Kapitel 14

Virtuelle Maschinen und Übersetzer

Wenn Sie mit einem Computer arbeiten, arbeiten Sie mit einem System, das aus Hardware und Software besteht. Die Hardware des Computers sind die Teile, die Sie sehen und anfassen können. Die Software besteht aus Programmen und Daten, die auf Speichermedien wie Festplatten und CDs abgelegt sind. Betriebssysteme, Programmiersysteme, Web-Browser und Editoren sind durch Software realisierte Computer-Werkzeuge.

Die Schnittstelle zwischen Hardware und Software wird durch den sogenannten *Prozessor* realisiert. Dabei handelt es sich um einen durch Hardware realisierten Interpreter für eine sehr primitive Sprache, die man als *Maschinensprache* oder *Befehlssatz* bezeichnet.

Maschinensprachen sind nicht für die Benutzung durch Programmierer konzipiert. Maschinenprogramme werden mithilfe von *Übersetzern* aus Programmen gewonnen, die in *Programmiersprachen* geschrieben sind. Die Übersetzer für Programmiersprachen werden durch Maschinenprogramme realisiert.

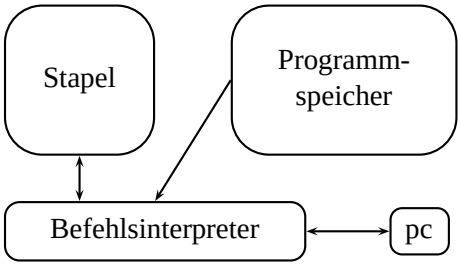
Eine Machinensprache sollte einerseits so entworfen sein, dass man dafür mit vernünftigem Aufwand Prozessoren bauen kann, die Maschinenprogramme schnell ausführen. Andererseits muss eine Maschinensprache so konzipiert sein, dass man in den gebräuchlichen Programmiersprachen Programme schreiben kann, die automatisch in schnelle Maschinenprogramme übersetzt werden können. In den letzten fünfzehn Jahren haben sich die in der Praxis vorkommenden Maschinensprachen kaum geändert. Diese Stabilität war eine Voraussetzung für die enorme Steigerung der Prozessorgeschwindigkeit, die in diesem Zeitraum erzielt wurde.

Eine *virtuelle Maschine* ist ein virtueller Computer, der mithilfe von Software auf einem richtigen Computer realisiert wird. In diesem Kapitel entwickeln wir eine virtuelle Maschine V, die wir mit Standard ML implementieren. Die Maschinensprache von V, die wir der Einfachheit halber auch V nennen, beinhaltet

wesentliche Konzepte von realen Maschinensprachen. Für zwei einfache Programmiersprachen W und S werden wir Übersetzer nach V entwickeln. Damit lernen wir einige grundlegende Techniken für die Implementierung von Programmiersystemen kennen.

14.1 Eine virtuelle Maschine

Wir definieren jetzt eine virtuelle Maschine V0 und realisieren sie in Standard ML. V0 hat vier Komponenten:



Der *Befehlsinterpreter* kann die in Abbildung 14.1 dargestellten *Befehle* ausführen. Bei den anderen drei Komponenten von V0 handelt es sich um *Speicher*, deren Inhalte vom Befehlsinterpreter gelesen werden können. Der *Programmspeicher* ist eine Reihung vom Typ

```
instruction array
```

und enthält die Befehle des auszuführenden Programms. Der *Programmzähler* (pc) ist eine Zelle

```
val pc : int ref
```

die die *Adresse* (den Index) des nächsten auszuführenden Befehls im Programmspeicher enthält. Der Wert des Programmzählers kann vom Befehlsinterpreter gelesen und geändert werden. Der *Stapel* ist eine Reihung vom Typ

```
int array
```

die wie ein Stapel verwaltet wird und in der der Befehlsinterpreter Zwischenergebnisse abgelegt.

Die Maschine wird mit einer Liste von Befehlen gestartet, die man als *Programm* bezeichnet. Zunächst werden die folgenden vorbereitenden Schritte durchgeführt:

- Die Befehle des Programms werden in die Zellen des Programmspeichers geschrieben, wobei der erste Befehl in die Zelle mit der Adresse 0 kommt.

```
type index = int
type noi   = int                (* number of instructions *)

datatype instruction =
    con     of int
  | add                         (* addition               *)
  | sub                         (* substraction           *)
  | mul                         (* multiplication         *)
  | leq                         (* less or equal test     *)
  | branch  of noi              (* unconditional branch   *)
  | cbranch of noi              (* conditional branch     *)
  | getS    of index            (* push value from stack  *)
  | putS    of index            (* update value in stack  *)
  | halt                        (* halt machine           *)

type code = instruction list
```

Abbildung 14.1: Befehlssatz von V0

- Der Programmzähler wird auf 0 gesetzt, also auf die Adresse des ersten Befehls des Programms.
- Der Stapel wird in den Zustand leer gebracht.

Danach beginnt die Maschine mit der Ausführung des Befehls, dessen Adresse im Programmzähler steht. Ein Befehl kann Zahlen vom Stapel nehmen und Zahlen auf den Stapel legen. Er kann auch den Wert des Programmzählers ändern. Nachdem ein Befehl ausgeführt ist, fährt die Maschine mit der Ausführung des Befehls fort, dessen Adresse dann im Programmzähler steht. Davon ausgenommen ist der Befehl `halt`, der die Maschine anhält. Sie liefert dann den obersten Wert des Stapels als Ergebnis der Programmausführung.

Arithmetische Befehle

Der Befehl

`con` *n*

legt die Zahl *n* auf den Stapel. Der Befehl

`add`

nimmt zwei Zahlen vom Stapel, addiert diese, und legt das Ergebnis auf den Stapel. Nach der Ausführung des Befehls `add` hat der Stapel also ein Element weniger als vorher. Beide Befehle erhöhen den Programmzähler jeweils um eins. Damit liefert das Programm

```
[con 4, con 7, add, halt]
```

die Zahl $4 + 7 = 11$.

Mit den Befehlen `sub` und `mul` kann man in analoger Weise subtrahieren und multiplizieren. Bei der Subtraktion ist die Reihenfolge der Argumente von Bedeutung. Die Argumente der Befehle werden immer in umgekehrter Reihenfolge auf den Stapel gelegt: das letzte Argument wird zuerst und das erste Argument zuletzt auf den Stapel gelegt. Damit bekommt ein Befehl seine Argumente in der richtigen Reihefolge, wenn er sie nacheinder vom Stapel nimmt. Beispielsweise liefert das Program

```
[con 4, con 7, sub, halt]
```

die Zahl $7 - 4 = 3$.

Den Wert des Ausdrucks

```
4*3-7*2
```

können wir mit dem Programm

```
[con 2, con 7, mul,        (*  7*2  *)
 con 3, con 4, mul,        (*  4*3  *)
 sub,                      (*   -   *)
 halt]
```

berechnen.

Offensichtlich können wir beliebig geschachtelte Ausdrücke in Befehlslisten übersetzen, die sie auswerten. Dabei wird eine baumartige Datenstruktur in eine lineare Struktur übersetzt, die mithilfe eines Stapels interpretiert wird. Dahinter steckt ein allgemeines Prinzip: Baumrekursion kann mithilfe eines Stapels auf Iteration zurückgeführt werden.

Konditionale

Für die Berechnung von konditionalen Ausdrücken

if e_1 then e_2 else e_3

hat V0 zwei sogenannte *Sprungbefehle*. Der *bedingte Sprungbefehl*

`cbranch` *i*

nimmt die oberste Zahl vom Stapel. Falls diese 0 ist, wird *i* zum Programmzähler hinzuaddiert:

```
pc:= !pc+i
```

Ansonsten wird der Programmzähler wie üblich um eins erhöht:

```
pc:= !pc+1
```

Der *unbedingte Sprungbefehl*

```
branch i
```

lässt den Stapel unverändert und addiert in jedem Fall *i* zum Programmzähler hinzu:

```
pc:= !pc+i
```

Mit den Sprungbefehlen hat man also die Möglichkeit, im Program vor (positives *i*) oder zurück (negatives *i*) zu springen.

