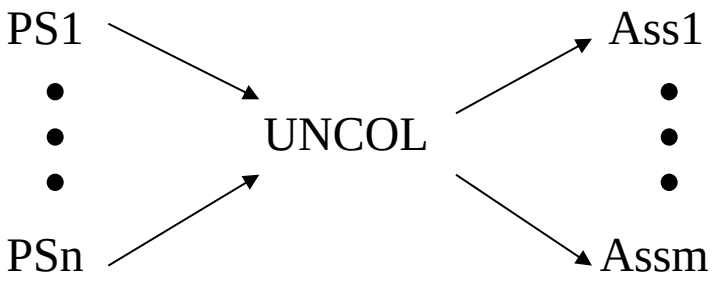
Übersetzung in Zwischencode

- **meist: nicht direkte Übersetzung in Assembler, sondern**
- **Übersetzung in Zwischencode Z: PS -> Z -> Assembler**
- Warum?
  - Z abstrakter als Assembler -> Problembeseitigung aufteilen
  - bessere Portabilität des Compilers, siehe **Java Virtual Machine**
  - leichtere Codeoptimierung, da abstrakter
- **Historisch**: UNCOL (Universal Computer Oriented Language) **(1960)**



statt n*m Übersetzungen
nur n+m

Strukturen von Programmiersprachen

- **Basistypen und –operationen**
  int, double, ..., +,*,...
- **Datenstrukturen (statische, dynamische)**
  Felder, Listen, Strukturen, ...
- **Kontrollstrukturen (Erzeugung von Anweisungen)**
  Schleifen, Verzweigungen, Zuweisungen, ...
- **Applikative und funktionale Konzepte**
  Ausdrücke, Prozeduren, Funktionen
- **Modulstrukturen**
  Blöcke, Moduln, Klassen
- **hier nur imperative Konzepte**
  - keine funktionalen Sprachen oder Logiksprachen
  - keine OO-Sprachen

Strukturen von Assemblercode

- **betrachten von Neumann-Typ** Single Instruction Single Data
- **Speicherhierarchie**
  - Register
  - Cache
  - Hauptspeicher
  - Hintergrundspeicher
- **Befehlsarten**
  - Transferbefehle LOAD, STORE, ...
  - Operationsbefehle ADD, MULT, ...
  - Test-/Sprungbefehle JMC adr, JMP adr, ...
- **Adressierungsarten**
  - relative Adressierung mit Index-/Basisregister
  - indirekte Adressierung (Adresse über Speicherplatz)

Strukturen von Z

- **3-Adress-Befehle** (durch Stack nur 1-Adress-Befehle)
- **Typen und Operationen aus PS übernehmen**
  - hier nur Typen int und boolean (auch als int dargestellt)
  - Operationen +: ADD, *: MULT, <: LT, ...
- **Sprungbefehle**
- **Datenkeller**
  - Auswertung von Ausdrücken
- **Prozedurkeller**
  - Blöcke — nächstes Kapitel
  - Prozeduren
- **Heap**
  - dynamische Datenstrukturen — übernächstes Kapitel

Vorgehen in Vorlesung

- **Beschreibung der Übersetzung als rekursive Funktionen über den Aufbau der PS**
- **Hierzu schrittweiser Aufbau von:**
  - Programmiersprache (incl. Definition der Semantik **M**)
  - Zwischensprache (incl. Definition der Semantik **MZ**)
  - Übersetzung **trans: PS -> Z**
    mit M(P) = MZ(trans(P)), für alle Programme P aus PS

in folgender Reihenfolge:

- Ausdrücke auf Stackmaschine — Prakt. 4
- Anweisungen durch Sprünge — Prakt. 5
- Funktionen und Prozeduren durch Prozedurstack — Prakt. 6

- Bsp.: **Übersetzung eines Ausdrucks:**
  `trans(7 + bez1) =  LIT 7; LOAD(addr(bez1)); ADD;`

