Übersetzung von Mini-Java-FunProc

- **bisher: Übersetzung von**
  - Mini-Java-Exp: Datenkeller zur Auswertung arithm. Ausdrücke
  - Mini-Java: Sprünge zur Simulation von Kontrollstrukturen
- **jetzt:**
  - Mini-Java-FunProc:
    - Erweiterung von Mini-Java um
      - Funktionen: liefern Rückgabewert
      - Prozeduren: strukturierte Programmentwicklung
    - erlaubt rekursive Programmierung:
      Bsp.:
      - Fibonacci
      - Fakultät
      - Ackermann-Funktion

} • **einfache Umsetzung in Iteration**

• **nicht so einfach in Assembler!**

Ackermann-Funktion

- meisten Funktionen sind primitiv-rekursiv
  -> einfache Transformation in iterative Lösung ohne Stack
- **große Ackermann-Funktion 1928:**
  - nicht primitiv-rekursiv (wächst schneller)
  - nicht so einfach zu transformieren, insb. in Assemblerprogramm
  - Def.: ack: Nat x Nat -> Nat

    ack (a,0) = 1, für alle nat. Zahlen a >= 0

    ack (0,1) = 2

    ack (0, b+2) = b + 4, für alle nat. Zahlen b >= 0

    ack (a+1,b+1) = ack (a,ack(a+1,b)), a,b >= 0
- **Übung:** in Assembler ziemlich schwierig
  darum: **Mini-Java-FunProc**
  direkte Umsetzung der rekursiven Definition

Prozeduren & Funktionen in PS

- **Fortran**
  - keine Rekursion & Unterprogramme nicht geschachtelt

  -> statische Speicherverwaltung,
  Speicherbedarf zur Compilezeit bekannt

- **C, Mini-Java-FunProc** **hier**
  - Rekursion & Prozeduren und Funktionen nicht geschachtelt

  -> dynamische Speicherverwaltung
  Speicherbedarf erst zur Laufzeit bekannt
  Prozedurkeller ohne statische Verweise

- **Pascal (Algol 60)**
  - Rekursion & Prozeduren und Funktionen geschachtelt

  -> dynamische Speicherverwaltung
  Prozedurkeller mit statischen Verweisen

Syntax von Mini-Java-FunProc

- **vollständige Syntax:**
  - siehe: ..\Sonstiges\miniJavaFunProc Syntax.pdf
- **Erweiterung der bisherigen Syntax um:**
  - neue Schlüsselwörter: *func, void, return*
  - in *program*: Prozeduren- & Funktionsdeklarationen
  - neu *procedure*: Prozedurdeklaration mit Par.-Liste & Block
  - neu *function*: Funktionsdeklaration mit Par.-Liste & Block & abschließendem return-Befehl
  - neu *routinenParameter*: Parameterliste ohne Klammern
  - neu *routinenBlock*: Block von Prozeduren & Funktionen mit Konstanten- und Variablendeklarationen & Anweisung
  - in *term*: Funktionsaufruf als Ausdruck
  - in *statement*: Prozeduraufruf, parallel zur Zuweisung

Bsp.: Mini-Java-FunProc-Programm

siehe Übungsaufgabe 49 oder:

```
int n = 10;

func a(){
      int m = 30;
      n = m - n;
      return(n);
}

func b(int n1, int n2){
      int n = 4;
      n = n1 + n2;
      return(n + a());
}

print(b(n+2,3));
```

Was kommt dabei raus?

Welches n wird bei Aufruf von a() verwendet?

n aus Hauptprogramm

Static Scope

Beispielrechnung

```
   n/10
   print(b(n+2,3));
```

-> Aufruf von `b(n+2,3)`

-> hierzu: Auswertung der Parameter: `n1/12, n2/3`

-> Deklaration von lokalem n: `n(b)/4`

-> Ausführung von n = n1+n2: `n(b)/15`

-> Aufruf von `a()`

-> Deklaration von lokalem m: `m(a)/30`

-> Ausführung von n = m-n: `n/20`

-> Rückgabe von globalem n an b(): `a() -> 20`

-> Rückgabe von lokalem n(b) + a() an Hauptprogramm: `b() -> 35`

-> Ausgabe von Hauptprogramm: `35` **in Semantik irrelevant**

-> Semantik: Zustand [`n/20`]

