

# Subrange Variablen, Arrays und For Schleifen für IML

## Schlussbericht

Benjamin Meyer, Christian Scheller und Tobias Bollinger

Compilerbau, HS 2016, Team 12

### Abstract

IML zum Vorbild wurde die eigene Programmiersprache H-IML mit zugehörigem Compiler (in Haskell für die Haskell IML-VM) sowie passenden Beispiel-Programmen entwickelt. Der Fokus in H-IML war IML mit folgenden Kernfeatures auszubauen: Arrays mit frei wählbarer Index-Palette, einen Subrange Variablen Mechanismus ("Variable-Clamp"), der die Werte auf ihre jeweiligen Maxima und Minima klemmt, und eine passende For-Schleife, die mit dem Variable-Clamp harmoniert. Während der Entwicklung von H-IML wurden zusätzliche Anpassungen an der IML Sprache wie auch der VM vorgenommen.

### Wer hat was gemacht

Während der Arbeit haben wir uns online getroffen und generell zusammen an allem gearbeitet. Es ergab sich jedoch für jedes Thema ein oder mehrere Verantwortliche.

Thema	Teil-Thema	Verantwortlicher
Parser	Funktionen, While, If, Statements	Benjamin Meyer
	Expression, Operation, Factor, For	Christian Scheller
	Variable-Clamp, Array, AST visualisieren	Tobias Bollinger
Code Generierung	Funktionen, For	Benjamin Meyer
	While, Array	Christian Scheller
	If, Variable-Clamp, For, Array	Tobias Bollinger
Checker	Alles	Christian Scheller

Eine feinkörnigere Übersicht über die einzelnen Arbeiten der jeweiligen Personen finden Sie auf Github in der History (metataro = Christian Scheller, swordbreaker = Tobias Bollinger, benikm91 = Benjamin Meyer) [https://github.com/metataro/CPIB\\_ILMCompiler](https://github.com/metataro/CPIB_ILMCompiler)

## Implementierte Features in H-IML

Name	Beschreibung
Programm	Ein Programm kann man als Funktion ansehen, dessen "In" Werte zu Beginn vom User eingelesen werden und dessen "Out" Werte am Ende ausgegeben werden.
Funktionen	Prozeduren und Funktionen wurden zusammengefasst.
Schleifen	While und For Schleifen wurden implementiert.
Branch-Befehle	If und If-Else Statements wurden implementiert
Variablen und Sichtbarkeit	Variablen können als Parameter oder lokal in einem Befehlsblock deklariert werden. Funktionsparameter sind nur in der jeweiligen Funktion sichtbar. Programmparameter sind nur im Hauptprogramm sichtbar. Die Sichtbarkeit einer lokalen Variable begrenzt sich immer auf den Block sowie dessen Kind-Blöcke in welchem die Variable deklariert wurde. Die Funktionen sind global sichtbar. Variablen werden bei ihrer Deklaration auf ihren Default-Wert gesetzt.
Typen	int: Integer mit 32 Bits Subrange int: int mit definierbaren Teilbereich int Array: Ein Array von int

## Entfernte Features von IML

Folgende Features wurden wegen anderen Philosophien oder Zeitgründen nicht in H-IML implementiert.

Name	Beschreibung
Globale Variablen	Globale Variablen bringen in IML nur den Vorteil, dass Funktionen und das Programm Sicht auf dieselben Variablen haben. In H-IML gilt: Braucht eine Funktion Sicht auf eine Variable des Programms, soll diese per Parameter übergeben werden.
Init	Werte werden bei uns immer auf einen Default Wert initialisiert. Dies erhöht die generierten VM-Instruktionen, dafür erleichtert es das Programmieren.

By reference	Parameter werden in H-IML immer by value übergeben (out und inout Parameter werden nach dem Funktionsaufruf zurückgeschrieben). Dieses Feature wurde aus Zeitgründen nicht implementiert.
--------------	--

## Kernfeatures

IML wurde in H-IML mit folgenden Kernfeatures ausgebaut.

### Subrange Variable (Variable-Clamp)

#### Syntax

```
var a: int($MINIMUM .. $MAXIMUM); // a = $MINIMUM
```

#### Grundidee

Der Wertebereich von Typen kann mit einem \$MINIMUM und \$MAXIMUM in einen Teilbereich begrenzt werden. Dadurch kann klarer definiert werden, was eine Variable enthalten kann. Zudem ist eine treffendere Beschreibung möglich, was eine Funktion entgegennimmt und/oder zurückgibt.

