

Nat32 und Casting für IML

Marco Romanutti und Benjamin Meyer^{1,2}

¹Fachhochschule Nordwestschweiz FHNW, Brugg

²Schlussbericht

Im Modul Compilerbau wird eine Erweiterung für die bestehende Sprache IML spezifiziert und implementiert. Die Implementierung beinhaltet einen neuen Datentyp für natürliche Zahlen, sowie eine Möglichkeit den Datentyp int32 in den neuen Datentypen zu casten und umgekehrt.

1 Erweiterung

1.1 Einleitung

Unter natürlichen Zahlen werden die positiven, ganzen Zahlen und 0 verstanden:

$$\mathbb{N} = \{0; 1; 2; 3; \dots\}$$

Die IML soll um einen neuen Datentyp nat32 erweitert werden. Der neue Datentyp soll solche positiven, ganzen Zahlen mit einer Länge von bis zu 32 Bits in Binärdarstellung abbilden können. Es sollen die bestehenden Operationen unterstützt werden. Ausserdem soll ein explizites Casting zwischen dem bestehenden Datentyp int32 und dem neuen Datentyp nat32 möglich sein.

1.2 Lexikalische Syntax

Für den neuen Datentyp wird das Keyword (TYPE, NAT32) und ein Castingoperator hinzugefügt.

Datentyp:	nat32	(TYPE, NAT32)
Brackets:	[]	LBRACKET, RBRACKET

Casting ist nur von (TYPE, INT32) zu (TYPE, NAT32) und umgekehrt möglich. Als Castingoperator werden rechteckige Klammern (nachfolgend Brackets genannt) verwendet. Innerhalb der Brackets befindet sich der Zieldatentyp ¹.

¹zum Beispiel [int32]

1.3 Grammatikalische Syntax

Das nachfolgende Code-Listing zeigt, wie der neue Datentyp nat32 eingesetzt werden kann.

```
// Deklaration
var natIdent1 : nat32;
var natIdent2 : nat32;
var natIdent3 : nat32;

// Initialisierung
natIdent1 init := [nat32] 50;
natIdent2 init := [nat32] 10;
natIdent3 init := natIdent1 + natIdent2;

// Casting von int32 nach nat32
var intIdent1 : int32;
intIdent1 init := 30;
natIdent3 := [nat32] intIdent1;

call functionWithNatParam([nat32] intIdent1);

// Casting von nat32 nach int32
var intIdent2 : int32;
intIdent2 init := [int32] natIdent3;

call functionWithIntParam([int32] natIdent3);
```

Literale werden standardmässig als int32 interpretiert - ein nat32-Literal bedingt vorab deshalb den Castingoperator. Falls zwei Datentypen nicht gecastet werden können, wird ein Kompilierungsfehler geworfen. Folgendes Code-Listing zeigt ein solches Beispiel mit dem bestehenden Datentyp bool:

```
// Deklaration
var boolIdent : bool;
boolIdent init := false;
var natIdent : nat32;
// Throws type checking error:
natIdent init := [nat32] boolIdent
```

Unsere Erweiterung unterstützt keine impliziten Castings. Weitere Code-Beispiele sind im Anhang zu finden.

Ebenfalls haben wir in unserer IML-Syntax die logischen Symbole für AND (\wedge) sowie OR (\vee) durch die Symbole `&&` resp. `||` ersetzt. Nach unserem Erachten ist die Bedeutung dieser Symbole den meisten Programmierern geläufiger.

1.4 Änderungen an der Grammatik

Zusätzlich zu den bestehenden Operatoren wurde ein neuer `castOpr` erstellt, welcher anstelle des Nichtterminal-Symbol `factor` verwendet werden kann.

```
castOpr ::= LBRACKET TYPE RBRACKET
```

Das bestehende Nichtterminal-Symbol `factor` wird um diese neue Produktion ergänzt:

```
factor ::= LITERAL
        | IDENT [INIT | exprList]
        | castOpr factor
        | monadicOpr factor
        | LPAREN expr RPAREN
```

1.5 Kontext- und Typen-Einschränkungen

Der `TYPE` zwischen `LBRACKET` und `RBRACKET` muss vom Datentyp `int32` oder `nat32` sein. Ein Casting zum Typ `bool` oder vom Typ `bool` zu `int32` resp. `nat32` führt zu einem Kompilierungsfehler.

