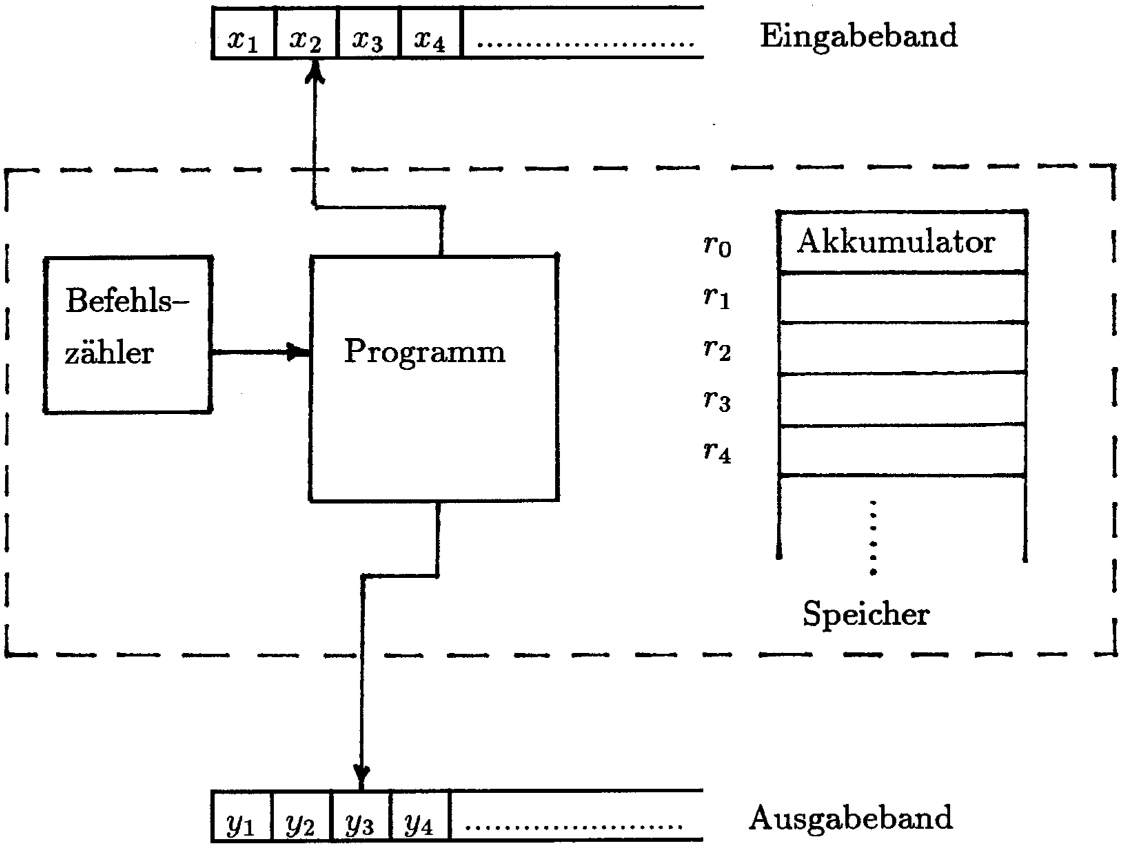
1.3. RAM und RASP

Definition 1.3.1: Eine *RAM (random access machine)* besteht aus einem *Eingabeband*, einem *Ausgabeband*, einem *Befehlszähler*, einem *Programm* und einem (unendlichen) *Speicher*.



Das Eingabeband ist eine unendliche Folge von *Feldern* und das Ausgabeband ist eine unendliche Folge von Feldern, von denen jedes eine ganze Zahl beinhalten kann. Der Speicher ist eine unendliche Folge von *Registern*, deren jedes eine ganze Zahl aufnehmen kann. Das Programm ist eine endliche Folge von (eventuell markierten) *Instruktionen.* Jede Instruktion besteht aus zwei Teilen, einem *Operationskode* und einer *Adresse* (die entweder ein *Operand* oder eine *Marke* ist).