Statische Semantik

- **Static Scope**
  - Deklarationsumgebung für Variablen gilt: n = 10
  - nicht Aufrufumgebung: n = n1+n2
- **Bezeichner einer Deklaration paarweise verschieden**
  - im Hauptprogramm & in verschiedenen Funktionen oder Prozeduren können aber die gleichen Bezeichner mehrmals verwendet werden.
  - innerste Deklaration ist gültig für ein Auftreten
- **Bezeichner müssen deklariert sein vor Verwendung**
  - entweder selbst in Funktion oder Prozedur
  - oder im Hauptprogramm

Semantik von Mini-Java-FunProc

- **analog zur Semantik von Java, hier nicht formal:**
  - Hauptprogramm wie bisher
  - wird Prozedur oder Funktion *f* aufgerufen:
    - Auswertung der aktuellen Parameter (Ausdrücke) von *f*
    - Zuweisung der aktuellen Werte an die formalen Parameter
    - Ausführung des Codes von *f*
    - tritt dabei Variable auf:
      Unterscheidung, ob Variable in *f* deklariert, wenn
      - ja: Verwendung der lokalen Variable
      - nein: Verwendung der Variable aus Hauptprogramm
    - Rücksprung in aufrufendes Programm bei
      - Abarbeitung des Prozedurrumpfes (Zustandstransformation)
      - Ausführung der return-Anweisung in Funktion mit Rückgabe des Ausdruckswertes (Zustandstransformation + Integerwert)

ZFP

- **Zwischencode für Funktionen und Prozeduren**
- **Übersetzung eines Funktions-/Prozeduraufrufs:**
  - zusätzlicher Speicherplatz für lokale Variablen & Parameter
  - Freigabe des zusätzlichen Platzes nach Ausführung des Aufrufs

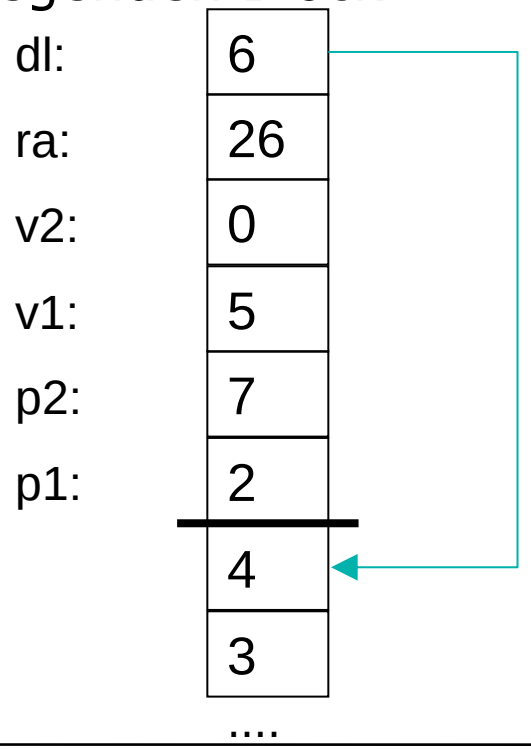
  ->Speicherbedarf für Prozeduraufrufe **laufzeitabhängig** & wegen Rekursion **unbeschränkt**
- **dynamische Speicherverwaltung**
  - wegen Schachtelung & Rekursion wird kein Heap benötigt
  - reicht Stack: **Laufzeitkeller (Prozedurkeller)**

Prozedurkeller bei Funktions-/Prozeduraufruf

- **Block auflegen mit Platz für:**
  - Parameter
  - lokale Variablen
  - Rücksprungadresse
  - dynamischen Verweis auf darunterliegenden Block

```
void f(int p1, int p2) {
  int v1=5, v2;
  v1=v2*2;
}
```

