

## 9. Fallstudie: Compiler-Verifikation

- Formalisierung einer einfachen, imperativen Programmiersprache
- mit Ausdrücken, einfachen Anweisungen einschließlich `while`-Schleifen
- Übersetzung in Maschinensprache für eine abstrakte Stack-Maschine
- Nachweis der Korrektheit der Übersetzung

# Übersicht

## Ausdrücke

### 1. Quellsprache

- Syntax
- Semantik

### 2. Maschinenebene

- Maschinenmodell
- Syntax der Zielsprache
- Semantik

### 3. Übersetzung

- Übersetzungsfunktion
- Korrektheitsbegriff
- (Beweis)

## Anweisungen

### 1. Quellsprache

- Syntax
- Semantik

### 2. Maschinenebene

- Maschinenmodell
- Syntax der Zielsprache
- Semantik

### 3. Übersetzung

- Übersetzungsfunktion
- Korrektheitsbegriff
- (Beweis)

# Quellsprache: Syntax der Ausdrücke

Die Quellsprache besitze Ausdrücke der folgenden Art:

- Konstanten: bezeichnen einen gewissen Wert; z. B. Zahlen
- Variablen: gekennzeichnet durch einen Bezeichner
- Operationen:
  - einstellige Operationen: z. B. Negation
  - binäre Operationen: z. B. Addition

Mögliche Facetten:

- Boolesche Werte und Operationen
- Zahlen und arithmetische Operationen
- Zeichen und Zeichenketten mit entsprechenden Operationen

Konkrete Ausprägung ist für Zielsetzung der Vorlesung nicht relevant, es wird von konkreten Wertmengen und Operationen abstrahiert.

# Abstrakte Werte

Wir abstrahieren von konkreten Werten und Operationen auf Werten:

Value : TYPE+;    Unop, BinOp : TYPE+

Bedeutung der Operatoren wird gegeben durch eine *Semantik-Funktion* (“Denotation”, s. später):

$[| |] : [\text{Unop} \rightarrow [\text{Value} \rightarrow \text{Value}]]$

$[| |] : [\text{Binop} \rightarrow [\text{Value}, \text{Value} \rightarrow \text{Value}]]$

Bem.: PVS erlaubt „mix-fix“-Anwendung:  $[| \circ |](v1, v2)$

Variablennamen bleiben ebenfalls abstrakt:

VarId : TYPE+

Diese Deklarationen sind eigentlich jeweils Theorie-Parameter.

# Syntax von Ausdrücken

Syntax wird nur *abstrakt* behandelt, d.h. nach Parsing, Typ-Überprüfung usw.; Aspekte der “statischen Semantik” werden als bereits abgehandelt angenommen.

~> nur “sinnvolle” Programme werden betrachtet

Modellierung der Ausdruckssyntax als induktiver Datentyp:

```
Expr : DATATYPE
```

```
BEGIN
```

```
  const (val : Value) : const?
```

```
  varid (name : VarId) : varid?
```

```
  unopr (unop : Unop, arg : Expr) : unopr?
```

```
  binopr(binop: Binop, left,right: Expr) : binopr?
```

```
END Expr
```

# Semantik der Ausdrücke

Nächster Schritt: Definition der Semantik

- Denotationelle Semantik: *was* ist der Wert eines Ausdrucks?
- Operationelle Semantik: *wie* wird ein Ausdruck ausgewertet?

Hier: denotationelle Semantik

Wovon hängt der Wert eines Ausdrucks ab?

- Was ist z. B. der Wert von  $5 + v_1$ ?

Aktueller Wert der Variablen ist maßgebend!

