

Compiler 1: Grundlagen

Code-Generierung

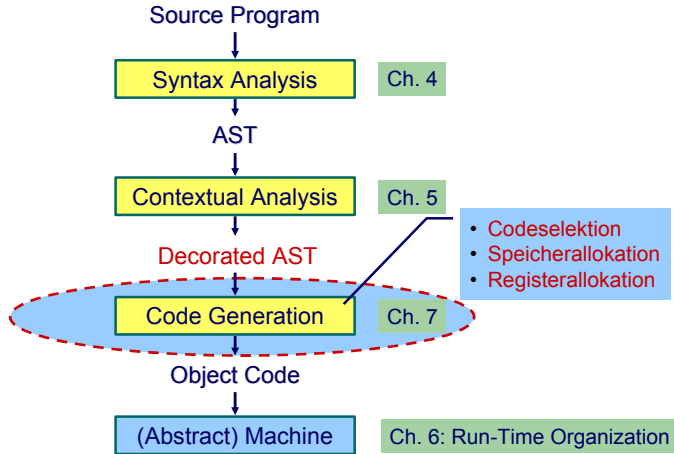


TECHNISCHE
UNIVERSITÄT
DARMSTADT

WS 2013/14

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt





- ▶ Abhängig von Eingabesprache
 - ▶ Syntaktische Analyse
 - ▶ Kontextanalyse
- ▶ Abhängig von Eingabesprache **und** Zielmaschine
 - ▶ Codegenerierung



- ▶ Abhängig von Eingabesprache
 - ▶ Syntaktische Analyse
 - ▶ Kontextanalyse
- ▶ Abhängig von Eingabesprache **und** Zielmaschine
 - ▶ Codegenerierung

➡ Schwierig allgemein zu formulieren

Codegenerierung befaßt sich mit **Semantik** der Eingabesprache

```
let
  var x: integer;
  var y: integer
in begin
  y := 2;
  x := 7;
  printint(y);
  printint(x);
end
```



```
PUSH      2
LOADL     2
STORE(1)  1[SB]
LOADL     7
STORE(1)  0[SB]
LOAD(1)   1[SB]
CALL      putint
LOAD(1)   0[SB]
CALL      putint
HALT
```

Codegenerierung befaßt sich mit **Semantik** der Eingabesprache

```
let
  var x: integer;
  var y: integer
in begin
  y := 2;
  x := 7;
  printint(y);
  printint(x);
end
```



```
PUSH      2
LOADL     2
STORE(1)  1[SB]
LOADL     7
STORE(1)  0[SB]
LOAD(1)   1[SB]
CALL      putint
LOAD(1)   0[SB]
CALL      putint
HALT
```

➔ Gleiche Semantik für Quellprogramm und Zielprogramm



Aufteilung in Unterprobleme

- ▶ **Code-Selektion**

Ordnet Phrasen aus Quellprogramm Folgen von Maschineninstruktionen zu

- ▶ **Speicherallokation**

Weist jeder Variablen Speicherplatz zu und führt über diesen Buch

- ▶ **Registerallokation**

Verwaltet Registerverwendung für Variablen und Zwischenergebnisse (nicht in TAM!)



- ▶ Semantik
 - ▶ In der Regel auf Phrasenebene beschrieben
 - ▶ Expressions, Commands, Declarations, ...



- ▶ Semantik
 - ▶ In der Regel auf Phrasenebene beschrieben
 - ▶ Expressions, Commands, Declarations, ...

Vorgehensweise

Induktives Herleiten der Übersetzung des gesamten Programmes aus Übersetzungen von Einzelphrasen

- ▶ Semantik
 - ▶ In der Regel auf Phrasenebene beschrieben
 - ▶ Expressions, Commands, Declarations, ...

Vorgehensweise

Induktives Herleiten der Übersetzung des gesamten Programmes aus Übersetzungen von Einzelphrasen

- ▶ Problem: Mehrere semantisch korrekte Übersetzungen für eine Phrase
- ▶ Wie konkrete Instruktionsfolge auswählen?

- ▶ Semantik
 - ▶ In der Regel auf Phrasenebene beschrieben
 - ▶ Expressions, Commands, Declarations, ...

Vorgehensweise

Induktives Herleiten der Übersetzung des gesamten Programmes aus Übersetzungen von Einzelphrasen

- ▶ Problem: Mehrere semantisch korrekte Übersetzungen für eine Phrase
- ▶ Wie konkrete Instruktionsfolge auswählen?

➡ Code-Selektion



Code-Funktion

Bildet Phrase auf Instruktionsfolge ab.



Code-Funktion

Bildet Phrase auf Instruktionsfolge ab.

Definition durch:

Code-Schablone

Ordnet jeder speziellen *Form* einer Phrase eine Definition in Form von Maschineninstruktionen oder Anwendungen von Code-Funktionen zu.



Code-Funktion

Bildet Phrase auf Instruktionsfolge ab.

Definition durch:

Code-Schablone

Ordnet jeder speziellen *Form* einer Phrase eine Definition in Form von Maschineninstruktionen oder Anwendungen von Code-Funktionen zu.

Wichtig: Eingabesprache muß **vollständig** durch Code-Schablonen **abgedeckt** werden.

Beispiel: Code-Funktion 1



TECHNISCHE
UNIVERSITÄT
DARMSTADT

execute : **Command** → **Instruction***

Beispiel: Code-Funktion 1



execute : **Command** → **Instruction***

Anweisungsfolge $C1; C2$

Semantik: Führe erst $C1$ aus, dann $C2$.



execute : **Command** → **Instruction***

Anweisungsfolge C1 ; C2

Semantik: Führe erst C1 aus, dann C2.

execute [[C1 ; C2]] =
 execute[[C1]]
 execute[[C2]]

Beispiel: Code-Funktion 2



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Zuweisung I := E

Beispiel: Code-Funktion 2



Zuweisung $I := E$

Semantik: Weise Wert von Ausdruck E and Variable bezeichnet durch I zu

Beispiel: Code-Funktion 2



Zuweisung $I := E$

Semantik: Weise Wert von Ausdruck E and Variable bezeichnet durch I zu

execute [[$I := E$]] =

evaluate[[E]]

STORE a , mit a =Adresse von Variable I

Beispiel: Code-Funktion 3



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Anweisungsfolge $f := f*n; n := n-1$

Beispiel: Code-Funktion 3



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Anweisungsfolge $f := f * n; n := n - 1$

execute $[[f := f * n; n := n - 1]] =$

Beispiel: Code-Funktion 3



Anweisungsfolge $f := f * n; n := n - 1$

$execute [[f := f * n; n := n - 1]] =$

$$\begin{aligned} & execute [[f := f * n]] \\ & execute [[n := n - 1]] \quad = \end{aligned}$$

Beispiel: Code-Funktion 3



Anweisungsfolge $f := f * n; n := n - 1$

$execute [[f := f * n; n := n - 1]] =$

$execute [[f := f * n]]$
 $execute [[n := n - 1]] \quad =$

$evaluate [[f * n]]$
STORE f

Beispiel: Code-Funktion 3



Anweisungsfolge $f := f * n; n := n - 1$

execute $[[f := f * n; n := n - 1]] =$

execute $[[f := f * n]]$
execute $[[n := n - 1]] \quad =$

evaluate $[[f * n]]$
STORE f

evaluate $[[n - 1]]$
STORE $n \quad =$

Beispiel: Code-Funktion 3



Anweisungsfolge $f := f*n; n := n-1$

execute $[[f := f*n; n := n-1]] =$

execute $[[f := f*n]]$
execute $[[n := n-1]] \quad =$

evaluate $[[f*n]]$
STORE f
evaluate $[[n - 1]]$
STORE n =

LOAD f
LOAD n
CALL mult
STORE f
LOAD n
CALL pred
STORE n



Orientiert sich an Subphrasenstruktur

$$f_P [[\dots Q \dots R \dots]] =$$

...

$$f_Q [[Q]]$$

...

$$f_R [[R]]$$

...



