

Industrieroboter

Programmiersprache

Industrial Robot Language (IRL)

DIN
66312

ICS 25.040.30; 35.060

Ersatz für
DIN 66312-1:1993-06

Deskriptoren: Industrieroboter, Programmiersprache, Software, Informationsverarbeitung, Maschinenbau

Manipulating industrial robots; Programming language; Industrial Robot Language (IRL)

Vorwort

Diese Norm wurde vom Normenausschuß Maschinenbau (NAM), Fachbereich Montage- und Handhabungstechnik (FB MHT), Arbeitsausschuß 81.4 "IRL + ICR", erarbeitet.

Diese Norm beschreibt den einheitlichen Aufbau einer höheren Programmiersprache für Industrieroboter (IRL Industrial Robot Language). Sie definiert die Syntax und die Semantik von IRL. IRL gestattet dem Anwender dieser Norm, die Bewegungen von Industrierobotern und den logischen Programmfluß in einer benutzerorientierten Form zu programmieren. Ein IRL-Programm kann gegebenenfalls in ICR (Intermediate Code for Robots, ISO/TR 10562) überführt werden.

Mit der Sprache IRL steht ein Hilfsmittel zur Verfügung, mit dem eine einheitliche, von speziellen Industrierobotern und -steuerungen unabhängige Programmierung ermöglicht wird. Zielgruppe für die Programmierung mit IRL sind alle Anwender, die in ihren Fertigungsanlagen Industrieroboter einsetzen.

Für das Bedienungspersonal ist im Anhang C ein Subset definiert, der die für die Erfüllung der Aufgaben des Bedienungspersonals notwendige Teilmenge von IRL beschreibt.

Die Darstellungsform, die für die Beschreibung der Syntax gewählt wurde, basiert auf der Backus-Naur-Form (BNF).

Der zuständige Arbeitsausschuß 81.4 im Fachbereich Montage- und Handhabungstechnik des Normenausschusses Maschinenbau (NAM) ist der Meinung, daß Anwender dieser Norm Englisch als Fachsprache verstehen. Daher wird der englische Originaltext veröffentlicht.

Das Präsidium des DIN hat auf seiner 36. Sitzung bestätigt, daß DIN-Normen, deren Inhalt sich auf internationale Arbeitsergebnisse aus dem Sachgebiet Informationsverarbeitung gründet, in englischer Sprache veröffentlicht werden dürfen.

Literaturhinweise

ISO 646:1983

Information processing – ISO 7-bit coded character set for information interchange

ISO 2382-1:1984

Data processing – Vocabulary – Part 1: Fundamental terms, Bilingual edition

ISO 2382-9:1984

Data processing – Vocabulary – Part 9: Data communication, Bilingual edition

ISO 2806:1980

Numerical control of machines – Vocabulary, Bilingual edition

ISO 7185:1983

Programming languages – PASCAL (Endorsement of British standard BS 6192 – 1982)

ISO/TR 10562

Manipulating industrial robots – Intermediate Code for Robots (ICR)

Fortsetzung Seite 2 bis 116

Normenausschuß Maschinenbau (NAM) im DIN Deutsches Institut für Normung e. V.
Normenausschuß Informationstechnik (NI) im DIN

Änderungen

Gegenüber DIN 66312-1:1993-06 wurden folgende Änderungen vorgenommen:

Allgemeines: Umstellung der Grammatikdarstellung und Vereinheitlichung der Darstellungen zu Konstanten, Typen und Variablen. Das betrifft die Struktur der Grammatik, deren Darstellung und deren Widerspiegelung im Aufbau des Textes, aber nicht das Erscheinungsbild oder den Aufbau der Sprache selbst oder deren Semantik. Einige der Ausnahmeregelungen (z. B. zu Teach Types) sind bereits in der kontextfreien Grammatik dargestellt – die automatisierbare Analyse ist weiterhin gewährleistet.

- Titel Teil 2 statt Teil 1 – enthält aber den aktualisierten Teil 1
- neue Formatierung mit Times-Roman, dadurch Komprimierung des Layouts und neues Inhaltsverzeichnis
- Korrektur diverser Verweise, z. B. auf 4.6.7 Multi Robot Handling (jetzt 4.6.9)

4.1.2.1 Word Symbols: Ergänzung neuer Schlüsselwörter

4.1.2.2 Identifiers: teilweise Einführung neuer Arten von Identifier und dementsprechend Ergänzung neuer Regeln

- Umbenennung von TeachConstantIdentifier zu PermanentConstantIdentifier

4.1.2.3 Predefined Identifiers: Änderung bzw. Erweiterung der Definition sowie Erläuterung der vordefinierten Variablen

- Ergänzung neuer vordefinierter Variablen (für SynactStatement) und Einführung vordefinierter Konstanten
- R_TARGET_BASE und R_BASE ergänzt bzw. modifiziert sowie die Tabelle der Konstanten ergänzt; R_SPEC_TYPE wird überall um R_BASE ergänzt
- Streichen von R_TARGET_BASE
- weitere vordefinierte Bezeichner (Funktionen, Konstanten, Variablen, Typen)
- neue Konstante NO_PRED

4.1.3.4 Record Constants: neue Grammatikregel CallOrRecord

4.1.3.5 Boolean Constants: Einführung einer Regel für Boolesche Konstanten, verbunden mit der Umwandlung von FALSE und TRUE von Konstanten zu Wortsymbolen und mit der einheitlichen Kleinschreibung von false (= 0) und true (> 0), soweit es sich nicht um diese Wortsymbole handelt

- durchgängige Schreibweise: true (≠ 0) statt bisher true (> 0)

4.2.2 Comment: neue Regel (geschachtelte Kommentare) und korrigierter Text

4.3.1 Constant Definition: neue Grammatikregel ConstantDefiniton

4.3.1.1 Usual Constant Definition: neue Grammatikregel UsualConstantDefiniton

4.3.1.2 Teach Constant Definition: neue Grammatikregel TeachConstantDefinition

- Ersetzung von TeachConstant... durch PermanentConstant...; hier und andernorts: Unterscheidung zwischen permanenten Variablen bzw. Konstanten und TeachTypes

4.3.2 Type Declaration: neuer Abschnitt und Neuorganisation der Grammatik der Typdeklarationen (predefined = simple or standard or system, userdefined = structured or teach or signal)

- Einordnung des neuen Datentyps LIST
- Einordnung des neuen Datentyps SemaphoreType
- D_SPEC_TYPE ergänzt
- Differenzierung zwischen teach und permanent

4.3.2.1 Type Description and Type Identifier: neuer Abschnitt und neue Regel

4.3.2.2 Predefined Data Types: neuer Abschnitt und neue Regel

4.3.2.2.1 Simple Data Types: neue Regel und neuer Identifier

4.3.2.2.2 Standard Data Types: neue Regel und neuer Identifier, neue Absätze zur Einordnung von Position, Orientation und Pose

- **Orientation:** Die neue Orientierungsdarstellung über die Einheitsvektoren wird ergänzt
- im Text durchgängige Schreibweise: "≤" statt bisher "<=" – in der IRL-Syntax und in den Beispielen bleibt es natürlich bei "<="
- **Predefined Geometric Data Types:** Im letzten Absatz wird vor R_SPEC_TYPE der neue Datentyp D_SPEC_TYPE ergänzt.

4.3.2.2.3 System Data Types: neue Regel und neuer Identifier, 5 neue predefined System Data Types (MAIN_JOINT, ADD_JOINT, JOINT, ROBTARGET and R_SPEC_TYPE)

- **Robot Specification Type:** Erweiterung des vordefinierten Records R_SPEC_TYPE als Konsequenz der Änderungen zu 4.1.2.3
- Es gibt jetzt 6 System Data Types; vor R_SPEC_TYPE muß D_SPEC_TYPE ergänzt werden
- Vor dem Unterabschnitt **Robot Specification Type** wird ein neuer Unterabschnitt **Device Specification Type** ergänzt.
- **Robot Specification Type:** Ergänzung von R_DEVICES

4.3.2.2.4 Cast Operators on Predefined Types: Neuordnung des Abschnitts, Präzisierung der Konvertierung

4.3.2.3 Userdefined Data Types: neuer Abschnitt und neue Regel

4.3.2.3.1 Structured Data Types: neue Regel und neuer Identifier; Erläuterung von Record Types, Array Types und File Types

4.3.2.3.1.3 List Types: Erweiterung der Structured Data Types und Einfügen der Listen nach Feldern

- Ergänzung der Alternative TeachType in den Syntaxregeln

4.3.2.3.2 **Teach Types:** Neueinordnung und neue Regeln

- Differenzierung zwischen teach und permanent

4.3.2.3.3 **Signal Types:** Neueinordnung und neue Regeln

4.3.2.3.4 **Semaphore Data Types:** Erweiterung der User Defined Data Types und Einfügen der Semaphoren nach Signals

4.3.3 **Declaration of Variables:** Neuorganisation der Abschnittsstruktur und neue Grammatikregeln

- permanente Variablen in der default data list
- Einführen von Shared Variables
- Differenzierung zwischen teach und permanent

4.3.3.1 **Declaration Without Initialisation:** neue präzisierte Grammatikregeln

- Listen als Variablen ohne Initialisierung
- Behandlung von "undef" Variablen
- Initialisierung von Semaphoren
- Differenzierung zwischen teach und permanent

4.3.3.2 **Declaration With Initialisation:** neue präzisierte Grammatikregeln

- Listen als Variablen mit Initialisierung

4.3.3.3 **Interface Declarations:** Neueinordnung und teilweise präzisierte Regeln zu Signals und Channels

4.3.4 **Interrupt Declaration:** Einführung der Grammatikregel zur Interrupt-Deklaration

4.4 **Compiler Directive:** neuer Absatz: implizit deklarierte ROBTARGET-Variablen sind auch automatisch permanent

4.5.1 **Operators:** Neuordnung der Regeln zur Unterscheidung unärer und binärer Operatoren – der logische Operator "NOT" sowie unäres "+" und "-" (Vorzeichen) werden statt bei Objects jetzt bei Expressions in die Syntax integriert