Übersetzung von Ausdrücken

- **Einschränkung von Mini-Java auf Mini-Java-Exp durch:**
  - Weglassen des Syntaxdiagramms für *condition*
  - Als *statement* nur *print(expression);* zulassen
  - Einfaches Syntaxdiagramm für *expression* (Prakt. 1 + ident)
- **Beispielprogramm:**

```
final int c1 = 10;          //Konstanten c1,c2,...
int v1, v2 = 5;             //Variablen v1/0,v2,...
print(((c1+v1)*v2));        //Ausgabe 50, nicht Sem.
```

//Vereinfachung:

- **Wiederholung statische Semantik:**
  - Bezeichner müssen paarweise verschieden sein (Symboltab.)
  - nicht initialisierte Variablen erhalten den Wert 0
  - Bezeichner, die verwendet werden, müssen vorher deklariert sein. (Symboltab.)

Zustandsraum & Umgebungen

- **Verwendung folgender Hilfskonstrukte, die Pendant in Z haben:**
  - Loc = {add1,add2,...} abstrakte Speicherplätze (Locations)
  - Sta = {sig | sig : Loc -> Int} Zustandsraum (States),
    sig0 : leerer Zustandsraum, nirgendwo definiert

    dynamischer Aspekt: Speicherwerte während Semantikberechnung veränderbar
  - Env = {rho | rho : Ide -> Int ∪ Loc} Umgebungen (Environments)
    rho0 : leere Umgebung, nirgendwo definiert

    statischer Aspekt: Werte von Konstanten & Speicherplätze der Variablen nach Deklaration nicht mehr veränderbar

- **Beachte: Speicherplätze nur über Vereinbarung durch Umgebung zugänglich**

Semantik von Mini-Java-Exp-Progs

- **Semantik (Meaning)**
  - $\mathbb{M}$: {*program*} -> Int
    - $\mathbb{M}$(*constDecl varDecl print(expression);*)
      = $\mathbb{E}$(*expression*,$\mathbb{DV}$(*varDecl*,$\mathbb{DC}$(*constDecl*,rho0),sig0))
  - $\mathbb{DC}$: {*constDecl*} x Env -> Env
    - $\mathbb{DC}$(`final int c1=z1,...,cm=zm;`,rho)
      = rho[c1/z1,...,cm/zm]
  - $\mathbb{DV}$: {*varDecl*} x Env x Sta -> Env x Sta
    - $\mathbb{DV}$(`int v1,...,vk,vk+1=z1,...,vk+n=zn;`,rho,sig)
      = (rho[v1/add1,...,vk/addk,vk+1/addk+1,...,vk+n/addk+n],
      sig[add1/0,...,addk/0,addk+1/z1,...addk+n/zn])

Ausdruckssemantik

– $\mathbb{E}$: *{expression}* x Env x Sta -> Int

- $\mathbb{E}$*(num,*rho,sig) = ***num***
- $\mathbb{E}$*((exp),*rho,sig) = $\mathbb{E}$*(exp,*rho,sig)

entfällt, s.u.

- $\mathbb{E}$*(ident,*rho,sig) = **if** rho(*ident*) **=** z **then** z
  if rho(*ident*) **=** addr **then** sig(addr)
- $\mathbb{E}$*(exp1+exp2,*rho,sig) = $\mathbb{E}$*(exp1,*rho,sig) **+** $\mathbb{E}$*(exp2,*rho,sig)
- $\mathbb{E}$*(exp1-exp2,*rho,sig) = $\mathbb{E}$*(exp1,*rho,sig) **-** $\mathbb{E}$*(exp2,*rho,sig)
- $\mathbb{E}$*(exp1*exp2,*rho,sig) = $\mathbb{E}$*(exp1,*rho,sig) ***** $\mathbb{E}$*(exp2,*rho,sig)
- $\mathbb{E}$*(exp1/exp2,*rho,sig) = $\mathbb{E}$*(exp1,*rho,sig) **/** $\mathbb{E}$*(exp2,*rho,sig)

syntaktisch **semantisch**

brauchen: syntaktische Struktur: Syntaxbaum, Auflösung der Klammern, Präferenzen

Beispielrechnung

- **Semantik des Programms P =**

```
final int c1 = 10;                    //Konstanten c1,c2,...
int v1, v2 = 5;                       //Variablen v1,v2,...
print(((c1+v1)*v2));
```