- ▶ Sammlung **aller**
 - ▶ Code-Funktionen
 - ▶ Code-Schablonen
- ▶ Muß Eingabesprache vollständig überdecken



Abstrakte Syntax

```
Program ::= Command
Command ::= V-name := Expression
          | Identifier ( Expression )
          | Command ; Command
          | if Expression then Command
            else Command
          | while Expression do Command
          | let Declaration in Command

Program ::= AssignCommand
          | CallCommand
          | SequentialCommand
          | IfCommand
          | WhileCommand
          | LetCommand
```



Abstrakte Syntax

```
Program ::= Command
Command ::= V-name := Expression
          | Identifier ( Expression )
          | Command ; Command
          | if Expression then Command
            else Command
          | while Expression do Command
          | let Declaration in Command
```

| | | | | |
|---------|-----|-----------------------------|--|-------------------|
| Program | ::= | Command | | Program |
| Command | ::= | V-name := Expression | | AssignCommand |
| | | Identifier (Expression) | | CallCommand |
| | | Command ; Command | | SequentialCommand |
| | | if Expression then Command | | IfCommand |
| | | else Command | | |
| | | while Expression do Command | | WhileCommand |
| | | let Declaration in Command | | LetCommand |

run : Program → Instruction*

execute : Command → Instruction*

evaluate : Expression → Instruction*

fetch : V-name → Instruction*

assign : V-name → Instruction*

elaborate : Declaration → Instruction*

Code-Spezifikation 2

| <i>class</i> | <i>code function</i> | <i>effect of the generated code</i> |
|--------------|----------------------|--|
| Program | <i>run P</i> | Run the program P and then halt, starting and finishing with an empty stack. |
| Command | <i>execute C</i> | Execute the command C , possibly updating variables, but neither expanding nor contracting the stack. |
| Expression | <i>evaluate E</i> | Evaluate the expression E , pushing its result on the stack top, but having no other effects. |
| V-name | <i>fetch V</i> | Push the value of the constant or variable named V on the stack. |
| V-name | <i>assign V</i> | Pop a value from the stack top, and store it in the variable named V . |
| Declaration | <i>elaborate D</i> | Elaborate the declaration D , expanding the stack to make space for any constants and variables declared therein. |



run [C]
= *execute* [C]
HALT


$$\begin{aligned} & \textit{execute} [C_1 ; C_2] \\ &= \textit{execute} [C_1] \\ & \quad \textit{execute} [C_2] \end{aligned}$$



execute [$V := E$]
= *evaluate* [E]
assign [V]



```
execute [if E then C1 else C2]  
=          evaluate [E]  
           JUMPIF (0)      Lelse  
           execute [C1]  
           JUMP          Lfi  
Lelse:   execute [C2]  
Lfi:
```



```
execute [while E do C] =  
    Lwhile:    evaluate [E]  
              JUMPIF (0)  Lend  
              execute [C]  
              JUMP  Lwhile  
  
    Lend:
```

$$\begin{aligned} \text{execute } [\textit{let } D \textit{ in } C] \\ = \quad \text{elaborate } [D] \\ \quad \text{execute } [C] \\ \quad \text{POP } (0) \quad s \end{aligned}$$

POP nur wenn $s > 0$ (zusätzlicher Speicher alloziert wurde)

Beispiel Code-Schablonen 1



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
while i > 0 do i := i - 2
```

Beispiel Code-Schablonen 1



```
while i > 0 do i := i - 2
```

```
execute [while i>0  
         do i:=i-2]
```

evaluate [*i>0*]

execute [*i:=i-2*]

```
50: LOAD      i  
51: LOADL    0  
52: CALL     gt  
53: JUMPIF(0) 59  
54: LOAD     i  
55: LOADL    2  
56: CALL     sub  
57: STORE    i  
58: JUMP     50  
59:
```

```
execute [while E do C]  
= Lwhile: evaluate [E]  
          JUMPIF(0)  Lend  
          execute [C]  
          JUMP      Lwhile  
Lend:
```



Integer-Literal

`evaluate [IL] =`
`LOADL v` ; v is the value of IL



Integer-Literal

```
evaluate [IL] =  
    LOADL v      ; v is the value of IL
```

Variable

```
evaluate [V] =  
    fetch V
```



Integer-Literal

```
evaluate[IL] =  
    LOADL v      ; v is the value of IL
```

Variable

```
evaluate[V] =  
    fetch V
```

Unärer Operator

```
evaluate[O E] =  
    evaluate E  
    CALL p      ; p is the address of the routine corresponding to O
```



Integer-Literal

```
evaluate [IL] =  
    LOADL v          ; v is the value of IL
```

Variable

```
evaluate [V] =  
    fetch V
```

Unärer Operator

```
evaluate [O E] =  
    evaluate E  
    CALL p          ; p is the address of the routine corresponding to O
```

Binärer Operator

```
evaluate [E1 O E2] =  
    evaluate E1  
    evaluate E2  
    CALL p          ; p is the address of the routine corresponding to O
```



Konstante

```
elaborate [const I ~ E] =  
    evaluate E ; ... and decorate the tree
```



Konstante

```
elaborate [const I ~ E] =  
    evaluate E ; ... and decorate the tree
```

- ▶ Beachte: Legt berechneten Wert auf Stack ab!



Konstante

```
elaborate [const I ~ E] =  
    evaluate E ; ... and decorate the tree
```

- ▶ Beachte: Legt berechneten Wert auf Stack ab!
- ▶ Optimierung möglich:



Konstante

`elaborate [const I ~ E] =`
`evaluate E ; ... and decorate the tree`

- ▶ Beachte: Legt berechneten Wert auf Stack ab!
- ▶ Optimierung möglich:
 - ▶ Setze Wert der Konstante direkt in Maschinencode ein



Konstante

`elaborate [const I ~ E] =`
`evaluate E` ; ... and decorate the tree

- ▶ Beachte: Legt berechneten Wert auf Stack ab!
- ▶ Optimierung möglich:
 - ▶ Setze Wert der Konstante direkt in Maschinencode ein
 - ▶ Dann leere Schablone



Konstante

`elaborate [const I ~ E] =`
`evaluate E` ; ... and decorate the tree

- ▶ Beachte: Legt berechneten Wert auf Stack ab!
- ▶ Optimierung möglich:
 - ▶ Setze Wert der Konstante direkt in Maschinencode ein
 - ▶ Dann leere Schablone



Konstante

elaborate [const I ~ E] =
 evaluate E ; ... and decorate the tree

- ▶ Beachte: Legt berechneten Wert auf Stack ab!
- ▶ Optimierung möglich:
 - ▶ Setze Wert der Konstante direkt in Maschinencode ein
 - ▶ Dann leere Schablone

Variable

elaborate [var I : T] =
 PUSH size(T) ; ... and decorate the tree



Konstante

```
elaborate [const I ~ E] =  
    evaluate E ; ... and decorate the tree
```

- ▶ Beachte: Legt berechneten Wert auf Stack ab!
- ▶ Optimierung möglich:
 - ▶ Setze Wert der Konstante direkt in Maschinencode ein
 - ▶ Dann leere Schablone

Variable

```
elaborate [var I : T] =  
    PUSH size(T) ; ... and decorate the tree
```

Deklarationsfolge

```
elaborate [D1; D2] =  
    elaborate D1  
    elaborate D2
```

Code-Schablone für Vname



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Beachte: Mini-Triangle, keine lokalen Variablen!



Beachte: Mini-Triangle, keine lokalen Variablen!

Lesen