Tabelle 1 zeigt die unterstützten Typumwandlungen der verschiedenen Datentypen. Typumwandlungen, welche zu potentiellen Laufzeitfehlern führen, sind mit `*` gekennzeichnet. Der Datentyp `int32` umfasst einen Wertebereich von -2147483648 bis 2147483647 , wobei das Most Significant Bit (MSB) als Vorzeichen verwendet wird. Weil der Datentyp `nat32` nur positive, ganze Zahlen und die Zahl 0 darstellt, wird kein Vorzeichenbit benötigt. Der Wertebereich verschiebt sich dadurch auf 0 bis 4294967295 . Falls bei Typumwandlungen der Wert ausserhalb des Wertebereichs des Zieldatentyps liegt, führt dies zu einem Laufzeitfehler. Bei der Umwandlung von `nat32` nach `int32` kann ein solcher Laufzeitfehler beispielsweise auftreten, falls es sich um einen Wert > 2147483647 handelt. Falls negative Werte von `int32` nach `nat32` umgewandelt werden, resultiert ebenfalls ein Laufzeitfehler.

Table 1: Casting zwischen Datentypen

Quell- \ Zieldatentyp	int32	nat32	bool
int32	✓	✓*	✗
nat32	✓*	✓	✗
bool	✗	✗	✗

2 Aufbau Compiler

Der Compiler basiert auf der IML (V2) und ist in Java geschrieben.

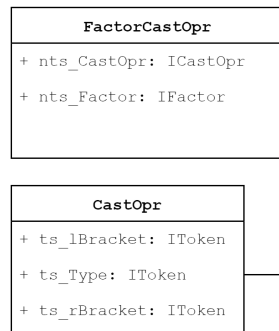
2.1 Scanner

Literale werden standardmässig als `int32` interpretiert - ein `nat32`-Literal bedingt vorab deshalb den Casting-operator. Vom Scanner werden Literale als `long` in Java eingelesen. Dieser Datentyp kann Werte von -9223372036854775808 bis 9223372036854775807 annehmen und deckt somit den gesamten Wertebereich der beiden Datentypen `int32` und `nat32` ab. Die Überprüfung, ob der Wert innerhalb des gültigen Wertebereichs des jeweiligen Datentyps liegt, erfolgt zum Zeitpunkt der Code-Generierung.

2.2 Parser

Der neu eingeführte `castOpr` und die neue Produktion `factor ::= castOpr factor` sind im Abstrakten Syntax Tree (AST) abgebildet. Dabei wird der Typ `CastOpr` zum Typ `CastFactor`, welcher Teil des AST ist. Abbildung 1 zeigt die Umwandlung von der konkreten in die abstrakte Syntax.

Concrete Syntax:



Abstract Syntax:

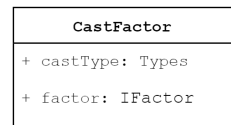


Figure 1: Umwandlung von der konkreten in die abstrakte Syntax

2.3 Statische Analyse

Scope checking

Für Routinen und Variablen liegen unterschiedliche Namespaces vor. Namen für Routinen und Variablen können deshalb identisch sein. Es wird zwischen globalen und lokalen Namespaces unterschieden: Bei lokalen Namespaces werden die Variablen innerhalb einer Routine definiert und haben dort ihre Gültigkeit. Globale Variablen dürfen nicht denselben Namen haben wie lokale Variablen. Überladene Signaturen für Routinen² wurden in dieser Implementation nicht umgesetzt.

²identischer Name, unterschiedliche Parameterlisten

Bei *FunCallFactor*, *ProcCallCmd*, *DebugInCmd* und *AssignCmd* muss überprüft werden, ob die Parameter den richtigen LValue, resp. RValue besitzen. Folgende Kombinationen sind dabei allgemein erlaubt:

Table 2: LRValue-Kombinationen

Callee	Caller	Resultat
LValue	LValue	Valid
RValue	LValue	Valid (LValue dereferenzieren)
RValue	RValue	Valid
LValue	RValue	LRValError \neq

Bei einem *AssignCmd* muss der Ausdruck links zudem zwingend ein LValue sein. Bei einem *DebugInCmd* muss es sich ebenfalls um einen LValue handeln, damit der Input-Wert dieser Variable zugewiesen werden kann.

Innerhalb des Scope checkings wird zudem überprüft, ob die Anzahl der erwarteten Parameter mit der Anzahl übergebener Parameter übereinstimmt.