$\mathbb{M}$(P) = $\mathbb{E}$(((c1+v1)*v2),$\mathbb{DV}$(int v1, v2 = 5;,
$\mathbb{DC}$(final int c1 = 10;,rho0),sig0))

= $\mathbb{E}$(((c1+v1)*v2),$\mathbb{DV}$(int v1, v2 = 5;,[c1/10],sig0))

= $\mathbb{E}$(((c1+v1)*v2),[c1/10,v1/add1,v2/add2],[add1/0,add2/5])

= $\mathbb{E}$((c1+v1)*v2,[c1/10,v1/add1,v2/add2],[add1/0,add2/5])

= $\mathbb{E}$((c1+v1),[c1/10,v1/add1,v2/add2],[add1/0,add2/5]) *
$\mathbb{E}$(v2,[c1/10,v1/add1,v2/add2],[add1/0,add2/5]) = ...

= 10 *$\mathbb{E}$(v2,[c1/10,v1/add1,v2/add2],[add1/0,add2/5])

= 10 * 5 = 50

ZE

- **Zwischencode für Ausdrücke**
- **Stackmaschine mit Datenkeller und Hauptspeicher**
- **Speicher (Zustandsraum)**
  - SM = BZ x DK x HS Stackmaschine
  - BZ = Nat Befehlszähler
  - DK = Int* Datenkeller (Spitze links)
  - HS = {h | h: Nat -> Int} Hauptspeicher
    h0: leerer Hauptspeicher, nirgends definiert
- **Zustand**
  - s = (m,d,h) ∈ SM
- **Befehlsvorrat Com: Menge der Befehle**
  - Arithmetische Befehle: ADD, MULT, SUB, DIV
  - Transportbefehle: LOD n, LIT z (n ∈ Nat, z ∈ Int)

Befehlssemantik

- **Command-Semantik**
  - beschreibt, was Befehle machen
  - Transformation von einem Zustand in anderen
  - $\mathbb{C}$: Com -> (SM -> SM)
    - $\mathbb{C}$ (LOD n)(m,d,h) = (m+1,h(n):d,h)
    - $\mathbb{C}$ (LIT z)(m,d,h) = (m+1,z:d,h)
    - $\mathbb{C}$ (ADD) (m, z1z2:d' ,h) = (m+1,(z1+z2):d',h)
    - $\mathbb{C}$ (MULT)(m, z1z2:d' ,h) = (m+1,(z1*z2):d',h)
    - $\mathbb{C}$ (SUB) (m, z1z2:d' ,h) = (m+1,(z2-z1):d',h)
    - $\mathbb{C}$ (DIV) (m, z1z2:d' ,h) = (m+1,(z2/z1):d',h)

nur für aufgeführte Fälle definiert

semantisch

ZE-Code & ZE-Semantik

- **Programm P aus ZE-Code hat die Form:**
  - P = 1: c1; 2: c2; ... ; p: cp; p > 0 und ci ist ein ZE-Befehl
- **Semantik von P**
  - $\mathbb{P}$: Progs x SM -> Int wie üblich definiert:
    - Beginn bei Befehlsmarke 1
    - sukzessive Befehlsausführung (Anwendung von $\mathbb{C}$)
    - Abbruch, wenn
      - Befehlszähler nicht mehr auf Befehl zeigt (m>p)
      - Befehl nicht ausführbar (bei ADD keine 2 Elemente auf Keller)
  - Startzustand: (1, , h)
  - Endzustand: (p+1, z, h)
  - Für obigen Endzustand ist Semantik von P: z Stackelement