4.5.2 **Objects:** Listen und Listenelemente als Bestandteile von Objekten

4.5.3 **Expressions:** Neuordnung der Regeln zur Unterscheidung unärer und binärer Operatoren – der logische Operator "NOT" sowie unäres "+" und "-" (Vorzeichen) werden statt bei Objects jetzt bei Expressions in die Syntax integriert

- Einordnung der BooleanConstants in die Expression-Syntax

4.5.3.2 **Function Call:** Geänderte Grammatikregel

4.6.1 **Construction of Statements:** Ergänzung der ParallelProcessStatements

4.6.2 **Assignment:** Zuweisung von Listen untereinander (assignment compatibility)

4.6.3 **Procedure Call:** Geänderte Grammatikregel

4.6.4 **Program Flow:** Einordnung der Grammatikregel zum InterruptStatement und zum ResumeStatement

- Einordnung der Grammatikregel zum SynactStatement in die ProgramFlowStatement-Syntax
- Umbenennung der Regel ProgramStopStatement in ProgramHaltStatement

4.6.4.4.1 **FOR Loop:** Präzisierung der Semantik

- Korrektur des Textes zum ForStatement

4.6.4.8 **Programhalt Statement:** Umbenennung der Regel ProgramStopStatement in ProgramHaltStatement bei Beibehaltung der Syntax mit "HALT"

4.6.4.9 **Interrupt Statement:** Einfügen des Interrupt-Statements mit neuer Grammatikregel

- Korrektur des Textes zum InterruptStatement

4.6.4.10 **Resume Statement:** neue Grammatikregel, neuer Abschnitt und Beispiel

4.6.4.11 **Synchronized Action Statement:** Einführung des SynactStatement

4.6.5 **Movement Statements:** Präzisierung der Grammatikregel für Path

- Einordnung der Grammatikregel zum BrakeStatement

4.6.5.1 **Path and PTP Statements:** neue Abschnittsnummer und Überarbeitung der Regeln und der Tabellen

- Überarbeitung der Tabellen (%-Zeichen bei Erläuterung und Beispielen der Faktor-Einheiten (Werte <= 100.0))
- MOVE-Befehl, insbesondere PATH
- **CP additional parameters:** Ergänzung der Regel und der Tabelle
- Ergänzung der Regeln und der Tabellen um die ALTER_BY-Spezifikation

4.6.5.2 **Brake statement:** neue Grammatikregel und neuer Abschnitt

4.6.6 **I/O-Statements:** Neunummerierung und Einordnung der Grammatikregel zum PulseStatement

4.6.6.1.2 **Pulse statement:** neue Grammatikregel und neuer Abschnitt

- Änderung der Syntax des PulseStatements

4.6.6.2 **I/O for Channels and Files:** Korrektur der Grammatikregel

4.6.6.3 **Processing Sensor Data in IRL:** Erwähnung der Möglichkeiten zum Handhaben von Sensoranweisungen

4.6.7 **Multi robot handling:** Präzisierung der Grammatikregel

- Verschiebung nach 4.6.9

4.6.7 List Control Statements: Ergänzung der Statements LISTINS, LISTADD und LISTDEL zum Einfügen, Anhängen bzw. Löschen eines Elementes sowie der Statements LISTLENGTH und LISTINDEX

4.6.8 Parallel Processing Statements: Einfügen aller parallele Prozeß- und Semaphoren betreffenden Anweisungen

4.6.8.3 Getting the status of a process: Korrektur der Abbildung

4.6.10 Devices and Groups (Additional Axes): neuer Abschnitt

- externe Routinen im Beispiel

4.7 Procedures, Functions and Parallel Processes: Ergänzung des Abschnittes über parallele Prozesse

4.7.1 Procedures: Präzisierung der Grammatikregel

4.7.1.2 External Procedures: neuer Unterpunkt und neue Regel zur Einführung der ExternalProcedureDeclaration

4.7.2 Functions: Präzisierung der Grammatikregel

4.7.2.2 External Functions: neuer Unterpunkt und neue Regel zur Einführung der ExternalFunctionDeclaration

4.7.3 Formal Parameters: Ergänzung bei parameter compatibility

- Parameterübergabe (parameter compatibility)

4.7.4 Predefined Functions: Verweis auf Orientierungsfunktionen

- Korrektur der Erklärung von TRAFO und TRAFOINV

4.7.5 Parallel Processes: Einfügen des Abschnittes über parallele Prozesse und dabei insbesondere deren Definition

- Streichen des dritten Absatzes

4.8.1 Programs and Program Parts: neuer Absatz zum Aufbau des IRL-Programms mit Unterscheidung von ProgramDeclarationPart und DeclarationPart

- Einordnung der External ... Declarations in die Syntax der RoutineList
- Ergänzung der Syntax

4.8.2 SystemSpecification: Überarbeitung des Beispiels (Default-Werte und Record-Struktur)

- Überarbeitung des Beispiels (Record-Struktur von R_SPEC_TYPE)
- Die System Specification wurde erheblich erweitert und teilweise korrigiert. R_DEF_ORI wird nicht unter den Konstanten, sondern unter den Variablen definiert. Bisher fehlende Deklarationen werden ergänzt
- hier und andernorts Berücksichtigung von NO_PRED
- hier und andernorts Record-Konstanten immer mit Typ-Prefix

4.8.3 Declaration Parts: Absätze und Regeln zur Unterscheidung von DeclarationPart und ProgramDeclarationPart

- Einordnung der Grammatikregel zur InterruptDeclaration

4.8.3.1 Import- and Export Declarations: Präzisierung des Textes

4.8.3.3 Data Lists: Präzisierung der Grammatikregel

- Ergänzungen
- statt des ersten Absatzes (bis vor die Spiegelstriche): Zuordnung von Variablen zu Datenlisten und Gültigkeit der Werte der Variablen
- weitere Differenzierung zwischen teach und permanent

4.9 Scopes: Präzisierungen in der Semantik

4.10 Units: %-Zeichen bei Erläuterung der Faktor-Einheiten

Annex A: Aktualisierung der Syntax entsprechend den Änderungen im Text

Annex B: Ergänzung der neuen vordefinierten Funktionen

Annex C: Übernahme der neuen Grammatikregeln, Schlüsselwörter, vordefinierten Variablen, Kommentare, Move-Befehle und Boolesche Konstanten

- R_BASE bei den vordefinierten Variablen ergänzt

Annex D: zuerst **keine** Änderungen notwendig, da die Sprache erhalten bleibt

- Anpassung der Beispiele an den neuen R_SPEC_TYPE
- Anpassung der Beispiele an den neuen R_SPEC_TYPE (%-Werte in den Beispielen der Faktor-Einheiten (Werte ≤ 100.0))

D.2 Example for Welding / D.3 Example for Painting with external axis: Deklaration von R_BASE und R_TOOL als permanente Konstanten mittels CONST und PERMANENT; statt R_BASE jetzt R_TARGET_BASE .

- Anpassung der Beispiele entsprechend den Festlegungen zur Nichtdefinition von R_TARGET_BASE
- Nutzen der additional axes in einer neuen Anweisung

D.4 Simple Example: die INPUT BOOL Variable continue muß umbenannt werden, da CONTINUE jetzt ein Schlüsselwort ist