Aufruf: `25: f(2,7);`



Zustand von ZFP

- **Stackmaschine mit Hauptspeicher, Datenkeller, Prozedurkeller**
- **Speicher (Zustandsraum)**
  - SM = BZ x DK x PK x HS Stackmaschine
  - BZ = Nat Befehlszähler
  - DK = Int* Datenkeller (Spitze links)
  - PK = Int* Prozedurkeller (Spitze links)
  - HS = {h | h: Nat -> Int} Hauptspeicher
    h0: leerer Hauptspeicher, nirgends definiert
- **Zustand**
  - s = (m,d,b,h) SM
  - schreiben: DK-Zustand: d = d.1 d.2 ... d.r
    PK-Zustand: b = b.1 b.2 ... b.s
    oder: d.1...d.n:d', dann d' Rest Spitze d.1 links

Befehlsvorrat von ZFP

- **Arithmetische Befehle, Logische Befehle und Sprungbefehle wie bisher, kein Einfluss auf Prozedurstack**
- **neuer Befehl:** - HALT //STOP wie in M32
- **Prozedurkellerbefehle:**
  - CALL(ca,npar,lv1...lvn) mit
    - ca: Codeadresse der Funktion/Prozedur
    - npar: Anzahl der Parameter
    - lv1...lvn: Initialwerte der lokalen Variablen
  - RET
- **Transportbefehle**
  - LIT z, LOD i, STO i // wie bisher für Hauptprogramm
  - LODLOCAL i // im obersten Block i-te Pos. von unten
  - STOLOCAL i // im obersten Block i-te Pos. von unten

Semantik der PK-Befehle

- $\mathbb{C}$ (HALT) (m, d, b, h) = (stopadr, d, b, h)
  Setze BZ direkt hinter letzten Befehl.

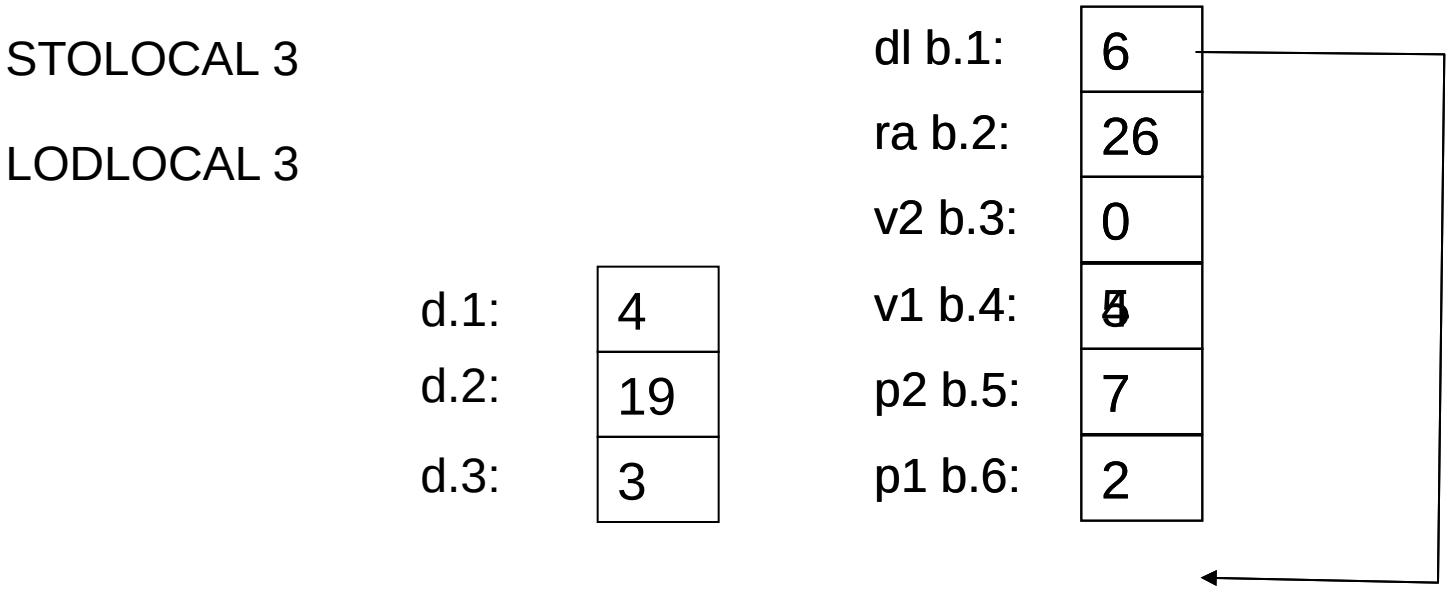
- $\mathbb{C}$ (CALL(ca,npar,lv1...lvn)) (m, pnpar...p1 : d', b, h)
  = (ca, d', (npar+n+2) (m+1) lvn...lv1 pnpar...p1 : b, h),
  dl ra lok. Var. Par.
  Par. vor Aufruf auf DK ausgewertet und dann auf PK gelegt
  für lokale Variablen wird Platz angelegt und deren Wert aus Initialisierung eingetragen, ggf. 0.
  Als Rücksprungadresse nächster Befehl
  Dynamischer Verweis zeigt auf drunter liegenden Block