Type checking

Das Casten zwischen zwei Datentypen ist nur für bestimmte Typen erlaubt (vgl. Tabelle 1). Zusätzlich sind beim Abarbeiten des AST nur folgende Typen erlaubt:

Table 3: Erlaubte Typen im AST

Klasse	Types
AddExpr, MultExpr, RelExpr	int32, nat32 *
BoolExpr, IfCmd	bool
AssignCmd	int32, nat32, bool *
FunCallFactor, ProcCallFactor	Typ von Caller muss Typ von Callee entsprechen
MonadicFactor	NOTOPR: bool ADDOPR: int32, nat32
CastFactor	Typ von Factor und Typ von CastFactor müssen gecastet werden können

Bei Einträgen, die mit * gekennzeichnet sind, müssen LValue und RValue vom selben Typ sein. Bei der Typenüberprüfung innerhalb vom CastFactor wird überprüft, ob der Typ vom CastFactor und jener des zugehörigen factors gecasted werden können (vgl. Tabelle 1). Die effektive Typ-Konversion wird erst bei der Codegenerierung durchgeführt.

Initialization checking

Beim Initialization checking wird überprüft, ob Variablen noch nicht initialisiert wurden, zuvor bereits einmal initialisiert wurden oder das Zuweisen eines Werts zur Variable untersagt ist. Folgendes Codebeispiel zeigt die verschiedenen Initialisierungsfehler:

```
program exampleInitErrors
global
  var x:int32;
  const y:int32
do
  x := 7; // NotInitializedError

  x init := 4;
  x init := 3; // AlreadyInitializedError

  y init := 1;
  y := 2 // AssignToConstError
endprogram
```

- NotInitializedError: Die Variable wurde noch nicht initialisiert.
- AlreadyInitializedError: Die Variable wurde bereits initialisiert.
- AssignToConstError: Die Variable ist eine Konstante und wurde bereits initialisiert.

Falls einer dieser Fehler auftritt, wird die Überprüfung sofort beendet und der Initialisierungsscheck ist fehlgeschlagen. Der AST wird während der Validierung Knoten für Knoten durchlaufen, wobei eine "Deep-First" Baumsuche angewendet wird.

2.4 Virtuelle Maschine

Grundlage für die Codegenerierung ist der AST. Vom Root-Knoten ausgehend fügt jeder Knoten seinen Code zum Code-Array.

Im Falle eines Castings zwischen zwei Datentypen befindet sich an mindestens einer Stelle in der AST-Struktur ein CastFactor-Element. Von diesem Element aus wird der Datentyp des zugehörigen factor geändert, indem dessen Attribut castFactor geändert wird. Dieses Attribut übersteuert den eigentlichen Datentyp des Elements innerhalb des AST. Weil der factor gemäss Grammatik unterschiedliche Produktionen besitzt (vgl. Änderungen an der Grammatik), muss die Typanpassung rekursiv weitergegeben werden. Die Rekursion wird durch Literale oder Expressions unterbrochen, wie am Beispiel in Anhang D aufgezeigt.

In der virtuellen Maschine wurde ein neuer generischer Typ NumData eingeführt. Dieser wird für die Konversion zwischen Daten vom Typ IntData³ und NatData⁴ verwendet. Abbildung 2 zeigt die Klassenhierarchie dieser Typen.

³für den Datentyp int32

⁴für den Datentyp nat32

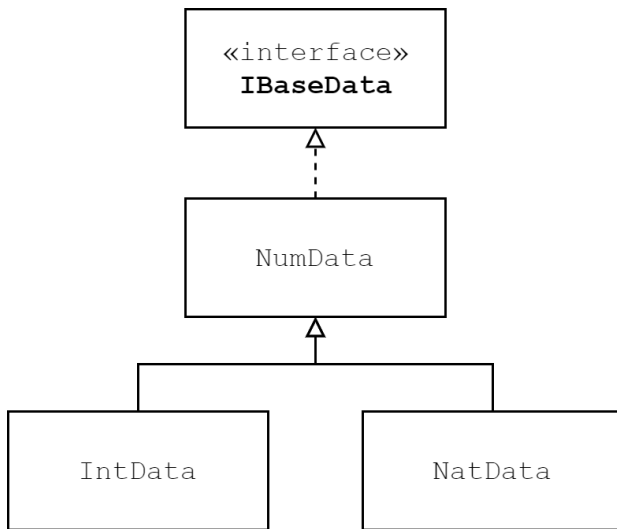


Figure 2: Daten in VM

Beim Dereferenzieren muss im Falle eines Castings der Datentyp von bereits typisierten Daten auf dem Stack geändert werden. Anhang E zeigt ein Beispiel, bei welchem der Datentyp aufgrund des neu propagierten Attributs `castType` umgewandelt wird.