- Zustand: Belegung der Variablen mit Werten

State : TYPE = [VarId -> Value]

# Denotationelle Semantik der Ausdrücke

Wert eines Ausdrucks relativ zu einem Zustand  $\sigma$ :

- Konstanten:  $\llbracket c \rrbracket_\sigma ::= c$
- Variablen:  $\llbracket v \rrbracket_\sigma ::= \sigma(v)$
- unäre Operatoren:  $\llbracket \sim e \rrbracket_\sigma ::= \llbracket \sim \rrbracket_\sigma(\llbracket e \rrbracket_\sigma)$
- binäre Operatoren:  $\llbracket e_1 \# e_2 \rrbracket_\sigma ::= \llbracket \# \rrbracket_\sigma(\llbracket e_1 \rrbracket_\sigma, \llbracket e_2 \rrbracket_\sigma)$

In PVS:

```
[[|]](e:Expr)(s:State) : RECURSIVE Value =
  CASES e OF
    const(val)           : val,
    varid(name)          : s(name),
    unopr(op,arg)        : [| op |]( [| arg |](s)),
    binopr(op,left,right) : [| op |]( [| left |](s),
                                     [| right |](s))
  ENDCASES MEASURE e BY <<
```

# Zielmaschine

## Auswertung von Ausdrücken

- auf einer Stack-Maschine:

```
T : TYPE
Stack : DATATYPE
BEGIN
  empty : empty?
  push(top: T, pop: Stack) : nonempty?
END Stack
```

- mit adressierbarem Speicher:

```
Addr : TYPE
Mem : TYPE = [Addr -> Value]
```

- Maschinenzustand:

```
MachineState : TYPE+ = [# stack: Stack, mem: Mem #]
```

# Maschinensprache

Maschinenbefehle:

- Laden eines konstanten Werts auf den Stack: lit
- Laden eines Werts aus dem Speicher: load
- Unäre und binäre Operationen: unop und binop

In PVS:

```
Instr : DATATYPE
BEGIN
  LIT(z: Value)      : lit?
  LOAD(la: Addr)     : load?
  UNOP(uop:Unop)     : unop?
  BINOP(bop: Binop) : binop?
END Instr
```

# Semantik der Maschinenbefehle

## Operationelle Semantik der Maschinsprache

- Semantikfunktion beschreibt, wie sich der Maschinenzustand ändert
- Zustandsübergangsfunktion
- Laden eines konstanten Werts auf den Stack:  $\text{lit}(c)$

$s \mapsto s'$ , wobei  $\text{mem}' = \text{mem}$  und  $\text{stack}' = \text{push}(c, \text{stack})$

```
litf(c:Value) : [MachineState -> MachineState] =  
  LAMBDA s: s WITH [(stack) := push(c,stack(s))]
```

- Laden eines Werts aus dem Speicher:  $\text{load}(a)$

```
loadf (a:Addr) : [MachineState -> MachineState] =  
  LAMBDA s: s WITH [(stack) := push(mem(s)(a),stack(s))]
```

## Semantik der Maschinenbefehle (2)

Unäre Operation:  $\text{unop}(\text{op})$

- Speicher bleibt unverändert:  $\text{mem}' = \text{mem}$
- Operator wird auf oberstes Stackelement angewandt . . .
- . . . dieses wird durch Ergebnis der Operation ersetzt:

$$\text{stack}' = \text{push}(\text{op}(\text{top}(\text{stack})), \text{pop}(\text{stack}))$$

Wo liegt das Problem? Stack darf nicht leer sein!

Was, wenn doch?

Generelle Lösungsmöglichkeiten:

- Einschränken des Definitionsbereichs auf *Subtyp*
- Erweitern des Bildbereichs um ein error-Element

Vorteile – Nachteile?

# Darstellung der Maschinensemantik

- Semantik der Maschinenbefehle ist nur *partiell* definiert
- (eine) Lösung: Semantik wird relational beschrieben
- Variante: Funktion, die auf eine *Menge* abbildet

A,B : TYPE

Relation : TYPE = [A -> set[B]]

- „error“-Element: leere Menge
- Bildbereich der Semantikfunktion: höchstens ein Folgezustand

R : VAR Relation;    S : VAR set[B]

unique?(S) : bool =

FORALL x,y: S(x) AND S(y) IMPLIES x = y

PartialFunction : TYPE = [A -> (unique?)]