Ein Over- bzw. Underflow von Subrange Variablen kann auf unterschiedliche Weise gehandhabt werden (Siehe Mögliche Erweiterungen), wir haben uns für die Clamp Variante entschieden. Clamp ist für spezifischen Fällen (wenn wir uns nicht für den Over- bzw. Underflow eines Wertes interessieren) nützlich, beispielsweise in der Bildverarbeitung beim Umgang mit Farbwerte von 0 bis 255.

#### Regeln

<b>Default Wert</b>	Der Default Wert einer Clamp-Variable ist sein \$MINIMUM. => In Range Garantie (\$MINIMUM aus Teilbereich ist willkürlich gewählt)
<b>Over- / Underflow</b>	Bei unter- bzw. überschreiten dieser Werte, wird der Wert auf das \$MINIMUM bzw. das \$MAXIMUM gesetzt. => In Range Garantie

## Garantien

<b>In Range</b>	Eine Clamp-Variable enthält einen Wert zwischen \$MINIMUM und \$MAXIMUM
-----------------	---

## Beispiel

```
var a: int(0 .. 100); // a = 0
a := 99;           // a = 99
a := a + 1;        // a = 100;
a := a + 1;        // a = 100;
a := 1;            // a = 1
a := a - 1;        // a = 0
a := a - 1;        // a = 0
var b: int(5 .. 100); // b = 5
```

## Vergleich

Subrange Variablen werden in verschiedenen Programmiersprachen verwendet wie z.B. in Pascal. Dort ist das Default Verhalten für einen Over- bzw. Underflow eine Exception zu werfen.

## Statischer Int Array

### Syntax

```
var a: int[$START .. $ENDE];
```

### Grundidee

Ein statischer Array geht im Speicher von seiner Basisadresse aus von 0 bis ans Ende des Arrays. In den meisten Programmiersprachen ist die Index-Palette ([..]) des Arrays ebenfalls fix von 0 bis ans Ende des Arrays - dies wollen wir verbessern.

Bei der Array Deklaration gibt man den \$START und das \$ENDE der Index-Palette mit, wodurch die Länge des Arrays gegeben ist. Ein Zugriff auf ein Element im Array ist nur durch diese Index-Palette möglich, oder aber führt zu einem Kompilationsfehler (bei statischem Index) oder undefiniertem Verhalten zur Laufzeit (bei dynamischen Index).

Aus Einfachheitsgründen haben wir nur den Int Array umgesetzt.

## Regeln

<b>Statische Arrays</b>	\$START und \$ENDE müssen Int-Literale sein. => Benötigter Speicherplatz zur Kompilationszeit bekannt. => Zugriffsgarantie
<b>Default Wert der Elemente</b>	Alle Elemente der Arrays werden mit dem Int-Default-Wert 0 initialisiert. => Erleichtert den Umgang mit Arrays.
$\$START \leq \$ENDE$	Der \$START Wert muss kleiner gleich dem \$ENDE Wert sein.

## Garantien

<b>Zugriffsgarantie</b>	Für alle $k \leq (\$ENDE - \$START)$ gilt: $a[\$START + k]$ greift auf das kte Element im Array zu.
-------------------------	---

## Beispiel

```
var a:int[ 0 .. 100]; // Array von index 0 bis und mit index 100
var b:int[ 50 .. 100]; // Array von index 50 bis und mit index 100
var c:int[-100 .. 100]; // Array von index -100 bis und mit index 100
a[0] := 100;
a[1] := 50;
b[50] := 50;
c[-100] := 50;
// a[101] => Compile Fehler;
// b[49] => Compile Fehler;
// c[-101] => Compile Fehler;
var i : int;
i := 101;
//a[i] => Laufzeitfehler (nicht implementiert)
```

## Vergleich

Dieselbe Idee ist bereits in der Programmiersprache Pascal implementiert.

## For Schleife

### Syntax

```
var $CLAMP : int($MINIMUM .. $MAXIMUM);  
$CLAMP := $START_WERT;  
for ($CLAMP) { ... }
```

### Grundidee

Eine For Schleife sollte (anders als in Java) nur eine fixe Anzahl an Durchläufen haben (LOOP-berechenbar) und weniger mächtig sein als eine While-Schleife (WHILE-berechenbar). Das Design der For-Schleife mit Clamp Variablen propagieren einen solchen Programmierstil.