Liste der RAM–Instruktionen:

Operationskode	Adresse	Operationskode	Adresse
1. `LOAD`	Operand	7. `READ`	Operand
2. `STORE`	Operand	8. `WRITE`	Operand
3. `ADD`	Operand	9. `JUMP`	Marke
4. `SUB`	Operand	10. `JGTZ`	Marke
5. `MULT`	Operand	11. `JZERO`	Marke
6. `DIV`	Operand	12. `HALT`	

Ein Operand kann wie folgt beschaffen sein:

1. `=i`, die Bedeutung ist die ganze Zahl i selbst,
2. `i`, wobei i eine nichtnegative ganze Zahl ist. Die Bedeutung ist der Inhalt des Registers i,
3. `*i`, die Bedeutung ist der Inhalt von Register j, wobei j die ganze Zahl in Register i ist (indirekte Adressierung). Ist $j < 0$, so bleibt die Maschine stehen.

Um die *Bedeutung* einer RAM R zu definieren, ziehen wir die *Speicherfunktion* $c : \mathbb{N}_0 \to \mathbb{Z}$ heran. $c(i)$ ist der Inhalt des Speicherregisters r_i.

Am Anfang der *Berechnung* ist $c(i) = 0$ für alle $i \geq 0$, der Befehlszähler ist auf 1 gesetzt, und jedes Feld des Ausgabebandes enthält Blank. Bis auf endlich viele Ausnahmen sind alle Felder des Eingabebandes leer. Der Lesekopf zeigt auf das erste Feld des Eingabebandes und der Schreibkopf zeigt auf das erste Feld des Ausgabebandes. Das Feld, auf welches der Lese– bzw. Schreibkopf gerade zeigt, heißt das *aktuelle* Eingabe– bzw. Ausgabefeld. In jedem Schritt der Berechnung wird diejenige Instruktion ausgeführt, auf die der Befehlszähler gerade zeigt, also die k–te Instruktion, wenn k der Inhalt des Befehlszählers ist. Nach Ausführung der k–ten Instruktion von R wird der Befehlszähler automatisch auf $k + 1$ gesetzt, wenn die k–te Instruktion nicht eine der Anweisungen `JUMP`, `HALT`, `JGTZ` oder `JZERO` war. Die Berechnung endet, wenn die Instruktion `HALT` abgearbeitet wird, oder zu einer Marke gesprungen wird, die im Programm nicht vorkommt.

Um die Bedeutung einer Instruktion anzugeben, definieren wir $v(a)$, den *Wert* des Operanden a, wie folgt:

$$v(=\mathtt{i}) = i, \quad v(\mathtt{i}) = c(i), \quad v(*\mathtt{i}) = c(c(i)).$$

Die folgende Tabelle gibt die Bedeutung der einzelnen Instruktionen an. Instruktionen, deren Bedeutung in der Tabelle nicht definiert ist, wie etwa `STORE =i`, haben dieselbe Bedeutung wie `HALT`. Ebenso beendet eine Division durch 0 die Berechnung.

Instruktion	Bedeutung
1. `LOAD a`	$c(0) \leftarrow v(a)$
2. `STORE i`	$c(i) \leftarrow c(0)$
`STORE *i`	$c(c(i)) \leftarrow c(0)$
3. `ADD a`	$c(0) \leftarrow c(0) + v(a)$
4. `SUB a`	$c(0) \leftarrow c(0) - v(a)$
5. `MULT a`	$c(0) \leftarrow c(0) \cdot v(a)$
6. `DIV a`	$c(0) \leftarrow \lfloor c(0)/v(a) \rfloor$
7. `READ i`	$c(i) \leftarrow$ Symbol im aktuellen Eingabefeld. Der Lesekopf wird um eine Position nach rechts gerückt.
`READ *i`	$c(c(i)) \leftarrow$ Symbol im aktuellen Eingabefeld. Der Lesekopf wird um eine Position nach rechts gerückt.

8. `WRITE a` $v(a)$ wird in das aktuelle Ausgabefeld geschrieben. Der Schreibkopf wird um eine Position nach rechts gerückt.
9. `JUMP b` Der Befehlszähler wird auf die Nummer der Instruktion mit der Marke b gesetzt.
10. `JGTZ b` Falls $c(0) > 0$, dann wird der Befehlszähler auf die Nummer der Instruktion mit der Marke b gesetzt. Andernfalls wird der Befehlszähler um 1 erhöht.
11. `JZERO b` Falls $c(0) = 0$, dann wird der Befehlszähler auf die Nummer der Instruktion mit der Marke b besetzt. Andernfalls wird der Befehlszähler um 1 erhöht.
12. `HALT` Die Berechnung endet.

Die *Bedeutung* der RAM R ist die durch R definierte Abbildung von Eingabebändern zu Ausgabebändern. Da die Berechnung von R nicht auf allen Eingabebändern terminieren muß, ist diese Abbildung nur eine partielle Abbildung. Diese Abbildung kann nun in verschiedener Weise interpretiert werden, etwa als Funktion oder als Sprache.