```
fetch[I] =  
    LOAD d[SB] ; d is the address of I
```



Beachte: Mini-Triangle, keine lokalen Variablen!

Lesen

```
fetch[I] =  
    LOAD d[SB] ; d is the address of I
```

Schreiben

```
assign[I] =  
    STORE d[SB] ; ditto
```

Beispiel Code-Schablonen 2



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
execute[let const n ~ 7; var i : Integer in i := n*n]
```

Beispiel Code-Schablonen 2



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
execute[let const n ~ 7; var i : Integer in i := n*n]
```

```
= elaborate[const n ~ 7; var i : Integer]  
   execute[i := n*n]
```


Beispiel Code-Schablonen 2



```
execute[let const n ~ 7; var i : Integer in i := n*n]
```

```
= elaborate[const n ~ 7; var i : Integer]  
   execute[i := n*n]
```

```
= elaborate[const n ~ 7]  
   elaborate[var i : Integer]  
   evaluate[n*n]  
   assign[i]
```

Beispiel Code-Schablonen 2



```
execute[let const n ~ 7; var i : Integer in i := n*n]
```

```
= elaborate[const n ~ 7; var i : Integer]  
   execute[i := n*n]
```

```
= elaborate[const n ~ 7]  
   elaborate[var i : Integer]  
   evaluate[n*n]  
   assign[i]
```

```
= LOADL 7  
   PUSH 1  
   LOAD n  
   LOAD n  
   CALL mult  
   STORE i  
   POP(0) 2
```



```
execute[let const n ~ 7; var i : Integer in i := n*n]
```

```
= elaborate[const n ~ 7; var i : Integer]  
   execute[i := n*n]
```

```
= elaborate[const n ~ 7]  
   elaborate[var i : Integer]  
   evaluate[n*n]  
   assign[i]
```

```
= LOADL 7  
   PUSH 1  
   LOAD n  
   LOAD n  
   CALL mult  
   STORE i  
   POP(0) 2
```

Kann noch optimiert werden (`const n`), → Inlining.



Spezialisierte Schablonen für Sonderfälle



Spezialisierte Schablonen für Sonderfälle

Beispiel: $i + 1$

Spezialisierte Schablonen für Sonderfälle

Beispiel: $i + 1$

Allg. Schablone

```
LOAD   i
LOADL  1
CALL   add
```

Spez. Schablone

```
LOAD   i
CALL   succ
```

Effizienterer Code für
“+1”.

Analoges Vorgehen für
Inlining von Konstanten



von Konstanten in Maschinen-Code

Konstante I mit statischem Wert $v = \text{valueOf}(IL)$

```
fetch [I] =  
    LOADL v ; ... v retrieved from DAST  
  
elaborate [const I ~ IL] =  
    ; ... just decorate the tree
```

Beispiel Sonderfallbehandlung



```
execute[let const n ~ 7; var i : Integer in i := n*n] =
  elaborate[const n ~ 7; var i : Integer]
  execute[i := n*n]

= elaborate[const n ~ 7]
  elaborate[var i : Integer]
  evaluate[n*n]
  assign[i]

=
  PUSH 1
  LOADL 7
  LOADL 7
  CALL mult
  STORE i
  POP(0) 2
```


Beispiel Sonderfallbehandlung



```
execute[let const n ~ 7; var i : Integer in i := n*n] =
  elaborate[const n ~ 7; var i : Integer]
  execute[i := n*n]

= elaborate[const n ~ 7]
  elaborate[var i : Integer]
  evaluate[n*n]
  assign[i]

=
  PUSH 1
  LOADL 7
  LOADL 7
  CALL mult
  STORE i
  POP(0) 2
```

Jetzt kein Speicherzugriff mehr für n erforderlich.



- ▶ Systematischer Aufbau
- ▶ Orientiert sich direkt an Code-Funktionen
- ▶ Code-Funktionen beschreiben rekursiven Algorithmus zur Traversierung vom DAST
- ▶ Wieder bewährtes Visitor-Entwurfsmuster verwenden



```
package TAM;

public class Instruction {
    public int op; // op-code (LOADop, LOADAop, etc.)
    public int n; // length field
    public int r; // register field (SBr, LBr, L1r, etc.)
    public int d; // operand field
}

public class Machine {
    public static final byte // op-codes (Table C.2)
        LOADop = 0, LOADAop = 1, ...;

    public static final byte // register numbers (Table C.1)
        CBr = 0, CTr = 1, PBr = 2, Ptr = 3, ...;

    private static Instruction[] code = new Instruction[1024];
}

public class Interpreter {
    ...
}
```



```
package Triangle.CodeGenerator;

public class Encoder extends Visitor {
    /** Append an instruction to the object program. */
    private void emit(int op, int n, int r, int d) {
        Instruction nextInstr = new Instruction();
        if (n > 255) {
            reporter.reportRestriction(
                "length of operand can't exceed 255 words");
            n = 255; // to allow code generation to continue
        }
        nextInstr.op = op;
        nextInstr.n = n;
        nextInstr.r = r;
        nextInstr.d = d;
        if (nextInstrAddr == Machine.PB)
            reporter.reportRestriction(
                "too many instructions for code segment");
        else {
            Machine.code[nextInstrAddr] = nextInstr;
            nextInstrAddr = nextInstrAddr + 1;
        }
    }
    private short nextInstrAddr = 0;
}
```

Code-Generierung via Visitor 1



Beispiel: Generiere Code für gesamtes Programm

```
public class Encoder implements Visitor {
    public Object visitProgram(Program prog, Object arg ) {
        prog.C.visit(this, arg);
        emit(Machine.HALTop, 0, 0, 0);
        return null;
    }
    ...
}
```

Code-Generierung via Visitor 2

Aufgaben der einzelnen Visitor-Methoden bei Code-Generierung

| <i>phrase class</i> | <i>visitor method</i> | <i>behaviour of the visitor method</i> |
|---------------------|-----------------------|---|
| Program | visitProgram | generate code as specified by run[P] |
| Command | visit..Cmd | generate code as specified by execute[C] |
| Expression | visit..Expr | generate code as specified by evaluate[E] |
| V-name | visit..Vname | return “ entity description ” for the visited variable or constant name (i.e. use the “decoration”). |
| Declaration | visit..Decl | generate code as specified by elaborate[D] |
| Type-Den | visit..TypeDen | return the size of the type |



Tritt je nach Umgebung mit zwei unterschiedlichen Bedeutungen auf

- ▶ Auslesen des Wertes einer Variablen
- ▶ Ziel einer Zuweisung



Tritt je nach Umgebung mit zwei unterschiedlichen Bedeutungen auf

- ▶ Auslesen des Wertes einer Variablen
- ▶ Ziel einer Zuweisung

Getrennt realisieren

```
public class Encoder implements Visitor {  
    ...  
    public void encodeFetch(Vname name) {  
        // as specified by fetch code template ...  
    }  
  
    public void encodeAssign(Vname name) {  
        // as specified by assign code template ...  
    }  
}
```

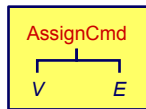
... aber nicht **in** einem Visitor, sondern **für** Visitor benutzbar.

Beispiel Benutzung von VName 1



Ziel einer Zuweisung

```
execute [V:=E] = evaluate [E]  
                assign [V]
```



```
public Object visitAssignCmd(AssignCmd cmd, Object  
    arg) {  
    cmd.E.visit(this, arg);  
    encodeAssign(cmd.V);  
}
```

Beispiel Benutzung von VName 2



Innerhalb eines Ausdrucks