### Regeln

<b>Hochzählen</b>	Die For-Schleife nimmt die gegebenen Clamp-Variable und zählt sie bis zu seinem Maximum hoch und bricht dann ab. => \$CLAMP = \$MAXIMUM => Minimum ein Durchlauf
-------------------	--

### Garantien

<b>\$CLAMP = \$MAXIMUM</b>	Am Ende der For-Schleife hat die \$CLAMP Variable den Wert \$MAXIMUM
<b>Minimum ein Durchlauf</b>	Eine For-Schleife läuft im Minimum einmal durch, da eine Subrange Variable mindestens einen möglichen Wert haben muss.

### Beispiel

```
int sum := 0;  
var i: int(5 .. 100);  
for (i) { // von 5 bis und mit 100  
    sum := sum + i;  
}  
// sum = 5 + 6 + ... + 99 + 100  
// i = 100  
  
i := 50;  
For (i) { ... } // von 50 bis und mit 100
```

# Mögliche Erweiterungen

Alle möglichen Erweiterungen sind nicht Teil von der jetzigen Version von H-IML und dienen allein als Anregungen für zukünftige Versionen.

## Subrange Variable - Verschiedenste Default Verhalten

### Syntax

```
var a = int($MINIMUM..$MAXIMUM, $VERHALTEN);
```

### Grundidee

Unser Clamp-Default-Verhalten bei Over- bzw. Underflow einer Subrange Variable ist nur eines vieler möglichen Verhaltensweisen. Beispielsweise könnte auch eine Exception geworfen werden, um den Programmierer darüber zu informieren. Wir wollen diesen Entscheid über das Verhalten dem Programmierer geben.

Mögliche Verhalten könnten sein:

- Exception: Over- bzw. Underflow Exception werfen
- Clamp: Das implementierte Verhalten.
- Ping-Pong: Bei einem Over- bzw. Underflow um den Wert d, wird vom Maximum bzw. Minimum -d addiert.
- Loop: Die Variable bildet einen endlichen Körper.

### Beispiel

```
var e = int(0..100, exception);  
e := -1; // Kompilation Fehler  
e := d; // Wenn d < 0 oder > 100 => Laufzeitfehler  
var p = int(0..100, ping-pong);  
p := 92; // p := 92  
p := p + 10; // p := 98  
var l = int(-100..100, loop);  
l := 100; // l := 100  
l := l + 1; // l := -100
```

## Foreach

### Syntax

```
var $ARRAY = int[$START..$ENDE];  
for (var $NEXT_ELEMENT <- $ARRAY) { ... }
```

### Grundidee

Wir wollen den Umgang mit Arrays vereinfachen. Die Idee der For-Each-Schleife ist, dass sie jedes Element eines Arrays einmal durchlaufen wird: Vom \$START bis und mit dem \$ENDE werden alle Elemente des Arrays nach und nach in \$NEXT\_ELEMENT geschrieben und der Schleifenblock durchgeführt.

### Regeln

<b>Hochzählen</b>	Die For-Each-Schleife nimmt einem Array entgegen und gibt jedes Element einzeln aus. => Fixe Anzahl Durchläufe
-------------------	---

### Garantien

<b>Fixe Anzahl Durchläufe</b>	Eine For-Each-Schleife läuft genau $\$ENDE - \$START + 1$ Mal durch.
-------------------------------	--

### Beispiel

```
var sum: int;  
sum := 0;  
for (var elem <- a) {  
    sum := sum + elem;  
}  
// sum ist die Summe von allen Elementen in a.
```



## Open Issues in H-IML

Name	Beschreibung
Arrays als Funktionsparameter	Nicht implementiert.
Arrays für beliebige Typen (in H-IML z.B. auch für Clamp-Variablen)	Nicht implementiert.
Rechts nach links Evaluierung von Ausdrücken ändern zu links nach rechts Evaluierung	Ein Ausdruck wie: 5 - 5 + 5 wird als 5 - (5 + 5) evaluiert und ergibt den Wert -5 und nicht den Wert 5 (wie ein Programmierer erwarten würden). Dies könnte man noch ändern.
ChangeMode	Im Parser implementiert. Im Checker nicht implementiert.

## Implementierung

Unsere Implementierung besteht aus einer Grammatik (Siehe Anhang), Änderungen an der VM und drei Hauptelementen: Der Parser, die Checker und der CodeGenerator.