Im Zusammenhang mit dem bedingten Sprungbefehl ist der Befehl

```
leq
```

hilfreich. Er holt sich zwei Zahlen vom Stapel und vergleicht diese. Wenn das erste Argument kleiner als das zweite ist, wird die Zahl 1 auf den Stapel gelegt, ansonsten die Zahl 0.

Als Beispiel betrachten wir den Ausdruck

```
if 1<=2 then 4-3 else 7*5
```

Diesen können wir mit dem folgenden Programm auswerten:

```
[con 2, con 1, leq, cbranch 5,    (* if 1<=2   *)
 con 3, con 4, sub, branch 4,     (* then 4-3  *)
 con 5, con 7, mul,               (* else 7*5  *)
 halt]
```

Zuerst kommen die Befehle für die Bedingung. Danach kommt ein bedingter Sprung, der die Konsequenz überspringt, falls die Bedingung zu 0 evaluiert. Danach kommen die Befehle für die Konsequenz und ein Sprung über die Alternative. Schließlich folgen die Befehle für die Alternative.

Imperative Variablen

Mit den Befehlen `getS` und `putS` kann man sogenannte *imperative Variablen* realisieren. Dabei handelt es sich um Zellen, die im Stapel alloziert sind. Der Befehl

```
getS i
```

legt den Wert auf den Stapel, der dort bereits an der Adresse *i* steht. Der Befehl

```
putS i
```

nimmt das oberste Element vom Stapel und überschreibt damit den Wert, der an der Adresse *i* des Stapels steht. Dabei hat die unterste Zelle des Stapels die Adresse 0. Das Programm

```
[con 5, con 7, getS 0, getS 1, add, putS 1, mul,halt]
```

liefert also $5 \cdot (7 + 5) = 60$.

Schleifen

Mit Sprüngen und imperativen Variablen können wir Schleifen programmieren. Als Beispiel übersetzen wir den Ausdruck

```
let
    val n = ref 12
    val a = ref 1
in
    while 2 <= !n do
        ( a := !a * !n ;
          n := !n - 1 ) ;
    !a
end
```

der 12! berechnet, in ein V0-Programm:

```
[con 12,                          (* val n = ref 12  [0] *)
 con 1,                           (* val a = ref 1   [1] *)
 getS 0, con 2, leq, cbranch 10,  (* while 2 <= !n do    *)
 getS 0, getS 1, mul, putS 1,     (*   a := !a * !n ;    *)
 con 1, getS 0, sub, putS 0,      (*   n := !n - 1       *)
 branch ~12,
 halt]                            (* !a                  *)
```

Die Referenzen `n` und `a` werden im Stapel an den Adressen 0 und 1 alloziert. Die Schleife wird mit einem bedingten *Vorwärtsspung* (positives Argument) und einem unbedingten *Rückwärtssprung* (negatives Argument) realisiert. Beim Erreichen des Haltebefehls liegen nur noch die Referenzen `n` und `a` im dem Stapel. Da `a` an oberster Stelle liegt, wird `!a` ausgegeben.

Realisierung der virtuellen Maschine

Wir realisieren die virtuelle Maschine V0 jetzt in Standard ML. Damit haben wir einerseits eine präzise Definition für V0. Andererseits haben wir eine Realisierung der Maschine, mit der wir Programme ausführen können.

Den Stapel und den Programmspeicher der Maschine realisieren wir durch die in den Abbildungen 14.2 und 14.3 gezeigten Strukturen. Abbildung 14.4 zeigt die Realisierung der kompletten Maschine. Die Prozedur `execute` realisiert den sogenannten *Befehlsinterpreter* der Maschine.

```
structure Stack :>
  sig
    val pushI   : int   -> unit
    val popI    : unit  -> int
    val pushS   : index -> unit
    val updateS : index -> unit
    val clear   : unit  -> unit
  end
=
struct
  val size  = 1000
  val store = Array.array(size, 0)
  val sp    = ref ~1  (* points to topmost cell *)

  fun clear () = sp:= ~1

  fun getS sa    = Array.sub(store, sa)
  fun putS(sa,v) = Array.update(store, sa, v)

  fun check sa = if 0<=sa andalso sa <= !sp then sa
                 else raise Error "illegal stack address"

  fun pushI v = if !sp+1 >= size
                  then raise Error "stack overflow"
                else (sp:= !sp+1 ; putS(!sp, v))

  fun popI () = getS(check(!sp)) before sp:= !sp-1

  fun pushS i   = pushI(getS(check i))
  fun updateS i = putS(check i, popI())
end
```

Abbildung 14.2: Realisierung des Stapels von V0

```
structure ProgramStore :>
  sig
    val load  : code -> unit
    val instr : int -> instruction
  end
=
struct
  val size  = 100
  val store = Array.array(size, halt)

  fun instr ca = if 0<=ca andalso ca<size
                   then Array.sub(store, ca)
                 else raise Error "illegal program address"

  fun load'(x,i) = if i < size
                     then i+1 before Array.update(store,i,x)
                   else raise Error "program too large"

  fun load code = (foldl load' 0 code; ())
end
```

Abbildung 14.3: Realisierung des Programmspeichers von V0

14.2 Übersetzung einer imperativen Sprache

Wir zeigen jetzt, wie man einen Übersetzer für eine einfache imperative Programmiersprache W schreibt.[1] Ein W-Programm (Abbildung 14.6 zeigt ein Beispiel) deklariert zunächst einige imperative Variablen, führt dann eine *Anweisung* aus, und liefert schließlich den Wert eines Ausdrucks. Die Ausdrücke von W können die Werte der Variablen lesen aber nicht ändern. Die Anweisungen (englisch „statements“) können die Werte der Variablen lesen und verändern. Wie in imperativen Sprachen üblich, liefern sie keine Werte. Alle Werte haben den Typ `int`. Die Booleschen Werte werden durch Zahlen repräsentiert, wobei das Konditional die Zahl 0 wie `false` behandelt und allen anderen Zahlen wie `true`.

Abbildung 14.5 zeigt die abstrakte Syntax von W. Wie in imperativen Sprachen üblich, ist das Konditional als Anweisung modelliert. Abbildung 14.6 zeigt die abstrakte Syntax eines W-Programms. Zusätzlich ist eine Darstellung des Programms in einer möglichen konkreten Syntax angegeben. Das Programm entspricht dem Fakultätsprogramm in Abschnitt 14.1.

Abbildungen 14.7 und 14.8 zeigen einen Übersetzer, der W-Programme in V0-Programme übersetzt. Die deklarierten Variablen werden, wie in Abschnitt 14.1 gezeigt, im Stapel alloziert. Mithilfe von Umgebungen merkt sich der Übersetzer,

[1] W steht für while.

```
structure VM :>
  sig
    exception Error of string
    val run : code -> int  (* Error *)
  end
=
struct

  exception Error of string

  structure ProgramStore   ···

  structure Stack   ···

  open ProgramStore Stack

  val pc = ref 0

  exception Halt of int

  fun execute instruction =
    case instruction of
      con i       => (pushI i ; pc:= !pc+1)
    | add         => (pushI (popI() + popI()) ; pc:= !pc+1)
    | sub         => (pushI (popI() - popI()) ; pc:= !pc+1)
    | mul         => (pushI (popI() * popI()) ; pc:= !pc+1)
    | leq         => (pushI (if popI() <= popI() then 1 else 0) ;
                      pc:= !pc+1)
    | branch noi  => pc:= !pc+noi
    | cbranch noi => if popI()=0 then pc:= !pc+noi else pc:= !pc+1
    | getS i      => (pushS i ; pc:= !pc+1)
    | putS i      => (updateS i ; pc:= !pc+1)
    | halt        => raise Halt(popI())

  fun run code =
    (load code ;
     pc:=0 ;
     clear() ;
     while true do execute(instr(!pc)) ;
     0 )
    handle Halt n   => n
         | Overflow => raise Error "Overflow"

end
```