```
public Object visitVnameExpression(VnameExpression expr,
                                   Object arg) {
    encodeFetch(expr.V);
    return new Short((short) 1);
}
```



Integer Literale

```
public Object visitIntegerExpression(IntegerExpression expr,
                                     Object arg) {
    short v = valuation(expr.I.spelling);
    emit(Instruction.LOADLop, (byte) 0, (byte) 0, v);
    return new Short((short) 1);
}
```



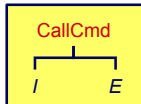
Vereinfacht für Mini-Triangle



Vereinfacht für Mini-Triangle

- ▶ Nur primitive Funktionen
- ▶ Mit maximal einem Parameter

execute [*I* (*E*)] = *evaluate* [*E*]
CALL *p*

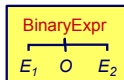


```
public Object visitCallCmd(CallCmd cmd, Object arg) {  
    cmd.E.visit(this, arg);  
    short p = address of primitive routine for name cmd.I  
    emit(Instruction.CALOp,  
        Instruction.SBr,  
        Instruction.PBr, p);  
    return null;  
}
```



Gleicher Mechanismus wie Prozeduraufruf

```
evaluate [ $E_1$  op  $E_2$ ] = evaluate [ $E_1$ ]  
                        evaluate [ $E_2$ ]  
                        CALL p
```



```
public Object visitBinaryExpression(  
    BinaryExpression expr, Object arg) {  
    expr.E1.visit(this, arg);  
    expr.E2.visit(this, arg);  
    short p = address for expr.O operation  
    emit(Instruction.CALlop,  
        Instruction.SBr,  
        Instruction.PBr, p);  
    return null;  
}
```



if/then, while, ...

```
execute [while E do C] = Lwhile: evaluate [E
                                JUMPIF (0)   Lend
                                execute [C]
                                JUMP         Lwhile
                                Lend:
```

- ▶ Realisiert durch bedingte und unbedingte Sprunginstruktionen
- ▶ Rückwärtssprünge einfach: Zieladresse bereits generiert und bekannt



if/then, while, ...

```
execute [while E do C] = Lwhile: evaluate [E]
                                JUMPIF (0)  Lend
                                execute [C]
                                JUMP        Lwhile
                                Lend:
```

- ▶ Realisiert durch bedingte und unbedingte Sprunginstruktionen
- ▶ Rückwärtssprünge einfach: Zieladresse bereits generiert und bekannt
- ▶ Vorwärtssprünge schwieriger
 - ▶ Instruktionen bis hin zur Zieladresse noch nicht generiert
 - ▶ Wert der Zieladresse damit unbekannt



```
execute [while E do C] = Lwhile: evaluate [E]  
                                JUMPIF (0)   Lend  
                                execute [C]  
                                JUMP         Lwhile  
                                Lend:
```

➔ “Nachbessern” bereits generierten Codes (*backpatching*)



```
execute [while E do C] = Lwhile: evaluate [E]
                                JUMPIF (0)   Lend
                                execute [C]
                                JUMP         Lwhile
                                Lend:
```

➔ “Nachbessern” bereits generierten Codes (*backpatching*)

1. Erzeuge Sprunginstruktion mit “leerer” (=0) Zieladresse



```
execute [while E do C] = Lwhile: evaluate [E]
                                JUMPIF (0)   Lend
                                execute [C]
                                JUMP         Lwhile
                                Lend:
```

➔ “Nachbessern” bereits generierten Codes (*backpatching*)

1. Erzeuge Sprunginstruktion mit “leerer” (=0) Zieladresse
2. Merke Adresse dieser unvollständigen Sprunginstruktion



```
execute [while E do C] = Lwhile: evaluate [E]
                                JUMPIF (0)   Lend
                                execute [C]
                                JUMP         Lwhile
                                Lend:
```

➔ “Nachbessern” bereits generierten Codes (*backpatching*)

1. Erzeuge Sprunginstruktion mit “leerer” (=0) Zieladresse
2. Merke Adresse dieser **unvollständigen** Sprunginstruktion
3. Wenn Code-Generierung gewünschte Zieladresse **erreicht**, trage **echten** Adresswert in gemerkte unvollständige Sprunginstruktion **nach**

Beispiel Backpatching 1



```
execute [while E do C] = Lwhile: evaluate [E]
                                JUMPIF (0)  Lend
                                execute [C]
                                JUMP      Lwhile
                                Lend:
```



```
public Object visitWhileCmd(WhileCmd cmd, Object arg) {
    short lwhile = nextInstrAddr;
    cmd.E.visit(this, arg);
    short jump2end = nextInstrAddr;
    emit(Instruction.JUMPIFop, 0, Instruction.CBr, 0);
    cmd.C.visit(this, arg);
    emit(Instruction.JUMPop, 0, Instruction.CBr, lwhile);
    short lend = nextInstrAddr;
    code[jump2end].d = lend;
}
```

Beispiel Backpatching 1



```
execute [while E do C] = Lwhile: evaluate [E]
                                JUMPIF (0)  Lend
                                execute [C]
                                JUMP        Lwhile
                                Lend:
```



```
public Object visitWhileCmd(WhileCmd cmd, Object arg) {
    short lwhile = nextInstrAddr;
    cmd.E.visit(this, arg);
    short jump2end = nextInstrAddr;
    emit(Instruction.JUMPIFop, 0, Instruction.CBr, 0);
    cmd.C.visit(this, arg);
    emit(Instruction.JUMPop, 0, Instruction.CBr, lwhile);
    short lend = nextInstrAddr;
    code[jump2end].d = lend;
}
```

backpatching

Beispiel Backpatching 2



```
execute [if E then C1 else C2]  
=      evaluate [E]  
      JUMPIF (0)      Lelse  
      execute [C1]  
      JUMP           Lfi  
Lelse: execute [C2]  
Lfi:
```

Doppeltes Backpatching bei if/then/else