## Änderungen an der VM

### readBool, readInteger

readBool und readInteger wurden implementiert.

<pre> 19 -- currently defined in Scanner 20 readBool :: String -&gt; Maybe Bool 21 - readBool s = Just \$ (read s :: Bool) 22 23 readInteger :: String -&gt; Maybe Integer 24 - readInteger s = Just \$ (read s :: Integer) 25 </pre>	<pre> 19 -- currently defined in Scanner 20 readBool :: String -&gt; Maybe Bool 21 + readBool = error "not yet implemented" 22 23 readInteger :: String -&gt; Maybe Integer 24 + readInteger = error "not yet implemented" 25 </pre>
---	--

## Debug

readS und updateS wurden so abgeändert, dass bei illegaler Adresse, die Adresse die Länge vom Stack und seinen Inhalt als Fehler ausgegeben wird. Dies erleichterte uns das Bugfixing.

```
readS :: Stack -> StoreAddress -> VmValue
+ readS stack addr =
+   | addr < 0 = error ("read: invalid stack address | address: " ++ show addr ++ " | s
+   | ((length stack - 1) - addr) < 0 = error ("read: invalid stack address | address:
+   | otherwise = stack !! ((length stack - 1) - addr)

+
+ updateS :: Stack -> (StoreAddress, VmValue) -> Stack
+ updateS stack (addr, val) =
+   | addr < 0 = error ("update: invalid stack address | address: " ++ show addr ++ " |
+   | ((length stack - 1) - addr) < 0 = error ("update: invalid stack address | address
+   | otherwise = stack'
```

Wir haben die “internal Error” Nachricht mit dem übergebenen State ergänzt.

```
execInstr instr state =
  · return (Left (ErrorMsg ([], "internal error VM state: " ++ show state ++ " : " ++ show instr)))
```

Wir haben dazu noch eine “debugProgrammStack” Funktion hinzugefügt welche uns am Ende den Stack zurückgibt.

## MoveSpUp

Die von uns hinzugefügte Instruktion “MoveSpUp” wandert den Stack um den mitgelieferten Int hoch.

```
execInstr :: Instruction -> State -> IO (Check State)
execInstr (MoveSpUp size) (pc, fp, stack) =
  · return2 (pc + 1, fp, drop size stack)
```

## Input

Wir haben den Input Befehl der VM umgeschrieben. Er legt den eingegebenen Wert einfach auf den Stack. Zuvor führte der Input Befehl zudem direkt ein Store aus.

```
+execInstr (Input BoolTy | loc indicator) (pc, fp, stack) =
  ..do putStr ("? " ++ indicator ++ " : " ++ show BoolTy ++ " = ");
  .....inputString <- getLine
  .....case readBool inputString of
  .....Nothing ->
  .....return (Left (ErrorMsg ([loc], "input: not a boolean literal")))
  .....Just b ->
  .....let inputVmVal = (IntVmVal . boolToInt) b
+ .....stack' = inputVmVal : stack
  .....in return2 (pc + 1, fp, stack')
+execInstr (Input (IntTy 32) loc indicator) (pc, fp, stack) =
  ..do putStr ("? " ++ indicator ++ " : " ++ show (IntTy 32) ++ " = ");
  .....inputString <- getLine
  .....case readInteger inputString of
  .....Nothing ->
  .....return (Left (ErrorMsg ([loc], "input: not an integer literal")))
  .....Just i ->
  .....case fromIntegerToInt32 i of
  .....Left Overflow ->
  .....return (Left (ErrorMsg ([loc], "input: overflow of 32 bit")))
  .....Right result ->
+ .....let stack' = Int32VmVal result : stack
  .....in return2 (pc + 1, fp, stack')
```

## Parser

Den Parser haben wir mit der Haskell Library “Parsec” umgesetzt und damit den Scanner und den Parser gemeinsam implementiert.