Abbildung 14.4: Realisierung der virtuellen Maschine V0

```
type id  = string            (* identifier            *)

datatype exp =               (* expression            *)
  Con    of int                  (* constant          *)
| Var    of id                   (* variable          *)
| Add    of exp * exp            (* addition          *)
| Sub    of exp * exp            (* subtraction       *)
| Mul    of exp * exp            (* multiplication    *)
| Leq    of exp * exp            (* less or equal test *)

datatype sta =               (* statement             *)
  Assign of id  * exp            (* assignment        *)
| If     of exp * sta * sta      (* conditional       *)
| While  of exp * sta            (* while loop        *)
| Seq    of sta list             (* sequentialization  *)

type declaration = id * exp

type program  = declaration list * sta * exp
```

Abbildung 14.5: Abstrakte Syntax von W

```
var n:= 12
var a:= 1
while 2<=n do
  a:= a*n;
  n:= n-1
end
return a


([ ("n", Con 12),
   ("a", Con 1 ) ],
 While(Leq(Con 2, Var"n"),
       Seq [Assign("a", Mul(Var"a", Var"n")),
            Assign("n", Sub(Var"n", Con 1))]),
 Var"a")
```

Abbildung 14.6: Konkrete und abstrakte Syntax eines W-Programms

welche Bezeichner an Variablen gebunden sind und an welchen Adressen diese im Stapel alloziert sind. Der Übersetzer setzt voraus, das Umgebungen über eine Struktur `Env` mit der Signatur in Abbildung 11.6 zur Verfügung gestellt werden. Außerdem werden die Typdeklaration für den Befehlssatz von V0 vorausgesetzt (siehe Abbildung 14.1). Einige wichtige Aspekte der Übersetzung von W nach V0 sind:

1. Variablen werden im Stapel alloziert und durch die entsprechende Stapeladresse identifiziert.
2. Ein Audruck wird in eine Befehlsliste übersetzt, deren Ausführung den Stapel um den Wert erweitert, den die Auswertung des Ausdrucks liefert. Die bereits existierenden Stapelelemente bleiben dabei unverändert.
3. Eine Anweisung wird in eine Befehlsliste übersetzt, deren Ausführung die Werte der im Stapel allozierten Variablen verändern kann. Nach Ausführung der Befehlsliste hat der Stapel genau so viele Elemente wie vorher.

14.3 Statische Prozeduren

Im Allgemeinen benötigt man für die Repräsentation einer Prozedur eine Befehlssequenz und eine Umgebung (siehe Kapitel 11). In vielen Fällen gelingt es jedoch, Prozeduren ohne Umgebung nur durch eine Befehlssequenz zu repräsentieren. Wir sprechen dann von einer *statischen Repräsentation*. Wenn die Repräsentation einer Prozedur dagegen aus einer Befehlssequenz und einer Umgebung besteht, sprechen wir von einer *dynamischen Repräsentation*. Einige Programmiersprachen, zum Beispiel C und Pascal, sind so entworfen, dass alle Prozeduren statisch repräsentiert werden können. In solchen Sprachen besteht eine besonders enge Beziehung zwischen einer Prozedur und ihrer Deklaration. Standard ML und Java sind dagegen Sprachen, in denen nicht alle Prozeduren statisch repräsentiert werden können.

Im Kontext einer Übersetzung sprechen wir oft von *statischen und dynamischen Prozeduren* und meinen damit, dass die Repräsentation der Prozedur statisch beziehungsweise dynamisch ist.

In Standard ML haben Prozeduren semantisch gesehen immer genau ein Argument. Die Übergabe von mehreren Argumenten erfolgt entweder mithilfe von Tupeln oder von Kaskadierung. Viele Programmiersprachen, zum Beispiel C, Pascal und Java, stellen jedoch weder Tupel noch Kaskadierung zur Verfügung. Stattdessen haben diese Sprachen *stellige Prozeduren*, die mit einer bestimmten Anzahl von Argumenten definiert werden.

```
structure WC :>
  sig val compile: program -> code  (* Unbound *) end
=
struct
  open Env
  type env = index env

  fun compileE (e:exp, env:env) : code = ···

  fun compileS (s:sta, env:env) : code =
    case s of
      Assign(i,e) => compileE(e, env) @ [putS (lookup(i,env))]
    | If(e,s1,s2) => let
                       val ce  = compileE(e, env)
                       val cs1 = compileS(s1, env)
                       val cs2 = compileS(s2, env)
                     in
                       ce @ [cbranch (length cs1 + 2)] @
                       cs1 @ [branch (length cs2 + 1)] @
                       cs2
                     end
    | While(e, s) => let
                       val ce = compileE(e, env)
                       val cs = compileS(s, env)
                     in
                       ce @ [cbranch (length cs + 2)] @
                       cs @ [branch (~(length cs + length ce + 1))]
                     end
    | Seq ss      => foldr (fn (s,c) => compileS(s,env) @ c) nil ss

  fun compileD (ds: declaration list, env:env, sa:index): code*env =
    case ds of
      nil       => (nil,env)
    | (i,e)::dr => let
                     val env'        = insert(i, sa+1, env)
                     val (cdr,env'') = compileD(dr, env', sa+1)
                   in
                     (compileE(e,env) @ cdr, env'')
                   end

  fun compile ((ds,s,e) : program) : code =
    let
      val (cds, env) = compileD(ds, empty, ~1)
    in
      cds @ compileS(s,env) @ compileE(e,env) @ [halt]
    end
end
```

Abbildung 14.7: Übersetzer von W nach V0

```
fun compileE (e:exp, env:env) : code =
  case e of
    Con i      => [con i]
  | Var i      => [getS (lookup(i,env))]
  | Add(e1,e2) => compileE(e2, env) @ compileE(e1, env) @ [add]
  | Sub(e1,e2) => compileE(e2, env) @ compileE(e1, env) @ [sub]
  | Mul(e1,e2) => compileE(e2, env) @ compileE(e1, env) @ [mul]
  | Leq(e1,e2) => compileE(e2, env) @ compileE(e1, env) @ [leq]
```

Abbildung 14.8: Übersetzung von W-Ausdrücken

Wir erweitern V0 jetzt um vier Befehle, mit denen wir statische Prozeduren mit einer beliebigen Anzahl von Argumenten realisieren können. Die erweiterte Maschine nennen wir V1. Die Semantik der bisher eingeführten Befehle bleibt unverändert. Abbildung 14.9 zeigt die vier neuen Befehle.

Die Semantik der neuen Befehle erklären wir mithilfe eines Beispiels. Abbildung 14.10 zeigt die Übersetzung eines Standard ML Ausdrucks in ein V1-Programm. Der Ausdruck deklariert eine Prozedur `exp` und berechnet damit die Potenz 2^{10}. Der erste Befehl der Übersetzung

```
proc(2,15)
```

leitet die Befehlssequenz für die Prozedur `exp` ein und sagt, dass die Prozedur zwei Argumente hat und dass die Befehlssequenz aus 15 Befehlen besteht. Die Maschine führt diesen Befehl aus, indem sie die Befehlssequenz für die Prozedur überspringt. Der letzte Befehl dieser Sequenz ist `return`. Danach kommen die Befehle für die Prozeduranwendung `exp(2,10)`. Die Argumente und das Ergebnis der Prozeduranwendung werden wie bei einer Operation mithilfe des Stapels übergeben. Der Befehl

```
call 0
```

leitet die Ausführung des Prozeduraufrufs ein. Dabei wird die Prozedur `exp` durch die *Anfangsadresse* ihrer Befehlssequenz im Programmspeicher identifiziert (im Beispiel 0). Die Ausführung des Prozeduraufrufs schließt die Ausführung weiterer Prozeduraufrufe ein. Nach der Ausführung eines Prozeduraufrufs fährt die Maschine mit der Ausführung des Befehls fort, der auf den den Aufruf einleitenden Call-Befehl folgt. Hier ist dies der Befehl `halt`.