# Semantik der Maschinenbefehle (3)

Unäre Operatoren:

```
uopf(uop:Unop) : PartialFunction =  
  LAMBDA s:  
    IF nonempty?(stack(s)) THEN  
      singleton(s WITH [  
        (stack) := push([| uop |](top(stack(s))),  
          pop(stack(s))))])  
    ELSE emptyset  
  ENDIF
```

# Semantik der Maschinenbefehle (4)

Binäre Operatoren:

```
bopf(bop:Binop) : PartialFunction =  
  LAMBDA s:  
    IF twotops?(stack(s)) THEN  
      singleton(s WITH [  
        (stack) := push([| bop |](top(pop(stack(s))),  
                          top(stack(s))),  
                          pop(pop(stack(s))))])  
      ])  
    ELSE emptyset  
  ENDIF
```

Typ der Semantikfunktionen `litf` und `loadf` wird entsprechend angepasst.

# Ein-Schritt-Interpretation

Zusammenfassung der Semantikfunktionen zu einem Ein-Schritt-Interpreter der Maschinenbefehle:

```
onestep(i:Instr) : PartialFunction =  
  CASES i OF  
    LIT(v)      : litf(v),  
    LOAD(a)     : loadf(a),  
    UNOP(op)    : uopf(op),  
    BINOP(op)   : bopf(op)  
  ENDCASES
```

# Vom Maschinenbefehl zum Maschinenprogramm

- Gegeben:
- Maschineninstruktionen
  - Struktur des Maschinenzustands
  - Semantik der Maschineninstruktionen

```
simple_interpreter
  [Instr  : TYPE,
   MState : TYPE+,
   effect : [Instr -> PartialFunction[MState,MState]]
  ] : THEORY
BEGIN
  ...
```

Maschinenprogramm ist eine Folge von Instruktionen

```
Code : TYPE = list[Instr]
++(l,k:Code) : Code = append(l,k)
```

# Interpretation von Maschinenprogrammen

Hintereinanderausführung der einzelnen Zustandsübergänge

- einfach, wenn „richtige“ Zustandsübergangsfunktion

```
interpret(c)(s:MState) : RECURSIVE MState =  
  IF null?(c) THEN s  
    ELSE interpret(cdr(c))(effect(car(c))(s))  
  ENDIF  
  MEASURE length(c)
```

Problem:  $\text{effect}(\text{car}(c))(s)$  liefert eine *Menge* von Zuständen

# Abstrakte Theorie für Zustandsrelationen

```
srel [State : TYPE] : THEORY
BEGIN
  IMPORTING relation[State,State]
  srel : TYPE = Relation
  s    : VAR State
  f,g  : VAR srel
  skip    : srel = LAMBDA s: singleton(s);
  ++(f, g) : srel = LAMBDA s: image(g)(f(s))
  % hier ist:
  % image(R)(S) : set[B] =
  % {y:B | EXISTS (s:(S)): member(y,R(s))}
END srel
```

# Interpretation von Maschinenprogrammen

Hintereinanderausführung der einzelnen Zustandsübergänge

```
interpret(c) : RECURSIVE srel =  
  IF null?(c) THEN skip  
    ELSE effect(car(c)) ++ interpret(cdr(c))  
  ENDIF  
  MEASURE length(c)
```

nützliche Eigenschaft:

```
interpret_split : LEMMA  
  interpret(l ++ k) = interpret(l) ++ interpret(k);
```

Etwas fürs Auge:  $\text{interpret}(c)(\text{start}) \gg \text{final}$

```
>>(S:set[MState],s:MState) : bool = S(s)
```

# Übersetzung von Ausdrücken

Compilierungsfunktion `compile`

- Konstanten:

$$\text{compile}(c) \mapsto \text{LIT}(c)$$

- Variablen:

$$\text{compile}(v) \mapsto \text{LOAD}(a)$$

- $a$  ist eine Speicheradresse, in der  $v$  abgelegt ist
- Abbildung von Variablen auf Adressen nötig: *idmap*

- Unäre Operatoren:

$$\text{compile}(op(e)) \mapsto \text{compile}(e) \text{ ++ UNOP}(op)$$

- Binäre Operatoren:

$$\text{compile}(op(e_1, e_2)) \mapsto \text{compile}(e_1) \text{ ++ compile}(e_2) \text{ ++ BINOP}(op)$$