  Bsp.: Für b(n1,n2) -> Call(adr(b),2,4)

- $\mathbb{C}$ (RET) (m, d, (npar+n+2) ra lvn...lv1 pnpar...p1 : b', h)
  = (ra, d, b' , h)
  lösche obersten Block auf Prozedurkeller und fahre bei Rücksprungadresse fort

Semantik der lokalen Transportbefehle

- $\mathbb{C}$ (LODLOCAL i) (m, d, b, h) = (m+1, b.((b.1)-i+1) : d, b, h)
  lade den Parameter oder die Variable an der Position i von unten im aktuellen Block auf den Datenkeller
- $\mathbb{C}$ (STOLOCAL i) (m, d.1:d', b, h) = (m+1, d', b[(b.1)-i+1 / d.1], h)
  schreibe in den Prozedurkeller an der Position i von unten im aktuellen Block den obersten Wert vom Datenkeller und lösche diesen auf dem Datenkeller



Übersetzung

- **jeweils eine Symboltabelle für**
  - Hauptprogramm
    - Konstanten und Variablen wie bisher belegt
    - Funktionen & Prozeduren f werden eingetragen mit: Tag f oder p, Anzahl der Parameter, Initialwerte der lokalen Variablen und symbolischer Startadresse adr(f)

Tab = { st | st : Ide -> ({const} x Int)$\cup$({var} x Nat) $\cup$
({f} x Nat x Int* x Adr) $\cup$ ({p} x Nat x Int* x Adr) }

  - jede Prozedur und Funktion f
    - Konstanten wie immer
    - Variablen & Parameter mit dem Tag var und der Nummer des Auftretens: 1: linkster Parameter ... n: rechteste Variable entspricht der Position im PK-Block von unten

Tabf = {stf | stf : Ide -> ({const} x Int)$\cup$({var} x Nat) }

Erweiterung von *update*

- **Aufbau der Symboltabellen mit *update***
- **für Hauptprogramm:**
  - für Konstanten und Variablen wie bisher
  - für Prozeduren und Funktionen:
    - *update* (`func/void fun(int p1,...,int pn) {... int v1,...,vk,vk+1=z1,...,vk+m=zm;...}` ,st,h)

      = (st[fun/(f oder p,n,0...0z1...zm,adr(fun))],h) //k Nullen
- **für Prozeduren und Funktionen f**
  - für Konstanten wie immer
  - für Variablen und Parameter:
    - *update* (`func/void f(int p1,...,int pn) {... int v1, ..., vm; ...}` ,stf,h)

      =(stf[p1/(var,1),...,pn/(var,n),v1/(var,n+1),...,vm/(var,n+m)],h)

Erweiterung der Übersetzung

- ***trans*: Mini-Java-FunProc-Progs -> ZFP x HS liefert**
  - *trans*(*constDecl varDecl profunDecl statement)* = (*cmdtrans(statement,*sts) HALT; *procfuntrans(profunDecl,sts)* ,h) ,
    wobei (sts, h) = *update(constDecl varDecl profunDecl,* st0, h0)
    und sts : Sammlung aller Symboltabellen
    und stfst : beiden Symboltabellen für f und das Hauptprogramm
- **Übersetzung von Prozedur-/Funktionsdeklarationen mittels Funktion *procfuntrans***
  - *procfuntrans*(`void f(int p1,...,int pn) { decls; stmt }`,sts)
    = adr(f): *cmdtransfunproc(stmt,*stfst)
    RET;
  - *procfuntrans*(`func f(int p1,...,int pn) { decls; stmt return(exp); }` ,sts)
    = adr(f): *cmdtransfunproc(stmt,*stfst)
    exptrans(exp, stfst) RET;