Wir kommen jetzt zur Erklärung der Befehlssequenz für die Prozedur `exp`. Diese hat die Form

```
proc(2,15),
Befehle für den Rumpf der Prozedurdeklaration
return
```

```
type index = int
type noi   = int            (* number of instructions      *)
type noa   = int            (* number of arguments         *)
type ca    = int            (* code address                *)

datatype instruction =
    ...
  | proc    of noa * noi    (* begin of procedure code     *)
  | getF    of index        (* push value from frame       *)
  | call    of ca           (* call procedure              *)
  | return                  (* return from procedure call  *)
```

Abbildung 14.9: Befehle für statische Prozeduren

```
let
  fun exp(x,n) = if n<=0 then 1 else x*exp(x,n-1)
in
  exp(2,10)
end


[proc(2,15),                          (*  fun exp(x,n) =     *)
 con 0, getF ~2, leq, cbranch 3,      (*  if n<=0            *)
 con 1, branch 8,                     (*  then 1             *)
 con 1, getF ~2, sub, getF ~1,        (*  else x*exp(x,n-1)  *)
 call 0, getF ~1, mul,
 return,
 con 10, con 2, call 0,               (*  exp(2,10)          *)
 halt]
```

Abbildung 14.10: Übersetzung einer statischen Prozedur

Der Rumpf der Prozedurdeklaration wird wie ein normaler Ausdruck übersetzt, wobei der Zugriff auf die Argumente der Prozedur mit dem Befehl

`getF` $-i$

erfolgt. Dieser legt das i-te Argument des jeweiligen Prozeduraufrufs auf den Stapel. Dabei werden die Argumente von links nach rechts beginnend mit eins durchnummeriert.

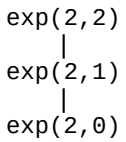
14.4 Realisierung der Prozedurbefehle

Die Prozedurbefehle `proc`, `call` und `return` sind spezielle Sprungbefehle, die den Programmzähler der Maschine manipulieren:

- `proc` überspringt die Befehlssequenz einer Prozedur.
- `call` springt zum zweiten Befehl einer Prozedur.
- `return` springt zum Nachfolger des Call-Befehls, mit dem die Befehlssequenz der Prozedur angesprungen wurde.

Die Realisierung des Return-Befehls ist nicht einfach, da er zum Nachfolger des richtigen Call-Befehls springen muss (im Beispiel gibt es zwei Call-Befehle für `exp`). Auch die Realisierung des GetF-Befehls ist nicht trivial, da sich bei einer rekursiven Prozedur gleichzeitig mehrere Aufrufe der Prozedur in der Ausführung befinden können und `getF` das Argument des richtigen Aufrufs liefern muss.

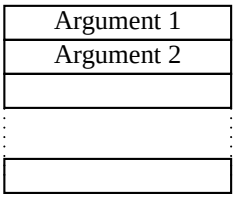
Wir zeigen jetzt, wie man die Implementierung der Maschine V0 (siehe Abschnitt 14.1) zu einer Implementierung der Maschine V1 erweitert. Der größte Teil der Erweiterungen betrifft den Stapel. Dieser kann bei V1 in sogenannten *Aufrufrahmen* (englisch „call frames“) unterteilt werden. Ein Aufrufrahmen repräsentiert einen sich in der Ausführung befindlichen Prozeduraufruf. Betrachten Sie dazu den Rekursionsbaum für den Aufruf `exp(2,2)`:



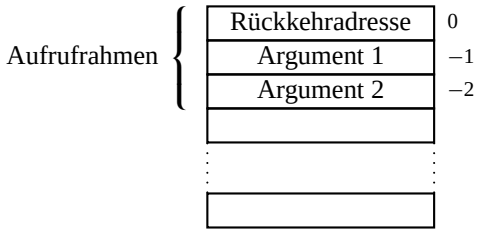
Wenn der letzte Aufruf `exp(2,0)` ausgeführt wird, sind alle drei Aufrufe des Baums durch Rahmen im Stapel der Maschine repräsentiert. Dabei liegt der Rahmen für den Aufruf `exp(2,0)` zuoberst. Wenn dieser Aufruf beendet ist, wird sein Rahmen vom Stapel genommen und mit dem Aufruf `exp(2,1)` fortgefahren, dessen Rahmen nun im Stapel zuoberst liegt.

Ein Aufrufrahmen enthält die Argumente sowie die sogenannte *Rückkehradresse* (englisch „return address“) des Aufrufs. Die Rückkehradresse ist die Adresse des Call-Befehls, der den Prozeduraufruf eingeleitet hat.

Stellen Sie sich vor, dass eine zweistellige Prozedur aufgerufen werden soll, deren Befehlsssequenz im Programmspeicher mit der Adresse a beginnt. Bevor die Prozedur aufgerufen werden kann, müssen die Argumente des Aufrufs auf den Stapel gelegt werden:



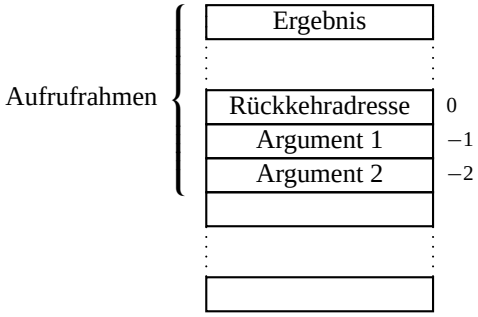
Die Prozedur wird jetzt mit dem Befehl `call` a aufgerufen. Dieser erweitert den Stapel um einen neuen Rahmen, der zunächst die Argumente des Aufrufs und die Rückkehradresse (also die Adresse des Befehls `call` a) enthält:



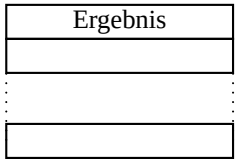
Die Anzahl der Argumente besorgt sich die Maschine aus dem ersten Befehl der Prozedur. Die Zellen eines Rahmens werden wie oben gezeigt *relativ adressiert*. Das i-te Argument hat also die Adresse $-i$ und die Rückkehradresse hat die Adresse 0. Nachdem der Call-Befehl den Rahmen im Stapel angelegt hat, setzt er den Befehlszähler auf die Adresse des zweiten Befehls der Prozedur. Damit ist seine Ausführung beendet.

Die Befehlssequenz der Prozedur kann wie üblich neue Werte auf den Stapel legen. Dabei wächst der Rahmen des Aufrufs nach oben. Der Befehl `getF` i legt den Wert der i-ten Zelle des obersten Rahmens auf den Stapel. Das erklärt die Realisierung der Argumentzugriffe in der Befehlssequenz einer Prozedur. Wenn die Befehlssequenz einer Prozedur eine Prozedur aufruft (zum Beispiel bei Rekursion), wird ein neuer Rahmen angelegt, der im Stapel über dem Rahmen des vorherigen Aufrufs liegt. Der untere Rahmen gibt dabei die Argumente an den oberen Rahmen ab.

Ein Prozeduraufruf wird mit dem Befehl `return` beendet. Wenn `return` ausgeführt wird, muss das Prozedurergebnis zuoberst auf dem Stapel liegen:



Der Befehl `return` nimmt den obersten Rahmen vom Stapel und legt danach das Ergebnis wieder auf den Stapel:



Außerdem setzt `return` den Programmzähler auf die um eins erhöhte Rückkehradresse des entfernten Rahmens. Damit hat der für den entfernten Rahmen verantwortliche Call-Befehl insgesamt den Effekt, dass er die Argumente des Aufrufs durch das Ergebnis des Aufrufs ersetzt hat.