# Übersetzung von Ausdrücken (2)

In PVS:

```
idmap : [VarId -> Addr]
```

```
compile(e:Expr) : RECURSIVE Code =
```

```
  CASES e OF
```

```
    const(val)      : (: LIT(val)::Instr :),
```

```
    varid(name)     : (: LOAD(idmap(name))::Instr :),
```

```
    unopr(op,a)     : compile(a) ++ (: UNOP(op)::Instr :),
```

```
    binopr(op,l,r) : compile(l) ++ compile(r) ++ (: BINOP(op)::Instr :)
```

```
  ENDCASES
```

```
  MEASURE e BY <<
```

(: *instr* :) ist PVS-Kurzschreibweise für `cons(instr,null)`

# Korrekte Übersetzung

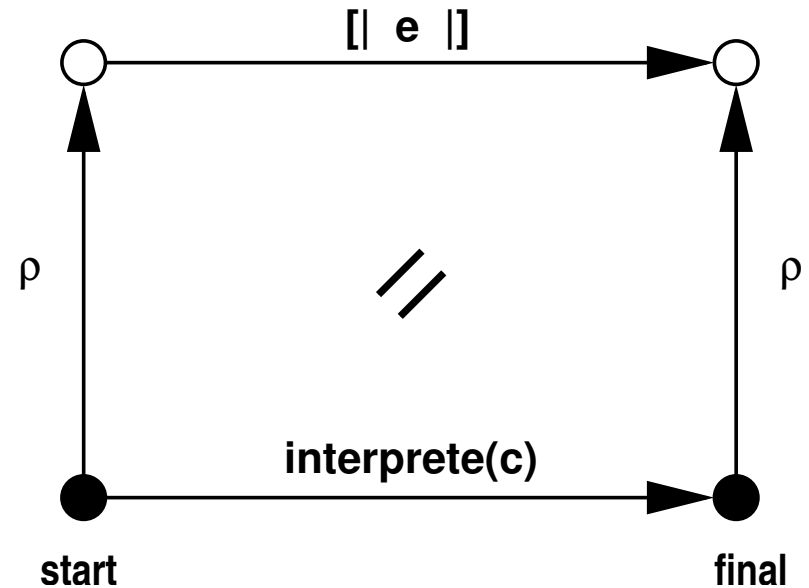
- Was soll *Korrektheit* bedeuten?
  - Maschine soll Wert des Ausdrucks berechnen
  - Interpretation des Maschinenprogramms liefert Wert des Ausdrucks als oberstes Element des Stacks
- *Aber*: Wert des Ausdrucks abhängig von Variablenbelegung
  - Speicher im Startzustand muss Variablenbelegung „entsprechen“
  - Zusammenhang Maschinenzustand und Programmzustand:

```
statemap(ms: MaschineState) : State =  
  LAMBDA (v: VarId): mem(ms)(idmap(v))
```
- Außerdem:
  - *keine Seiteneffekte!*
  - Speicher bleibt unverändert

# Korrektheitsbegriff

Korrekte Übersetzung formal:

```
correct(e:Expr,c:Code) : bool =  
  FORALL (start,final:MachineState):  
    interpret(c)(start) >> final =>  
      nonempty?(stack(final)) AND  
      [| e |](statemap(start))  
      =  
        top(stack(final))  
      AND statemap(final)  
      =  
        statemap(start)
```



Bem.: Diese Definition erlaubt Seiteneffekte auf den Speicher – aber nur, wenn dies auch in der denotationellen Semantik passiert.

# Korrektheitsbeweis

Korrektheitstheorem:

```
correctness : THEOREM
  correct(e, compile(e))
```

Hilfreiches Lemma:

```
interprete_invariant : LEMMA
  interprete(compile(e))(start) >> final IMPLIES
    EXISTS (v:Value): stack(final) = push(v, stack(start))
```

# Quellsprache: Syntax der Anweisungen

Arten von Anweisungen in der Quellsprache:

- leere Anweisung: `skip`
- Zuweisung eines Werts an eine Variable: `x := e`
- Hintereinanderausführung von Anweisungen: `c1; c2`
- bedingte Verzweigung: `if b then c1 else c2`
- Wiederholung (Schleife): `while b do c`