Bsp.: ZE-Programm & Rechnung

- **ZE-Programm:**

  1: LIT 10;

  2: LOD 1;

  3: ADD;

  4: LOD 2;

  5: MULT;

- **Rechnung für Startzustand (1, , [1/0, 2/5]):**

  (1, , [1/0,2/5])

  (2, 10, [1/0,2/5])

  (3, 0 10, [1/0,2/5])

  (4, 10, [1/0,2/5])

  (5, 5 10, [1/0,2/5])

  (6, 50, [1/0,2/5]) -> 50

Übersetzung

- **Hilfsmittel**
  - Symboltabelle (entspricht der Umgebung in Semantik)

    Tab = {st | st : Ide -> ({const} x Int) ∪({var} x Nat)}
    st0: leere Symboltabelle, nirgendwo definiert

  - Funktion *update* besorgt
    - Aufbau einer Symboltabelle gemäß einer Deklaration
    - Belegung des Hauptspeichers

    - *update*: *constDecl varDecl* x Tab x HS -> Tab x HS

      update(`final int c1=z1,...,cm=zm;`
      `int v1,...,vk,vk+1=z1',...,vk+n=zn';`st,h) =
      (st[c1/(const,z1),...,cm/(const,zm),v1/(var,1),...vk+n/(var,k+n)],
      h[1/0,...,k/0,k+1/z1',...,k+n/zn'])

Übersetzung 2

- Übersetzung der Ausdrücke mittels Funktion *exptrans*
  - verwendet Symboltabelle, die von *update* aufgebaut
  - geht induktiv über den Aufbau von Ausdrücken

  - *exptrans* : *{expression}* x Tab -> ZE-Code
  - *exptrans(num,*st) = LIT *num*; entfällt, s.o.
  - *exptrans((exp),*st) = *exptrans (exp,*st)
  - *exptrans(ident,*st) = **if** st(*ident*) **=** (const, z) **then** LIT z;
    if st(*ident*) **=** (var,i) **then** LOD i;
  - *exptrans(exp1+exp2,*st) = *exptrans (exp1,*st)
    *exptrans (exp2,*st) ADD;
  - analog für - : SUB
    * : MULT
    / : DIV

Funktion trans

- ***trans*: Mini-Java-Exp-Progs -> ZE-Code x HS liefert**
  - zu einem Mini-Java-Exp-Programm
  - ein ZE-Programm durch *exptrans* und
  - eine Hauptspeicherbelegung durch *update*
  - *trans*(*constDecl varDecl print(expression);*)
    = (*exptrans(expression,*st),h) ,
    wobei (st, h) = *update(constDecl varDecl,* st0, h0)
    Befehlsmarken werden seriell hinzu gefügt

- **Korrektheit von *trans***

  Für jedes Mini-Java-Exp-Programm P:
  - $\mathbb{M}$(P) = $\mathbb{P}$(*trans*(P))
  - *trans*(P) liefert direkt den Startzustand mit

Beispielübersetzung

- **für Programm P von Folie 10:**

```
final int c1 = 10;          //Konstanten c1,c2,...
int v1, v2 = 5;             //Variablen v1,v2,...
print(((c1+v1)*v2));
```

- ***trans*(P):**
  - Berechnung der Symboltabelle und Hauptspeicherbelegung mittels *update*
    - *update(*`final int c1=10;int v1, v2 = 5;` st0 ,h0) = ([c1/(const,10), v1/(var,1),v2/(var,2)], [1/0,2/5])
  - Berechnung des ZE-Codes mittels *exptrans* unter Verwendung der Symboltabelle st = [c1/(const,10), v1/(var,1),v2/(var,2)]
    - *exptrans*(`((c1+v1)*v2)`, st) = ZE-Programm auf Folie 14
  - Beispielrechnung für obige Hauptspeicherbelegung Folie 14