D.5 Parallel Process Example und D.6 Simple Sensor Example: neu

Frühere Ausgaben

DIN 66312-1:1993-06

Inhalt

	Seite		Seite
Contents			
1 Field of Application	7	4.6.2 Assignment	42
2 Terms	7	4.6.3 Procedure Call	43
3 Definition Conventions	7	4.6.4 Program Flow	43
4 IRL Language Description	7	4.6.4.1 Goto Statement	43
4.1 Lexical Elements	7	4.6.4.2 Conditional Branch Statement	43
4.1.1 General Remarks	8	4.6.4.3 Case Statement	44
4.1.1.1 Character Set	8	4.6.4.4 Loop Statement	44
4.1.1.2 Letters	8	4.6.4.4.1 FOR Loop	44
4.1.1.3 Digits	8	4.6.4.4.2 REPEAT Loop	45
4.1.2 Symbols and Identifiers	8	4.6.4.4.3 WHILE Loop	45
4.1.2.1 Word Symbols	8	4.6.4.5 Wait Statement	46
4.1.2.2 Identifiers	9	4.6.4.6 Pause Statement	46
4.1.2.3 Predefined Identifiers	9	4.6.4.7 Return Statement	46
4.1.3 Constants	13	4.6.4.8 Programhalt Statement	46
4.1.3.1 Integer Constants	13	4.6.4.9 Interrupt Statement	47
4.1.3.2 Real Constants	14	4.6.4.10 Resume Statement	47
4.1.3.3 Characters and Strings	15	4.6.4.11 Synchronized Action Statement	48
4.1.3.4 Record Constants	15	4.6.5 Movement Statements	49
4.1.3.5 Boolean Constants	15	4.6.5.1 Path and PTP Statements	49
4.1.3.6 Range of Value	16	4.6.5.2 Brake statement	55
4.2 Separation of Lexical Elements	16	4.6.6 I/O-Statements	55
4.2.1 General Remarks	16	4.6.6.1 I/O for Signals	55
4.2.2 Comment	16	4.6.6.1.1 Signals	55
4.3 Declarations and Definitions	16	4.6.6.1.2 Pulse statement	56
4.3.1 Constant Definition	16	4.6.6.2 I/O for Channels and Files	56
4.3.1.1 Usual Constant Definition	16	4.6.6.2.1 OpenStatement	56
4.3.1.2 Permanent Constant Definition	17	4.6.6.2.2 CloseStatement	57
4.3.2 Type Declaration	17	4.6.6.2.3 WriteStatement	57
4.3.2.1 Type Description and Type Identifier	17	4.6.6.2.4 ReadStatement	59
4.3.2.2 Predefined Data Types	18	4.6.6.3 Processing Sensor Data in IRL	59
4.3.2.2.1 Simple Data Types	18	4.6.7 List Control Statements	59
4.3.2.2.2 Standard Data Types	18	4.6.8 Parallel processes statements	60
4.3.2.2.3 System Data Types	24	4.6.8.1 Controlling parallel processes	60
4.3.2.2.4 Cast Operators on Predefined Types	26	4.6.8.1.1 Starting a process	60
4.3.2.3 Userdefined Data Types	27	4.6.8.1.2 Stopping a process	61
4.3.2.3.1 Structured Data Types	27	4.6.8.1.3 Continuing a process	61
4.3.2.3.1.1 Record Types	27	4.6.8.1.4 Terminating a process	61
4.3.2.3.1.2 Array Types	28	4.6.8.2 Controlling shared variables	61
4.3.2.3.1.3 List Types	28	4.6.8.2.1 Locking variables	61
4.3.2.3.1.4 File Types	28	4.6.8.2.2 Unlocking variables	61
4.3.2.3.2 Teach Types	29	4.6.8.3 Getting the status of a process	62
4.3.2.3.3 Signal Types	29	4.6.8.4 Synchronization of parallel processes using semaphores	63
4.3.2.3.4 Semaphore Data Types	29	4.6.8.4.1 Binary Semaphores	63
4.3.3 Declaration of Variables	30	4.6.8.4.2 Integer Semaphores	64
4.3.3.1 Declaration Without Initialization	30	4.6.9 Multi robot handling	64
4.3.3.2 Declaration With Initialization	31	4.6.10 Devices and Groups (Additional Axes)	65
4.3.3.3 Interface Declarations	32		
4.3.3.3.1 Signals	33	4.7 Procedures, Functions and Parallel Processes	69
4.3.3.3.2 Channels	33	4.7.1 Procedures	69
4.3.4 Interrupt Declaration	34	4.7.1.1 IRL Procedures	69
4.4 Compiler Directive	34	4.7.1.2 External Procedures	69
4.5 Operators, Objects, Expressions	34	4.7.2 Functions	69
4.5.1 Operators	34	4.7.2.1 IRL Functions	69
4.5.2 Objects	35	4.7.2.2 External Functions	70
4.5.3 Expressions	36	4.7.3 Formal Parameters	70
4.5.3.1 Interpretation of Expressions	36	4.7.4 Predefined Functions	73
4.5.3.1.1 Interpretation of Arithmetical Expressions	36	4.7.5 Parallel processes	75
4.5.3.1.2 Interpretation of Logical Expressions	37	4.7.5.1 Definition of parallel processes	75
4.5.3.1.3 Interpretation of Geometric Expressions	37		
4.5.3.2 Function Call	41	4.8 Program	76
4.6 Statements	41	4.8.1 Programs and Program Parts	76
4.6.1 Construction of Statements	41	4.8.2 System Specification	77
		4.8.3 Declaration Parts	80
		4.8.3.1 Import- and Export Declarations	81
		4.8.3.2 Modular Programming and Testing	82
		4.8.3.3 Data Lists	82

	Seite		Seite
4.9 Scopes	83	C.5 Compiler Directives	103
4.10 Units	84	C.6 Operators	103
Annex A – BNF syntax	85	C.7 Statements	104
A.1 Syntax rules	85	C.8 Procedures and Functions	106
A.2 Alphabetical index of syntax rules	97	C.9 Program	106
Annex B – Declaration of predefined functions	99	Annex D – Examples	108
Annex C – Worker Subset	101	D.1 Complex Example	108
C.1 Introduction	101	D.2 Example for Welding	110
C.2 Lexical Elements	101	D.3 Example for Painting with external axis	112
C.3 Separation of Lexical Elements	102	D.4 Simple Example	113
C.4 Declarations and Definitions	103	D.5 Parallel Process Example	114
		D.6 Simple Sensor Example	116

1 Field of Application

This standard describes the common construction of a higher programming language for industrial robots (IRL = Industrial Robot Language). It defines the syntax and the semantics of IRL. IRL allows the user of this standard to program the movement of the robot and the logical program flow in an user oriented form. An IRL program can be transferred into an ICR (Intermediate Code for Robots) form by an appropriate service program.

2 Terms

See the German Standards:

DIN 44300	Informationsverarbeitung; Begriffe
DIN 44302	Informationsverarbeitung; Datenübertragung, Datenübermittlung; Begriffe
DIN 66256	Informationsverarbeitung; Programmiersprache Pascal
DIN 66257	Numerisch gesteuerte Arbeitsmaschinen; Begriffe

3 Definition Conventions

The meta language used to describe the syntax of IRL in this standard is based on the Backus Naur Form (BNF). The notation was changed and extended to accomplish an easy description and to convert recursive productions into iterative ones. For the definition of the syntax of the IRL language in this standard, the meta symbols according to table 1 are used.

Table 1 Meta symbols for the description of the IRL language

meta symbol	description
=	defined as
	alternative
.	end of definition
[x]	no or one appearance of x
{x}	no, one or multiple appearance of x
(x y)	select x or y
"xyz"	terminal symbol xyz
Identifier	nonterminal symbol

Example:

RealConstant =
DecimalConstant Exponent |
DecimalConstant "." { *Digit* } [*Exponent*] |
 "." *DecimalConstant* [*Exponent*].

DecimalConstant, Digit and Exponent are nonterminal symbols.
 "." is a terminal symbol.

4 IRL Language Description

4.1 Lexical Elements

The syntax in this section describes the formation of lexical elements out of characters and the separation of these elements from each other. Therefore, the syntax is not of the same kind as that used in the rest of this standard.

4.1.1 General Remarks

The lexical elements of an IRL program are:

- Word symbols
- Identifiers
- Predefined Identifiers
- Characterstrings
- Numbers

4.1.1.1 Character Set

The character set is taken from the ISO 646 (7 bits coded character set for information processing interchange) in its current edition.

4.1.1.2 Letters

Letter =

"a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" |
"n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" |
"A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" |
"N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" |
" _ .

Outside of strings the case is ignored.

Example:

The identifiers HOME, Home, hOme are equivalent.

4.1.1.3 Digits

Digit =

"0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".

4.1.2 Symbols and Identifiers

4.1.2.1 Word Symbols

Word symbols are reserved which means they are not allowed to be identical with identifiers.

ACC	ACC_PTP	ACT_ROB
ADAX CONTROL	ALL_	AND_
ARRAY	AT	BRAKE
BEGIN	CANCEL	CASE
C PASS	CIRCLE	CLOSE
CONST	CONTINUE	C_CP
C_PTP	C_SPEED	DATALIST
DECL	DECLOFF	DECLON
DECLSYS	DEFAULT	DELAY
DEVICE	DISABLE	DIV
DO	DURATION	ELSE
ENABLE	ENDCASE	ENDERROR
ENDFCT	ENDFOR	ENDIF
ENDPROC	ENDPROCESS	ENDPROGRAM
ENDRECORD	ENDWHILE	ERROR
EXOR	EXPORT	EXTERNALFUNCTION
EXTERNALPROCEDURE	FALSE	FAST
FILE	FOR	FROM
FUNCTION	GETSTATUS	GOTO
HALT	HIGH	IF
IMPORT	IN	INOUT
INPUT	INTERRUPT	INTO
LIN	LIST	LISTADD
LISTDEL	LISTINDEX	LISTINS
LISTLENGTH	LOCK	LOW

MOD	MOVE	MOVE_INC
NOT	OF	OFF
ON	OPEN	OR
OUT	OUTPUT	PATH
PAUSE	PERMANENT	PROCEDURE
PROCESS	PROGRAM	PTP
PULSE	READ	READLN
READRAC	RECORD	REPEAT
RESUME	RETURN	ROBOT
SEC	SEMAPHORE	SEMA_SIGNAL
SEMA_WAIT	SHARED	SPEED
SPEED_ORI	SPEED_PTP	STANDARD_IN
STANDARD_OUT	START	STEP
STOP	SYNACT	SYSTEM_SPECIFICATION
TEACH	THEN	TIME
TIMEOUT	TO	TRUE
TYPE	UNLOCK	UNTIL
VAR	WAIT	WHEN
WHILE	WOBBLE	WRITE
WRITELN	WRITERAC	

4.1.2.2 Identifiers

Identifiers are allowed to be of any length. All characters of one identifier are significant. All identifiers have to be different from word symbols.

Identifier =
Letter { Letter | Digit }.

For a better readableness the following identifiers are distinguished in this standard:

<i>ProgramIdentifier</i> =	<i>Identifier.</i>
<i>PermanentConstantIdentifier</i> =	<i>Identifier.</i>
<i>ConstantIdentifier</i> =	<i>Identifier.</i>
<i>StructuredTypeIdentifier</i> =	<i>Identifier.</i>
<i>SimpleTypeIdentifier</i> =	<i>Identifier.</i>
<i>StandardTypeIdentifier</i> =	<i>Identifier.</i>
<i>SystemTypeIdentifier</i> =	<i>Identifier.</i>
<i>TeachTypeIdentifier</i> =	<i>Identifier.</i>
<i>SignalTypeIdentifier</i> =	<i>Identifier.</i>
<i>SemaphoreTypeIdentifier</i> =	<i>Identifier.</i>
<i>VariableIdentifier</i> =	<i>Identifier.</i>
<i>SignalIdentifier</i> =	<i>Identifier.</i>
<i>ProcedureIdentifier</i> =	<i>Identifier.</i>
<i>FunctionIdentifier</i> =	<i>Identifier.</i>
<i>ProcessIdentifier</i> =	<i>Identifier.</i>
<i>TypeIdentifier</i> =	<i>Identifier.</i>
<i>LabelIdentifier</i> =	<i>Identifier.</i>
<i>ComponentIdentifier</i> =	<i>Identifier.</i>

4.1.2.3 Predefined Identifiers

System Constants and System Variables

Predefined constants and variables do not have to be declared by the user and they are to be used just as normal constants or variables, respectively.

The CSR name is the name of the variable or component with the same meaning in MMS Companion Standard for Robots (ISO 9506-3).

The names of the predefined constants, variables and components always start with "R_".

