

Einführung in den Compilerbau

Code-Generierung



TECHNISCHE
UNIVERSITÄT
DARMSTADT

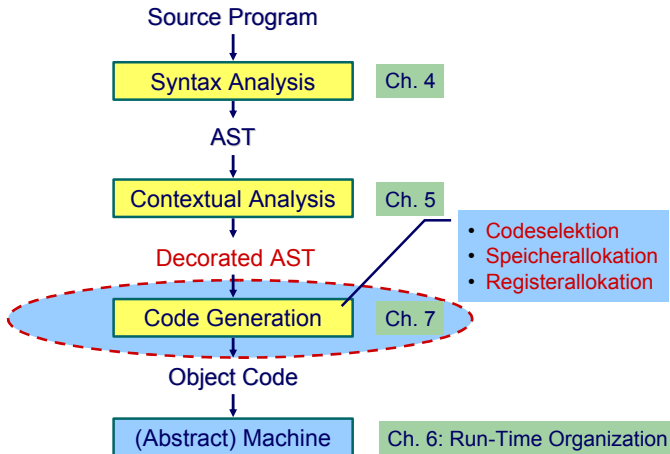
WS 2018/19

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt



Embedded Systems & Applications



- ▶ 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
```

➔ 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, ...

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.

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.



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

Anweisungsfolge C1; C2

Semantik: Führe erst C1 aus, dann C2.

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

Beispiel: Code-Funktion 2

Zuweisung $I := E$

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

$execute [[I := E]] =$
 $evaluate[[E]]$
STORE a , mit a =Adresse von Variable I

Beispiel: Code-Funktion 3



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

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

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

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

$$\begin{aligned} f_P [[\dots Q \dots R \dots]] = \\ \dots \\ f_Q [[Q]] \\ \dots \\ f_R [[R]] \\ \dots \end{aligned}$$



- ▶ 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

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:
```

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

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

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

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

- ▶ 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



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



```
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

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



Inlining 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) 1
```

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
- ▶ **Achtung:** Aus Gründen der Übersichtlichkeit wird der Code hier etwas vereinfacht dargestellt
- ▶ Der vollständigen Code-Generator steht als Teil des Triangle-Compilers auf der Website der Vorlesung zur Verfügung

Repräsentation vom TAM-Instruktionen



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

public class Machine {
    public final static int
        LOADop = 0, LOADAop = 1, LOADIop = 2, ...; // op-codes (Table C.2)
    public final static int
        CBr = 0, CTr = 1, PBr = 2, ...; // register numbers (Table C.1)

    public final static int maxCodeSize = 32767 - maxPrimitives;
    public static Instruction[] code = new Instruction[maxCodeSize];
}

public class Interpreter { ... }
```

package TAM;

Erzeugen vom TAM-Instruktionen

```
public class Encoder {  
    ... // visitor classes  
  
    // Appends an instruction to the object code.  
    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 int nextInstrAddr;  
}
```

```
package Triangle.CodeGenerator;
```



Beispiel: Generiere Code für gesamtes Programm

```
public class Encoder {  
    ...  
    private class ProgramEncoder  
        extends VisitorBase<Void, Void> {  
  
        public Void visitProgram(Program ast, Void __) {  
            ast.C.visit(commandEncoder, null);  
            emit(Machine.HALTop, 0, 0, 0);  
            return null;  
        }  
    }  
}  
    ...  
}
```




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

Getrennt realisieren

```
public class Encoder {  
    ...  
    private void encodeFetch(Vname v, int valSize, ...) {  
        // as specified by the fetch code template  
    }  
  
    private void encodeStore(Vname v, int valSize, ...) {  
        // as specified by the store/assign code template  
    }  
}
```

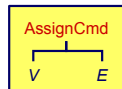
... aber nicht **in** einem Visitor, sondern **für** Visitor benutzbar.

Beispiel Benutzung von VName 1



Ziel einer Zuweisung

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



```
public class Encoder {  
    ...  
    private class CommandEncoder extends VisitorBase<Void, Void> {  
        public Void visitAssignCommand(AssignCommand ast, Void __) {  
            int valSize = ast.E.visit(expressionEncoder, null);  
            encodeStore(ast.V, valSize);  
            return null;  
        }  
        ...  
    }  
}
```

Beispiel Benutzung von VName 2



Innerhalb eines Ausdrucks