Übersetzung von Kontrollstrukturen

- **Bisher: Übersetzung von Ausdrücken -> Mini-Java-Exp**
- **Jetzt: Übersetzung von Kontrollstrukturen, d.h. Mini-Java**
- **neu zu betrachten:**
  - Übersetzung von *condition*: `(c1+v1) <= v2`
  - Zuweisung *assignment*: `v1 = 7*c1;`
  - Block *compound statement*: `{v1=7*c1; v2=2*v1;}`
  - Verzweigung *if-statement*: `if c1<v2 v2=2*v1;`
  - Schleife *while-statement*: `while c1<v2 v2=2*v1;`
- **Wiederholung statische Semantik:**
  - alles analog zu Mini-Java-Exp
  - zusätzlich: bei Zuweisung muss links ein Variablenbezeichner stehen *(Symboltabelle)*

Semantikerweiterungen

- ***Condition*semantik**

  $\mathbb{C}$: *{condition}* x Env x Sta -> {true, false}

  - $\mathbb{C}$(*exp1 compOP exp2*,rho,sig) syntaktisch
    = $\mathbb{E}$*(exp1,*rho,sig) ***compOP*** $\mathbb{E}$*(exp2,*rho,sig) **semantisch**

- ***Statement*semantik**

  $\mathbb{S}$: *{statement}* x Env x Sta -> Sta

  - $\mathbb{S}$(*ident=exp;,*rho,sig) = **if** rho(*ident*) **=** addr
    then sig[addr/ $\mathbb{E}$(*exp*,rho,sig)]

  - $\mathbb{S}$(*{stmt1 stmt2 ... stmtn},*rho,sig) =
    $\mathbb{S}$(*stmtn,*rho, $\mathbb{S}$(... $\mathbb{S}$(*stmt2,*rho, $\mathbb{S}$(*stmt1,*rho,sig))...))

  - $\mathbb{S}$(*{}*,rho,sig) = sig

  - $\mathbb{S}$(*print(exp);,*rho,sig) = sig

Semantikerweiterungen 2

- $\mathbb{S}$(*if cond stmt,*rho,sig) =
  if $\mathbb{C}$(*cond*,rho,sig) **= true** **then** $\mathbb{S}$(*stmt,*rho,sig)
  if $\mathbb{C}$(*cond*,rho,sig) **= false** **then** sig

- $\mathbb{S}$(*if cond stmt1 else stmt2,*rho,sig) =
  if $\mathbb{C}$(*cond*,rho,sig) **= true** **then** $\mathbb{S}$(*stmt1,*rho,sig)
  if $\mathbb{C}$(*cond*,rho,sig) **= false** **then** $\mathbb{S}$(*stmt2,*rho,sig)

- $\mathbb{S}$(*while cond stmt,*rho,sig) =
  if $\mathbb{C}$(*cond*,rho,sig) **= true** **then**
  $\mathbb{S}$(*while cond stmt,*rho,$\mathbb{S}$(*stmt,*rho,sig))
  if $\mathbb{C}$(*cond*,rho,sig) **= false** **then** sig

- ***Program*semantik**
  - $\mathbb{M}$: {*program*} -> Sta
    - $\mathbb{M}$(*constDecl varDecl statement)*
      = $\mathbb{S}$*(statement,*$\mathbb{DV}$(*varDecl*,$\mathbb{DC}$(*constDecl*,rho0),sig0))

Beispielrechnung

- **Semantik des Mini-Java-Programms P =**

```
final int c1 = 10;    //Konstanten c1,c2,...
int v1, v2 = 5;       //Variablen v1/0,v2,...
while c1/v2!=1 {      //abgekürzt als statement
   v1=2+v2;
   v2=v2*2;
}
```

$\mathbb{M}$(P)
= $\mathbb{S}$(statement,$\mathbb{DV}$(int v1, v2 = 5;,
$\mathbb{DC}$(final int c1 = 10;,rho0),sig0))
= $\mathbb{S}$(statement,$\mathbb{DV}$(int v1, v2 = 5;,[c1/10],sig0))
= $\mathbb{S}$(statement,[c1/10,v1/add1,v2/add2],[add1/0,add2/5])

Abkürzungen: rho = [c1/10,v1/add1,v2/add2]
sig = [add1/0,add2/5]

Beispielrechnung 2

= 𝕊(`while cond stmt`,rho,sig)

= **if** ℂ(`c1/v2 != 1`, rho, sig) **= true then** ...