Damit der Befehlsinterpreter den beschriebenen Ablauf realisieren kann, stellt die den Stapel realisierende Struktur `Stack` drei weitere Operationen zur Verfügung:

- `pushF` i legt einen neuen Rahmen auf den Stapel, der die i obersten Zellen des Stapels als Argumente bekommt (mit den Relativadressen $-1, \ldots, -i$).
- `pushFF` i legt den Wert in der i-ten Zelle des obersten Rahmens auf den Stapel.
- `popF` i nimmt den obersten Rahmen vom Stapel und legt danach die Werte in den i obersten Zellen des entfernten Rahmens wieder auf den Stapel.

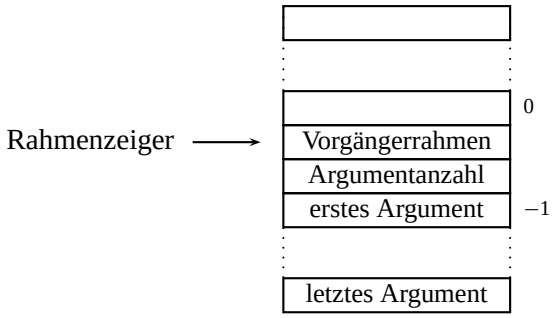
Damit kann der Befehlsinterpreter die Prozedurbefehle wie in Abbildung 14.11 gezeigt realisieren. Dabei wird der Befehl `popF` i nur für $i = 1$ benötigt. Mit der allgemeinen Form `popF` i können i Ergebnisse übergeben werden. Die allgemeinen Form werden wir im nächsten Abschnitt für die effiziente Realisierung von Endaufrufen verwenden.

```
fun getNoa ca = case instr ca of
                  proc(noa,_) => noa
                | _ => raise Error "wrong program address"

fun getRA () = (pushFF 0 ; popI())

fun execute instruction =
  case instruction of
    ...
  | proc(noa, noi) => pc:= !pc+noi
  | getF i         => (pushFF i ; pc:= !pc+1)
  | call ca        => (pushF(getNoa ca) ; pushI(!pc) ; pc:= ca+1)
  | return         => (pc:= getRA()+1 ; popF 1)
```

Abbildung 14.11: Realisierung der Prozedurbefehle im Befehlsinterpreter

Realisierung des Stapels

Wir kommen jetzt zur Realisierung des Stapels. Dazu erweitern wir die Implementierung des Stapels für V0 (siehe Abbildung 14.2) um die Verwaltung der Aufrufrahmen. Wie zu erwarten repräsentieren wir die Zellen eines Rahmens durch aufeinander folgende Zellen des Stapels. Jeder Rahmen bekommt jedoch zwei zusätzliche *interne Zellen*, die nach außen verborgen werden. Die internen Zellen werden zwischen den *Argumentzellen* und den *Hauptzellen* des Rahmens untergebracht:



Die untere interne Zelle enthält die Anzahl der Argumente des Rahmens. Die obere interne Zelle heißt *Ankerzelle* des Rahmens. In der Ankerzelle eines Rahmens ist die Adresse (im Stapel) der Ankerzelle des Vorgängerrahmens abgelegt (wenn es einen gibt). Die Struktur `Stack` verwaltet einen sogenannten *Rahmenzeiger* (`fp`), der auf die Ankerzelle des obersten Rahmens im Stapel zeigt (wenn es einen gibt). Wenn ein neuer Rahmen auf den Stapel gelegt wird, wird der bisherige Wert des Rahmenzeigers in die Ankerzelle des neuen Rahmens geschrieben und der Rahmenzeiger wird auf die Ankerzelle des neuen Rahmens gesetzt. Die Operation `pushFF` kann mithilfe des Rahmenzeigers realisiert werden. Die Operation

```
val fp = ref ~1  (* frame pointer *)

fun pushF noa = (pushI noa ; pushI(!fp) ; fp:= !sp)

fun pushFF i = if ~(getS(!fp-1)) <= i andalso i <= !sp
               then pushI(getS(!fp+i+(if i<0 then ~1 else 1)))
               else raise Error "illegal frame address"

val wc = ref ~1  (* counter for while loop  *)

fun moveS (start, noa) = (wc:= start ;
                          while !wc <= start+noa-1 do
                            (pushI(getS(!wc)) ; wc:= !wc+1))

fun popF noa = if !fp>=0 then
                 let
                   val start = !sp-noa+1
                 in
                   sp:= !fp-2-getS(!fp-1) ;
                   fp:= getS(!fp) ;
                   moveS(start, noa)
                 end
               else raise Error "illegal pop frame"
```

Abbildung 14.12: Realisierung der neuen Stapeloperationen

`popF` lässt sich mithilfe des Rahmenzeigers und der Werte in den internen Zellen des obersten Rahmens realisieren. Abbildung 14.12 zeigt die Implementierung der neuen Stapeloperationen.

14.5 Endaufrufe

Stellen Sie sich einen Prozeduraufruf vor, dessen Rekursionsbaum die Tiefe n hat. Wenn die Maschine V1 den Aufruf ausführt, dann hat sie bei der Ausführung des tiefsten Aufrufs im Rekursionsbaum $n + 1$ Rahmen im Stapel (einen für jeden Knoten auf dem Pfad von der Wurzel zum tiefsten Aufruf). Der Platzbedarf im Stapel wächst also linear mit der Rekursionstiefe eines Aufrufs.

In Abschnitt 7.11 haben wir gelernt, dass iterative Rekursion mit konstantem Platzbedarf realisiert werden kann, also unabhängig von der Rekursionstiefe. Durch die Einführung eines neuen Befehls `callR` werden wir unsere virtuelle Maschine jetzt so verbessern, dass sie iterative Rekursion in der Tat mit konstantem Platzbedarf ausführen kann. Die so verbesserte Maschine nennen wir V.

Als Beispiel betrachten wir die iterative Prozedur

```
fun gcd(x,y) = if x<=y
               then if y<=x then x else gcd(x, y-x)
               else gcd(x-y, y)
```

die den größten gemeinsamen Teiler zweier natürlicher Zahlen berechnet. Diese Prozedur können wir wie folgt nach V1 übersetzen:

```
[proc(2,23),                              (* fun gcd(x,y) =  *)
 getF~2, getF~1, leq, cbranch 13,         (* if x<=y         *)
 getF~1, getF~2, leq, cbranch 3,          (* then if y<=x    *)
 getF~1, branch 12,                       (* then x          *)
 getF~1, getF~2, sub, getF~1,             (* else gcd(x,y-x) *)
 call 0, branch 6,
 getF~2, getF~2, getF~1, sub,             (* else gcd(x-y,y) *)
 call 0, return]
```

Da die beiden unbedingten Sprungbefehle zu dem abschließenden Return-Befehl springen, können wir sie durch den Return-Befehl ersetzen, ohne den Effekt der Befehlssequenz zu ändern:

```
[proc(2,23),                              (* fun gcd(x,y) =  *)
 getF~2, getF~1, leq, cbranch 13,         (* if x<=y         *)
 getF~1, getF~2, leq, cbranch 3,          (* then if y<=x    *)
 getF~1, return,                          (* then x          *)
 getF~1, getF~2, sub, getF~1,             (* else gcd(x,y-x) *)
 call 0, return,
 getF~2, getF~2, getF~1, sub,             (* else gcd(x-y,y) *)
 call 0, return]
```

Der neue Befehl `callR` *a* ersetzt eine Kombination

```
call a, return
```

Damit bekommen wir die folgende Befehlssequenz:

```
[proc(2,21),                              (* fun gcd(x,y) =  *)
 getF~2, getF~1, leq, cbranch 12,         (* if x<=y         *)
 getF~1, getF~2, leq, cbranch 3,          (* then if y<=x    *)
 getF~1, return,                          (* then x          *)
 getF~1, getF~2, sub, getF~1, callR 0,    (* else gcd(x,y-x) *)
 getF~2, getF~2, getF~1, sub, callR 0]    (* else gcd(x-y,y) *)
```

Wir werden gleich zeigen, dass man CallR so realisieren kann, dass der bestehende Aufrufrahmen durch den neuen Aufrufrahmen ersetzt wird. Im Gegensatz zu Call legt CallR also keinen zusätzlichen Rahmen auf den Stapel. Damit realisiert die obige Befehlssequenz die Prozedur `gcd` mit konstantem Platzbedarf, da die Rekursion über den CallR-Befehl läuft.