```
public Object visitIfCommand(IfCommand com, Object arg) {  
    com.E.visit(this, arg);  
    short i = nextInstrAddr;  
    emit(Instruction.JUMPIFPop, (byte) 0,  
         Instruction.CBr, (short) 0);  
    com.C1.visit(this, arg);  
    short j = nextInstrAddr;  
    emit(Instruction.JUMPPop, (byte) 0,  
         Instruction.CBr, (short) 0);  
    short Lelse = nextInstrAddr;  
    patch(i, Lelse);  
    com.C2.visit(this, arg);  
    short Lfi = nextInstrAddr;  
    patch(j, Lfi);  
    return null;  
}
```



$execute [let D in C] = \begin{matrix} elaborate [D] \\ execute [C] \\ POP (0) \end{matrix} \quad s$

nur wenn $s > 0$,
wobei s =
Speichermenge
alloziert für D .

execute [*let D in C*] = *elaborate* [*D*]
execute [*C*]
POP (0) s

nur wenn $s > 0$,
wobei s =
Speichermenge
alloziert für D .

- ▶ ... aber wie eine Deklaration “elaborieren”?
- ▶ Weise Variablen und unbekanntem Konstanten (?) Speicherort zu
- ▶ Bei Ende von Geltungsbereich: Betroffene Speicherbereiche freigeben

$execute [let D in C] = elaborate [D]$
 $execute [C]$
 $POP (0)$ s

nur wenn $s > 0$,
wobei $s =$
Speichermenge
alloziert für D .

- ▶ ... aber wie eine Deklaration “elaborieren”?
- ▶ Weise Variablen und unbekannten Konstanten (?) Speicherort zu
- ▶ Bei Ende von Geltungsbereich: Betroffene Speicherbereiche freigeben

Ziel: Bestimme d in

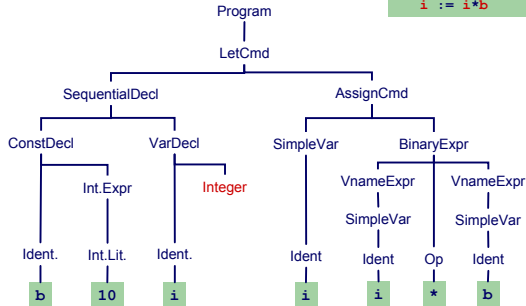
$fetch [V] = LOAD (1) d[SB]$

$assign [V] = STORE (1) d[SB]$

Beispiel Konstanten und Variablen



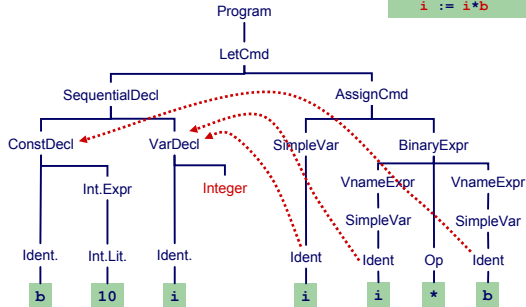
```
let  
  const b ~ 10;  
  var i: Integer  
in  
  i := i*b
```



Beispiel Konstanten und Variablen



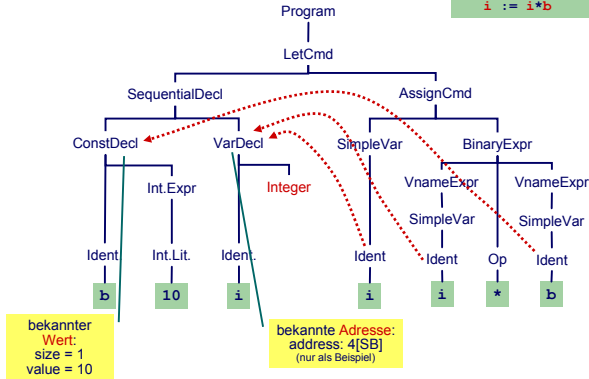
```
let  
  const b ~ 10;  
  var i: Integer  
in  
  i := i*b
```



Beispiel Konstanten und Variablen



```
let  
  const b ~ 10;  
  var i: Integer  
in  
  i := i*b
```



Bekannter Wert und bekannte Adresse

```
let
  const b ~ 10;
  var i: Integer
in
  i := i*b
```

```
PUSH      1
LOAD(1)   4[SB]
LOADL     10
CALL      mult
STORE(1)  4[SB]
POP(0)    1
```

Platz für *i*

Bekannter Wert und bekannte Adresse

```
let
  const b ~ 10;
  var i: Integer
in
  i := i*b
```

```
PUSH      1
LOAD(1)   4[SB]
LOADL     10
CALL      mult
STORE(1)  4[SB]
POP(0)    1
```

Platz für *i*

Bekannter Wert und bekannte Adresse

```
let
  const b ~ 10;
  var i: Integer
in
  i := i*b
```

```
PUSH      1
LOAD(1)   4[SB]
LOADL     10
CALL      mult
STORE(1)  4[SB]
POP(0)    1
```

Platz für *i*

Unbekannter Wert und bekannte Adresse

```
let
  var x: Integer
in let
  const y ~ 365 + x
  in putint(y)
```

bekannte Adresse:
address = 5

Unbekannter Wert:
size = 1
address = 6

```
PUSH      1      ; room for x
PUSH      1      ; room for y
LOADL     365
LOAD(1)   5[SB]  ; load x
CALL      add    ; 365+x
STORE(1)  6[SB]  ; y ~ 365+x
LOAD(1)   6[SB]
CALL      putint
POP(0)    1
POP(0)    1
```


| | |
|--------------------|--|
| Bekannter Wert | const Deklaration mit einem Literal |
| Unbekannter Wert | const Deklaration mit einem Ausdruck |
| Bekannte Adresse | var Deklaration |
| Unbekannte Adresse | Argument-Adresse gebunden an var-Parameter |



Deklaration eines Bezeichners `id`: Binde `id` an neuen **Entitätsdeskriptor**

- ▶ **Bekannter Wert**: Speichere **Wert** und seine **Größe**
- ▶ **Bekante Adresse**: Speichere **Adresse** und fordere **Platz** an



Deklaration eines Bezeichners `id`: Binde `id` an neuen **Entitätsdeskriptor**

- ▶ **Bekannter Wert**: Speichere **Wert** und seine **Größe**
- ▶ **Bekante Adresse**: Speichere **Adresse** und fordere **Platz** an

Benutzung von `id`: Rufe passenden Deskriptor ab und erzeuge Code, um auf beschriebene Entität zuzugreifen

- ▶ Lade Konstante direkt via `LOADL`
- ▶ Lade Variable von bekannter Adresse via `LOAD`



Implementierung des Entitätsdeskriptors durch `RuntimeEntity`

```
public abstract class RuntimeEntity {
    public short size;
    ...
}
public class KnownValue extends RuntimeEntity {
    public short value;
    ...
}
public class UnknownValue extends RuntimeEntity {
    public short address;
    ...
}
public class KnownAddress extends RuntimeEntity {
    public short address;
    ...
}

public abstract class AST {
    public RuntimeEntity entity;
    ...
}
```



Wie mit unbekanntem Werten oder Adressen verfahren?

- ▶ Erzeuge Code zur Evaluation der Entität **zur Laufzeit**
- ▶ Speichere Ergebnis an **bekannter** Adresse ab
- ▶ Erzeuge **Entitätsdeskriptor** für diese Adresse
- ▶ Nutze Entitätsdeskriptor, um Inhalt der Adresse bei **Verwendung** der unbekanntem Entität auszulesen



Globale Variablen

```
let
  var a: Integer;
  var b: Boolean;
  var c: Integer
in begin
  ...
end
```

| var | size | address |
|-----|------|---------|
| a | 1 | [0] SB |
| b | 1 | [1] SB |
| c | 1 | [2] SB |

In TAM, echte Maschinen
haben hier wahrscheinlich
unterschiedliche Größen



Globale Variablen

```
let
  var a: Integer;
  var b: Boolean;
  var c: Integer
in begin
  ...
end
```

| var | size | address |
|-----|------|---------|
| a | 1 | [0] SB |
| b | 1 | [1] SB |
| c | 1 | [2] SB |

In TAM, echte Maschinen
haben hier wahrscheinlich
unterschiedliche Größen

Globale Variablen

```
let
  var a: Integer;
  var b: Boolean;
  var c: Integer
in begin
  ...
end
```

| var | size | address |
|-----|------|---------|
| a | 1 | [0] SB |
| b | 1 | [1] SB |
| c | 1 | [2] SB |

In TAM, echte Maschinen haben hier wahrscheinlich unterschiedliche Größen

Verschachtelte Blöcke

```
let var a: Integer
in begin
  ...
  let var b: Boolean;
    var c: Integer
  in ...