Bei der Codegenerierung wird nun anhand des Datentyps des Knotens im Abstract Syntax Tree der entsprechende Befehl zum Code-Array hinzugefügt. Wie bereits erwähnt, wird der eigentliche Typ übersteuert, falls das Attribut `castType` gesetzt ist. Das folgende Codebeispiel `exampleLoad` zeigt je das Konvertieren eines Literals und einer Variable. Im daraus generierten Code (Abbildung 3) ist gut ersichtlich, dass ein Literal standardmässig als `int32` interpretiert wird.

```

program exampleLoad
global
  var x:int32;
  var y:nat32
do
  x init := 4;
  y init := [nat32] 3;
  x := [int32] y
endprogram
  
```

0: AllocBlock(1)	}	Allozieren von zwei Speicherplätzen
1: AllocBlock(1)		
2: UncondJump(3)	}	Sprung aus Allokationsblock
3: LoadAddrAbs(0)		
4: LoadImInt(4)	}	Laden/Speichern von Int Wert
5: Store		
6: LoadAddrAbs(1)	}	Laden/Speichern von Nat Wert
7: LoadImNat(3)		
8: Store	}	Laden/Speichern von Wert an Adresse 1 nach Adresse 0
9: LoadAddrAbs(0)		
10: LoadAddrAbs(1)		
11: Deref		
12: Store	}	Dereferenzierung von Wert an Adresse 1
13: Stop		
		Typumwandlung von bereits typisierten Daten
		Beenden des Programms

Figure 3: Generierter Code aus "exampleLoad"

3 Vergleich mit anderen Programmiersprachen

3.1 Ganzzahlige Werte

In Java wird bei Zuweisungen die Länge einer Zahl in Binärdarstellung überprüft: Beim Datentyp `long` wird beispielsweise geprüft, ob der Wert als ganzzahliger Wert von 64-bit Länge dargestellt werden kann. Falls dies nicht der Fall ist, wird ein Fehler zur Kompilierungszeit geworfen. Das MSB wird als Vorzeichenbit verwendet, womit rund die Hälfte der vorzeichenlos darstellbaren `long`-Werte entfällt, resp. zur Darstellung von negativen Zahlen eingesetzt wird. Falls bei fortlaufenden Berechnungen Wertebereiche unter- resp. überschritten werden, führt dies zu einem arithmetischen Überlauf. Abbildung 4 zeigt den Überlauf bei ganzzahligen, vorzeichenbehafteten Datentypen (am Beispiel von Bitlänge 3 + 1).

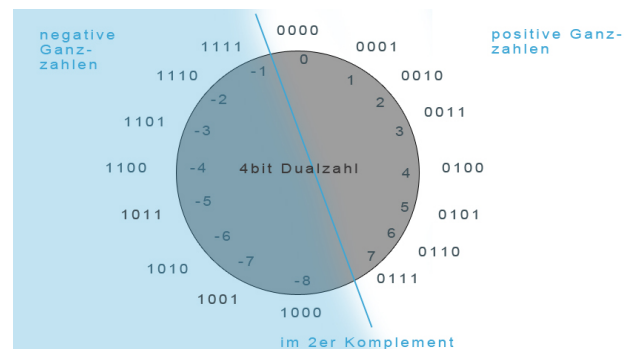


Figure 4: Überlauf mit Integerzahlen

Dadurch führt z.B. beim Datentyp `int` der Ausdruck `Integer.MAX_VALUE + 1` zum Wert `Integer.MIN_VALUE`. Dies kann dazu führen, dass mit „falschen“ Werten gerechnet wird, ohne dass der Entwickler dies bemerkt.

3.2 Fließkommazahlen

Im Gegensatz zur Darstellung im Zweierkomplement, welches für Integer-Typen in Java verwendet wird, werden Fließkommazahlen intern nach IEEE Standard dargestellt. Anders als bei der Zweierkomplement-Darstellung sieht dieses Format spezielle, konstante Werte vor. So sind in Java beispielsweise Konstanten für `Double.POSITIVE_INFINITY`, `Double.NEGATIVE_INFINITY` und `Double.NaN` definiert⁵.