```
public class Encoder {  
    ...  
    private class ExpressionEncoder extends VisitorBase<Void, Void> {  
        public Void visitVnameExpression(VnameExpression ast, Void __) {  
            int valSize = ast.type.visit(typeEncoder, null);  
            encodeFetch(ast.V, valSize);  
            return null;  
        }  
        ...  
    }  
}
```



Integer-Literale

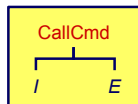
```
public class Encoder {  
    ...  
    private class ExpressionEncoder extends VisitorBase<Void, Void> {  
        public Void visitIntegerExpression(IntegerExpression ast, Void __) {  
            emit(Machine.LOADLop, 0, 0, Integer.parseInt(ast.IL.spelling));  
            return null;  
        }  
        ...  
    }  
}
```



Vereinfacht für Mini-Triangle

- ▶ Nur primitive Funktionen
- ▶ Mit maximal einem Parameter

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

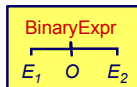


```
// CommandEncoder  
public Void visitCallCmd(CallCmd cmd, Void __) {  
    cmd.E.visit(expressionEncoder, null);  
    int p = // Adresse der primitiven Routine zu cmd.I  
    emit(Machine.CALLOp, Machine.SBr, Machine.PBr, p);  
    return null;  
}
```



Gleicher Mechanismus wie Prozeduraufruf

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



```
// ExpressionEncoder
```

```
public void visitBinaryExpression(BinaryExpression ast, Void __) {  
    ast.E1.visit(expressionEncoder, null);  
    ast.E2.visit(expressionEncoder, null);  
    ast.O.visit(operatorEncoder, null);  
    return null;  
}
```

```
// TerminalEncoder
```

```
public void visitOperator(Operator op, Void __) {  
    int displacement = // Adresse fuer primitive Routine  
    emit(Machine.CALLop, Machine.SBr, Machine.PBr, displacement);  
    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
- ▶ 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:
```

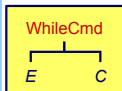
➔ “Nachbessern” bereits generierten Codes (*backpatching*)

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:
```



```
// CommandEncoder
```

```
public Void visitWhileCommand(WhileCommand cmd, Void __) {  
    int lwhile = nextInstrAddr;  
    cmd.E.visit(expressionEncoder, null);  
    int jump2end = nextInstrAddr;  
    emit(Machine.JUMPIFop, 0, Machine.CBr, 0);  
    cmd.C.visit(commandEncoder, null);  
    emit(Machine.JUMPop, 0, Machine.CBr, lwhile);  
    int lend = nextInstrAddr;  
    code[jump2end].d = lend;  
}
```

Backpatching



Doppeltes Backpatching bei Verzweigung (if/then/else)

```
// CommandEncoder
```

```
public Void visitIfCommand(IfCommand cmd, Void __) {  
    ast.E.visit(expressionEncoder, null);  
    int jumpifAddr = nextInstrAddr;  
    emit(Machine.JUMPIFop,  
         Machine.falseRep,  
         Machine.CBr, 0);  
  
    ast.C1.visit(commandEncoder, null);  
    int jumpAddr = nextInstrAddr;  
    emit(Machine.JUMPop, 0, Machine.CBr, 0);  
    patch(jumpifAddr, nextInstrAddr);  
  
    ast.C2.visit(commandEncoder, null);  
    patch(jumpAddr, nextInstrAddr);  
    return null;  
}
```

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

$execute [let D in C] = elaborate [D]$
 $execute [C]$
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

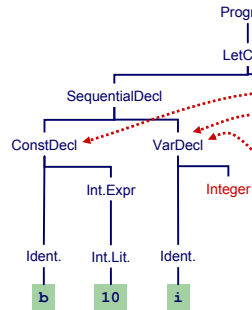
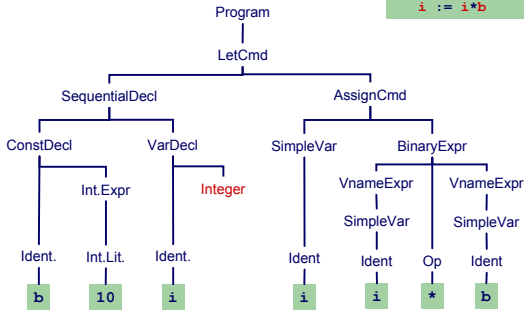
Ziel: Bestimme d in