= **if** 𝔼(`c1/v2`, rho, sig) **!=** 𝔼(`1`, rho, sig) **= true then** ...

= **if** (𝔼(`c1`, rho, sig) **/** 𝔼(`v2`, rho, sig) **!=** 𝟙) **= true then** ...

= **if** (𝟙𝟘 **/** 𝟝 **!=** 𝟙) **= true then** ...

= **if** (𝟚 **!=** 𝟙) **= true then** ...

= **if true = true then** ...

= 𝕊(`while cond stmt`,rho, 𝕊(`stmt`,rho,sig))

----------Nebenrechnung----------

𝕊(`stmt`,rho,sig) = 𝕊(`v2=v2*2;`,rho, 𝕊(`v1=2+v2;`,rho,sig))

= 𝕊(`v2=v2*2;`,rho, sig[add1/𝔼(`2+v2;`, rho, sig)]) = ...

= 𝕊(`v2=v2*2;`,rho, sig[add1/7]) -> sig'=[add1/7,add2/5]

= sig'[add2/𝔼(`v2*2;`, rho, sig')] = ... -> sig''=[add1/7,add2/10]

Beispielrechnung 3

= $\mathbb{S}$(`while cond stmt`,rho, sig'')

= **if** $\mathbb{C}$(`c1/v2 != 1`, rho, sig'') **= false then** sig''

= **if** $\mathbb{E}$(`c1/v2`, rho, sig'') **!=** $\mathbb{E}$(`1`, rho, sig'') **= false then** sig''

= **if** ($\mathbb{E}$(`c1`, rho, sig'') **/** $\mathbb{E}$(`v2`, rho, sig'') **!=** $\mathbb{1}$) **= false then** sig''

= **if** ($\mathbb{10}$ **/** $\mathbb{10}$ **!=** $\mathbb{1}$) **= false then** sig''

= **if** ($\mathbb{1}$ **!=** $\mathbb{1}$) **= false then** sig''

= **if false = false then** sig''

= sig''

-> $\mathbb{M}$(P) = sig'' = [add1/7,add2/10]

Zwischencode für Mini-Java (Z-Mini-Java)

- **Erweiterung von ZE-Code um**
  - Vergleichsbefehle: RELOP (EQ, NE, LT, GT, LE, GE)
  - Sprungbefehle: JMP n, JMC n (n aus Nat)
  - Speicherbefehl: STO n
  - Datenkellerbefehl: POP

- **Semantik der neuen Befehle**
  - $\mathbb{C}$: Com -> (SM -> SM)//nicht verwechseln mit Conditionsemantik!
    - $\mathbb{C}$ (RELOP) (m, z1z2:d', h) =
      if z2 **RELOP** z1 **= true then** (m+1, 1:d', h)
      if z2 **RELOP** z1 **= false then** (m+1, 0:d', h)
    - $\mathbb{C}$ (JMP n) (m, d, h) = (n, d, h)
    - $\mathbb{C}$ (JMC n) (m, b:d', h) = **if** b **= 0 then** (n, d', h)
      if b **= 1 then** (m+1, d', h)
    - $\mathbb{C}$ (STO n) (m, z:d', h) = (m+1, d', h[n/z])
    - $\mathbb{C}$ (POP) (m, z:d', h) = (m+1, d', h)

