Records in IML - Zwischenbericht, Compilerbau HS 2014, Team 01

Abstract

Im Modul Compilerbau wird den Stiduerenden die Funktionsweise, sowie der Aufbau eines Compilers vermittelt. Um diese Erkentnisse zu vertiefen, soll jedes Team eine Spracherweiterung zur IML-Sprache entwickeln. In diesem Dokument werden die Ansätze für die Erweiterung der Sprache IML um Records erläutert.

Grundsätzliche Idee der Erweiterung

Als Erweiterung sollen Records implementiert werden. Ein Record soll dabei als neuer Datentyp zur Verfügung stehen, welcher beliebig viele, zusammengesetzte Felder beinhalten kann. Die Felder können vom Datentyp Integer oder Boolean sein. Einmal definierte Records stehen im ganzen Programm als Datentyp bereit. Er sollte zu Beginn definiert werden und kann dann beliebig oft verwendet werden. Eine Definition eines Records könnte so aussehen:

$record\ vector(var\ x : int32,\ var\ y : int32,\ var\ b : boolean);$

Die Felder eines Records(z.B. $x,\ y,\ b$) verhalten sich wie ”normale”Variablen. Um sie verwenden zu können, müssen sie daher zuvor initialisiert werdendarum müssen sie auch zuerst initialisiert werden. Zusätzlich muss jedes Feld innerhalb eines Records über einen eindeutigen Namen verfügen. Ein Beispiel soll nun verdeutlichen, wie Feldern eines Records verwendet werden können:

```
1 var v1:vector;

  ? v1.x init;
  v1.y init := 2;
  ! v1.y;
```

Ein Record soll auch mit konstanten Werten definiert werden können. Dazu muss lediglich das Keyword var weggelassen werden:

```
  record vector(var x:int32, y:int32);
  var v1:vector;
  v1.x init := 2;
  v1.y init := 2;
5 v1.x := 3;
  v1.y := 3; // Fehler, da v1.y konstant ist.
```

Überprüfungen zur Compilezeit

Die Erweiterung soll das IML-Programm auf folgende Fehler prüfen:

- Sind die Namen der Felder innerhalb eines Records eindeutig?
- Sind die Namen der Records im gesamten Programm eindeutig?
- Sind Operationen mit Variablen, die als Record definiert sind, vorhanden?
- Werden Records durch andere Records definiert?
- Wird versucht ein Feld eines Typs mit einem Wert eines andern Typs zu überschreiben?
  (z.B. <boolean>:= <integer>)
- Wird versucht auf undefinierte Record-Felder zuzugreifen?.

Fehlerausgabe

Fehler sollen mit Zeilen und Spaltennummer dem Programmierer gemeldet werden. Deshalb wird im Scanner jeweils die aktuelle Position zu jedem erkannten Token in der Tokenlist hinterlegt. Deshalb sind auch Positionsangaben bei Grammatik-, Kontext- und Typefehlern möglich.

Kontext- und Typeinschränkung

Da die Grösse und der Aufwand der Erweiterung noch nicht genau abschätzbar ist, wurde beschlossen, dass die nun folgenden Funktionen wenn möglich im Design der Erweiterung berücksichtigt werden. Vorerst werden diese aber nicht implementiert.

Ein Beispiel wäre hier die Definition von Records durch andere Records, was beispielsweise bei einer Matrix sinnvoll wäre. Dies bedingt jedoch, dass als Datentyp nicht nur primitive Datentypen erlaubt wären. In einem ersten Schritt werden jedoch nur primitive Datentypen erlaubt, was zur Folge hat, dass das folgende Beispiel nicht verwendet werden kann:

```
record vector(var x:int32, var y:int32, var b:boolean);
record matrix(var v1:vector, var v2:vector);
var m1:matrix;

m1.v1.x init := 0; // noch nicht unterstützt
m1.v2.y init := 2; // noch nicht unterstützt
```

Es wäre schön, wenn es eine kurze Schreibweise zur Initialisierung der Felder gäbe. Dies wird vorerst jedoch nicht unterstützt. Ein Beispiel könnte so aussehen:

```
record vector(var x:int32, var y:int32, var active:boolean);
var vl:vektor;

v1 init := (22, 13, true); //nicht implementiert
```

Operationen auf Records sind nicht zugelassen, da nur die Felder Eigenschaften enthalten, nicht jedoch der Record selbst.

```
record vector (var x:int32, var y:int32, var b:boolean);
var vl:vector;
var v2:vector;

//Arithmetische Operationen auf Felder von Records sind nicht zuglassen
v1 := v1 + v2
v1 := v1 - v2
v1 := v1 * v2
v1 := v1 div div2

//Zudem bestitzt die Variable keinen Zustand, somit auch keinen Boolean
if (v1) ..
```