Unser Parser generiert direkt einen Abstrakten Syntaxbaum. Folgende Struktur hat unser generierter Baum. Ein Beispiel Programm und der dazu generierte Baum befindet sich im Anhang.

```
data IMLType = Int
  ... | ClampInt Int Int -- min max
  ... | ArrayInt Int Int -- min max
  ... deriving Show

data IMLFlowMode = In | Out | InOut
  ... deriving Show

data IMLChangeMode = Const | Mutable
  ... deriving Show

data IMLOperation = Times | Div | Mod | Plus | Minus | Lt | Ge | Eq | Ne | Gt | Le | And | Or | Not
  ... deriving Show

data IMLLiteral = IMLBool Bool | IMLInt Int
  ... deriving Show

data IMLVal = Program IMLVal [IMLVal] [IMLVal] [IMLVal] SourcePos -- ident [paramDeclarations] [functionDeclarations] [statements]
  ... | Ident String SourcePos -- name
  ... | IdentDeclaration IMLChangeMode IMLVal IMLType SourcePos -- changeMode (ident | identArray) type
  ... | ParamDeclaration IMLFlowMode IMLChangeMode IMLVal IMLType SourcePos -- flowMode changeMode (ident | identArray) type
  ... | IdentFactor IMLVal (Maybe IMLVal) SourcePos -- (ident | identArray) _
  ... | IdentArray IMLVal IMLVal SourcePos -- ident indexExpression
  ... | DyadicOpr IMLOperation IMLVal IMLVal SourcePos -- operation expression expression
  ... | MonadicOpr IMLOperation IMLVal SourcePos -- operation expression
  ... | Literal IMLLiteral SourcePos -- literal
  ... | Init -- not used
  ... | ExprList [IMLVal] SourcePos -- not used
  ... | Message String -- for printing error message
  ... | FunctionDeclaration IMLVal [IMLVal] [IMLVal] SourcePos -- ident [parameters] [statements]
  ... | FunctionCall IMLVal [IMLVal] SourcePos -- ident [parameters]
  ... | If IMLVal [IMLVal] [IMLVal] SourcePos -- condition [if Statements] [else Statement]
  ... | While IMLVal [IMLVal] SourcePos -- condition [statements]
  ... | For IMLVal [IMLVal] SourcePos -- condition [statements]
  ... | Assignment IMLVal IMLVal SourcePos -- (ident | identArray) expression
  ... deriving Show
```

## Checker

Der Checker erhält als Input den Abstract-Syntax-Tree vom Parser. Auf diesen führt er die unten beschriebenen Checks aus. Falls ein Check fehlschlägt wirft der Checker einen Error mit eindeutiger Fehlerbeschreibung und der Position (im Source Code) des fehlerhaften Ausdrucks.

## Beispiel

### Input:

```
prog div0(out o: int) {
  o := 10 / 0;
}
```

### Output:

```
*** Exception: division by zero! "Program" (line 2, column 10)
```

## Scope Checks

Ist die verwendete Identifier im aktuellen Scope definiert?
Ist ein Identifier im selben Scope mehrmals definiert?

## Type Checks

Haben die Argumente der Operationen die korrekten Typen?
Stimmen Anzahl und Typen von Funktionsaufrufen mit denen der Funktionsdefinition überein?
Ist der Typ von If-Bedingungen und While-Bedingungen Bool?
Ist der Typ der Zählvariable von For-Schleifen Subrange Int?
Ist der Typ von Array-Index-Expressions Int oder Subrange Int?

## Sonstige Checks

Division mit dem Literal 0 (Div 0)?
Gilt $\min \leq \max$ für alle Subrange Ints (min: Untergrenze, max: Obergrenze)?
Gilt $\text{start} \leq \text{end}$ für alle Arrays (start: Start-Index, end: End-Index)?

## Nicht notwendige Checks

Check	Begründung
Initialization	Alle Stores werden bei Deklaration mit ihrem Default-Wert initialisiert. Deshalb sind keine Initialization-Checks nötig.
Flow Analysis	Nicht nötig bei H-IML. Es gibt keine By-Reference-Parameter und Out-Flow-Parameter werden automatisch initialisiert.

## Codeerzeugung

Für die Codegenerierung führen wir ein Environment mit. Dieses enthält den Programcounter den Stackpointer, den Framepointer und die Symboltabelle. Die Symboltabelle besteht aus dem Globalen Scope und einem Stack von lokalen Scopes. Im globalen Scope stehen nur die Funktionen und jeder lokaler Scope entspricht dem Scope eines Befehlsblocks. Bei einem Zugriff auf einen Identifier werden alle lokalen Scopes der Reihe nach durchsucht und zuletzt noch der globale Scope.

Die Codegenerierung selbst besteht hauptsächlich aus Pattern Matches von Nodes im Abstract Syntax Tree zu Instruktionen und Änderungen am Environment.