  let var d: Integer
  in ...
end
```

| var | size | address |
|-----|------|---------|
| a | 1 | [0] SB |
| b | 1 | [1] SB |
| c | 1 | [2] SB |
| d | 1 | [1] SB |

d verwendet Platz von b wieder (anderer Geltungsbereich)



- ▶ Code-Generator führt Buch über Größe des belegten Speichers



- ▶ Code-Generator führt Buch über Größe des belegten Speichers
- ▶ In Abhängigkeit von Deklarationen und Geltungsbereichen



- ▶ Code-Generator führt Buch über Größe des belegten Speichers
- ▶ In Abhängigkeit von Deklarationen und Geltungsbereichen
- ▶ Implementierung: Erweitern des Visitors



- ▶ Code-Generator führt Buch über Größe des belegten Speichers
- ▶ In Abhängigkeit von Deklarationen und Geltungsbereichen
- ▶ Implementierung: Erweitern des Visitors
 - ▶ Verwende Parameter `Object arg` zur Eingabe des **aktuell** belegten Speicherplatzes



- ▶ Code-Generator führt Buch über Größe des belegten Speichers
- ▶ In Abhängigkeit von Deklarationen und Geltungsbereichen
- ▶ Implementierung: Erweitern des Visitors
 - ▶ Verwende Parameter `Object arg` zur Eingabe des **aktuell** belegten Speicherplatzes
 - ▶ Verwende Funktionsergebnis zur Rückgabe des **zusätzlich** benötigten Speicherplatzes



- ▶ Code-Generator führt Buch über Größe des belegten Speichers
- ▶ In Abhängigkeit von Deklarationen und Geltungsbereichen
- ▶ Implementierung: Erweitern des Visitors
 - ▶ Verwende Parameter `Object arg` zur Eingabe des **aktuell** belegten Speicherplatzes
 - ▶ Verwende Funktionsergebnis zur Rückgabe des **zusätzlich** benötigten Speicherplatzes
 - ▶ Verpacken der Angaben in ein `Short`-Objekt



- ▶ Code-Generator führt Buch über Größe des belegten Speichers
- ▶ In Abhängigkeit von Deklarationen und Geltungsbereichen
- ▶ Implementierung: Erweitern des Visitors
 - ▶ Verwende Parameter `Object arg` zur Eingabe des **aktuell** belegten Speicherplatzes
 - ▶ Verwende Funktionsergebnis zur Rückgabe des **zusätzlich** benötigten Speicherplatzes
 - ▶ Verpacken der Angaben in ein `Short`-Objekt

```
public Object visitXYZ(XYZ xyz, Object arg)
```

Falls nicht Null, **Short**-Objekt
mit zusätzlich benötigtem Platz

Short-Objekt mit bisher
benötigtem Speicherplatz



Allgemeines Schema

Weitergabe der **bisherigen** Belegung in `gs`

```
public Object visit...Command(..., Object arg) {  
    short gs = ((Short) arg).shortValue();  
    ...  
}
```




Allgemeines Schema

Weitergabe der **bisherigen** Belegung in `gs`

```
public Object visit...Command(..., Object arg) {  
    short gs = ((Short) arg).shortValue();  
    ...  
}
```

➡ Ist auch nächste **freie** Adresse!



Allgemeines Schema

Weitergabe der **bisherigen** Belegung in `gs`

```
public Object visit...Command(..., Object arg) {  
    short gs = ((Short) arg).shortValue();  
    ...  
}
```

➡ Ist auch nächste **freie** Adresse!

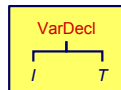
Weitergabe der **Erhöhung** des Speicherbedarfs im Ergebnis

```
public Object visit...Declaration(...) {  
    ...  
    return new Short(...);  
}
```



Elaboriere Variablendeklaration

elaborate [**var** *I* : *T*] = PUSH *s* where *s* = size of *T*



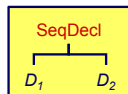
```
public Object visitVarDecl(VarDecl decl, Object arg) {
    short gs = shortValueOf(arg);
    short s = shortValueOf(decl.T.visit(this, null));
    decl.entity = new KnownAddress(s, gs);
    emit(Instruction.PUSHop, 0, 0, s);
    return new Short(s);
}
```

Remember the **size** and
address of the variable.



Elaboriere Folge von Deklarationen

$elaborate [D_1; D_2] = \begin{matrix} elaborate [D_1] \\ elaborate [D_2] \end{matrix}$



```
public Object visitSeqDecl(SeqDecl decl, Object arg) {
    short gs = shortValueOf(arg);
    short s1 = shortValueOf(decl.D1.visit(this, gs));
    short s2 = shortValueOf(decl.D2.visit(this,
                                        new Short(gs+s1)));
    return new Short(s1+s2);
}
```

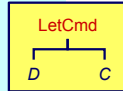


Führe kompletten `let`-Block aus

```
execute [let D in C] = elaborate [D]  
                        execute [C]  
                        POP(0)      s
```

nur wenn $s > 0$, wobei s
Größe des durch D
angeforderten
Speichers ist.

```
public Object visitLetCmd(LetCmd cmd, Object arg) {  
    short gs = shortValueOf(arg);  
    short s = shortValueOf(cmd.D.visit(this, gs));  
    cmd.C.visit(this, new Short(gs+s));  
    if (s > 0)  
        emit(Instruction.POPop, 0, 0, s);  
    return null;  
}
```



```
private static short shortValueOf(Object obj) {  
    return ((Short)obj).shortValue();  
}
```

Beispiel Speicherverwaltung im Visitor



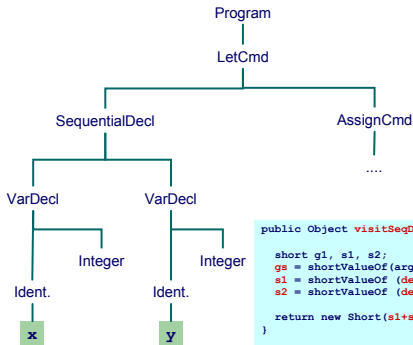
TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
let  
  var x: Integer;  
  var y: Integer  
in  
  x := y
```

Beispiel Speicherverwaltung im Visitor



```
let  
  var x: Integer;  
  var y: Integer  
in  
  x := y
```

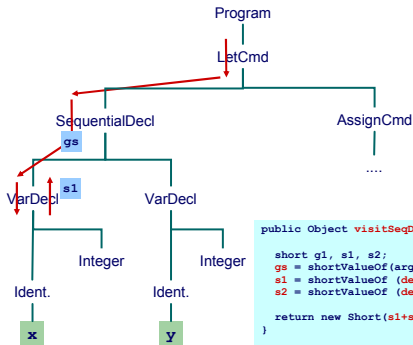


```
public Object visitSeqDecl(SeqDecl decl,  
                           Object arg) {  
  short g1, s1, s2;  
  gs = shortValueOf(arg);  
  s1 = shortValueOf (decl.D1.visit(this, gs));  
  s2 = shortValueOf (decl.D2.visit(this,  
                                   new Short(gs+s1)));  
  return new Short(s1+s2);  
}
```

Beispiel Speicherverwaltung im Visitor



```
let
  var x: Integer;
  var y: Integer
in
  x := y
```

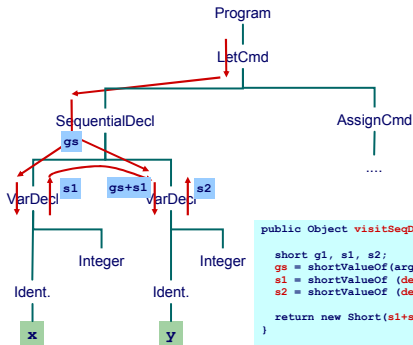


```
public Object visitSeqDecl(SeqDecl decl,
                           Object arg) {
    short g1, s1, s2;
    gs = shortValueOf(arg);
    s1 = shortValueOf (decl.D1.visit(this, gs));
    s2 = shortValueOf (decl.D2.visit(this,
                                     new Short(gs+s1)));
    return new Short(s1+s2);
}
```


Beispiel Speicherverwaltung im Visitor



```
let  
  var x: Integer;  
  var y: Integer  
in  
  x := y
```