Es werden *Boolesche* Ausdrücke benötigt:

```
trueV    : Value
BExpr    : TYPE FROM Expr
[| |](b:BExpr)(s:State) : bool =
  [| b |](s) = trueV
```

# Syntax von Anweisungen

Abstrakte Syntax der Anweisungen als induktiver Datentyp:

```
Command : DATATYPE
BEGIN
  skip : skip?
  seq(first,second:Command) :seq?
  assign(varid:VarId, exp:Expr) : assign?
  if_(ifcond: BExpr, thn, els: Command) : if?
  while(whilecond: BExpr, body: Command) : while?
END Command
```

# Denotationelle Semantik der Anweisungen

- Was ist die Bedeutung einer Anweisung?

*Transformation des Zustands*

- Erinnerung: der Zustand wird durch die Variablenbelegung gebildet:

State : TYPE = [VarId -> Value]

- Was bedeutet dann Zustandsänderung?

*Änderung von Werten von Variablen*

update(s: State)(x: VarId, v: Value) : State =  
s WITH [(x) := v]

# Semantik-Funktionale (1)

- leere Anweisung:  $\llbracket \text{skip} \rrbracket_\sigma ::= \text{skip}$

```
skip : PartialFunction = LAMBDA s: singleton(s);
```

- Zuweisung:  $\llbracket x := e \rrbracket_\sigma ::= \{s[x \leftarrow \llbracket e \rrbracket_\sigma]\}$

```
e : VAR [State -> Value]
```

```
<<(x,e) : PartialFunction =
```

```
LAMBDA s: singleton(update(s)(x,e(s)))
```

- Hintereinanderausführung:  $\llbracket c_1; c_2 \rrbracket_\sigma ::= \llbracket c_1 \rrbracket_\sigma ++ \llbracket c_2 \rrbracket_\sigma$

```
f,g : VAR PartialFunction
```

```
++(f, g) : PartialFunction = LAMBDA s: image(g)(f(s))
```

## Semantik-Funktionale (2)

- bedingte Verzeigung:

$$\llbracket \text{if } b \text{ then } c_1 \text{ else } c_2 \rrbracket_{\sigma} ::= \begin{cases} \llbracket c_1 \rrbracket_{\sigma} & \text{falls } \llbracket b \rrbracket_{\sigma} = \text{true} \\ \llbracket c_2 \rrbracket_{\sigma} & \text{sonst} \end{cases}$$

`b` : VAR [State -> bool]

IF(b, f, g) : PartialFunction =

LAMBDA s: IF b(s) THEN f(s) ELSE g(s) ENDIF

Bem.: IF wird hier in 2 unterschiedlichen Weisen benutzt:

1. als Bezeichner der neu definierten Funktion und
2. als Schlüsselwort aus der PVS-Sprache.

Die neue Funktion IF kann in IF-THEN-ELSE-ENDIF-Schreibweise verwendet werden.

## Semantik-Funktionale (3)

- Wiederholungsanweisung:

$$\llbracket \text{while } b \text{ do } c \rrbracket_{\sigma} ::= \begin{cases} \llbracket c \rrbracket_{\sigma} ++ \llbracket \text{while } b \text{ do } c \rrbracket_{\sigma} & \text{falls } \llbracket b \rrbracket_{\sigma} = \text{true} V \\ \text{skip} & \text{sonst} \end{cases}$$

- Problem: Das ist eine *rekursive* Definition!

- terminiert die Funktion?
- wenn ja, welches Maß ist zum Nachweis zu verwenden?