## Subrange Variable (Variable-Clamp)

### Deklaration

Es wird ein Int32 mit dem \$MINIMUM auf dem Stack angelegt.

```
LoadIm Int32VmTy $MINIMUM
```

### Zuweisung

Bild	Code
<pre> graph TD     Start(( )) --&gt; Lade[Lade Adresse]     Lade --&gt; Fuehre[Führe Expression aus x liegt auf dem Stack]     Fuehre --&gt; MaxCheck{MaxCheck}     MaxCheck -- "[x &gt; \$MAXIMUM]" --&gt; MaxStore[\$MAXIMUM speichern]     MaxCheck -- "[x &lt;= \$MAXIMUM]" --&gt; MinCheck{MinCheck}     MinCheck -- "[x &lt; \$MINIMUM]" --&gt; MinStore[\$MINIMUM speichern]     MinCheck -- "[x &gt;= \$MINIMUM]" --&gt; XStore[x Speichern]     MaxStore --&gt; End(( ))     MinStore --&gt; End     XStore --&gt; End     </pre>	<p><b>Lade Adresse &amp; führe Expression aus</b></p> <pre> LoadIm IntVmTy \$ADDRESS \$EXPRESSION </pre> <p><b>MaxCheck</b></p> <pre> Dup LoadIm Int32VmTy \$MAXIMUM Le Int32VmTy CondJump ::STOREMIN </pre> <p><b>MinCheck</b></p> <pre> Dup LoadIm Int32VmTy \$MINIMUM Ge Int32VmTy CondJump ::STOREMAX </pre> <p><b>x Speichern</b></p> <pre> Store UncondJump ::FIN ::STOREMIN MoveSpUp 1 LoadIm Int32VmTy \$MAXIMUM Store UncondJump ::FIN ::STOREMAX MoveSpUp 1 LoadIm Int32VmTy \$MINIMUM Store ::FIN </pre>

Stack Beispiel einer Zuweisung für x grösser als \$MAXIMUM

Schritt	Stack
Clamp Variabel hat die Stack Adresse 5	5
Die Expression wird ausgewertet somit steht das Resultat auf dem Stack z.B. 9	5, 9
Resultat wird dupliziert (damit man es nach dem folgenden Vergleich noch hat).	5, 9, 9
Das Clamp Maximum wird in den Stack geladen z.B. 7	5, 9, 9, 7
Es wird mittel Le (Less Equals) verglichen	5, 9, 0
Der Conditional Jump spring in diesem Fall	5, 9
Es wird der Befehl MoveSpUp 1 aufgerufen um den letzten Wert auf dem Stack zu entfernen	5
\$MAXIMUM wird geladen	5, 7
Es wird Store aufgerufen und der Unconditional Jump wird ausgeführt jetzt steht das Maximum an der Adresse 5.	Stack ist leer