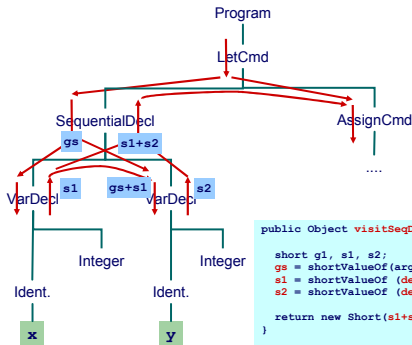


```
public Object visitSeqDecl(SeqDecl decl,  
                           Object arg) {  
  short g1, s1, s2;  
  gs = shortValueOf(arg);  
  s1 = shortValueOf (decl.D1.visit(this, gs));  
  s2 = shortValueOf (decl.D2.visit(this,  
                                   new Short(gs+s1)));  
  return new Short(s1+s2);  
}
```

Beispiel Speicherverwaltung im Visitor



```
let  
  var x: Integer;  
  var y: Integer  
in  
  x := y
```



```
public Object visitSeqDecl(SeqDecl decl,  
                           Object arg) {  
  short g1, s1, s2;  
  gs = shortValueOf(arg);  
  s1 = shortValueOf (decl.D1.visit(this, gs));  
  s2 = shortValueOf (decl.D2.visit(this,  
                                   new Short(gs+s1)));  
  return new Short(s1+s2);  
}
```



Bekannte Werte, Variablen und unbekannte Werte

```
fetch [I] = LOADL v wobei v = Wert gebunden an I  
fetch [I] = LOAD(s) d[SB] wobei d = Adresse gebunden  
an I und s = size(Typ von I)
```

```
public Object encodeFetch(Vname name, short s) {  
    RuntimeEntity entity =  
        (RuntimeEntity) name.visit(this, null);  
    if (entity instanceof KnownValue) {  
        short v = ((KnownValue) entity).value;  
        emit(Instruction.LOADLop, 0, 0, v);  
    } else {  
        short d = (entity instanceof UnknownValue) ?  
            ((UnknownValue) entity).address :  
            ((KnownAddress) entity).address;  
        emit(Instruction.LOADop, s, Instruction.SBr, d);  
    }  
}
```



Bisher diskutiert: Mini-Triangle

- ▶ Flache Block-Struktur
- ▶ Verschachtelte Deklarationen
- ▶ Adressierung der ...
 - ▶ globalen Variablen über `+offset [SB]`
 - ▶ lokalen Variablen über `+offset [SB]`



Nun Erweiterung auf Triangle mit Prozeduren und Funktionen

- ▶ Verschachtelte Block-Struktur
- ▶ Lokale Variablen (adressiert über `+offset [LB]`)
- ▶ Parameter (adressiert über `-offset [LB]`)
- ▶ Nicht-lokale Variablen (adressiert über `+offset [reg]`)
 - ▶ `reg` ist statisches Verkettungsregister `L1, L2, ...`

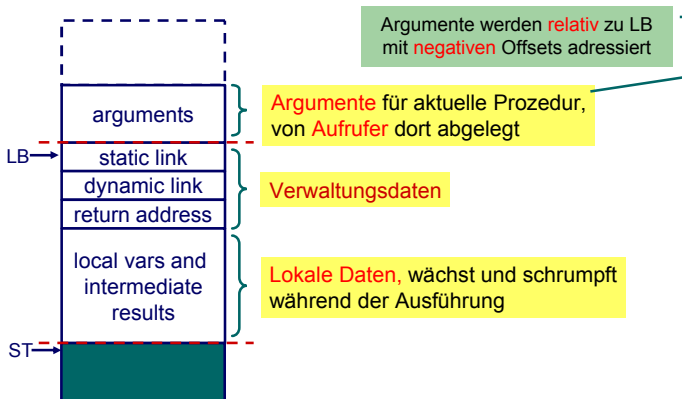


Nun Erweiterung auf Triangle mit Prozeduren und Funktionen

- ▶ Verschachtelte Block-Struktur
- ▶ Lokale Variablen (adressiert über `+offset [LB]`)
- ▶ Parameter (adressiert über `-offset [LB]`)
- ▶ Nicht-lokale Variablen (adressiert über `+offset [reg]`)
 - ▶ `reg` ist statisches Verkettungsregister `L1, L2, ...`

➡ Viele verschiedene zu verwaltende Entitäten

Wichtigste Struktur der Laufzeitumgebung: Stack Frame



Adressierung von Variablen 1

Jetzt alle Spielarten berücksichtigen

- ▶ Jede Prozedur ist auf bestimmter **Schachtelungstiefe** definiert
- ▶ Speichere zu jeder Variablen die Schachtelungstiefe der **umschließenden** Prozedur
 - ▶ **Globale** Variablen haben dabei die Tiefe 0
- ▶ Verwalte Offsets jetzt **je** Schachtelungstiefe

```
let var a: array 8 of Integer;  
var b: Char;  
proc foo() ~  
  let var c: Integer;  
  var d: Integer;  
  proc bar() ~  
    let var e: Integer;  
    in ... d:=  
  in ... d:=  
in ...
```

| var | size | address |
|-----|------|---------|
| a | 8 | (0,0) |
| b | 1 | (0,8) |
| c | 1 | (1,3) |
| d | 1 | (1,4) |
| e | 1 | (2,3) |

4 [L1]

Laufzeitadressen von Variablen
nun von **Kontext** abhängig!

4 [LB]

Adressierung von Variablen 2



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Bisher:

fetch [*I*] = LOAD (*s*) d[*SB*] where *d* is address bound to *I*
and *s* = size(type of *l*)

Adressierung von Variablen 2



Bisher:

~~*fetch [I] = LOAD(s) d[SB] where d is address bound to I
and s = size(type of I)*~~

Nun komplizierter:

Adressierung von Variablen 2



Bisher:

~~*fetch [I] = LOAD(s) d[SB] where d is address bound to I
and s = size(type of I)*~~

Nun komplizierter:

Adressierung von Variablen 2



Bisher:

~~*fetch* [I] = LOAD(s) d[SB] where *d* is address bound to I
and *s* = size(type of I)~~

Nun komplizierter:

fetch [I] = LOAD(s) d[r] *s* = size(type of I)
(level, d) is declaration address of I
if (level == 0) then *r* = SB
elif (level == currentLevel) then *r* = LB
else *r* = I(currentLevel – level)



- ▶ Bei Besuch einer Deklaration abspeichern
 - ▶ Offset innerhalb des Frames
 - ▶ Schachtelungsebene des Frames
- ▶ Angaben ersetzen nun `Short` Parameter



- ▶ Bei Besuch einer Deklaration abspeichern
 - ▶ Offset innerhalb des Frames
 - ▶ Schachtelungsebene des Frames
- ▶ Angaben ersetzen nun `Short` Parameter

```
public class Frame {  
    public byte level;  
    public byte displacement;  
}
```



Jetzt Verwaltung des belegten Speicherplatzes je Ebene

```
public class EntityAddress {
    public byte level;
    public short displacement;
}
```

```
public abstract class RuntimeEntity {
    public short size;
    ...
}

public class UnknownValue extends RuntimeEntity {
    public EntityAddress address;
    ...
}

public class KnownAddress extends RuntimeEntity {
    public EntityAddress address;
    ...
}
```



Adressvergabe und Einträgen in den DAST

```
elaborate [var! :T] = PUSH s      where s = size of T
```

```
public Object visitVarDecl(VarDecl decl, Object arg) {  
    Frame frame = (Frame)arg;  
    short s      = shortValueOf(decl.T.visit(this, null));  
    decl.entity = new KnownAddress(s, frame.level,  
                                   frame.displacement);  
  
    emit(Instruction.PUSHop, 0, 0, s);  
    return new Short(s);  
}
```