Funktion *cmdtransfunproc*

- **Übersetzung der Statements in Prozedur /Funktionsdeklarationen mittels Funktion *cmdtransfunproc***
- **Unterscheidung, ob globale oder lokale Variable oder Parameter**
  - *st* globale Symboltabelle
  - *stf* Symboltabelle für Funktion

  – *cmdtransfunproc*(*ident*=*exp*,stfst) = **if** stf(*ident*) **=** (var,i)
  then *exptrans(exp*,stfst)
  STOLOCAL i;
  else if st(*ident*) **=** (var,i)
  then *exptrans(exp*,stfst)
  STO i;

- **Rest analog zu *cmdtrans* mit Übergabe von stfst**

Erweiterung von *cmdtrans*

- Prozeduraufruf in *cmdtrans*
  - *cmdtrans*(*proc(exp1,...,expnpar)* ,sts)
    = **if** st(*proc*) **=** (p,npar,0...0z1...zn,adr(proc))
    then *exptrans(exp1* ,sts) ...
    exptrans(expnpar ,sts)
    CALL(adr(proc),npar,0...0z1...zn);

Auswertung der Parameter und Aufruf von p

Symboltabelle kommt von aufrufender Umgebung

sts = st oder stfst

Erweiterung von *exptrans*

- **bei Parameter- oder Variablenbezeichner Unterscheidung, ob globale oder lokale Variable oder Parameter**
  - *st* globale Symboltabelle
  - *stf* Symboltabelle für Funktion

  – *exptrans*(*ident*,stfst) = **if** stf(*ident*) **=** (var,i)
  then LODLOCAL i;
  else if st(*ident*) **=** (var,i)
  then LOD i;

- **Funktionsaufruf**

  – *exptrans*(*fun(exp1,...,expnpar)* ,sts)
  = **if** st(*fun*) **=** (f,npar,0...0z1...zn,adr(fun))
  then *exptrans(exp1* ,sts) ...
  exptrans(expnpar ,sts)
  CALL(adr(fun),npar,0...0z1...zn)*;*

  sts = st oder stfst

  nach Auswertung liegt Ergebnis auf Datenkeller, s. procfuntrans

Beispielübersetzung

- **Übersetzung des Programms auf Folie 5:**

update(`int n=10;`
`func a() {int m=30; ...}`
`func b(int n1, int n2) {int n=4; ...}`, st0, h0)
=(st0[n/(var,1), a/(f, 0, 30, adr(a)), b/(f, 2, 4, adr(b))], h0[1/10])
=**(st, h)**

update(`func a() {int m=30; ...}`, sta0, h)
= (sta0[m/(var,1)], h) = **(sta,** h**)**

update(`func b(int n1, int n2) {int n=4; ...}`, stb0, h)
= (stb0[n1/(var,1), n2/(var,2), n/(var,3)], h) **= (stb,** h**)**

--> **Sei 'sts' die Sammlung der Symboltabellen 'st'(Hauptprogramm), 'sta'(Funktion a), 'stb'(Funktion b)**

Beispielübersetzung 2

- **trans(P)**

= (*cmdtrans*(`print(...)`, sts)
HALT;
procfuntrans(`func a() {...}` , sts)
procfuntrans(`func b(…) {...}` , sts), h)

- **h bleibt gleich, darum nur noch 1. Komponente**

= *cmdtrans*(`print(b(n+2,3));` , sts)
HALT;
procfuntrans(`func a() {… n=m-n; return(n);}` , sts)
procfuntrans(`func b(…){… n=n1+n2; return(n+a());}` , sts)