Der Call-Befehl der Kombination

```
call a, return
```

erzeugt einen neuen Aufrufrahmen, bei dessen Entfernung zum Return-Befehl gesprungen wird. Der Return-Befehl nimmt seinen Aufrufrahmen vom Stapel und springt zu der in diesem Rahmen vermerkten Rückkehradresse. Man kann denselben Effekt erreichen, wenn man den bestehenden Rahmen sofort vom Stapel nimmt und seine Rückkehradresse zur Rückkehradresse des neuen Rahmens macht. Genau das macht der CallR-Befehl, den wir wie folgt im Befehlsinterpreter (siehe Abbildung 14.11) realisieren:

```
| callR ca => let
                val ra = getRA()
                val noa = getNoa ca
              in
                popF noa ; pushF noa; pushI ra ; pc:=ca+1
              end
```

Eine Prozeduranwendung *a* im Rumpf einer Prozedurdeklaration steht in *Endposition*, wenn die Ausführung von *a* das Ergebnis der deklarierten Prozedur liefert. Alle Prozeduranwendungen in Endposition können mit dem CallR-Befehl übersetzt werden (unabhängig davon, ob es sich um rekursive oder nichtrekursive Anwendungen handelt). Eine Prozedurdeklaration erzeugt genau dann eine iterative Prozedur, wenn alle rekursiven Anwendungen der Prozedur in Endposition stehen. Prozeduranwendungen, die in Endposition stehen, bezeichnet man auch als *Endaufrufe*.[2]

14.6 Übersetzung einer funktionalen Sprache

Abbildung 14.13 zeigt die abstrakte Syntax einer funktionalen Programmiersprache S, die nur statische Prozedurdeklarationen zulässt.[3] Ein S-Programm deklariert zunächst einige Prozeduren und wertet dann einen Ausdruck aus. Genau wie W hat S nur Werte des Typs `int`. Abbildung 14.14 zeigt die abstrakte Syntax eines S-Programms, das mithilfe einer Prozedur `exp` die Potenz 2^{10} berechnet. Das Programm ist zusätzlich mit der konkreten Syntax von Standard ML dargestellt.

Abbildungen 14.15 und 14.16 zeigen einen Übersetzer, der S-Programme und nach V übersetzt. Der Übersetzer merkt sich mithilfe von Umgebungen, welche Bezeichner an Prozeduren oder Argumente gebunden sind, und wie diese im übersetzten Programm zu adressieren sind. Die Prozedur

[2] Diese Sprechweise ist bequem aber ungenau, da eine Prozeduranwendung und ein Prozeduraufruf eigentlich verschiedene Dinge sind. Eine Prozeduranwendung ist ein Ausdruck (also ein syntaktisches Objekt). Dagegen ist ein Prozeduraufruf ein Paar aus einer Prozedur und einem Wert (also ein Paar aus zwei semantischen Objekten).

[3] S steht für Simple.

```
type id = string                  (* identifier            *)

datatype exp =                    (* expression            *)
    Con  of int                      (* constant           *)
  | Id   of id                       (* argument           *)
  | Add  of exp * exp                (* addition           *)
  | Sub  of exp * exp                (* subtraction        *)
  | Mul  of exp * exp                (* multiplication     *)
  | Leq  of exp * exp                (* less or equal test *)
  | App  of id  * exp list           (* application        *)
  | If   of exp * exp * exp          (* conditional        *)

type declaration = id * id list * exp

type program = declaration list * exp
```

Abbildung 14.13: Abstrakte Syntax von S

```
let
  fun exp(x,n) = if n<=0 then 1 else x*exp(x,n-1)
in
  exp(2,10)
end


([
  ("exp", ["x", "n"],
   If(Leq(Id"n", Con 0),
      Con 1,
      Mul(Id"x", App("exp", [Id"x", Sub(Id"n", Con 1)])))))
  ],
 App("exp", [Con 2, Con 10])
)
```

Abbildung 14.14: Konkrete und abstrakte Syntax eines S-Programms

```
compileE : exp * env * code -> code
```

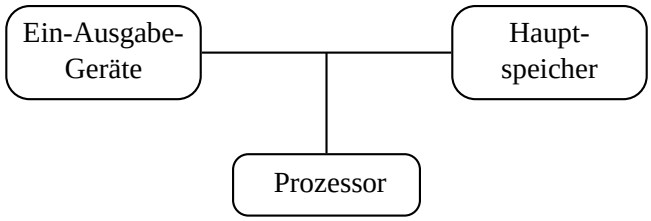
übersetzt einen Ausdruck relativ zu einer Umgebung und einem *Schwanz*. Der Schwanz zeigt an, ob sich der Ausdruck in Endposition innerhalb einer Prozedurdeklaration befindet. Der Schwanz kann nur die Werte `[return]` (in Endposition) und `nil` (nicht in Endposition) annehmen.

Hier sind einige wichtige Aspekte der Übersetzung von S nach V:

1. Prozeduren werden durch die Anfangsadresse ihrer Befehlssequenz im Programmspeicher adressiert.
2. Argumente werden duch ihren Index in der Argumentliste der Prozedurdeklaration adressiert (beginnend mit 1).
3. Der Aufruf einer Operation oder Prozedur ersetzt die im Stapel stehenden Argumente durch das Ergebnis.
4. Die Argumente für den Aufruf einer Operation oder Prozedur werden in invertierter Reihenfolge auf den Stapel gelegt (das heißt, das erste Argument wird als letztes auf den Stapel gelegt).
5. Vor der Ausführung des Return-Befehls liegt hat der Aufrufrahmen nur eine Hauptzelle, in der das Ergebnis des Aufrufs steht.
6. Vor der Ausführung des CallR-Befehls hat der Aufrufrahmen nur soviele Hauptzellen, wie die aufzurufende Prozedur Argumente hat. In diesen Zellen liegen die Argumente des einzuleitenden Aufrufs.

14.7 Die Software-Hardware-Schnittstelle

Die Hardware-Architektur eines Computers lässt sich wie folgt darstellen:



Der *Prozessor* definiert die Maschinensprache und führt deren Befehle aus. Der *Hauptspeicher* ist eine große Reihung, in der die auzuführenden Programme und die für die Ausführung benötigten Daten abgelegt werden. Bei den *Ein-Ausgabe-Geräten* handelt es sich unter anderem um den Plattenspeicher, den CD-Leser,

```
structure SC :>
  sig
    exception Error of string
    val compile: program -> code  (* Error *)
  end
=
struct
  exception Error of string

  datatype idType =
      Arg of index
    | Proc of ca

  open Env
  type env = idType env

  fun insertArgs' (i, (env, ai)) = (insert(i,Arg ai,env), ai+1)

  fun insertArgs (is, env) = #1(foldl insertArgs' (env,1) is)

  fun compileE (e:exp, env:env, tail:code) : code = ···

  fun compileD (ds: declaration list, env:env, ca:ca) : code*env =
    case ds of
      nil          => (nil,env)
    | (i,is,e)::dr =>
        let
          val env'        = insert(i, Proc(ca+1), env)
          val env''       = insertArgs(is, env')
          val ce          = compileE(e, env'', [return])
          val cd          = [proc(length is, 1+length ce)] @ ce
          val (cdr,env'') = compileD(dr, env', ca + length cd)
        in
          (cd @ cdr, env'')
        end

  fun compile ((ds,e) : program) : code =
    let
      val (cds,env) = compileD(ds, empty, ~1)
    in
      cds @ compileE(e,env,nil) @ [halt]
    end
    handle
    Unbound i => raise Error("Unbound identifier: " ^ i)
end
```