The following predefined variables exist:

variable name	type	write protection meaning
R_ACT_ROB	R_SPEC_TYPE	n actual robot
R_DEF_ORI	STRING	n default orientation mode
trigger variables used in SynactStatements (see 4.6.4.11)		
R_MOVE_TIME	REAL	y time gone from the start of the actual robot movement
R_SCALE_FORW	REAL	y factor (%) of the movement distance already covered
R_SCALE_BACK	REAL	y factor (%) of the movement distance still to cover
R_DISTANCE_FORW	REAL	y distance from the start point (only LIN and CIRCLE movements)
R_DISTANCE_BACK	REAL	y distance to the end point (only LIN and CIRCLE movements)

The following predefined constants exist. Their values have to be defined in the System Specification:

constant name	type	fixed value	meaning
R_NTURNS	INT		number of joints that can rotate more then 360 degrees
R_NADDDJ	INT		max. number of additional joints
R_NJ	INT		number of joints
RS	STRING	'RS'	constants specifying different
XYZ	STRING	'XYZ'	descriptions of orientations
YXZ	STRING	'YXZ'	used in system functions to assign
ZYX	STRING	'ZYX'	orientations (4.3.2.2.2) , write
ZYZ2	STRING	'ZYZ2'	statements (4.6.6.2.3) and read
VANG	STRING	'VANG'	statements (4.6.6.2.4)
QUAT	STRING	'QUAT'	
NO_ADAX_CONTROL	INT	0	constants specifying the type of coordinated movement (see 4.6.5.1 and 4.6.10), here the default
TCP_KEEP	INT	1	world coordinate TCP position is kept
TCP_KEEP_REL	INT	2	TCP position relative to a device is kept
DEVICE_REL	INT	3	movement execution relative to a device
R_PROCESS_READY	INT	0	constants returned by the system function getstatus (4.6.8.3), here process ready
R_PROCESS_RUNNING	INT	1	process running
R_PROCESS_STOPPED	INT	2	process stopped
R_PROCESS_BLOCKED	INT	3	process blocked
R_DEVICE_NUMBER	INT		number of devices in the cell
AX_NONE	INT	-1	placeholder for a non device in the device group specifications (4.6.10)
AX_MAIN	INT	0	constants specifying different device types, here the main axes (robot)
ADAX_IS	INT	1	additional axes starting the kinematical chain, e. g. a rail
ADAX_IE	INT	2	additional axes ending the kinematical chain
ADAX_EX	INT	3	external additional axes, e. g. a positioner
NO_PRED	INT	-1	tag for devices having no predecessor in the kinematical chain (see D_SPEC_TYPE)

For every robot defined by a Robot Specification (see 4.6.9) a predefined record with the name of the robot exists. Every record consists of the following components. Some of these components are write protected (y) outside the System Specification (see 4.8.2) and are not allowed to be used on the left side of an user written assignment. The rest of them is not write protected (n).

component name	CSR name	type	write protection meaning
R_ACC		REAL	n acceleration (path)
R_ACC_PTP		ARRAY[1..R_NJ] OF REAL	n acceleration factor (axes, %)
R_C_PTP		REAL	n path smoothing: axis distance (factor, %)
R_C_CP		REAL	n path smoothing: distance to the intermediate point
R_C_SPEED		REAL	n path smoothing: (% of programmed speed)
R_JOINT_ACT	R_AJV	JOINT	y actual robot position in robot coordinate system
R_ROBTARGET_ACT		ROBTARGET	y actual robot position
R_ROBTARGET_ACT.PSE			
R_ROBTARGET_START	R_CTUP	POSE	y actual tool point (commanded pose)
		ROBTARGET	y robot position at the start of the actual movement statement
R_ROBTARGET_END		ROBTARGET	y robot position to be reached at the end of the actual movement statement
R_ROBTARGET_INTER		ROBTARGET	y robot position at the last executed interrupt
R_SPEED_ACT		REAL	y actual speed (path)
R_SPEED	R_PSTU	REAL	n speed (path)
R_SPEED_PTP		ARRAY[1..R_NJ] OF REAL	n speed factor (axes, %)
R_SPEED_ORI		REAL	n orientation speed
R_BASE	R_TUB	POSE	n actual robot base coordinate system
R_TOOL	R_TTM	POSE	n actual tool correction

For more explanation see 4.6.5.

A commanded pose describes that pose which will be commanded to the transformation algorithm.

Predefined system variables which are not write protected had to be set by a default value inside the System Specification.

Functions

ABS	ACOS	ANGLEX
ANGLEY	ANGLEZ	ANGLEZ2
APPENDC	ASIN	ATAN2
CHR	COMPARE	CONCATS
COS	CROSS	DELETC
DOT	EOF	EOL
EXTRACTS	FLOAT	GETCHR
INV	LENGTH	LOCATE
ORD	ORIMAT	ORQUAT
ORIRS	ORIVANG	ORIXYZ
ORIYZX	ORIZYX	ORIZYZ2
QUATA	QUATB	QUATC
QUATD	ROTANGLE	ROTAXIS
ROUND	SETCHR	SIN

SIZE
TRAFO
VECTORA

SQRT
TRAFOINV
VECTORN

TAN
TRUNC
VECTORO

Channels

SER_1 .. SER_x
serial interfaces (x = number of interfaces)
PAR_1 .. PAR_x
parallel interfaces (x = number of interfaces)
KEYBOARD
PANEL
PRINTER
SCREEN
TTY
HPU (hand programming unit)
MCP (manual control panel)

Constants

ADAX_EX	ADAX_IE	ADAX_IS
AX_MAIN	AX_NONE	DEVICE_REL
NO_ADAX_CONTROL	NO_PRED	QUAT
RS	R_DEVICE_NUMBER	R_NADDJ
R_NJ	R_NTURNS	R_PROCESS_BLOCKED
R_PROCESS_READY	R_PROCESS_RUNNING	R_PROCESS_STOPPED
TCP_KEEP	TCP_KEEP_REL	VANG
XYZ	YXZ	ZYX
ZYZ2		

Types

ADD_JOINT	BOOL	CHAR
D_SPEC_TYPE	INT	JOINT
MAIN_JOINT	ORIENTATION	POSE
POSITION	REAL	R_SPEC_TYPE
STRING	TEXT	ROBTARGET

Variables

R_ACT_ROB	R_DEF_ORI	R_DISTANCE_BACK
R_DISTANCE_FORW	R_MOVE_TIME	R_SCALE_BACK
R_SCALE_FORW		

4.1.3 Constants

4.1.3.1 Integer Constants

IntegerConstant =
DecimalConstant |
BinaryConstant |
OctalConstant |
HexadecimalConstant.

DecimalConstant:

A decimal constant consists of a sequence of digits. It is of type "INT" (Integer) and interpreted as decimal.

DecimalConstant =
Digit { *Digit* }.

A decimal constant is unsigned and it is positive. Negative constants are built with Expression.

The maximum number of digits is limited by the range of the integer data type.

Note:

The number of digits depends on the implementation; for constant computation the compiler shall use always at least 32 bit (see 4.1.3.6).

BinaryConstant:

The following notation stands for the constant which is interpreted to base 2:

$$\text{BinaryConstant} = \\ \text{"2\#" ("0" | "1" | " ")} \\ \{ \text{"0" | "1" | "_"} \}.$$

The symbol "2#" can be followed by the digits 0 or 1 or by underlines. The maximum number of digits is limited by the range of the integer data type. The last digit is the least significant one.

OctalConstant:

The following definition stands for the constant which is interpreted to base 8:

$$\text{OctalConstant} = \\ \text{"8\#" ("0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | " ")} \\ \{ \text{"0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "_"} \}.$$

The symbol "8#" can be followed by the digits 0 to 7 or by underlines. The maximum number of digits is limited by the range of the integer data type. The last digit is the least significant one.

HexadecimalConstant:

Constants can also be hexadecimal which means that they are interpreted to base 16. The letters A to F correspond to the decimal values 10 to 15.

$$\text{HexadecimalConstant} = \\ \text{"16\#" (Digit | "A" | "B" | "C" | "D" | "E" | "F" | " ")} \\ \{ \text{Digit | "A" | "B" | "C" | "D" | "E" | "F" | "_"} \}.$$

The symbol "16#" may be followed by hexadecimal characters or by underlines. The maximum number of the hexadecimal characters is limited by the range of the integer data type. The character which is positioned on the far right side has the least significance.

Note:

The character '_' in non decimal integer constants serves only to format the constant for human convenience.

4.1.3.2 Real Constants

$$\text{RealConstant} = \\ \text{DecimalConstant Exponent} | \\ \text{DecimalConstant "." \{ Digit \} [Exponent]} | \\ \text{"." DecimalConstant [Exponent]}.$$

Exponent =
("E" | "e") ["+" | "-"] *DecimalConstant*.

A real constant is unsigned and it is positive. Negative constants are built with Expression.

Example:

5.
100.07
.3
23.4E5
10.2E-3

4.1.3.3 Characters and Strings

CharacterString =
"" { *Any-Character-Except-""* | "" } "".

All characters of the character set are permitted.

A CharacterString which consists of one character only is of type CHAR and is called a character constant.

A CharacterString with more than one character is of type STRING and is called a string constant. The maximum length of a string constant should be restricted to 255.

Note:

To include a quote mark (character ') in a CharacterString, it has to be written twice.

Example:

'alarm'
'abc' 'def'

4.1.3.4 Record Constants

CallOrRecord =
Object "(" [*ValueList*] ")".

ValueList =
Expression { "," *Expression* }.

Every Expression in ValueList represents the value of a record component. The number and the sequence of components is fixed by the type declaration.

Object describes the type of the record constant.

Example:

POSITION(17.2, -20.4, 33.4)

In the case of nested records the component value is again a record constant.

Example :

JOINT (MAIN_JOINT(10.2, -220.4, 135.5, -90.0, 0.0, 90.0),
ADD_JOINT(-90.0))

4.1.3.5 Boolean Constants

BooleanConstant = "TRUE" | "FALSE".

Boolean constants can only be true ($\neq 0$) or false ($= 0$). They are represented by the word symbols TRUE or FALSE, respectively.