Die RAM R lese immer genau n Zahlen $x_1, \ldots, x_n$ vom Eingabeband, und schreibe genau eine Zahl y auf das Ausgabeband oder terminiere nicht. Damit ist implizit die (partielle) Funktion

$$f(x_1, \ldots, x_n) = \begin{cases} y & \text{falls } R \text{ auf } x_1, \ldots, x_n \text{ terminiert und } y \text{ schreibt} \\ \perp & \text{falls } R \text{ auf } x_1, \ldots, x_n \text{ nicht terminiert} \end{cases}$$

erklärt. Wir sagen R *berechnet* die Funktion f.

Man kann R auch auffassen als Akzeptor einer Sprache *) über einem Alphabet $\{\alpha_1, \ldots, \alpha_k\}$. Das Symbol α_i kann kodiert werden als die ganze Zahl i. Ein Wort wird so auf das Eingabeband geschrieben, daß die Kodes für die einzelnen Symbole nacheinander in die Eingabefelder geschrieben werden und als Abschluß in das nächste Feld eine 0 geschrieben wird. Man sagt nun das Wort s wird von R *akzeptiert*, wenn R das Wort s und das Markierungssymbol 0 vom Eingabeband liest, eine 1 in das erste Feld des Ausgabebandes schreibt, und hält. Die *von R akzeptierte Sprache* ist die Menge aller Wörter, die von R akzeptiert werden. ■

Beispiel 1.3.1: Wir betrachten die Funktion $f(n)$, gegeben als

$$f(n) = \begin{cases} n^n & \text{falls } n \text{ positiv ist} \\ 0 & \text{sonst.} \end{cases}$$

---

*) Vgl. Sektion 1.2. Ein Alphabet Σ ist eine endliche Menge von Symbolen. Ein Wort w über Σ ist eine endliche Folge von Symbolen aus Σ. Eine Sprache L über Σ ist eine Teilmenge von Σ^*, der Menge aller Wörter über Σ.

Ein Programm zur Berechnung von $f(n)$ (in einer PASCAL–ähnlichen Sprache) könnte wie folgt aussehen:

begin
  read $r1$;
  if $r1 \leq 0$ **then write** 0
  else
    begin
      $r2 \leftarrow r1$;
      $r3 \leftarrow r1 - 1$
      while $r3 > 0$ **do**
        begin
          $r2 \leftarrow r2 * r1$;
          $r3 \leftarrow r3 - 1$
        end;
      write $r2$
    end
end

Ein entsprechendes RAM Programm ist

RAM Programm:			entsprechende PASCAL Statements:
	`READ`	`1`	**read** $r1$
	`LOAD`	`1`	
	`JGTZ`	`pos`	**if** $r1 \leq 0$ **then write** 0
	`WRITE`	`=0`	
	`JUMP`	`endif`	
`pos:`	`LOAD`	`1`	$r2 \leftarrow r1$
	`STORE`	`2`	
	`LOAD`	`1`	
	`SUB`	`=1`	$r3 \leftarrow r1 - 1$
	`STORE`	`3`	
`while:`	`LOAD`	`3`	
	`JGTZ`	`continue`	**while** $r3 > 0$ **do**
	`JUMP`	`endwhile`	

```
continue:   LOAD    2
            MULT    1           r2 ← r2 * r1
            STORE   2
            LOAD    3
            SUB     =1          r3 ← r3 − 1
            STORE   3
            JUMP    while
endwhile:   WRITE   2           write r2
endif:      HALT
```

■

Beispiel 1.3.2: Wir erzeugen ein RAM Programm, welches die Sprache über dem Alphabet $\{1, 2\}$ akzeptiert, die aus allen Wörtern mit gleich vielen 1en wie 2en besteht. Das Programm liest jedes Eingabesymbol in Register 1 ein. In Register 2 wird jeweils die Differenz der Anzahl der bereits gelesenen 1en und 2en gespeichert. Sobald das Programm zum Endsymbol 0 kommt, prüft es, ob die Differenz 0 ist. Wenn ja, so wird 1 ausgedruckt und das Programm hält. Wir nehmen an, daß 0, 1 und 2 die einzigen möglichen Eingabesymbole sind.