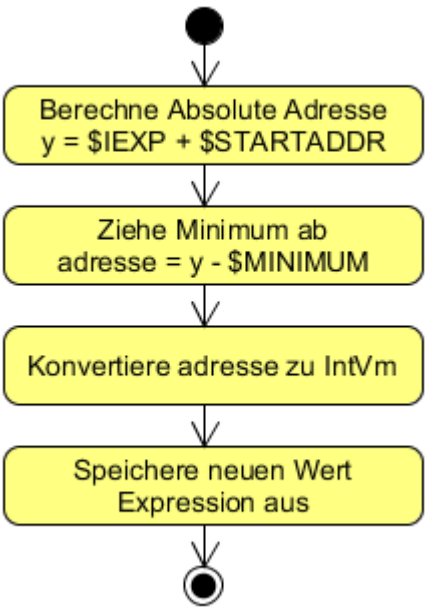
## Statischer Int Array

### Deklaration

Es wird n mal eine Null (Int Default) auf den Stack geschrieben. Wobei

$$n = \$MAXIMUM - \$MINIMUM + 1$$

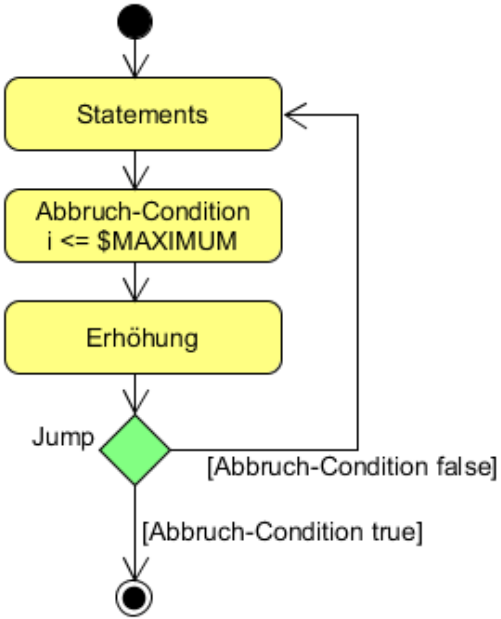
### Zuweisung

Bild	Code
 <pre> graph TD     Start(( )) --&gt; Box1[Berechne Absolute Adresse y = \$IEXP + \$STARTADDR]     Box1 --&gt; Box2[Ziehe Minimum ab adresse = y - \$MINIMUM]     Box2 --&gt; Box3[Konvertiere adresse zu IntVm]     Box3 --&gt; Box4[Speichere neuen Wert Expression aus]     Box4 --&gt; End((( )))         </pre>	<pre> <u>Berechne Absolute Adresse</u> \$IEXP LoadIm Int32VmTy \$STARTADDR Add <u>Ziehe Minimum ab</u> LoadIm IntVmTy \$MINIMUM Sub <u>Konvertiere Adresse zu IntVm</u> Convert Int32VmTy IntVmTy <u>Speichere neuen Wert</u> \$AEXP STORE         </pre>

### For Schleife

Die For Schleife muss vom \$START Wert bis und mit dem \$MAXIMUM Wert der Clamp-Variable durchlaufen. Um diese Durchläufe zu erreichen, werden zuerst die \$STATEMENTS durchgeführt, dann die Abbruch-Condition berechnet, dann die Erhöhung der Laufvariable getätigt und dann nach Condition gejumpet. Die Abbruch-Condition kann nach den \$STATEMENTS vollzogen werden, da mindestens ein Durchlauf garantiert ist: die Clamp Variable ist zu Beginn kleiner gleich dem \$MAXIMUM. Die Abbruch-Condition wird vor der Erhöhung der Laufvariable gemacht, dass er erst nach dem Durchlauf mit dem \$MAXIMUM anschlagen soll bzw. die Schleife verlassen werden soll.



Bild	Code
 <pre> graph TD     Start(( )) --&gt; Statements[Statements]     Statements --&gt; Condition[Abbruch-Condition i &lt;= \$MAXIMUM]     Condition --&gt; Increase[Erhöhung]     Increase --&gt; Jump{Jump}     Jump -- "[Abbruch-Condition false]" --&gt; Statements     Jump -- "[Abbruch-Condition true]" --&gt; End((( )))         </pre>	<pre> <b>::STATEMENT_BLOCK</b> \$STATEMENTS <b>Abbruch-Condition</b> LoadAddrRel \$ADDRESS Deref LoadIm Int32VmTy \$MAXIMUM, Eq <b>Erhöhung</b> Erhöhe Clamp Variable + 1 (Siehe Clamp) <b>Jump</b> CondJump ::STATEMENT_BLOCK         </pre>

## Rechtliches

Hiermit erklären wir, dass wir den Code selbständig erarbeitet haben, sofern nicht anderes deklariert.