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

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

Beispiel Konstanten und Variablen

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



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

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 int size;
    ...
}
public class KnownValue extends RuntimeEntity {
    public int value;
    ...
}
public abstract class AddressableValue extends RuntimeEntity {
    public int address;
    ...
}
public class UnknownValue extends AddressableValue { ... }
public class KnownAddress extends AddressableValue { ... }

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

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
- ▶ In Abhängigkeit von Deklarationen und Geltungsbereichen
- ▶ Implementierung: Erweitern der relevanten Teil-Visitors (z.B. ExpressionEncoder)
 - ▶ Verwende Parameter Integer arg zur Eingabe des **aktuell** belegten Speicherplatzes
 - ▶ Verwende Funktionsergebnis zur Rückgabe des **zusätzlich** benötigten Speicherplatzes

```
private class ExpressionEncoder extends VisitorBase<Integer, Integer> {  
    public Integer visitXYZ(XYZ xyz, Integer arg) { ... }  
}
```

Integer-Wert mit
zusätzlich benötigtem Platz

Integer-Wert mit
bisher benötigtem Platz

Allgemeines Schema

Weitergabe der **bisherigen** Belegung in gs

```
public Integer visit...Expression(..., Integer gs) {  
    ...  
}
```

➔ Ist auch nächste **freie** Adresse!

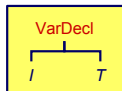
Weitergabe der **Erhöhung** des Speicherbedarfs im Ergebnis

```
public Integer visit...Declaration(..., Integer gs) {  
    ...  
    gs += moreMemRequired;  
    ...  
    return moreMemRequired;  
}
```



Elaboriere Variablendeklaration

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



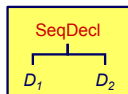
```
public Integer visitVarDeclaration(VarDeclaration decl,
    Integer gs) {
    int s = decl.T.visit(typeEncoder, null);
    decl.entity = new KnownAddress(s, gs); ←
    emit(Machine.PUSHop, 0, 0, s);
    return s;
}
```

Merke die *Größe* und *Adresse* der Variablen



Elaboriere Folge von Deklarationen

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



```
public Integer visitSequentialDeclaration(SequentialDeclaration
    decl, Integer gs) {
    int s1 = decl.D1.visit(entityEncoder, gs);
    int s2 = decl.D2.visit(entityEncoder, gs+s1);
    return 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 Void visitLetCommand(LetCommand cmd, Integer gs) {  
    int s = ast.D.visit(entityEncoder, gs);  
    ast.C.visit(commandEncoder, gs+s);  
    if (s > 0)  
        emit(Machine.POPop, 0, 0, s);  
    return null;  
}
```




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)
```

```
private void encodeFetch(Vname name, int s) (  
    RuntimeEntity entity = name.visit(vnameEncoder, null);  
  
    entity.visit(new RuntimeEntityVisitorBase<Void, Void> {  
        public Void visitKnownValue(KnownValue ent, Void __) {  
            emit(Machine.LOADLop, 0, 0, ent.value /* v */);           return null;  
        }  
        public Void visitAddressableValue(AddressableValue ent, Void __) {  
            emit(Machine.LOADop, s, Machine.SBr, ent.address /* d */); return null;  
        }  
    }, null);  
}
```



Bisher diskutiert: Mini-Triangle

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

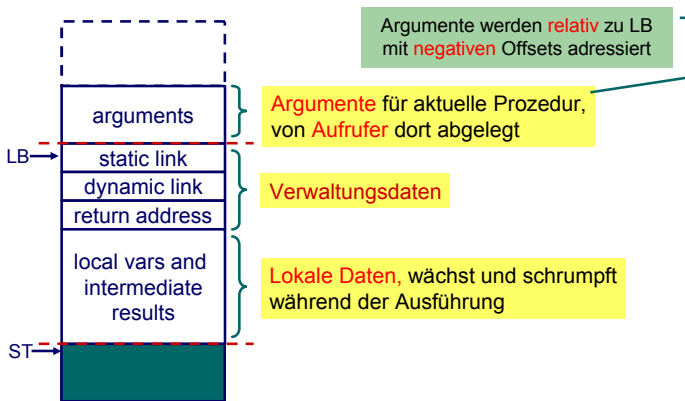


Nun Erweiterung auf Triangle mit Prozeduren und Funktionen

- ▶ 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, ...

➡ 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



Bisher:

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

~~*fetch* [**I**] = LOAD (**s**) d[**SB**] where
and~~

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 Integer Parameter

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




Jetzt Verwaltung des belegten Speicherplatzes je Ebene