Ein Pidgin–PASCAL Programm könnte wie folgt aussehen:

begin
  $d \leftarrow 0$;
  read x;
  while $x \neq 0$ **do**
    begin
      if $x \neq 1$ **then** $d \leftarrow d - 1$ **else** $d \leftarrow d + 1$;
      read x
    end;
  if $d = 0$ **then write** 1
end

Ein entsprechendes RAM Progamm ist

```
            LOAD    =0          d ← 0
            STORE   2
            READ    1           read x
while:      LOAD    1           while x ≠ 0 do
            JZERO   endwhile
            LOAD    1
            SUB     =1          if x ≠ 1
            JZERO   one
            LOAD    2
            SUB     =1          then d ← d − 1
            STORE   2
            JUMP    endif
one:        LOAD    2
            ADD     =1          else d ← d + 1
```

```
              STORE   2
endif:        READ    1              read x
              JUMP    while
endwhile:     LOAD    2
              JZERO   output         if d = 0 then write 1
              HALT
output:       WRITE   =1
              HALT
```

■

Das Programm einer RAM ist nicht im Speicher abgelegt, es kann sich also nicht selbst modifizieren. Wir wollen nun ein Maschinenmodell betrachten, das ganz gleich ist wie eine RAM, dessen Programm aber im Speicher abgelegt ist.

Definition 1.3.2: Die Instruktionen einer *RASP (random access stored program* machine) sind identisch mit den Instruktionen einer RAM, indirekte Adressierung ist jedoch nicht gestattet (eine RASP kann indirekte Adressierung simulieren durch Modifikation der Instruktionen während der Ausführung des Programms). Die Struktur einer RASP ist auch gleich wie die einer RAM, nur daß das Programm im Speicher abgelegt ist. Jede RASP Instruktion belegt zwei aufeinanderfolgende Register. Im ersten Register steht eine Kodierung des Operationskodes, im zweiten Register eine Adresse. Ist die Adresse von der Form `=i`, dann kodiert der Inhalt des ersten Registers auch den Umstand, daß der Operand ein Literal ist, und das zweite Register enthält i. Die Instruktionen werden als Zahlen kodiert, etwa auf folgende Weise:

Instruktion:	Kode:	Instruktion:	Kode:
`LOAD i`	1	`DIV i`	10
`LOAD =i`	2	`DIV =i`	11
`STORE i`	3	`READ i`	12
`ADD i`	4	`WRITE i`	13
`ADD =i`	5	`WRITE =i`	14
`SUB i`	6	`JUMP i`	15
`SUB =i`	7	`JGTZ i`	16
`MULT i`	8	`JZERO i`	17
`MULT =i`	9	`HALT`	18

So würde etwa die Instruktion `LOAD =32` abgespeichert als 2 in einem Register und 32 im nächsten Register.

Die Abarbeitung eines RASP Programms erfolgt im wesentlichen gleich wie die Abarbeitung eine RAM Programms, mit den folgenden geringfügigen Änderungen: zu Beginn der Berechnung ist das Programm in zusammenhängenden Speicherregistern abgelegt, und der Befehlszähler ist auf die erste Instruktion gesetzt. Die übrigen Register enthalten alle 0. Nach der Ausführung jeder Instruktion wird der Befehlszähler um 2 erhöht, außer nach einer Instruktion der Form `JUMP i`, `JGTZ i` (wenn der Inhalt des Akkumulators positiv ist) oder `JZERO i` (wenn der Inhalt des Akkumulators 0 ist). In diesen Fällen wird der Befehlszähler zu i gesetzt. ■

Jede Funktion (Sprache), welche von einem RAM Programm berechnet (akzeptiert) werden kann, kann auch von einem RASP Programm berechnet (akzeptiert) werden und umgekehrt; das wird im folgenden Satz ausgesprochen.

Satz 1.3.1: *Zu jedem RAM Programm gibt es ein äquivalentes RASP Programm und umgekehrt.*

Beweis: (a) Zunächst zeigen wir, wie ein RAM Programm P durch ein RASP Programm P' simuliert werden kann. Register 1 der RASP wird zur Zwischenspeicherung des RAM Akkumulators benutzt. Das Programm P' belegt die nächsten $r-1$ Register der RASP. Die Konstante r hängt ab vom RAM Programm P. Der Inhalt des RAM Registers i, $i \geq 1$, wird im RASP Register $r+i$ abgespeichert, sodaß alle Speicherzugriffe in P' um r gegenüber P verschoben sind.
Jede RAM Instruktion, die keine indirekte Adressierung benutzt, wird in die identische RASP Instruktion (mit entsprechend verschobenen Speicherzugriffen) übersetzt. Jede RAM Instruktion mit indirekter Adressierung wird übersetzt in eine Folge von sechs RASP Instruktionen; dabei wird die indirekte Adressierung durch Instruktionsmodifikation simuliert. Als Beispiel dafür betrachten wir die RAM Instruktion `SUB *i`, $i \geq 1$. Diese RAM Instruktion wird in eine Folge von RASP Instruktionen übersetzt, die