4.1.3.6 Range of Value

The minimum range of the integer data type is:

-2147483647 to +2147483647 (32 bits)

The minimum range of the real data type is (32 bits):

-1.7 E38 to -1.7 E-38
1.7 E-38 to 1.7 E38
and 'TrueZero'

The accuracy is not fixed.

Note:

For calculation of constant expressions the compiler always works with a precision of 32 bits.

The range of the boolean data type is true ($\neq 0$) and false ($= 0$).

These values represent a minimum demand for an IRL implementation.

4.2 Separation of Lexical Elements

4.2.1 General Remarks

Blanks (except in character strings, blank = ASCII-Code according to ISO 646: '32'decimal), tabs (ASCII-Code '9'decimal), end-of-lines (ASCII-Code '13'decimal and '10'decimal) and comments are used as separators between lexical elements. Zero, one or more separators are allowed to appear between lexical elements. A separator is not allowed within lexical elements.

4.2.2 Comment

Comment =
"*" { CommentCharStream | Comment } "*"

CommentCharStream =
{ Any-Character-Exept-"{"-or-"} ".

Accepted are all characters of the character set (incl. end-of-line), except of the opening and closing curly bracket. Comments may be nested.

4.3 Declarations and Definitions

4.3.1 Constant Definition

ConstantDefiniton =
PermanentConstantDefinition | UsualConstantDefinition.

4.3.1.1 Usual Constant Definition

The constant definition introduces a constant name which represents a fixed value.

UsualConstantDefinition =
Type ":" ConstantIdList ":" Expression ",".

ConstantIdList =
ConstantIdentifier { "," ConstantIdentifier }.

Expression and Type must be of same data type.

Example:

```
REAL : pi := 3.1415926;
```

4.3.1.2 Permanent Constant Definition

```
PermanentConstantDefinition =  
    ( TeachType | "PERMANENT" Type ) ":" PermanentConstantIdList ";"
```

```
PermanentConstantIdList =  
    PermanentConstantIdentifier  
    { "," PermanentConstantIdentifier }.
```

For the syntax of the term *TeachType* including the word symbol TEACH see 4.3.2.3.2.

The values of permanent constants are taken from a data list file, cf. section 4.8.6. The constants are set to an "undef"-state unless a data list file is specified which contains these constants. Any application of an "undef" constant will cause a run time error.

Each permanent constant is of global scope and there is no local *PermanentConstantDefinition* allowed.

4.3.2 Type Declaration

IRL includes predefined data types and allows the definition of several user specific data types. The predefined data types consist of simple data types (BOOL, INT, REAL or CHAR), standard data types (STRING, TEXT, POSITION, ORIENTATION or POSE) and the so called system data types (MAIN_JOINT, ADD_JOINT, JOINT, ROBTARGET, D_SPEC_TYPE or R_SPEC_TYPE). The properties of the system data types are partially determined in the system specification.

Only structured data types can be determined by the user, namely structures (RECORD) arrays (ARRAY), lists (LIST) or files (FILE), and only predefined data types can be more precisely specified by additional informations namely by the key words TEACH, SIGNAL or SEMAPHORE. TEACH-specifications for teach data types are allowed for all predefined data types of geometric nature including the system specific data types, but no device or robot specification type (D_SPEC_TYPE or R_SPEC_TYPE). The SIGNAL-specification for signal data types is only allowed for simple data types, the SEMAPHORE-specification is only possible in connection with boolean or integer types.

The predefined data types TEXT up to R_SPEC_TYPE simultaneously are structured data types, TEXT is a special file type (FILE OF CHAR), the other types except ORIENTATION are structures. For the data types POSITION up to ADD_JOINT the term geometric data types is also allowed.

```
TypeDeclaration =  
    StructuredTypeDeclaration |  
    TeachTypeDeclaration |  
    SignalTypeDeclaration |  
    SemaphoreTypeDeclaration.
```

4.3.2.1 Type Description and Type Identifier

By the needs of variable declaration (see 4.3.3) and constant definition it is necessary to have possibilities to reference to specified data types. This can be done by explicit type description or by using type identifiers.

A type identifier represents the name of an existing predefined or userdefined data type.

```
Type =  
    PredefinedType |  
    StructuredTypeIdentifier |  
    StructuredTypeDescription.
```

4.3.2.2 Predefined Data Types

PredefinedType =
 SimpleType |
 StandardType |
 SystemType.

4.3.2.2.1 Simple Data Types

SimpleType = *SimpleTypeIdentifier*.

There are 4 predefined Simple Data Types with the predefined identifiers BOOL, CHAR, INT and REAL (see 4.1.2.3).

Boolean Type

The predefined data type name BOOL describes objects which can only hold the values true (≠0) and false (=0).

Character Type

The predefined data type name CHAR describes objects which can hold exactly one ASCII character.

Integer Type

The predefined data type INT describes integer values. For the range of values see chapter 4.1.3.6.

Real Type

The predefined data type REAL describes real numbers. For the range of values see chapter 4.1.3.6.

4.3.2.2.2 Standard Data Types

StandardType = *StandardTypeIdentifier*.

There are 5 predefined Standard Data Types with the predefined identifiers STRING, TEXT, POSITION, ORIENTATION and POSE. (see 4.1.2.3).

String Type

The predefined data type STRING describes objects which can hold a number of ASCII characters. The maximum length of STRING type objects is restricted to 255. The first element of a string is referenced with the index 1. The controller should reserve additional space to store the maximum and the actual size of any string. For manipulating objects of type STRING the predefined functions mentioned in chapter 4.7.4 can be used.

Text Type

The predefined data type TEXT is used for file handling in sequential text files and for channel handling. The data type bases on data type CHAR and is defined as FILE OF CHAR.

Predefined Geometric Data Types

Geometric data types and the attached operators enable the programmer to handle datas from CAD-systems, to optimize robot movements under various aspects, or to calculate the path of a robot under sensor definition.

The geometric data types are used for description of position and orientation of robots or effectors or other objects. These types are underlying a cartesian coordinate system whose origin and orientation will be defined during the installation of the robot control system. It is called world coordinate system and all other coordinate systems are directly or indirectly related to it. The values of an object description can be related to any reference system, but the language does not support the explicit specification of this relation. Therefore the programmer using the geometric operations defined later to combine such object descriptions (see 4.5.3.1.3) is responsible for matching the relations between the corresponding reference systems assumed by these operations.

If the reference system of an object description is not the world coordinate system in the definitions and in the examples of this standard variables are used with an upper left index specifying the reference system. In most cases the index identifier corresponds to a POSE or ROBTARGET variable representing the reference coordinate system.

For the record types MAIN_JOINT, ADD_JOINT, JOINT, ROBTARGET, D_SPEC_TYPE and R_SPEC_TYPE see the following chapter.

Position

TYPE

```
RECORD REAL: X, Y, Z ENDRECORD = POSITION;
```

The predefined data type POSITION is given by three real numbers, describing the coordinate values in X-, Y- and Z-direction. The three values will be assigned in a single way to a record component or can be treated as an entity.

Assignment to a position variable:

```
VAR
  POSITION: p;
  REAL: x := -10.5;

BEGIN
  p.X := x;
  p.Y := p.Y * 0.5;
  p.Z := 2.01;

  p := POSITION(x, p.y*0.5, 2.01);
  p := POSITION(100.0, 150.0, 200.0);
```

Selection of a component of POSITION:

```
x := p.X;
```

Orientation

The inner structure of the predefined data type ORIENTATION is fixed by the producer of the IRL-compiler.

This type describes the direction of a coordinate system in space in relation to a reference system. The zero orientation is identical with the orientation of the reference system. The internal realization of the data type ORIENTATION in the programming system is no part of this standard, but for the exact semantic specification of the geometric operations the representation of an orientation by a rotation matrix is used.

If an orientation describes the direction of a coordinate system R1 in relation to a reference system R the corresponding rotation matrix $M = (R_n \ R_o \ R_a)$ consists of three columns equal to the R-coordinates of the three basic vectors R_n , R_o and R_a of the coordinate system R1.

For every orientation ori there is one unique rotation matrix here written as $M = M(\text{ori})$, and because of the properties of the basic vectors of a cartesian coordinate system the identity $M^{-1} = M^T$ is true for all M .

Besides other possible interpretations an ORIENTATION type variable can specify a rotation or the direction of a coordinate system describing a transformation. This can be seen in the following equations using normal matrix operations.

$$\text{Rotation: } R_p' = M(R_{\text{ori}}) * R_p$$

Here R_p and R_p' represent the vector p before and after the rotation represented by R_{ori} , all variables specified in relation to the same reference system R .

$$\text{Transformation: } R_p = M(R_{\text{ori}}) * R_{l_p}$$

Here R_{l_p} represents the local coordinates of the Point P in relation to the coordinate system R_l , R_{ori} describing the direction of R_l in relation to R . The result R_p represents the coordinates of the same Point P in relation to R .

It is sufficient to define the orientation by some rotations, but there are different ways to specify the behavior of the rotations. Therefore the rotations are notified as different function calls, each one is defining another meaning of the rotation values. The calculation of the internal representation from the given orientation function is a task of the programming system.