- Semantik ist der (kleinste) Fixpunkt einer Funktionstransformation

```
while(b,f) : PartialFunction =  
  mu! (x:PartialFunction):  
    IF b THEN f ++ x ELSE skip ENDIF
```

# Denotationelle Semantik der Anweisungen (2)

Semantik-Definition in PVS:

```
[[|]](c) : RECURSIVE PartialFunction =
  CASES c OF
    skip      : skip,
    seq(c1,c2) : [| c1 |] ++ [| c2 |],
    assign(x,e) : x << [| e |],
    if_(b,ct,cf) : IF [| b |] THEN [| ct |]
                  ELSE [| cf |] ENDIF,
    while(b,cb) : while([| b |],[| cb |])
  ENDCASES
  MEASURE c BY <<
```

Pretty-Printing:  $[[c]]_{\sigma} \gg s$

```
>>(S:set[State],s:State) : bool = S(s)
```

# Übersetzung von Anweisungen: Maschinenmodell

## Erweiterung der Stack-Maschine

- IF-Verzweigung und `while`-Schleife haben keine lineare Struktur
- Sprünge im Maschinencode: entsprechende Instruktionen nötig
- Maschine benötigt *Programmzähler*

```
IMPORTING stack[Value]
```

```
Mem : TYPE = [Addr -> Value]
```

```
MachineState : TYPE+ =
```

```
  [# PC : int, stack: Stack, mem: Mem #]
```

# Erweiterung der Maschinensprache

Zusätzliche Maschinenbefehle:

- Speichern eines Werts vom Stack in den Speicher: store(a)
- Sprung um eine Anzahl von Instruktionen: jmp(k)
- bedingter Sprung: jmc(k)

In PVS:

```
Instr : DATATYPE
BEGIN
  % LIT, LOAD, UNOP, BINUP wie bisher
  STORE(a:Addr)      : store?
  JMP(k:int)         : jmp?
  JMC(k:int)         : jmc?    % --- jump on FALSE
END Instr
```

# Semantik der Maschinenbefehle (1)

Semantik für lit, load, unop und binop wird um PC ergänzt.

```
litf(v:Value) : PartialFunction =  
  LAMBDA s: singleton(  
    s WITH [(PC)      := PC(s) + 1,  
            (stack) := push(v,stack(s))])
```

## Semantik der Maschinenbefehle (2)

Speichern eines Werts vom Stack in den Speicher: store(a)

```
storef(a:Addr) : PartialFunction =  
  LAMBDA s:  
    IF nonempty?(stack(s)) THEN  
      singleton(s WITH [  
        (PC)      := PC(s) + 1,  
        (stack)   := pop(stack(s)),  
        (mem)     := mem(s) WITH [(a) := top(stack(s))])  
      ]  
    ELSE emptyset  
    ENDIF
```

## Semantik der Maschinenbefehle (3)

Sprung um eine Anzahl von Instruktionen:  $\text{jmp}(k)$

```
jmpf(k:int) : PartialFunction =  
  LAMBDA s: singleton(s WITH [(PC) := PC(s) + k])
```

## Semantik der Maschinenbefehle (4)

bedingter Sprung: `jmc(k)`

```
jmcf(k:int) : PartialFunction =  
  LAMBDA s:  
    IF nonempty?(stack(s)) THEN  
      IF top(stack(s)) = trueV THEN  
        singleton(s WITH [(PC)      := PC(s) + 1,  
                          (stack) := pop(stack(s))])  
      ELSE  
        singleton(s WITH [(PC)      := PC(s) + 1 + k,  
                          (stack) := pop(stack(s))])  
      ENDIF  
    ELSE emptyset  
  ENDIF
```

Sprung erfolgt, wenn das oberste Stack-Element nicht den Wert `trueV` besitzt.

# Semantik der Maschinenbefehle (5)

Zusammenfassung der Effekte der einzelnen Instruktionen:

```
effect(i:Instr) : PartialFunction =  
  CASES i OF  
    LIT(v)      : litf(v),  
    LOAD(a)     : loadf(a),  
    UNOP(op)    : uopf(op),  
    BINOP(op)   : bopf(op),  
    STORE(a)    : storef(a),  
    JMP(k)      : jmpf(k),  
    JMC(k)      : jmcf(k)  
  ENDCASES
```

# Ein-Schritt-Interpretation

- Aktuelle Instruktion soll ausgeführt werden
- Wert des Programmzählers bestimmt aktuelle Instruktion
- Kein Folgezustand, wenn PC außerhalb des Programmbereichs zeigt

```
onestep(c) : PartialFunction =  
  LAMBDA s:  
    IF PC(s) < 0 OR PC(s) > length(c)-1  
      THEN emptyset [MachineState]  
      ELSE effect(nth(c,PC(s)))(s)  
    ENDIF
```