= *exptrans*(`b(n+2,3)`, st) POP; HALT;
adr(a): *cmdtransfunproc*(`n=m-n;`, stast)
exptrans(`n`, stast) RET;
adr(b): *cmdtransfunproc*(`n=n1+n2;`, stbst)
exptrans(`n+a()`, stbst) RET;

Beispielübersetzung 3

```
=           exptrans(n+2, st)
            exptrans(3, st)
            CALL(adr(b), 2, 4);                    POP;   HALT;
adr(a):     exptrans(m-n, stast)
            STO 1;
            LOD 1;                                 RET;
adr(b):     exptrans(n1+n2, stbst)
            STOLOCAL 3;
            exptrans(n, stbst)
            exptrans(a(), stbst)
            ADD;                                   RET;
=       ....
```

Beispielübersetzung 4

```
=           LOD 1;LIT 2; ADD;
            LIT 3;
            CALL(adr(b), 2, 4);                    POP;   HALT;
adr(a):     LODLOCAL 1; LOD 1; SUB;
            STO 1;
            LOD 1;                                 RET;
adr(b):     LODLOCAL 1; LODLOCAL 2; ADD;
            STOLOCAL 3;
            LODLOCAL 3;
            CALL(adr(a),0,30);
            ADD;                                   RET;
```

Beispielübersetzung 5 mit Adressen

= main:

```
1:  LOD 1;
2:  LIT 2;
3:  ADD;
4:  LIT 3;
5:  CALL(14, 2, 4);
6:  POP;
7:  HALT;
```

a:

```
8:   LODLOCAL 1;
9:   LOD 1;
10:  SUB;
11:  STO 1;
12:  LOD 1;
13:  RET;
```

b:

```
14:  LODLOCAL 1;
15:  LODLOCAL 2;
16:  ADD;
17:  STOLOCAL 3;
18:  LODLOCAL 3;
19:  CALL(8,0,30);
20:  ADD;
21:  RET;
```

Beispielrechnung

- **Rechnung analog zu Folie 6**

  (BZ, DK, PK, HS)

(1, , , [1/10])	// LOD 1;
(2, 10, , [1/10])	// LIT 2;
(3, 2 10, , [1/10])	// ADD;
(4, 12, , [1/10])	// LIT 3;
(5, 3 12, , [1/10])	// CALL(14,2,4);
(14, , 5 6 4 3 12, [1/10])	// LODLOCAL 1;
(15, 12, 5 6 4 3 12, [1/10])	// LODLOCAL 2;
(16, 3 12, 5 6 4 3 12, [1/10])	// ADD;
(17, 15, 5 6 4 3 12, [1/10])	// STOLOCAL 3;
(18, , 5 6 15 3 12, [1/10])	// LODLOCAL 3;

Beispielrechnung 2

(19, 15, 5 6 15 3 12, [1/10])	// CALL(8,0,30);
(8, 15, 3 20 30 5 6 15 3 12, [1/10])	// LODLOCAL 1;
(9, 30 15, 3 20 30 5 6 15 3 12, [1/10])	// LOD 1;
(10, 10 30 15, 3 20 30 5 6 15 3 12, [1/10])	// SUB;
(11, 20 15, 3 20 30 5 6 15 3 12, [1/10])	// STO 1;
(12, 15, 3 20 30 5 6 15 3 12, [1/20])	// LOD 1;
(13, 20 15, 3 20 30 5 6 15 3 12, [1/20])	// RET;
(20, 20 15, 5 6 15 3 12, [1/20])	// ADD;
(21, 35, 5 6 15 3 12, [1/20])	// RET;
(6, 35, , [1/20])	// POP;
(7, , , [1/20])	// HALT;
(22, , , [1/20])	**-> Semantik: [1/20]**

Umsetzung in Praktikum

- **nur 1 Stack: Prozedur- und Datenkeller zusammenfassen Übungsaufgabe**
  - bei Prozedur/Funktionsaufruf Auswertung der Parameter
  - damit liegen deren Werte schon oben auf Stack
  - zusätzlich noch Platz für
    - lokale Variablen
    - Rücksprungadresse
    - dynamischer Verweis
- **javaCC: One-Parse-Compiler nur einmal durchlaufen**