Assignment to an orientation variable:

```
VAR
  ORIENTATION: ori;
  POSITION: n1,o1,a1,v;
  REAL: a,b,c,d;

BEGIN
  ori := ORIRS(a,b,c);
    first a rotates about the x-axis of the reference system, then b about the y-axis of
    the original reference system, third c rotates about the z-axis of the original
    reference system

  ori := ORIXYZ(a,b,c);
    first a rotates about the x-axis of the reference system, then b about the new y-axis
    of the rotated reference system, third c rotates about the new z-axis of the two
    times rotated reference system

  ori := ORIYXZ(a,b,c);
    first a rotates about the y-axis of the reference system, then b about the new x-axis
    of the rotated reference system, third c about the new z-axis of the two times
    rotated reference system

  ori := ORIZYX(a,b,c);
    first a rotates about the z-axis of the reference system, then b about the new y-axis
    of the rotated reference system, third c about the new x-axis of the two times
    rotated reference system (equal to ORIRS(c,b,a))

  ori := ORIZYZ2(a,b,c);
    first a rotates about the z-axis of the reference system, then b about the new y-axis
    of the rotated reference system, third c about the new z-axis of the two times
    rotated reference system

  ori := ORIVANG(v,a);
    a rotates about the axis v, described in the reference system
```

```
ori := ORQUAT(a,b,c,d);
      a, b, c, d are parameters of the quaternion specification

ori := ORIMAT(n1,o1,a1);
      n1, o1, a1 are unit vectors representing the three columns of a rotation matrix,
      they build a righthanded coordinate system
```

Selection of a component of ORIENTATION:

Selecting a component of an orientation, i.e. rotation angle or axis of rotation or quaternion value, there is the need to specify the kind of orientation which should underlie the values; this is done by an additional parameter of the select function.

```
VAR
  ORIENTATION: ori;
  POSITION: n1,o1,a1,v;
  REAL: r;

BEGIN
  r := ANGLEX(RS,ori);
      result: rotation about the x-axis under ORIRS definition

  r := ANGLEX(XYZ,ori); ... under ORIXYZ definition
  r := ANGLEX(YXZ,ori); ... under ORIYXZ definition
  r := ANGLEX(ZYX,ori); ... under ORIZYX definition

  r := ANGLEY(RS,ori);
      result: rotation about the y-axis under ORIRS definition

  r := ANGLEY(XYZ,ori); ... under ORIXYZ definition
  r := ANGLEY(YXZ,ori); ... under ORIYXZ definition
  r := ANGLEY(ZYZ2,ori);... under ORIZYZ2 definition
  r := ANGLEY(ZYX,ori); ... under ORIZYX definition

  r := ANGLEZ(RS,ori);
      result: rotation about the z-axis under ORIRS definition

  r := ANGLEZ(XYZ,ori); ... under ORIXYZ definition
  r := ANGLEZ(YXZ,ori); ... under ORIYXZ definition
  r := ANGLEZ(ZYZ2,ori);... under ORIZYZ2 definition
  r := ANGLEZ(ZYX,ori); ... under ORIZYX definition

  r := ANGLEZ2(ZYZ2,ori);
      result: rotation about the z2-axis under ORIZYZ2 definition

  v := ROTAXIS(ori);
      result: vector of rotation under ORIVANG definition

  r := ROTANGLE(ori);
      result: rotation about the v-axis under ORIVANG definition

  r := QUATA(ori);
      result: parameter a of the quaternion specification
```

```
r := QUATB(ori);           result: parameter b ...
r := QUATC(ori);           result: parameter c ...
r := QUATD(ori);           result: parameter d ...
```

```
a1 := VECTORA(ori);
      result: unit vector representing the third column of the rotation matrix
```

```
n1 := VECTORN(ori);
      result: unit vector representing the first column of the rotation matrix
```

```
o1 := VECTORO(ori);
      result: unit vector representing the second column of the rotation matrix
```

ENDPROGRAM;

It is important, that the definition of an orientation can use values greater than 90 or 180 degree, but that the selection of an orientation component will always yield a value between -180 and 180 degree. Furthermore the results of a select operation can differ from the input value which are used for generating the orientation, because of the ambiguity of angle statements in the mathematical world.

To solve the following three problems it is necessary to specify the results of these select operations in detail. Even if all angle results are defined as degree between -180 and 180 there is no unique solution for every orientation value. That's why some of the results must be between -90 and 90 degree or between 0 and 180 degree, respectively. Secondly even then there exist orientation values with more than one possible correct solution of the select operations. Then some default values should be used. Thirdly the results of the select operations are not independent from each other. Especially if one angle was set to such a default value then the select operations must give results which yield to the original orientation using the underlying orientation definition function.

The exact specification of the possible results of the select operators for every kind of orientation definition is given in a short form, specifying the select operators of the same kind of orientation in complex.

For $a := \text{ANGLEX}(\text{RS}, \text{ori})$, $b := \text{ANGLEY}(\text{RS}, \text{ori})$ and $c := \text{ANGLEZ}(\text{RS}, \text{ori})$ the following must be true:

- 180 ≤ a ≤ 180 - first rotation about the x-axis
- 90 ≤ b ≤ 90 - second rotation about the y-axis
- 180 ≤ c ≤ 180 - third rotation about the z-axis
- if b = -90 or b = 90 then a := 0 (no rotation about the x-axis) and the value of c must realize that $\text{ORIRS}(a, b, c)$ is equal to the original orientation

For $a := \text{ANGLEX}(\text{XYZ}, \text{ori})$, $b := \text{ANGLEY}(\text{XYZ}, \text{ori})$ and $c := \text{ANGLEZ}(\text{XYZ}, \text{ori})$ the following must be true:

- 180 ≤ a ≤ 180 - first rotation about the x-axis
- 90 ≤ b ≤ 90 - second rotation about the y-axis
- 180 ≤ c ≤ 180 - third rotation about the z-axis
- if b = -90 or b = 90 then a := 0 (no rotation about the x-axis) and the value of c must realize that $\text{ORIXYZ}(a, b, c)$ is equal to the original orientation

For $a := \text{ANGLEY}(\text{YXZ}, \text{ori})$, $b := \text{ANGLEX}(\text{YXZ}, \text{ori})$ and $c := \text{ANGLEZ}(\text{YXZ}, \text{ori})$ the following must be true:

- 180 ≤ a ≤ 180 - first rotation about the y-axis
- 90 ≤ b ≤ 90 - second rotation about the x-axis
- 180 ≤ c ≤ 180 - third rotation about the z-axis
- if b = -90 or b = 90 then a := 0 (no rotation about the y-axis) and the value of c must realize that $\text{ORIYXZ}(a, b, c)$ is equal to the original orientation

For $a := \text{ANGLEZ}(\text{ZYZ2}, \text{ori})$, $b := \text{ANGLEY}(\text{ZYZ2}, \text{ori})$ and $c := \text{ANGLEZ2}(\text{ZYZ2}, \text{ori})$ the following must be true:

- 180 ≤ a ≤ 180 - first rotation about the z-axis
- 0 ≤ b ≤ 180 - second rotation about the y-axis

-180 ≤ c ≤ 180 - third rotation about the new z-axis
if b = 0 or b = 180 then a := 0 (no rotation about the y-axis) and the value of c must realize that ORIZYZ2(a,b,c) is equal to the original orientation

For a := ANGLEZ(ZYX,ori), b := ANGLEZ(ZYX,ori) and c := ANGLEZ(ZYX,ori) the following must be true:

-180 ≤ a ≤ 180 - first rotation about the z-axis
-90 ≤ b ≤ 90 - second rotation about the y-axis
-180 ≤ c ≤ 180 - third rotation about the x-axis
if b = -90 or b = 90 then a := 0 (no rotation about the y-axis) and the value of c must realize that ORIZYZ2(a,b,c) is equal to the original orientation

For v := ROTAXIS(ori) and r := ROTANGLE(ori) the following must be true:
v must be a unit vector

0 ≤ r ≤ 180 - rotation about the v-axis
if r = 0 then v := (0, 0, 1) (parallel to z-axis)
if r = 180 then ((v.x > 0) or ((v.x = 0) and (v.y > 0)) or ((v.x = 0) and (v.y = 0) and (v.z = 1)))

For a := QUATA(ori), b := QUATB(ori), c := QUATC(ori) and d := QUATD(ori) the Quatrupel (a,b,c,d) must be a unit quaternion with:

0 ≤ a - per definition
if a = 0 then ((b > 0) or ((b = 0) and (c > 0)) or ((b = 0) and (c = 0) and (d = 1)))

Example:

```
VAR
  ORIENTATION: ori1, ori2;
  REAL: r;

BEGIN
  ori1 := ORIRS(90,90,0);
  the orientation of the reference system will be rotated about the x-axis of the
  reference system with 90 degree and afterwards about the y-axis of the original
  reference system with 90 degree

  ori2 := ORIXYZ(90,90,0);
  the orientation of the reference system will be rotated about the x-axis of the
  reference system with 90 degree and afterwards about the new y-axis of the rotated
  reference system with 90 degree

  r := ANGLEZ(RS,ORIZYZ2(90,90,90));
  assigns to r the rotation angle 180 degree about the original z-axis; the other
  solution with a rotation of 0 degree about the z-axis including a rotation of 180
  degree about the y-axis is not possible because of
  ANGLEZ(RS,ORIZYZ2(90,90,90)) must between -90 and 90
ENDPROGRAM;
```

Note:

Even with the given exact specification the use of the select operators on a sequence of continuous orientation values can lead to non-continuous results which is very dangerous if the results should be used to control some robot or additional axis. But only this specification makes it possible to detect such problems by the user and to write algorithms that may handle such non-continuous values.

Pose

TYPE

```
RECORD
  POSITION      : POS;
  ORIENTATION : ORI
ENDRECORD = POSE;
```

The predefined data type POSE is composed of the components POS and ORI of POSITION and ORIENTATION type, respectively. Besides other possible interpretations a variable of type POSE can specify a transformation between two coordinate systems, the components representing the origins position and the direction of the second coordinate system in relation to the first one or the position and orientation of an object or the robot TCP in relation to the world coordinate system.

In order to select a POSE component, the POSE structure must be splitted in its POSITION and ORIENTATION part and then these datatypes have to be handled in the above described way. Besides this the type conversion in assignment statements (see 4.6.2) can be used to select the POSITION or ORIENTATION part of a POSE type expression.

Assignment to a POSE variable:

```
VAR
  POSITION: first_pos, next_pos;
  ORIENTATION: ori;
  POSE: f,g;

BEGIN
  f.POS := first_pos;
  f.ORI := ori;
  g := POSE (next_pos, f.ORI);
```

Selection of a component of POSE:

```
next_pos := f.POS;
ori := g.ORI;
```

4.3.2.2.3 System Data Types
SystemType = SystemTypeIdentifier.

There are 6 predefined System Data Types with the predefined identifiers MAIN_JOINT, ADD_JOINT, JOINT, ROBTARGET, D_SPEC_TYPE and R_SPEC_TYPE (see 4.1.2.3).

These predefined record types are system specific and have to be defined in the System Specification (see 4.8.2.).