⁵vgl. [4]

4 Designentscheidungen

4.1 Spezifiziertes Verhalten

Der neue Datentyp `nat32` unterstützt die bestehenden Operationen aus IML⁶. Sofern sich die einzelnen Operanden und auch das Resultat im Wertebereich⁷ befinden, entspricht das Verhalten vom Datentyp `nat32` jenem vom Datentyp `int32`. Andernfalls wird folgendes Verhalten festgelegt:

- **Wertebereich:** Wertebereichsüber- resp. Unterschreitungen resultieren in einem Laufzeitfehler⁸. Dies erhöht die Typsicherheit beim Einsatz der verschiedenen Datentypen.
- **Nachkommastellen:** Gemäss IML-Spezifikation sind als Literale nur ganzzahlige Werte erlaubt. Falls z.B. bei einer Division ein Rest resultiert, wird ein ganzzahliges Resultat zurückgegeben. Der Wert des Resultats ist abhängig von der gewählten Operation⁹.

4.2 Alternative Ansätze

Folgende weiteren Ansätze wurden für die Umsetzung der Erweiterung in Betracht gezogen:

- **Arithmetischer Überlauf:** Wertebereichsüber- resp. unterschreitungen resultieren in einem arithmetischen Überlauf. Gegenüber der Darstellung in Abbildung 4 müsste kein negativer Wertebereich verwendet werden und das Addieren von +1 zum grössten darstellbaren Wert des Datentyps `nat32` führt zum Wert 0. Weil auf ein Vorzeichenbit verzichtet werden kann, verdoppelt sich der Wertebereich gegenüber dem Datentyp `int32`. Nachteilig ist dabei, dass der Entwickler verantwortlich ist für das Einhalten der Wertebereichsgrenzen.
- **Vordefinierte Konstanten:** Ähnlich wie bei Fließkommazahlen (vgl. Kapitel 3.2) könnten konstante Werte für z.B. `POSITIVE_INFINITY` und `NEGATIVE_INFINITY` vorgesehen werden. Das Verhalten bei Wertebereichsüberschreitungen müsste definiert werden. Nachteilig bei dieser Variante ist, dass der Wertebereich um die Anzahl solcher konstanter Werte verringert wird.
- **Absolute Werte:** Bei dieser Variante wird, bei negativen Werten, deren Absolutwert verwendet. Von dieser initial angedachten Variante wurde abgesehen, weil das Verhalten schnell zu ungewollten Resultaten führen kann.

⁶Aktuell sind dies

- `MULTOPR(*, divE, modE)`
- `ADDOPR(+, -)`
- `RELOPR(<, <=, >, >=, =, /=)`
- `BOOLOPR(&&, ||, &, |)`

⁷[0, 4294967295]

⁸Negative Zahlen entsprechen Wertebereichsunterschreitungen

⁹`DivFloor`, `DivTrunc`, etc.

Aufgrund der beschriebenen Nachteile, haben wir uns für die im Kapitel 4.1 spezifizierte Variante entschieden.

5 Diskussion und Ausblick

Die Erweiterung des Compilers um einen neuen Datentypen für natürliche Zahlen, sowie eine Möglichkeit den Datentyp `int32` in den neuen Datentypen zu casten und umgekehrt, konnten erfolgreich umgesetzt werden. Der Scanner wurde grösstenteils von Marco Romanutti implementiert, während Benjamin Meyer die Grammatik für Fix und Foxi erstellte. Alle übrigen Arbeiten (Parser, Statische Analyse, Virtuelle Berichte, Berichte, etc.) wurden aufgrund des grossen Aufwands gemeinsam und in fortwährendem Austausch durchgeführt.

Wie in Kapitel 2.1 beschrieben, werden Literale standardmässig als `int32` interpretiert. Literale für den Datentyp `nat32` müssen deshalb mit dem Castingoperator gecastet werden. In einer nächsten Erweiterung des Compilers könnte ein neues Syntaxelement eingefügt werden, mithilfe welchem `nat32`-Literals "schöner" gekennzeichnet werden könnten¹⁰. Weiter könnten Datentypen mit einer Länge von mehr als 32 Bit in Binärdarstellung oder auch Floatingdatentypen eingeführt werden. Ebenfalls denkbar wären zusätzliche mathematische Funktionen einzubauen, wie z.B. eine Funktion zur Berechnung des grössten gemeinsamen Teilers (ggT) oder des kleinsten gemeinsamen Vielfachen (kgV).