Z-Mini-Java-Code

- **Verwendung symbolischer Adressen**
  - wegen Sprungbefehlen kompliziertere Bestimmung der Befehlsadressen
  - Adressen haben Form: **an** n: natürliche Zahl
  - gehen davon aus, dass es Adresspool gibt, der immer die nächste Adresse **anext** liefert und sich darum kümmert, dass keine Adresse zweimal vergeben wird.
  - Befehle können mehrere Adressen haben
- **Semantik von P**
  - $\mathbb{P}$: Progs x SM -> SM
  - erfordert zunächst Ersetzen der symbolischen Adressen durch 1,2,...,p vor Befehlen und in Befehlen.
  - Dann Iterieren von $\mathbb{C}$ bis Befehlszähler m > p

Erweiterung der Übersetzung

- ***trans*: Mini-Java-Progs -> Z-Mini-Java-Code x HS liefert**
  - *trans*(*constDecl varDecl statement)*
    = (*cmdtrans*(*statement,*st),h) ,
    wobei (st, h) = *update(constDecl varDecl,* st0, h0)
- **Übersetzung von Condition mittels Funktion *condtrans***
  - *condtrans* : *{condition}* x Tab -> Z-Mini-Java-Code
  - *condtrans(exp1 compOP exp2,* st) = *exptrans (exp1,*st)
    *exptrans (exp2,*st) RELOP;
- **Übersetzung von Statements mittels Funktion *cmdtrans***
  - *cmdtrans(ident=exp;,*st) = **if** st(*ident*) **=** (var,i)
    then *exptrans(exp,*st) STO i;
  - *cmdtrans({stmt1 stmt2 ... stmtn},*st) = *cmdtrans*(*stmt1,*st)
    cmdtrans(*stmt2,*st)
    ...
    cmdtrans(*stmtn,*st)

Erweiterung der Übersetzung 2

- *cmdtrans*(*if cond stmt,*st) = *condtrans(cond,*st)
  JMC anext;
  cmdtrans(*stmt,*st)
  anext:

- *cmdtrans*(*if cond stmt1 else stmt2,*st) = *condtrans(cond,*st)
  JMC anext;
  cmdtrans(*stmt1,*st)
  JMP a(next+1);
  anext: *cmdtrans*(*stmt2,*st)
  a(next+1):

- *cmdtrans*(*while cond stmt,*st) = anext: *condtrans(cond,*st)
  JMC a(next+1);
  cmdtrans(*stmt,*st)
  JMP anext;
  a(next+1):

- *cmdtrans*(*print(exp);,*st) = *exptrans*(*exp*,st) POP;

Beispielübersetzung

- Mini-Java-Programm:

```
final int c1 = 10;              //Konstanten c1,c2,...
int v1, v2 = 5;                 //Variablen v1,v2,...
while c1/v2!=1 {
  v1=2+v2;
  v2=v2*2;
}
```

- ***trans*(P):**
  - Berechnung der Symboltabelle und Hauptspeicherbelegung mittels *update*
    - *update(*`final int c1=10;int v1, v2 = 5;` st0 ,h0) = ([c1/(const,10), v1/(var,1),v2/(var,2)], [1/0,2/5])

Abkürzungen: st = [c1/(const,10), v1/(var,1),v2/(var,2)]

h = [1/0,2/5]

Beispielübersetzung 2

– Berechnung des Z-Mini-Java-Codes mittels *cmdtrans* unter Verwendung der Symboltabelle st

cmdtrans(`while cond stmt`, st)

= a1: *condtrans*(`c1/v2 != 1`, st)
JMC a2;
cmdtrans(`{v1=2+v2; v2=v2*2;}`, st)
JMP a1;
a2:

= a1: *condtrans*(`c1/v2 != 1`, st)
JMC a2;
cmdtrans(`v1=2+v2;`, st)
cmdtrans(`v2=v2*2;`, st)
JMP a1;
a2:

Beispielübersetzung 3

condtrans(`c1/v2 != 1`, st)

= *exptrans*(`c1/v2`, st)
   exptrans(`1`, st)
   NE;