Des weiteren ist die Übergabe eines gesamten Records an eine Funktion nicht möglich. Es können jedoch einzelne Werte der Felder eines Records übergeben werden:

```
proc divide(in copy vec:vector, out ref q:vector){}
proc divide2(in copy x:int32, out ref q:int32) {}
record vector(var x:int32, var y:int32, var z:int32);
var q:int32;
var v1:vektor;
v1.x init := 5;
v1.y init := 6;
v1.z init := 7;
call divide(v1, q init); //nicht unterstützt
call divide2(v1.x,q init); //erlaubt
```

Vergleich mit anderen Programmiersprachen

Programmiersprache	Deklaration	Instanziierung

Pascal	`type`
	`  TVector = record`
	`  x : Integer;`
	`  y : Integer;`
	`  z : Integer;`
	`end;`
		`var`
		`  v1 : TVector`
		`begin`
		`  v1.x := 5;`
		`  v1.y := 6;`
		`  v1.z := 7;`
		`end;`

Haskell	`data vector = vector (`
	`  x .. Int,`
	`  y .. Int,`
	`  z .. Int`
	`)`
	`deriving (Show)`
		`v1 = vector 5 6 7`

C/C++	`struct vector {`
	`  int x;`
	`  int y;`
	`  int z;`
	`} ;`
		`vector v1;`
		`v1.x = 5;`
		`v1.x = 6;`
		`v1.z = 7;`

IML mit Record Spracherweiterung	`record vector (`
	`  var x:int32,`
	`  var y:int32,`
	`  var z:int32`
	`);`
		`var v1:vector;`
		`v1.x := 5;`
		`v1.x := 6;`
		`v1.z := 7;`

Gründe für diesen Aufbau der Erweiterung

Syntax

Damit die Syntax bestmöglich zur IML-Syntax passt, wurde eine Mischung zwischen Pascal und C++ gewählt. Dabei wird die eigentliche Definition des Records ähnlich wie ein Struct aus C++ definiert. Die Definition der Felder entspricht dann eher der Umsetzung in Pascal.

Grammatik

”param” erlaubt die Verwendung von FLOWMODE und MECHMODE. Allerdings benötigen wir nur CHANGEMODE für die Umsetzung von Records, da wir nur const und var erlauben. Deshalb haben wir das neue NTS ”paramCmList” definiert.

Lexikalische Syntax

Um die von uns vorgesehenen Operationen auf die Felder von Recordszu gewährleisten, muss neben dem Token RECORD auch ein Token DOT eingeführt werden. Dabei wird gilt: ”.”$\longrightarrow$ DOT und ”record”$\longrightarrow$ RECORD.
Für die Definition der Records soll der Scanner folgende Tokenlist erstellen:

`record vector(var x:int32, var y:int32, var z:int32);`
RECORD, (IDENT,”vector”), LPAREN, (CHANGEMODE, VAR), (IDENT, ”x”), COLON, (TYPE, INT32), COMMA, (CHANGEMODE, VAR), (IDENT, ”y), COLON, (TYPE, INT32), COMMA, (CHANGEMODE, VAR), (IDENT, ”y”), COLON, (TYPE, INT32), RPAREN, SEMICOLON

```
v1.x init := 5;
(IDENT, "v1"), DOT, (IDENT, "x"), INIT, BECOMES, (LITERAL, INTVALUE, 5), SEMICOLON
```

Grammatikalische Syntax

Um die Erweiterung umzusetzen werden folgende neue bzw. geänderte Produktionen benötigt:

```
decl          := storeDecl
               | funDecl
               | prodDecl
               | recordDecl                    //Decl erweitern für Records

recordDecl    := RECORD IDENT paramCmList      //paramCmList, da nur CHANGEMODE benötigt

paramCmList   := LPAREN optstoreDecls RPAREN   //(LISTE VON FELDERN)

optstoreDecls := storeDecl repCOMMAstoreDecls  //storeDecl und weitergabe übrige Felder
               | ϵ

repCOMMAstoreDecls := COMMA storeDecl repCOMMAstoreDecls //Verarbeitung übrige Felder
               | ϵ

factor        := LITERAL
               | monadicOpr factor
               | LPAREN expr RPAREN
               | IDENT optDOTIDENT optINITexprList //Einführung DOT Operator

optDOTIDENT   := DOT IDENT optDOTIDENT            //Deklaration DOT Operator
               | ϵ
```

IML Beispiel

```
program vecScalProd

    record vector(var x:int32, var y:int32, var z:int32);

    var v1:vector;
    var v2:vector;
    var r:vector;

    ? v1.x init;
    ? v1.y init;
    ? v1.z init;

    ? v2.x init;
    ? v2.y init;
    ? v2.z init;

    r.x = v1.x * v2.x;
    r.y = v1.y * v2.y;
    r.z = v1.z * v2.z;

    ! r.x;
    ! r.y;
    ! r.z;

endprogram
```