References

- [1] Wikipedia: Natürliche Zahl
https://de.wikipedia.org/wiki/Nat%C3%BCrliche_Zahl
- [2] Wikipedia: Natural numbers (engl.)
https://en.wikipedia.org/wiki/Natural_number
- [3] Wikipedia: Integer (Datentyp)
[https://de.wikipedia.org/wiki/Integer_\(Datentyp\)](https://de.wikipedia.org/wiki/Integer_(Datentyp))
- [4] Stackoverflow: Purpose of defining `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`, `NaN` constants only for floating-point data types, but not for integral data types
<https://stackoverflow.com/questions/41312477/>

¹⁰vgl. Java Suffix für `long` (z.B. `39537L`), `float` (z.B. `1.4f`) und `double` (z.B. `-7.439d`)

Brugg, 07.01.2020

Marco Romanutti

Benjamin Meyer

A Vollständige Grammatik

```

program      ::= PROGRAM IDENT progParamList [GLOBAL cpsDecl] DO cpsCmd ENDPROGRAM

decl         ::= stoDecl
              | funDecl
              | procDecl

stoDecl      ::= [CHANGEMODE] typedIdent

funDecl      ::= FUN IDENT paramList RETURNS stoDecl [GLOBAL globImps] [LOCAL cpsStoDecl]
              DO cpsCmd ENDFUN
procDecl     ::= PROC IDENT paramList [GLOBAL globImps] [LOCAL cpsStoDecl] DO cpsCmd ENDPROC

globImps     ::= globImp {COMMA globImp}

globImp      ::= [FLOWMODE] [CHANGEMODE] IDENT

cpsDecl      ::= decl {SEMICOLON decl}

cpsStoDecl   ::= stoDecl {SEMICOLON stoDecl}

progParamList ::= LPAREN [progParam {COMMA progParam}] RPAREN

progParam    ::= [FLOWMODE] [CHANGEMODE] typedIdent

paramList    ::= LPAREN [param {COMMA param}] RPAREN

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

typedIdent   ::= IDENT COLON TYPE

cmd          ::= SKIP
              | expr BECOMES expr
              | IF expr THEN cpsCmd ELSE cpsCmd ENDIF
              | WHILE expr DO cpsCmd ENDWHILE
              | CALL IDENT exprList [globInits]
              | DEBUGIN expr
              | DEBUGOUT expr

cpsCmd       ::= cmd {SEMICOLON cmd}

globInits    ::= INIT idents

idents       ::= IDENT {COMMA IDENT}

expr         ::= term1 {BOOLOPR term1}

term1        ::= term2 [RELOPR term2]
term2        ::= term3 {ADDOPR term3}
term3        ::= factor {MULTOPR factor}

factor       ::= LITERAL
              | IDENT [INIT | exprList]
              | monadicOpr factor
              | LPAREN expr RPAREN
              | castOpr factor

castOpr      ::= LBRACKET TYPE RBRACKET

exprList     ::= LPAREN [expr {COMMA expr}] RPAREN

monadicOpr   ::= NOT | ADDOPR

```

B Beispiel Scanner

IML:

This file should lex successfully:

Liebe Grossmutter:

Zu Deinen 67-ten Geburtstag wuensche ich Dir
alles Gute,

Dein Beat

&& || program+-*endprogram

whilex==17do//
xwhile:=5

x_'_'17 1000''100'10

// A Comment
no comment

mod modE
[nat32]

Token-List:

```
[ (IDENT, "This"), (IDENT, "file"), (IDENT,
  "should"), (IDENT, "lex"), (IDENT,
  "successfully"), COLON, (IDENT, "Liebe"),
  (IDENT, "Grossmutter"), COLON, (IDENT,
  "Zu"), (IDENT, "Deinen"), (LITERAL, 67),
  (ADDOPR, MINUS), (IDENT, "ten"), (IDENT,
  "Geburtstag"), (IDENT, "wuensche"), (IDENT,
  "ich"), (IDENT, "Dir"), (IDENT, "alles"),
  (IDENT, "Gute"), COMMA, (IDENT, "Dein"),
  (IDENT, "Beat"), (BOOLOPR, CAND), (BOOLOPR,
  COR), PROGRAM, (ADDOPR, PLUS), (ADDOPR,
  PLUS), (ADDOPR, MINUS), (MULTOPR, TIMES),
  ENDPROGRAM, (IDENT, "whilex"), (RELOPR,
  EQ), (RELOPR, EQ), (LITERAL, 17), DO,
  (IDENT, "xwhile"), BECOMES, (LITERAL, 5),
  (IDENT, "x__17"), (LITERAL, 100010010),
  (IDENT, "no"), (IDENT, "comment"),
  (MULTOPR, MOD), (MULTOPR, MOD_E), LBRACKET,
  (IDENT, "NAT32"), RBRACKET, SENTINEL ]
```

C Beispiel init

IML:

```

program exampleInit
global
  var x:int32;
  var y:nat32
do
  x init := [int32] 4;
  y init := [nat32] 3
endprogram