= *exptrans*(`c1`, st)
   exptrans(`v2`, st)
   DIV;
   LIT 1;
   NE;

= LIT 10;
   LOD 2;
   DIV;
   LIT 1;
   NE;

Beispielübersetzung 4

cmdtrans(`v1=2+v2;` , st)

= *exptrans*(`2+v2`, st)
   STO 1;

= *exptrans*(`2`, st)
   exptrans(`v2`, st)
   ADD;
   STO 1;

= LIT 2;
   LOD 2;
   ADD;
   STO 1;

cmdtrans(`v2=v2*2;` , st)

= *exptrans*(`v2*2`, st)
   STO 2;

= *exptrans*(`v2`, st)
   exptrans(`2`, st)
   MULT;
   STO 2;

= LOD 2;
   LIT 2;
   MULT;
   STO 2;

Beispielübersetzung 5

Einsetzen ergibt:

```
a1:     LIT 10;
        LOD 2;
        DIV;
        LIT 1;
        NE;
        JMC a2;
        LIT 2;
        LOD 2;
        ADD;
        STO 1;
        LOD 2;
        LIT 2;
        MULT;
        STO 2;
        JMP a1;
a2:
```

- **Ersetzen der Adressen:**

```
1:      LIT 10;
2:      LOD 2;
3:      DIV;
4:      LIT 1;
5:      NE;
6:      JMC 16;
7:      LIT 2;
8:      LOD 2;
9:      ADD;
10:     STO 1;
11:     LOD 2;
12:     LIT 2;
13:     MULT;
14:     STO 2;
15:     JMP 1;
16:
```

Z-Mini-Java-Beispielrechnung

Beispielrechnung für Hauptspeicherbelegung h = [1/0,2/5]:

```
->      (1,          , [1/0, 2/5])      // LIT 10;
->      (2, 10       , [1/0, 2/5])      // LOD 2;
->      (3, 5 10     , [1/0, 2/5])      // DIV;
->      (4, 2        , [1/0, 2/5])      // LIT 1;
->      (5, 1 2      , [1/0, 2/5])      // NE;
->      (6, 1        , [1/0, 2/5])      // JMC 16;
->      (7,          , [1/0, 2/5])      // LIT 2;
->      (8, 2        , [1/0, 2/5])      // LOD2;
->      (9, 5 2      , [1/0, 2/5])      // ADD;
->      (10, 7       , [1/0, 2/5])      // STO 1;
->      (11,         , [1/7, 2/5])      // LOD 2;
->      (12, 5       , [1/7, 2/5])      // LIT 2;
->      (13, 2 5     , [1/7, 2/5])      // MULT;
```

Z-Mini-Java-Beispielrechnung 2

```
->      (14, 10      , [1/7, 2/5])      // STO 2;
->      (15,         , [1/7, 2/10])     // JMP 1;
->      (1,          , [1/7, 2/10])     // LIT 10;
->      (2, 10       , [1/7, 2/10])     // LOD 2;
->      (3, 10 10    , [1/7, 2/10])     // DIV;
->      (4, 1        , [1/7, 2/10])     // LIT 1;
->      (5, 1 1      , [1/7, 2/10])     // NE;
->      (6, 0        , [1/7, 2/10])     // JMC 16;
->      (16,         , [1/7, 2/10])     // STOP, da kein Befehl!!!
```

Korrektheit

- **nicht einfach Gleichheit**
- ***Programm*semantik: Zustand**
- **Maschinenprogrammsemantik: Maschinenzustand**
- **Zustand und Hauptspeicherinhalt müssen übereinstimmen:**
  - $\mathbb{M}$(P) = [add1/z1,...,addm/zm]
  - $\mathbb{P}$(trans(P),Startzustand) = (bz, ,[1/z1,...,m/zm]) mit
    - bz > p
    - Keller ist leer
    - Hauptspeicher stimmt mit Zustand der Semantik überein
  - Startzustand mittels *update*