Abbildung 14.15: Übersetzer von S nach V

```
fun lookupA (i,env) =
  case lookup(i,env) of
    Arg i => i
  |  _    => raise Error("Argument expected: " ^ i)

fun lookupP (i,env) =
  case lookup(i,env) of
    Proc ca => ca
  | _       => raise Error("Procedure expected: " ^ i)

fun compileEs (es : exp list, env:env) : code =
  foldl
  (fn (e,c) => compileE(e, env, nil) @ c)
  nil es

and compileE (e:exp, env:env, tail:code) : code =
  case e of
    Con i        => [con i] @ tail
  | Id i         => [getF(~(lookupA(i,env)))] @ tail
  | Add(e1,e2)   => compileEs([e1,e2], env) @ [add] @ tail
  | Sub(e1,e2)   => compileEs([e1,e2], env) @ [sub] @ tail
  | Mul(e1,e2)   => compileEs([e1,e2], env) @ [mul] @ tail
  | Leq(e1,e2)   => compileEs([e1,e2], env) @ [leq] @ tail
  | If(e1,e2,e3) => let
                      val c1 = compileE(e1,env,nil)
                      val c2 = compileE(e2,env,tail)
                      val c3 = compileE(e3,env,tail)
                    in
                      if null tail
                      then
                        c1 @ [cbranch (2+length c2)] @
                        c2 @ [branch (1+length c3)] @
                        c3
                      else
                        c1 @ [cbranch (1+length c2)] @
                        c2 @
                        c3
                    end
  | App(i, es)   => compileEs(es,env) @
                    [(if null tail then call else callR)
                     (lookupP(i,env))]
```

Abbildung 14.16: Übersetzung von S-Ausdrücken

die Tastatur, die Maus, den Bildschirm und die Netzanbindung. Die Verbindung zwischen dem Prozessor, dem Hauptspeicher und den Ein-Ausgabe-Geräten wird durch einen sogenannten *Bus* realisiert.

Die Hardware eines Computers arbeitet nur mit sehr einfachen Werten, die man *Worte* nennt. Ein Wort ist ein Tupel über der Menge $\{0, 1\}$. Die meisten Computer arbeiten heute mit Worten der Länge 32; man spricht von *32-Bit-Computern*. Es gibt auch *64-Bit-Computer*. 64-Bit-Computer sind leistungsfähiger als 32-Bit-Computer, aber dafür aufwändiger zu bauen und daher teurer. Worte kann man mithilfe der Binärdarstellung als ganze Zahlen interpretieren (siehe Abschnitt 10.2.1). Eine solche Interpretation und entsprechende arithmetische Operationen sind in den Prozessor eingebaut.

Der Hauptspeicher realisiert eine Reihung fester Länge, deren Zellen (Komponenten) der Prozessor und die Ein-Ausgabe-Geräte lesen und schreiben können. Jede Zelle des Hauptspeichers speichert ein Wort. Die Zellen des Hauptspeichers werden ebenfalls durch Worte adressiert. Damit erreicht man, dass der Hauptspeicher Adressen speichern kann, und dass der Prozessor mit Adressen rechnen kann.

Das vom Prozessor ausgeführte Maschinenprogramm muss während der Ausführung im Hauptspeicher abgelegt sein. Der Prozessor kann neue Maschinenprogramme in den Hauptspeicher schreiben und diese dann ausführen. Er kann auch das gerade ausgeführte Programm ändern. Damit Befehlssequenzen im Hauptspeicher ablegt werden können, müssen sie durch Folgen von Wörtern repräsentiert werden. Am Beispiel von V0 zeigen wir, wie man das macht. Zunächst nummerieren wir die Befehlsnamen (siehe Abbildung 14.1) mit null beginnend durch. Mithilfe dieser Codierung können wir Befehle und Programme durch Zahlenfolgen darstellen, die wir wiederum durch Folgen von Wörtern repräsentieren können. Beispielsweise repräsentiert die Folge

```
[0, 7, 0, 4, 2, 9]
```

das Programm

```
[con 7, con 4, sub, halt]
```

Heutige Prozessoren sind in der Regel sogenannte *Registermaschinen*, die keinen Stapel voraussetzen. Stattdessen haben sie einen eingebauten, frei adressierbaren Speicher, dessen Zellen als *Register* bezeichnet werden. Ein Additionsbefehl hat bei Registermaschinen beispielsweise die Form

`add` *a b c*

und bewirkt, dass die Summe der Zahlen in den Registern *b* und *c* in das Register *a* geschrieben wird. Zusätzlich gibt es die Befehle `load` und `store`, mit denen Werte von einer Hauptspeicherzelle in ein Register beziehungsweise von

einem Register in eine Hauptspeicherzelle geschrieben werden können. Während die Register mit einer sehr schnellen aber teuren Technologie realisiert sind, ist der Hauptspeicher mit einer vergleichsweise langsamen aber billigen Technologie realisiert. Daher haben heutige Computer aus Kostengründen kleine Registerspeicher und große Hauptspeicher.

Wie unsere virtuelle Maschine verwalten Prozessoren einen Programmzähler, der die Anfangsadresse des nächsten auszuführenden Befehls im Hauptspeicher vorgibt. Der Programmzähler wird durch ein spezielles Register realisiert, das durch entsprechende Befehle gelesen und geschrieben werden kann. Mit diesen Befehlen können die Call- und Return-Operationen für Prozeduren realisiert werden. Der für Prozeduren erforderliche Stapel wird mithilfe des Register- und Hauptspeichers realisiert.

Das Grundzüge des gerade beschriebenen Computer-Modells wurde 1949 von John von Neumann formuliert. Computer, die nach diesem Modell konstruiert sind, bezeichnet man als *von-Neumann-Computer*. Die meisten heutigen Computer sind von-Neumann-Computer.

Wenn Sie mehr über Maschinensprachen und die Software-Hardware-Schnittstelle wissen wollen, empfehlen wir Ihnen das einführende Buch von Patterson und Hennessy [**?**].

14.8 Realisierung von Programmiersystemen

Programmiersprachen werden durch sogenannte Programmiersysteme realisiert. Wir haben das Programmiersystem Moscow ML kennengelernt, das unter anderem einen Interpreter, einen Übersetzer und diverse Standardstrukturen zur Verfügung stellt.

Die Übersetzung von Programmiersprachen in Maschinensprachen ist eine komplexe Angelegenheit, bei der eine Vielzahl von Techniken zum Einsatz kommt. Eine grundlegende Idee ist die Einführung einer *Zwischensprache*, mit der man die Übersetzung einer Programmiersprache in eine Maschinensprache in zwei unabhängige Schritte zerlegen kann:

Programmiersprache $\longrightarrow$ Zwischensprache $\longrightarrow$ Maschinensprache

Dabei übersetzt man zuerst von der Programmiersprache in die Zwischensprache und dann von der Zwischensprache in die Maschinensprache. Die Zwischensprache kann ähnlich aussehen wie V (die Sprache unserer virtuellen Maschine). Die Übersetzung von der Zwischensprache in die Maschinensprache bezeichnet man auch als *Code-Erzeugung*.

Ein Übersetzer stellt die Programme der drei Sprachen jeweils gemäß einer abstrakten Syntax dar. Bevor die eigentliche Übersetzung eines Programms der Programmiersprache beginnt, liest der Übersetzer zunächst die textuelle Darstellung des Programms, überführt diese in die abstrakte Darstellung (siehe Kapitel 12), und überprüft dann die Konsistenzbedingungen der statischen Semantik (siehe Kapitel 11). Nachdem der Übersetzer das Maschinenprogramm in abstrakter Darstellung erzeugt hat, schreibt er die konkrete Darstellung (eine Folge von Maschinenwörtern) in eine Datei.