Robot Target

ROBTARGET has to be defined suitable for all robots in a common cell; the exact semantic of the structure ROBTARGET should be explained in a comment.

ROBTARGET describes the position and orientation of the tool center point (TCP) of an industrial robot. ROBTARGET consists of a pose type part which defines the position and the orientation of the TCP, some parts which describes the exact status of the robot axes and if necessary the position of additional axes.

Objects of this data type may be operands of geometric operations or in special statements (e.g. MOVE PTP).

Example: A robot with additional axis
 (for ARRAY type description see 4.3.2.3.1.2)

TYPE

```

RECORD
  POSE          : PSE;
  INT           : STATUS;
  ARRAY [1..R_NTURNS] OF INT: TURNS;
  ADD_JOINT    : A_JOINT;
ENDRECORD = ROBTARGET;

```

In this example STATUS describes an additional degree of freedom (for example 'shoulder right or left'). The relation between values of data type INT and the robot configuration is system dependent. TURNS describes the number of rotations of 360 degree performed by some of the axes. R_NTURNS defines the maximum number of joints in a cell which can rotate more than 360 degrees.

Joints

The predefined data type JOINT describes the position of the TCP of an industrial robot in the robot coordinate system. Its definition depends on the number of axis of the industrial robot. According to the axis type, the components are either angles (rotation axis) or distances (translation axis).

JOINT is composed of the components M_JOINT and A_JOINT of MAIN_JOINT and ADD_JOINT type, respectively to distinguish between main axis and additional axis. The components A_JOINT of this type JOINT and of the above defined type ROBTARGET must be identical. This implicates, that the predefined functions TRAFO and TRAFOINV (see 4.7.4) must not change the values of this ADD_JOINT type component.

The predefined data types MAIN_JOINT, ADD_JOINT and JOINT have to be defined in the System Specification.

Example:

```

TYPE

RECORD REAL: A1,A2,A3,A4,A5 ENDRECORD = MAIN_JOINT;

RECORD REAL: AA1,AA2,AA3 ENDRECORD = ADD_JOINT;

RECORD
  MAIN_JOINT: M_JOINT;
  ADD_JOINT : A_JOINT
ENDRECORD = JOINT;

```

Device Specification Type

The Type D_SPEC_TYPE is used to define the devices that are controlled by the robot controller. These devices are the robot(s) and all devices that are controlled via additional axes like rails, positioners mounted at the robot mounting flange and external positioners.

Example:

```
TYPE
RECORD
  INT : D_NO; { Unique identification number of this device;
               must be >= 0 }
  INT : D_AXES_TYPE; { Type of additional axes: AX MAIN,
                      ADAX_IS, ADAX_IE, ADAX_EX; for information only }
  INT : D_PRED_DEVICE; { Identification number of the
                       preceding device in the kinematical chain; NO_PRED if
                       there is no predecessor }
  INT : D_NJ; { Number of joints of this device }
ENDRECORD = D_SPEC_TYPE;
```

All devices together are described in the Array D_DEVICES with R_DEVICE_NUMBER elements. To define groups of devices, that are controlled together by MOVE statements, the R_SPEC_TYPE is used.

Robot Specification Type

The Type R_SPEC_TYPE consists of the robot specific predefined system variables (see 4.1.2.3). As well as the system specific geometric data types it must be declared in the System Specification (see 4.8.2). For every robot used within an IRL program a record of that type has to be initialized by a Robot Specification (see 4.6.9).

Example:

```
TYPE
RECORD
  REAL:      R_ACC;
  ARRAY[1..R_NJ] OF REAL: R_ACC_PTP;
  REAL:      R_C_PTP;
  REAL:      R_C_CP;
  REAL:      R_C_SPEED;
  ARRAY[1..R_DEVICE_NUMBER] OF INT : R_DEVICES;
  JOINT:      R_JOINT_ACT;
  ROBTARGET:  R_ROBTARGET_ACT;
  ROBTARGET:  R_ROBTARGET_START;
  ROBTARGET:  R_ROBTARGET_END;
  ROBTARGET:  R_ROBTARGET_INTER;
  REAL:      R_SPEED_ACT;
  REAL:      R_SPEED;
  ARRAY[1..R_NJ] OF REAL: R_SPEED_PTP;
  REAL:      R_SPEED_ORI;
  POSE:      R_BASE;
  POSE:      R_TOOL;
ENDRECORD = R_SPEC_TYPE;
```

4.3.2.2.4 Cast Operators on Predefined Types

The names of the predefined data types POSITION, ORIENTATION, POSE and ROBTARGET may also be used as cast operators to describe explicit type conversion in geometric expressions (see also 4.5.3.1.3).

This explicit type conversion is possible according to the following table:

geometric data type	of cast operator POSITION		ORIENTAT.	POSE	ROBTARGET
of expression	POSITION	=	-	x ₂	x ₂
	ORIENTAT.	-	=	x ₂	x ₂
	POSE	x ₁	x ₁	=	x ₂
	ROBTARGET	x ₁	x ₁	x ₁	=

- = = explicit type conversion useless
- x = type conversion by cast operator allowed
- = type conversion not allowed

In the cases where an expression of a complex data type has to be converted in a simpler data type (x₁), this will be done only by ignoring the nonrelevant record components. In the other cases where the expression is simpler than the cast operator (x₂) the type conversion will be done by adding if necessary the zero orientation, the null vector and the actual values of the additional axes and the status, respectively.

4.3.2.3 Userdefined Data Types

Userdefined data types are restricted to structured, teach, signal or semaphore types.

4.3.2.3.1 Structured Data Types

StructuredTypeDeclaration =
StructuredTypeDescription ":" *StructuredTypeIdentifier*.

StructuredTypeDescription =
RecordTypeDescription |
ArrayTypeDescription |
ListTypeDescription |
FileTypeDescription.

4.3.2.3.1.1 Record Types

A userdefined record type introduces a new data type.

RecordTypeDescription =
"RECORD" *ComponentList* "ENDRECORD".

ComponentList =
Type ":" *ComponentIdList*
{ ":" *Type* ":" *ComponentIdList* }.

ComponentIdList =
ComponentIdentifier { ":" *ComponentIdentifier* }.

A record type is a collection of several objects which may be of different data types. These objects are called components. If one component is a record type itself the record is nested.

Each component has an identifier by which it can be accessed. Identifier and data type of the components are described in the definition of the record type. The identifier has to be unique within one record type description.

Example:

TYPE

```

RECORD
  position_type: p1, p2;
  REAL: current;
ENDRECORD = test_type;
```

4.3.2.3.1.2 Array Types

An array is a data structure consisting of a fixed number of components which are all of the same type. An array element is accessed by indices.

ArrayTypeDescription =
"ARRAY" "[" *IndexBoundList* "]" "OF" (*Type* | *TeachType*).

IndexBoundList =
IndexBound { "," *IndexBound* }.

The dimension of an array is determined by the number of indices. *IndexBound* describes the lower and upper bounds of an array.

IndexBound =
Expression ".." *Expression*.

Expressions are allowed to be only constants of the data type INT or CHAR.

Example:

TYPE

```
ARRAY [4..10] OF ROBTARGET = p_type;  
    {one dimensional array, with elements of type ROBTARGET}
```

```
ARRAY [1..100,1..80] OF CHAR = text_type;  
    {two dimensional array, elements of type CHAR}
```

4.3.2.3.1.3 List Types

A list is a data structure consisting of a variable number of components which are all of the same type. A list element can be accessed by indices, the first index is 1, the highest index is equal to the length of the list. Additional there are statements to insert, append or delete single list elements (see 4.6.7).

ListTypeDescription =
"LIST" "OF" (*Type* | *TeachType*).

Example:

TYPE

```
LIST OF ROBTARGET = dynamic_path;  
    {list type with variable number of elements of type  
    ROBTARGET}
```

4.3.2.3.1.4 File Types

FileTypeDescription =
"FILE" "OF" *Type*.

IRL defines symbolic names of file types and files. The "real" structure of the files is not treated by IRL, because the structure depends on the operating system.

IRL allows the following file types:

- Sequential text files

These files are declared as FILE OF CHAR and are stored as ASCII-files. They are treated sequentially. The data type TEXT is predefined as FILE OF CHAR .

- Sequential typed files

These files only store values of a certain type defined by Type given in the FileType definition. They are treated sequentially.

- Random access typed files

These files only store values of a certain type defined by Type given in the FileType definition. They are treated in a random access manner.

- Channels (see 4.3.3.3.2)

4.3.2.3.2 Teach Types

TeachTypeDeclaration =
TeachTypeDescription ":" *TeachTypeIdentifier*.

TeachTypeDescription =
"TEACH" *PredefinedType*.

The global declaration of permanent constants (see 4.3.1.2) and permanent variables (see 4.3.3.1) is possible with the key word PERMANENT or in connection with the specification of so called teach types. The latter is only allowed with predefined data types of geometric nature.

TeachType =
TeachTypeIdentifier |
TeachTypeDescription.

4.3.2.3.3 Signal Types

SignalTypeDeclaration =
SignalTypeDescription ":" *SignalTypeIdentifier*.

SignalTypeDescription =
("INPUT" | "OUTPUT") *SimpleType*.

The declaration of Signals is allowed (see 4.3.3.3.1) and so attached data types may be defined.

SignalType =
SignalTypeIdentifier |
SignalTypeDescription.

4.3.2.3.4 Semaphore Data Types

SemaphoreTypeDeclaration =
SemaphoreTypeDescription ":" *SemaphoreTypeIdentifier*.

SemaphoreTypeDescription =
"SEMAPHORE" *SimpleType*.

The semaphore data type in IRL is dedicated to the synchronization of multiple processes (see 4.6.8.4).

There are two types of semaphores in IRL. Boolean semaphores that can only be true (=0) or false (≠0) and integer semaphores with the range of the integer data type.

SemaphoreType =
SemaphoreTypeIdentifier |
SemaphoreTypeDescription.

4.3.3 Declaration of Variables

A variable is always of a certain data type. This data type can be an existing predefined or userdefined type or an other structured type, described as part of the variable declaration. It is to be determined by a declaration.