1. den Inhalt des Akkumulators in Register 1 zwischenspeichert,
2. den Inhalt von Register $r+i$ in den Akkumulator lädt (Register $r+i$ der RASP entspricht dem Register i der RAM),
3. r zum Akkumulator addiert,
4. den Inhalt des Akkumulators in das Adressenfeld einer `SUB` Instruktion abspeichert,
5. den Inhalt von Register 1 wieder in den Akkumulator zurücklädt, und schließlich
6. die in Schritt 4 erzeugte `SUB` Instruktion benutzt, um die Subtraktion auszuführen. So wird, unter der Annahme, daß die Folge der RASP Instruktionen in Register 100 beginnt, die RAM Instruktion `SUB *i` wie folgt simuliert:

Register	Inhalt	Bedeutung	
100	3	`STORE 1`	
101	1		
102	1	`LOAD` $r+i$	
103	$r+i$		
104	5	`ADD =`r	
105	r		
106	3	`STORE 111`	
107	111		
108	1	`LOAD 1`	
109	1		
110	6	`SUB` b`,`	wobei b der Inhalt des RAM
111	–		Registers i ist.

Enthält P n Instruktionen mit m indirekten Adressierungen, so enthält P' $(n-m)+6m$ Instruktionen. Der Verschiebungsfaktor r ist also $2n+10m+1$.

(b) Zu einem gegebenen RASP Programm P' konstruieren wir ein äquivalentes RAM Programm P. Einige RAM Register werden für spezielle Zwecke benutzt:

Register 1:	für indirekte Adressierung,
Register 2:	der Befehlszähler der RASP,
Register 3:	der Akkumulator der RASP.

Register i, $i \geq 1$, der RASP wird in Register $i+3$ der RAM gespeichert.

Die RAM beginnt mit dem RASP Programm im Speicher ab Register 4. Register 2, der RASP Befehlszähler, enthält 4. Die Register 1 und 3 enthalten 0. Das RAM Programm ist eine Simulationsschleife, welche damit beginnt eine RASP Instruktion zu lesen (mit einer `LOAD *2` Instruktion), diese dekodiert und zu einer von 18 Instruktionsfolgen verzweigt, deren jede einen Typ von RASP Instruktionen simuliert. Als Beispiel betrachten wir die RAM Instruktionsfolge zur Simulation der RASP Instruktion 6, also `SUB i`:

`LOAD 2`	erhöhe den Befehlszähler um 1, sodaß er nun auf das Register
`ADD =1`	zeigt, welches den Operanden i der `SUB i` Instruktion enthält;
`STORE 2`	
`LOAD *2`	lade i in den Akkumulator, addiere 3 und speichere
`ADD =3`	das Ergebnis in Register 1;
`STORE 1`	
`LOAD 3`	lade den Inhalt des RASP Akkumulators aus Register 3, sub–
`SUB *1`	trahiere den Inhalt des Registers $i+3$ und lege das Ergebnis
`STORE 3`	in Register 3 ab;
`LOAD 2`	erhöhe den Befehlszähler um 1, sodaß er nun auf die nächste
`ADD =1`	RASP Instruktion zeigt;
`STORE 2`	
`JUMP a`	verzweige zum Anfang der Simulationsschleife (hier
	mit “`a`” bezeichnet).

Zu diesem Programmstück wird verzweigt, wenn $c(c(2)) = 6$, der Befehlszähler also auf ein Register mit Inhalt 6, dem Kode von `SUB`, zeigt. ■

Beispiel 1.3.3: Es ist nicht schwer zu zeigen, daß die Sprachen aus den Beispielen 1.2.5 und 1.2.6 von einer RAM akzeptiert werden können. ■

Wie man auch unschwer sieht, kann jeder endliche Automat durch eine RAM simuliert werden. Wegen Beispiel 1.3.3 wissen wir also, daß das Konzept der RAM stärker ist als dasjenige endlicher Automaten, da mehr Sprachen akzeptiert werden können.