Einen Übersetzer von einer Programmiersprache P in eine Maschinensprache M entwickelt man am besten auf einem Computer R, auf dem P bereits verfügbar ist. Dazu schreibt man in P einen Übersetzer U von P nach M. Auf R kann man dann U mit sich selbst nach M übersetzen. Damit erhält man eine Version von U, die als M-Programm vorliegt. Man kann also einen Übersetzer von P nach M erhalten, ohne in M zu programmieren.

Man unterscheidet zwischen *Vollübersetzung* und *Halbübersetzung*. Bei der Vollübersetzung wird die Programmiersprache wie oben beschrieben in die Maschinensprache des jeweiligen Computers übersetzt. Bei der Halbübersetzung wird die Programmiersprache in eine maschinennahe Zwischensprache übersetzt, die durch eine virtuelle Maschine ausgeführt wird. Die virtuelle Maschine implementiert man mithilfe einer bereits verfügbaren maschinennahen Programmiersprache (typischerweise C). Die Realisierung einer Vollübersetzung ist erheblich aufwendiger als die Realisierung einer Halbübersetzung. Dafür kann man mit einer Vollübersetzung eine höhere Ausführungsgeschwindigkeit für die Programme der Programmiersprache erzielen.

Wir beschreiben jetzt genauer, wie man eine Programmiersprache mittels einer Halbübersetzung implementiert. Sei P die zu implementierende Programmiersprache und Z eine für P geeignete, maschinenorientierte Zwischensprache. Wir benötigen einen Übersetzer U von P nach Z und eine virtuelle Maschine V für Z. Die Maschine V schreiben wir in der maschinenorientierten Programmiersprache C (oder ihrer objektorientierten Erweiterung C++), für die auf allen Computern gute Übersetzer existieren. Den Übersetzer U schreiben wir in P. Mit einem *Hilfsübersetzer*, den wir in einer beliebigen, bereits implementierten Programmiersprache realisieren können, übersetzen wir dann U nach Z. Die übersetzten Versionen von U und V ergeben dann ein Maschinenprogramm, das zunächst von P nach Z übersetzt und dann Z interpretiert. Mit dieser Methode kann man eine Programmiersprache P auf einem Computer implementieren, ohne die Maschinensprache des Computers zu kennen.

Die Zwischenschaltung einer virtuellen Maschine hat den großen Vorteil, dass dieselbe Implementierung von P für alle Computer wiederverwendbar ist, für die

ein C-Übersetzer existiert (was in der Praxis immer der Fall ist). Sie hat jedoch den Nachteil, dass in *P* geschriebene Programme langsamer ausgeführt werden als dies bei einer vollständigen Übersetzung der Fall wäre.

Moscow ML ist übrigens mit einer virtuellen Maschine realisiert, die ursprünglich für den französischen ML-Dialekt Caml entwickelt wurde.

Im Zusammenhang mit der Übersetzung von Programmiersprachen sagt man, dass die Hardware nur über *lineare Strukturen* verfügt. Die typischen Beispiele für lineare Strukturen sind Reihungen und Schleifen. Entsprechend haben wir bei der Implementierung unserer virtuellen Maschine in Standard ML nur lineare Strukturen verwendet. Insbesondere haben wir keine rekursiven Prozeduren, keine Listen und keine rekursiven Konstruktortypen verwendet. Die einzige Ausnahme haben wir bei der Übergabe des auszuführenden Programms gemacht, dass bei der Übergabe mit einer Liste dargestellt wird.

Im nächsten Kapitel werden wir lernen, wie man Programme mit Listen, rekursiven Konstruktortypen und höherstufigen Prozeduren in maschinennahe Strukturen übersetzt. Wenn Sie noch mehr wissen wollen, empfehlen wir Ihnen das Buch von Wilhelm und Maurer [**?**].

Übungsaufgaben

Aufgabe 14.1 (Größte gemeinsame Teiler in V) Die Prozedur

```
fun gcd(x,y) = if x<>y then
                  if x<=y then gcd(x, y-x) else gcd(x-y, y)
               else x
```

berechnet den größten gemeinsamen Teiler zweier positiven ganzen Zahlen.

1. Schreiben Sie mithilfe einer Schleife eine nichtrekursive Variante `gcdi` der Prozedur `gcd`.
2. Schreiben Sie eine Befehlssequenz *gcdi* ohne Prozedurbefehle, sodass das V-Programm

   `[con` *x*`, con` *y*`] @` *gcdi* `@ [halt]`

   den größten gemeinsamen Teiler zweier positiven ganzen Zahlen *x* und *y* berechnet.

Aufgabe 14.2 (Übersetzung und Rückübersetzung) Seien die folgenden Ausdrücke gegeben:

```
datatype exp =
    Con    of int              (* constant           *)
  | Add    of exp * exp        (* addition           *)
  | Sub    of exp * exp        (* subtraction        *)
  | Mul    of exp * exp        (* multiplication     *)
```

1. Schreiben Sie eine Prozedur

   compile : exp -> code

   die einen Ausdruck in ein V0-Programm übersetzt, dessen Ausführung den Wert des Ausdrucks liefert.

2. Schreiben Sie eine Prozedur

   decompile : code -> exp

   sodass für alle Ausdrücke *e* gilt:

   decompile(compile *e*) = *e*

Aufgabe 14.3 (Endaufrufe) Markieren Sie in den folgenden Prozedurdeklarationen alle Endaufrufe.

```
fun f(x,y) = if x<y then y else f(x+1,y)

fun g x = if x<0 then x else f(x, 2*x)

fun h (1::xs) = h xs
  | h xs      = p xs

and p (2::xs) = h xs
  | p (x::xs) = g x + 5
```

Aufgabe 14.4 (Fibonacci-Prozedur in V) Übersetzen Sie die Prozedur

```
fun fib n = if n<2 then n else fib(n-2) + fib(n-1)
```

in eine Befehlssequenz für V. Nehmen Sie dabei an, dass die Befehlssequenz mit der Adresse 0 beginnend im Programmspeicher abgelegt wird.

Übersetzen Sie die Prozeduren

```
fun fibi'(n,a,b) = if n<=0 then a else fibi'(n-1, b, a+b)

fun fibi n = fibi'(n,0,1)
```

in eine Befehlssequenz für V. Nehmen Sie dabei an, dass die Befehlssequenz mit der Adresse 0 beginnend im Programmspeicher abgelegt wird. Übersetzen Sie alle Endaufrufe mit `callR`.

Aufgabe 14.5 (Verschränkte Rekursion in V) Übersetzen Sie die Prozeduren

```
fun even x = if x=0 then true else odd(x-1)
and odd  x = if x=0 then false else even(x-1)
```

in eine Befehlssequenz für V. Nehmen Sie dabei an, dass die Befehlssequenz mit der Adresse 0 beginnend im Programmspeicher abgelegt wird. Übersetzen Sie alle Endaufrufe mit `callR`.

Aufgabe 14.6 (Syntax und Semantik von W) Definieren Sie die abstrakte Syntax von W mithilfe einer abstrakten Grammatik. Definieren Sie die statische und die dynamische Semantik von W mithilfe von Inferenzregeln. Definieren Sie eine konkrete Syntax für W. Schreiben Sie einen Lexer und einen Parser für W.

Aufgabe 14.7 (Syntax und Semantik von S) Definieren Sie die abstrakte Syntax von S mithilfe einer abstrakten Grammatik. Definieren Sie die statische und die dynamische Semantik von S mithilfe von Inferenzregeln. Definieren Sie eine konkrete Syntax für S. Schreiben Sie einen Lexer und einen Parser für S. Erweitern Sie den Übersetzer für S (siehe Abschnitt 14.6) so, dass er prüft, ob das zu übersetzende Programm gemäß der statischen Semantik zulässig ist.

Aufgabe 14.8 (Erweiterung von V) Erweitern Sie die Implementierung der virtuellen Maschine V so, dass neben dem Ergebnis auch die Anzahl der ausgeführten Befehle und die maximale Anzahl der Aufrufrahmen, die sich während der Ausführung auf dem Stapel befanden, ausgegeben wird.