Benjamin Meyer

Christian Scheller

Tobias Bollinger

## Anhang: Grammatik

Für die Erstellung des Parsers und Codeerzeugers haben wir zuerst unsere Grammatik definiert, die sich im Verlauf der Entwicklung stets mit veränderte:

```
program ::= PROGRAM IDENT LPAREN [ cpsParam ] RPAREN LBRACE [ cpsFunDecl ] [ cpsCmd ] RBRACE
```

```
decl ::= stoDecl | funDecl
```

```
stoDecl ::= VAR typedIdent SEMICOLON | VAL typedIdent SEMICOLON
```

```
cpsFunDecl ::= funDecl { funDecl }
```

```
funDecl ::= def IDENT LPAREN [ cpsParam ] RPAREN LBRACE cpsCmd RBRACE
```

```
cpsParam ::= param {COMMA param}
```

```
param ::= [FLOWMODE] [CHANGEMODE] typedIdent
```

```
typedIdent ::= IDENT COLON ATOMTYPE
```

```
          | IDENT COLON ATOMTYPE [ LPAREN range RPAREN ]
```

```
          | IDENT COLON ATOMTYPE [ LBRACKET range RBRACKET ]
```

```
range ::= literalRange
```

```
literalRange ::= LITERAL POINT POINT LITERAL
```

```
cmd ::= expr BECOMES expr SEMICOLON
```

```
      | IF LPAREN expr RPAREN LBRACE cpsCmd RBRACE [ ELSE LBRACE cpsCmd RBRACE ]
```

```
      | WHILE LPAREN expr RPAREN LBRACE cpsCmd RBRACE
```

```
      | IDENT LPAREN [idents] RPAREN
```

```
      | FOR LPAREN IDENT RPAREN LBRACE cpsCmd RBRACE
```

```
      | stoDecl
```

```
cpsCmd ::= cmd {SEMICOLON cmd}
```

```
idents ::= IDENT {COMMA IDENT}
```

```
expr ::= term1 {BOOLOPR term1}
```

```
term1 ::= term2 [RELOPR term2]
```

```
term2 ::= term3 {ADDOPR term3}
```

```
term3 ::= factor {MULTOPR factor}
```

```
factor ::= LITERAL
```

```
      | IDENT
```

```
      | monadicOpr factor
```

```
      | LPAREN expr RPAREN
```

```
      | IDENT LBRACKET expr RBRACKET
```

```
exprList ::= LPAREN [expr {COMMA expr}] RPAREN  
monadicOpr ::= NOT | ADDOPR
```

## Anhang: Parser Beispiel

### Code

```
prog sample1(in m: int, out o: int) {  
  def func1(in m: int, out o: int) {  
    m := m * 5;  
    o := m;  
    o := o / 2;  
  }  
  def func2(out o: int) {}  
  
  var i :int;  
  var c :int(0 .. 100);  
  var a :int[50 .. 100];  
  
  c := 50;  
  a[50] := 10;  
  
  //a Comment  
  while(i < c) {  
    a[i+50] := i;  
  }  
  
  if(i < 5) {  
    i := i + 5;  
  }  
  else {  
    i := i - 5;  
  }  
  
  func1(m, o);  
}
```

## Abstract Syntax Tree

```
Program(Ident sample1)
[
    ParamDeclaration In Mutable (Ident m) Int,
    ParamDeclaration Out Mutable (Ident o) Int
]
[
    FunctionDeclaration (Ident func1)
    [
        ParamDeclaration In Mutable (Ident m) Int,
        ParamDeclaration Out Mutable (Ident o) Int
    ]
    [
        Assignment (Ident m) :=
            DyadicOpr Times
            (
                IdentFactor (Ident m),
                Literal IMLInt 5
            ),
        Assignment (Ident o) :=
            IdentFactor (Ident m),
        Assignment (Ident o) :=
            DyadicOpr Div
            (
                IdentFactor (Ident o),
                Literal IMLInt 2
            )
    ],
    FunctionDeclaration (Ident func2)
    [
        ParamDeclaration Out Mutable (Ident o) Int
    ]
    [
    ]
]
[
    IdentDeclaration Mutable (Ident i) Int,
    IdentDeclaration Mutable (Ident c) ClampInt 0 100,
    IdentDeclaration Mutable (Ident a) ArrayInt 50 100,
    Assignment (Ident c) :=
        Literal IMLInt 50,
    Assignment IdentArray (Ident a)
        Literal IMLInt 50 :=
        Literal IMLInt 10,
    While
    (
```

```

        DyadicOpr Lt
        (
            IdentFactor (Ident i),
            IdentFactor (Ident c)
        )
    )
    [
        Assignment IdentArray (Ident a)
            DyadicOpr Plus
            (
                IdentFactor (Ident i),
                Literal IMLInt 50
            ) :=
            IdentFactor (Ident i)
    ],
    If
    (
        DyadicOpr Lt
        (
            IdentFactor (Ident i),
            Literal IMLInt 5
        )
    )
    [
        Assignment (Ident i) :=
            DyadicOpr Plus
            (
                IdentFactor (Ident i),
                Literal IMLInt 5
            )
    ]
    [
        Assignment (Ident i) :=
            DyadicOpr Minus
            (
                IdentFactor (Ident i),
                Literal IMLInt 5
            )
    ],
    FunctionCall (Ident func1)
    [
        IdentFactor (Ident m),
        IdentFactor (Ident o)
    ]
]

```