Adressvergabe und Eintragen in den DAST

```
elaborate [var! :T] = PUSH s      where s = size of T
```

```
public Object visitVarDecl(VarDecl decl, Object arg) {  
    Frame frame = (Frame)arg;  
    short s      = shortValueOf(decl.T.visit(this, null));  
    decl.entity = new KnownAddress(s, frame.level,  
                                   frame.displacement);  
  
    emit(Instruction.PUSHop, 0, 0, s);  
    return new Short(s);  
}
```

- ▶ Schachtelungstiefe `level` erhöhen bei Besuch von Prozedurdeklaration
- ▶ Offset `displacement` erhöhen bei Besuch von Var/Const-Deklaration

Zugriff auf bekannte Werte, Variablen und unbekannte Werte

```
fetch [I] = LOAD(s) d[r]    s = size(type of I)  
                           (level, d) is address of I  
                           if (level == 0) then r = SB  
                           elif (level == currentLevel) then r = LB  
                           else r = I(currentLevel - level)
```

```
public Object encodeFetch(Vname name, Frame frame, short s) {  
    RuntimeEntity entity =  
        (RuntimeEntity) name.visit(this, null);  
    if (entity instanceof KnownValue) {  
        short v = ((KnownValue) entity).value;  
        emit(Instruction.LOADLop, 0, 0, v);  
    } else {  
        EntityAddress address =  
            (entity instanceof UnknownValue) ?  
                ((UnknownValue) entity).address :  
                ((KnownAddress) entity).address;  
        emit(Instruction.LOADop, s,  
            displayRegister(frame.level, address.level),  
            address.displacement);  
    }  
}
```

Frame der
aktuellen Prozedur

Einfache Berechnung des Basisregisters
der Frame von name.

Einfachster Fall: **Globale** Prozeduren, keine Parameter, kein Ergebnis

| | | | |
|-------------|---|--|----------|
| Declaration | ::= ... | | |
| | <code>proc Identifier () ~ Command</code> | | ProcDecl |
| Command | ::= ... | | |
| | <code>Identifier ()</code> | | CallCmd |

```
elaborate [proc I () ~ C]  
=      JUMP      g  
      e: execute [C]  
      RETURN (0) 0  
      g:
```

```
execute [I ()]  
= CALL (SB) e
```

e ist Startadresse der
Prozedur *I*

Globale Funktionen **identisch** bis auf
Rückgabewert mit Größe <> 0

Verschachtelte Unterprogramme 1



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Bei Aufruf von Y statische Verkettung auf umschliessende Prozedur X .

Verschachtelte Unterprogramme 1



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Bei Aufruf von Y statische Verkettung auf umschliessende Prozedur X .
↳ Gleiches Vorgehen wie bei lokalen Variablen



Bei Aufruf von Y statische Verkettung auf umschliessende Prozedur X .

↳ Gleiches Vorgehen wie bei lokalen Variablen

execute [I ()]

= CALL (r) e ($level, e$) is routine bound to I
if ($level == 0$) then $r = SB$
elif ($level == currentLevel$) then $r = LB$
else $r = L(currentLevel - level)$

Speichere Startadressen von Prozeduren und Funktionen als Paar ($level$, start address) in Klasse **KnownRoutine**, einer Subklasse von **RuntimeEntity**, ab.



Behandlung des Prozeduraufrufes

```
execute [I()]  
    = CALL(r) e    (level, e) is routine bound to I  
                    if (level == 0) then r = SB  
                    elif (level == currentLevel) then r = LB  
                    else r = I(currentLevel - level)
```

```
public Object visitCallCmd(CallCmd cmd, Object arg) {  
    Frame frame = (Frame) arg;  
  
    EntityAddress address =  
        ((KnownRoutine) cmd.I.decl.entity).address;  
  
    emit(Instruction.CALLOp, s,  
        displayRegister(frame.level, address.level),  
        address.displacement);  
}
```

Verweis auf Prozedurdeklaration ist
gespeichert im **decl**-Feld des für das
CallCmd verwendeten Bezeichners

Behandlung der Deklaration



```
elaborate [proc I () ~ C] =           JUMP      g  
                                         e: execute [C]  
                                         RETURN(0) 0  
                                         g:
```

```
public Object visitProcDecl(ProcDecl decl, Object arg) {  
    Frame outerFrame = (Frame)arg;  
  
    short j = nextInstrAddr;  
    emit(Instruction.JUMPop, 0, Instruction.CBr, 0);  
  
    short e = nextInstrAddr;  
    decl.I.entity = new KnownRoutine(outerFrame.level, e);  
    Frame localFrame = new Frame(outerFrame.level+1, 3);  
  
    decl.C.visit(this, localFrame);  
    emit(Instruction.RETURNop, 0, 0, 0);  
  
    short g = nextInstrAddr;  
    code[j].d = g; _____  
    return new Short(0);  
}
```

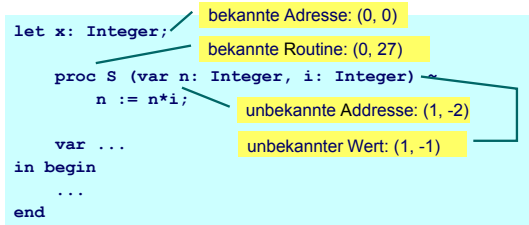
Offset der ersten lokalen Variable

Nachtragen der Sprungadresse



- ▶ **Aufrufer** legt aktuelle Parameter auf Stack
- ▶ **Gerufener** greift mit negativem Offset via `LB` auf Parameter zu
- ▶ **Wertparameter**: Handhabung als **unbekannter Wert**
- ▶ **Variablenparameter**: Handhabung als **unbekannte Adresse**

- ▶ **Aufrufer** legt aktuelle Parameter auf Stack
- ▶ **Gerufener** greift mit negativem Offset via `LB` auf Parameter zu
- ▶ **Wertparameter**: Handhabung als **unbekannter Wert**
- ▶ **Variablenparameter**: Handhabung als **unbekannte Adresse**



Behandlung von Parametern 2



```
Declaration ::= ...
             | proc Identifier (Formal) ~ CommandProcDecl

Command ::= ...
          | Identifier (Actual) CallCmd

Formal ::= Identifier : TypeDenoter
        | var Identifier : TypeDenoter

Actual ::= Expression
        | var Vname
```

Hier vereinfacht:
Nur ein Parameter

```
execute [I (AP)]
    = pass-argument [AP]
      CALL (SB) e
```

```
pass-argument [E]
    = evaluate [E]
```

```
pass-argument [var V]
    = fetch-address [V]
```

wobei *fetch-address* Code zur Bestimmung der Adresse einer Variablen ausgibt



Variablenparameter

- ▶ werden mit der `UnknownAddress` Subklasse von `RuntimeEntity` behandelt
- ▶ Die `fetch` und `assign`-Schablonen müssen erweitert werden

Variablenparameter

- ▶ werden mit der `UnknownAddress` Subklasse von `RuntimeEntity` behandelt
- ▶ Die `fetch` und `assign`-Schablonen müssen erweitert werden

```
fetch [I] = // KnownValue, KnownAddress Fälle nicht gezeigt
...
LOAD (1)  d[r]  if I is bound to an UnknownAddress
LOADI (s)  where
              s = size(type of I)
              (level, d) is address of I
              if (level == 0) then r = SB
              elif (level == currentLevel) then r = LB
              else r = I(currentLevel - level)
```

d wird negativ sein

nicht möglich!

Auch **innere** Prozeduren können auf **formale** Parameter zugreifen!



- ▶ Code-Selektion, -Funktionen, -Schablonen
- ▶ Implementierung als Visitor
- ▶ Zugriff auf bekannte/unbekannte Werte/Adressen
- ▶ Adressvergabe
 - ▶ Statische Blockstruktur
 - ▶ Dynamisch auf Stack
- ▶ Prozeduren
 - ▶ Deklaration
 - ▶ Parameterübergabe