```
public class ObjectAddress {
    int level;
    int displacement;
}

public abstract class RuntimeEntity {
    int size;
    ...
}

public abstract class AddressableValue extends RuntimeEntity {
    ObjectAddress address;
    ...
}

public class UnknownValue extends AddressableValue { ... }
public class KnownAddress extends AddressableValue { ... }
```

Adressvergabe und Eintragen in den DAST

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

```
public Integer visitVarDeclaration(VarDeclaration decl, Frame frame) {  
    int s = decl.T.visit(typeEncoder, null);  
    decl.entity = new KnownAddress(s, frame.level,  
                                   frame.displacement);  
  
    emit(Machine.PUSHop, 0, 0, s);  
    return s;  
}
```

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

Handhabung im Code-Generator 3

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 = L(currentLevel - level)
```

```
private void encodeFetch(Vname name, Frame frame, int s) (
    RuntimeEntity entity = name.visit(vnameEncoder, null);

    entity.visit(new RuntimeEntityVisitorBase<Void, Void> {
        public Void visitKnownValue(KnownValue ent, Void __) { /* wie bisher */ }
        public Void visitAddressableValue(AddressableValue ent, Void __) {
            ObjectAddress address = ent.address;
            emit(Machine.LOADop, s,
                displayRegister(frame.level, address.level) ←
                address.displacement);
            return null;
        }
    }, null);
}
```

Frame der aktuellen Prozedur

Einfache Berechnung des Basisregisters des Frames 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



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 (*x*) *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 Void visitCallCommand(CallCommand ast, Frame frame) {  
    // ... visit cmd.I ...  
    ObjectAddress address = ((KnownRoutine) cmd.I.decl.entity).address; ←  
  
    emit(Machine.CALLOp, s  
         displayRegister(frame.level, address.level),  
         address.displacement);  
  
    return null;  
}
```

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

Behandlung der Deklaration

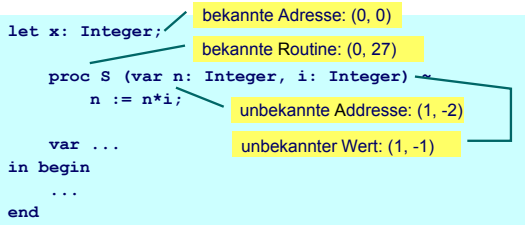
```
elaborate [proc I () ~ C] =  
    e: JUMP g  
      execute [C]  
      RETURN (0) 0  
    g:
```

```
public Integer visitProcDeclaration(ProcDeclaration decl, Frame outerFrame) {  
    int j = nextInstrAddr;  
  
    emit(Machine.JUMPop, 0, Machine.CBr, 0);  
  
    int e = nextInstrAddr;  
    decl.I.entity = new KnownRoutine(outerFrame.level, e);  
    Frame localFrame = new Frame(outerFrame.level + 1, Machine.linkDataSize);  
  
    decl.C.visit(commandEncoder, localFrame);  
    emit(Machine.RETURNop, 0, 0, 0);  
  
    int g = nextInstrAddr;  
    patch(j, g);  
    return 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**



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

```
fetch [I] = // KnownValue, KnownAddress Fälle nicht gezeigt
```

```
...
```

```
LOAD (1)  d[r]  
LOADI (s)
```

d wird negativ sein

if *I* is bound to an *UnknownAddress*
where

s = *size*(type of *I*)

(*level*, *d*) is address of *I*

nicht möglich!

if (*level* == 0) then *r* = *SB*

elif (*level* == *currentLevel*) then *r* = *LB*

else *r* = *I*(*currentLevel* - *level*)

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