# Interpretation des Maschinenprogramms

- Maschinenprogramm kann Sprünge enthalten
- Schleifen sind möglich, Termination nicht unbedingt gegeben
- Lösung auch hier: Maschinensemantik ist *Fixpunkt* einer Funktionstransformation

```
interprete(c) : PartialFunction =  
  mu! (X: PartialFunction): onestep(c) == X;
```

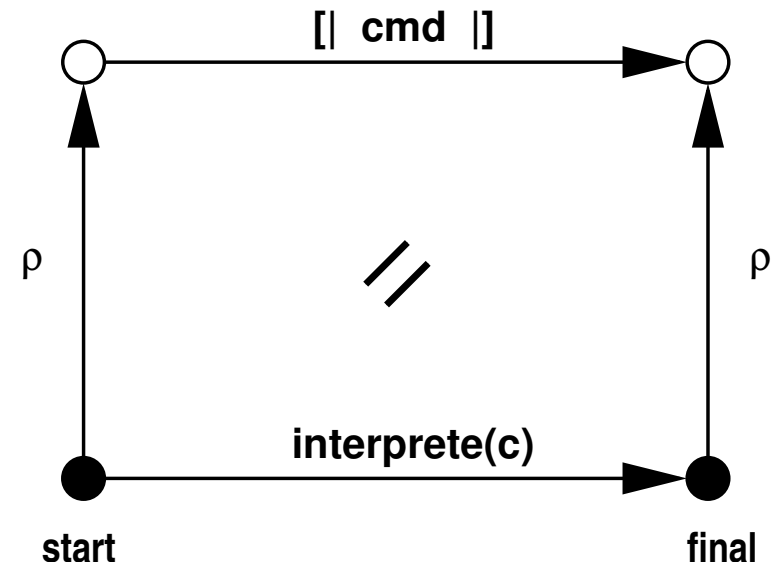
# Übersetzung von Anweisungen (2)

```
compile(c:Command) : RECURSIVE Code =
  CASES c OF
    skip          : (: :),
    seq(c1,c2)    : compile(c1) ++ compile(c2),
    assign(x,e)   : compile(e) ++ (: STORE(idmap(x)) :),
    if_(b,c1,c2) : LET t = compile(c1), e = compile(c2)
                  IN compile(b)
                    ++ (: JMC(length(t)+1) :)
                    ++ t ++ (: JMP(length(e)) :)
                    ++ e,
    while(b,c)    : LET e = compile(b), r = compile(c) IN
                  e ++ (: JMC(length(r)+1) :) ++
                  r ++
                  (: JMP(-(length(r)+length(e)+2)) :)
  ENDCASES
  MEASURE c BY <<
```

# Korrektheitsbegriff

Ausführung des Maschinenprogramms erhält die Semantik des Quellprogramms

```
correct(cmd:Command,c:Code) : bool =  
  FORALL (start,final:MachineState):  
    interprete(c)(start) >> final  
      IFF  
    [| cmd |](statemap(start))  
      >> statemap(final)
```



# Korrekte Übersetzung

Korrektheitstheorem:

```
correctness_compile_command : THEOREM
  correct(cmd, compile(cmd))
```

Beweis-Idee:

- *Basic blocks*: Sequenzen ohne Sprünge
- Korrektheit der Ausführung von *basic-block*-Graphen
- Linearisierung
- jeweils: mittels Fixpunktinduktion, bzw. Parks Lemma

# Zusammenfassung

Die wichtigsten Stichworte:

- Syntax der Quell- und Zielsprache: DATATYPE
- Semantik: denotationell – operationell
  - Ausdrücke: Werte relativ zu einer Variablenbelegung
  - Anweisungen: Zustandstransformationen
  - partielle Funktionen, Fixpunkte
  - Interpreter für Maschineninstruktionen
- Korrektheitsbegriff: Verfeinerungsbeziehung
- Fixpunkt-Induktion für Beweise nötig