```

Concrete Syntax Tree:

```

PROGRAM
(IDENT, "exampleInit")
GLOBAL
  (CHANGEMOD, VAR)
  (IDENT, "x")
  COLON
  (TYPE, INT32)
  SEMICOLON
  (CHANGEMOD, VAR)
  (IDENT, "y")
  COLON
  (TYPE, NAT32)
DO
  (IDENT, "x")
  INIT
  BECOMES
  LBRACKET
  (TYPE, INT32)
  RBRACKET
  (LITERAL, 4)
  SEMICOLON
  (IDENT, "y")
  INIT
  BECOMES
  LBRACKET
  (TYPE, NAT32)
  RBRACKET
  (LITERAL, 3)
ENDPROGRAM

```

Abstract Syntax Tree:

```

ch.fhnw.edu.cpiib.ast.Program
[localStoresNamespace]:
[globalStoresNamespace]:
[globalRoutinesNamespace]:
<ident>: (IDENT, "exampleInit")
<globalDeclarations>:
  ch.fhnw.edu.cpiib.ast.StoDecl
  [localStoresNamespace]:
  <changeMode>: VAR
  <typeIdent>:
    ch.fhnw.edu.cpiib.ast.TypedIdent
    [localStoresNamespace]:
    (<ident>, <type>): ((IDENT, "x"), INT32)
  ch.fhnw.edu.cpiib.ast.StoDecl
  [localStoresNamespace]:
  <changeMode>: VAR
  <typeIdent>:
    ch.fhnw.edu.cpiib.ast.TypedIdent
    [localStoresNamespace]:
    (<ident>, <type>): ((IDENT, "y"), NAT32)
<cpsCmd>: ch.fhnw.edu.cpiib.ast.CpsCmd
[localStoresNamespace]:
<commands>:
  ch.fhnw.edu.cpiib.ast.AssignCmd
  [localStoresNamespace]:
  <exprLeft>:
    ch.fhnw.edu.cpiib.ast.InitFactor
    [localStoresNamespace]:
    <init>: true
  <exprRight>:
    ch.fhnw.edu.cpiib.ast.CastFactor
    [localStoresNamespace]:
    <CastType>: INT32
    <factor>:
      ch.fhnw.edu.cpiib.ast.LiteralFactor
      [localStoresNamespace]:
      <literal>: (LITERAL, 4)
  ch.fhnw.edu.cpiib.ast.AssignCmd
  [localStoresNamespace]:
  <exprLeft>:
    ch.fhnw.edu.cpiib.ast.InitFactor
    [localStoresNamespace]:
    <init>: true
  <exprRight>:
    ch.fhnw.edu.cpiib.ast.CastFactor
    [localStoresNamespace]:
    <CastType>: NAT32
    <factor>:
      ch.fhnw.edu.cpiib.ast.LiteralFactor
      [localStoresNamespace]:
      <literal>: (LITERAL, 3)

```

D Beispiel doppeltes Casting

IML:

```

program exampleDouble
global
  var value:int32
do
  value init := [int32] [nat32] ((4 + 1) + 1)
endprogram

```

Code-Array:

```

0: AllocBlock(1)
1: UncondJump(2)
2: LoadAddrAbs(0)
3: LoadImInt(4)
4: LoadImInt(1)
5: AddInt
6: LoadImInt(1)
7: AddInt
8: Store
9: Stop

```

UML:

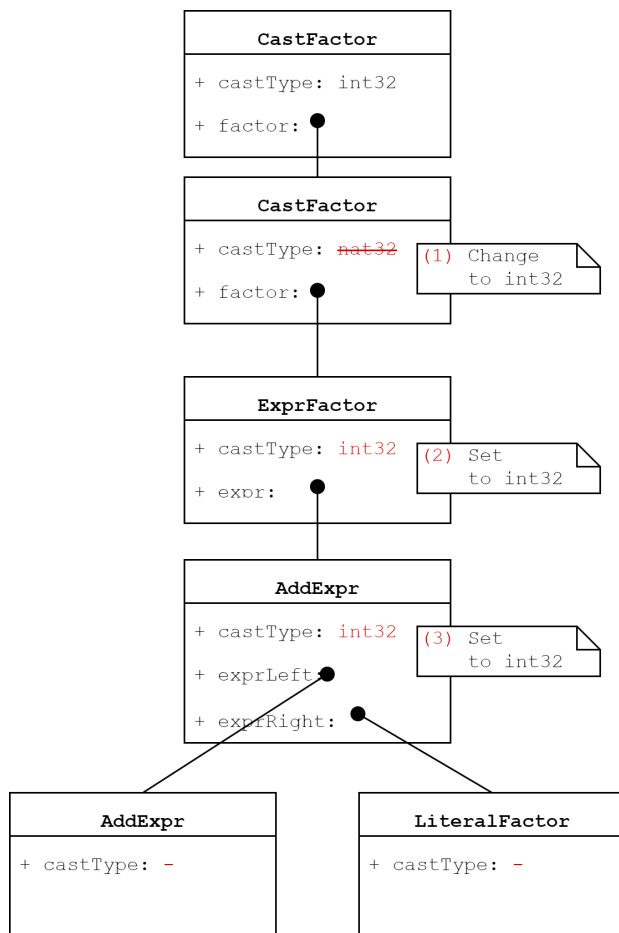


Figure 5: Auszug aus AST

E Beispiel Deref

IML:

```

program exampleDeref
global
  var x:int32;
  var y:nat32
do
  x init := [int32] 4;
  y init := [nat32] 3;
  x := [int32] y
endprogram

```

Code-Array:

```

0: AllocBlock(1)
1: AllocBlock(1)
2: UncondJump(3)
3: LoadAddrAbs(0)
4: LoadImInt(4)
5: Store
6: LoadAddrAbs(1)
7: LoadImNat(3)
8: Store
9: LoadAddrAbs(0)
10: LoadAddrAbs(1)
11: Deref
12: Store
13: Stop

```

Stack:

```

...

(8) Executing instruction Store
pc: 8
sp: 4
Stack content:
0: IntData: 4
1: NatData: 3

(9) Executing instruction LoadAddrAbs(0)
pc: 9
sp: 2
Stack content:
0: IntData: 4
1: NatData: 3
2: IntData: 0

(10) Executing instruction LoadAddrAbs(1)
pc: 10
sp: 3
Stack content:
0: IntData: 4
1: NatData: 3
2: IntData: 0
3: IntData: 1

(11) Executing instruction Deref
pc: 11
sp: 4
Stack content:
0: IntData: 4
1: NatData: 3
2: IntData: 0
3: IntData: 3

(12) Executing instruction Store
pc: 12
sp: 4
Stack content:
0: IntData: 3
1: NatData: 3

```

```

...

```

F Beispiel Casting expr

IML:

```

program exampleCastExpression
global
  var x:int32;
  var y:int32;
  var z:nat32;
  var b:bool;
  var c:bool
do
  x init := 4;
  y init := 2;
  z init := [nat32] (x * y + y + x);
  b init := x >= y;
  c init := x >= [int32] z
endprogram

```

Code-Array:

```

0: AllocBlock(1)
1: AllocBlock(1)
2: AllocBlock(1)
3: AllocBlock(1)
4: AllocBlock(1)
5: UncondJump(6)
6: LoadAddrAbs(0)
7: LoadImInt(4)
8: Store
9: LoadAddrAbs(1)
10: LoadImInt(2)
11: Store
12: LoadAddrAbs(2)
13: LoadAddrAbs(0)
14: Deref
15: LoadAddrAbs(1)
16: Deref
17: MultInt
18: LoadAddrAbs(1)
19: Deref
20: AddInt
21: LoadAddrAbs(0)
22: Deref
23: AddNat
24: Store
25: LoadAddrAbs(3)
26: LoadAddrAbs(0)
27: Deref
28: LoadAddrAbs(1)
29: Deref
30: GeInt
31: Store
32: LoadAddrAbs(4)
33: LoadAddrAbs(0)
34: Deref
35: LoadAddrAbs(2)
36: Deref
37: GeInt
38: Store
39: Stop

```

G Beispiel Factorial

IML:

```
program exampleFactorial
  global
  const input:nat32;
  const result:int32;
  proc factorial(copy const value:int32, ref
    var factorial:int32)
  local
    var counter:int32
    do
      counter init := value;
      factorial := 1;
      while counter > 1 do
        factorial := factorial * counter;
        counter := counter - 1
      endwhile
    endproc
  do
    debugin input init;
    call factorial([int32] input, result init);
    debugout result
  endprogram
```

Output bei Input 4:

result:int32 => 24 ✓

H Beispiel Fibonacci

IML:

```
program exampleFibo
  global
  fun recursiveFibonacci(copy const n:int32)
    returns m: int32
  do
    if n < 2 then
      m init := 1
    else
      m init := recursiveFibonacci(n-1) +
        recursiveFibonacci(n-2)
    endif
  endfun
do
  debugout recursiveFibonacci(8)
endprogram
```

Output für Input 8:

result:int32 => 34 ✓