A declaration of variables of the data type ROBTARGET can be omitted if the compiler directive "DECLOFF" is set. These undeclared variables are permanent and part of the default data list. See also 4.4 and 4.8.3.3. With the key word PERMANENT the explicit declaration of permanent (and global) variables is possible.

To handle the problem of accessing variables from parallel processes (see 4.6.8) all variables used by more than one process should be defined as shared variables using the the Word Symbol SHARED. SHARED and PERMANENT are mutually exclusive.

VariableDeclaration =
["SHARED" | "PERMANENT "] *OrdinaryVariableDeclaration*.

OrdinaryVariableDeclaration =
DeclarationWithoutInit |
DeclarationWithInit |
SignalDeclaration.

4.3.3.1 Declaration Without Initialization

DeclarationWithoutInit =
(*TeachType* | *Type*) ":" *VariableIdList* ";"

VariableIdList =
VariableIdentifier { "," *VariableIdentifier* }.

The declaration of a file type variable announces a file to a program.

A list type variable declared without initialisation is defined to be empty with a length of zero.

Permanent variables (declared as PERMANENT and all variables of some TeachType) may be defined by tools of the operating system of the robot controller outside of the scope of an IRL program (cf. section 4.9). If they are of geometric nature they might be defined by "Teach In" methods. Therefore permanent variables of geometric nature may be called teach variables for matters of convenience.

The values of permanent variables are taken from a data list file (see also 4.8.3.3). The variables are set to an "undef"-state unless the actual data list file contains them. Any application of an "undef" variable except teach variables of geometric nature will cause a run time error. The reaction in the case of "undef" teach variables of geometric nature is system dependent, but it is recommended that this will cause a program PAUSE and together with the START signal the actual robot position (R_ROBTARGET_ACT) then defines the new value of the variable. The assignment of a new value to a permanent variable means also to change the value of that variable within the associated data list.

Every permanent variable is of global scope and no local declaration of variables is allowed.

Semaphores (or variables of type SemaphoreType) must not be declared without initialization because the declaration is the only place where the value of a semaphore can be set directly.

Example:

TYPE

```
    ARRAY [1..10,1..10,1..10] OF REAL = palette_type;
```

VAR

```
    ROBTARGET: pl;
    REAL: palette_height, palette_width;
    palette_type: palette_position;
    TEACH POSE: carbody;
    RECORD REAL: X,Y,Z ENDRECORD: vector;
    ARRAY [1..10] OF POSE: work_point;
```

4.3.3.2 Declaration With Initialization

With the declaration variables of most data types can be initialized.

*DeclarationWithInit =
(SemaphoreType | Type) ":" VariableIdList ":=" Expression ";"*

To initialize variables of any data type except array, list and file types Expression must be a constant of the data type which is determined by Type.

To initialize arrays or lists Expression has to be an ArrayInit.

*ArrayInit =
"[" InitValueList "]"*

*InitValueList =
InitValue { "," InitValue }*

*InitValue =
Expression ["OF" Expression]*

The Expression OF clause allows repetitions of the following expression. The replicator value has to correspond with the number of elements of the corresponding dimension. For multidimensional arrays the indices are incremented from left to right. The following example of a two dimensional array x may illustrate this:

VAR

```
    ARRAY [1..2,1..3] OF REAL: x:= [x11, x12, x13, x21, x22, x23];
```

For multidimensional character arrays there is the possibility to initialize the array elements with character string constants, instead of initializing them separately.

For lists the replicator defines the initialization of as many list elements as the value of the constant expression of the replicator. Because lists have only one dimension multiple replication is not possible but a sequence of replications is allowed.

Example:

TYPE

```
ARRAY [1..2,1..2] OF REAL = matrix_type;  
ARRAY [1..2,1..80] OF CHAR = text_type;  
RECORD  
  BOOLEAN: occupied;  
  REAL:    object_weight;  
  POSE:    gripper_stand;  
ENDRECORD = palette_occupation;  
ARRAY[1..6,1..10] OF palette_occupation = palette_dimension;  
LIST OF ROBTARGET = dynamic_path;
```

VAR

```
matrix_type: matrix := [ [3.0,4.5] , [15.2,17.8] ];  
text_type: text := ['first message','second message'];  
palette_dimension: palette :=  
  [ 6 OF [ 10 OF palette_occupation (  
    FALSE,  
    0.0,  
    POSE ( POSITION (0.0,0.0,100.0),  
              ORIXYZ (0.0,180.0,0.0)  
          )  
    ]  
  ];  
  { Each place of the palette has a default definition of its gripperstand (position and orientation) which  
  is 100 mm above the bottom of the palette and the z-axis of the gripper is oriented downwards. }  
dynamic_path: path1 :=  
  [ ROBTARGET(  
    POSE(POSITION(100.0,1000.0,0.0),ORIRS(90.0,0.0,90.0)),  
    0,[0,0],0.0),  
    ROBTARGET(  
    POSE(POSITION(200.0,1200.0,0.0),ORIRS(90.0,0.0,90.0)),  
    0,[0,0],0.0),  
    ROBTARGET(  
    POSE(POSITION(0.0,800.0,1200.0),ORIRS(90.0,0.0,90.0)),  
    0,[0,0],0.0)  
  ];  
dynamic_path: path2 := [ 10 OF  
  ROBTARGET(  
    POSE(POSITION(50.0,800.0,100.0),ORIXYZ(0.0,0.0,90.0)),  
    0,[0,0],0.0)  
  ];
```

4.3.3.3 Interface Declarations

There are 2 classes of interfaces:

Signals are simple process interfaces. They include:

- binary in- and outputs
- digital in- and outputs
- analog in- and outputs

A set of binary inputs or outputs respectively can be combined to digital inputs and outputs. With such digital interfaces bit patterns can be processed simultaneously.

Logical interfaces are called **channels**. They transfer data by using a specific protocol. Serial and parallel interfaces, for example, are channels.

All interfaces are identified by symbolic names. Certain interfaces are predefined. Based on those names, the user can define his own names for the interfaces.

4.3.3.3.1 Signals

User defined signal names can be handled like usual variables and also have a data type.

The type REAL is mapped to analog I/O, the type INT to digital I/O and the type BOOL to binary I/O. In addition they are mapped by the user to certain I/O-port addresses.

Other types are not allowed.

SignalDeclaration =
SignalType ":" *AddressList* ";"

AddressList =
SignalIdentifier "AT" *Address*
{ "," *SignalIdentifier* "AT" *Address* }

Address =
SingleAddress |
ContinuousAddress |
NonContinuousAddresses.

SingleAddress = *DecimalConstant*.

ContinuousAddress =
"(" *DecimalConstant* ".." *DecimalConstant* ")".

NonContinuousAddresses =
"(" *DecimalConstant* { "," *DecimalConstant* } ")".

Example:

```
VAR
  INPUT BOOL: gripper AT 2;
    { binary input }

  INPUT INT: sensordata AT (2..4);
    { integer input mapped to addresses 2..4}

  OUTPUT INT: controldata AT (2,5,1,4);
    { integer output mapped to addresses 2,5,1,4}

  OUTPUT REAL: drive AT 2;
    { analog output No. 2 }
```

4.3.3.3.2 Channels

Channels are special files and have predefined identifiers.

Predefined channel identifiers are:

SER_1...SER_x	serial ports	(x = number of ports)
PAR_1...PAR_x	parallel ports	(x = number of ports)
PRINTER		
SCREEN		
KEYBOARD		
PANEL		
TTY		
HPU		(hand programming unit)
MCP		(manual control panel)

Based on these predefined channel identifiers the programmer can define his own channel identifiers and use them in the I/O-statements.

4.3.4 Interrupt Declaration

InterruptDeclaration =
("DECL" (*IntegerConstant* | *ConstantIdentifier*) "WHEN" *Expression*
| "DECLSYS" (*IntegerConstant* | *ConstantIdentifier*))
"DO" *ProcedureCall*

The INTERRUPT DECL allows the definition of an interrupt (see 4.6.4.9).

4.4 Compiler Directive

CompilerDirective =
["DECLON" ";" | "DECLOFF" ";"].

DECLOFF which is the default enables implicit declaration of all undeclared variables as global variables having data type ROBTARGET. These undeclared variables are permanent and part of the actual data list (if no IMPORT DATALIST statement is given, this is the default data list). See also 4.3.3 and 4.8.3.3.

DECLON means that all variables must be declared explicitly. In that case the occurrence of an undeclared identifier yields an error and all entities except of procedures and functions shall be declared before they are used.

4.5 Operators, Objects, Expressions

4.5.1 Operators

Operator =
"*" | "/" | "DIV" | "MOD" | "@" |
"+" | "-" |
"<" | ">" | "<=" | ">=" |
"=" | "<>" |
"AND" |
"OR" | "EXOR".

arithmetical operators:

+	addition
-	subtraction, sign operator
/	division
*	multiplication
DIV	integer division
MOD	modulo function

boolean operators:

AND	logical and
OR	logical or
NOT	logical not (unary operator, not in syntax rule)
EXOR	exclusive or

logical operators:

=	equal
>	greater
<	less
<>	not equal
>=	greater equal
<=	less equal

geometric operators:

+	addition
-	subtraction
/	division
*	multiplication
@	relative translation and rotation

4.5.2 Objects

Objects have a predefined or structured data type.

```

Object =
    Identifier |
    ArrayOrListElement |
    RecordComponent.

RecordComponent =
    Object "." ComponentIdentifier.

ArrayOrListElement =
    Object "[" ArrayExpressionList "]"

ArrayExpressionList =
    Expression { "," Expression }.

```

The number of the arithmetical expressions in an ArrayExpressionList must be equal to the dimension of the array or list. So in the case of lists or one dimensional arrays there must be exactly one Expression. The expressions to index the array or list elements have to be of data type INT or CHAR, respectively.

Example:

Given the following declarations:

TYPE

```

ARRAY [1..5] OF ROBTARGET = position_type;
    { Array with elements of type ROBTARGET }

RECORD
    ROBTARGET: p;
    REAL: current;
ENDRECORD = spot;      { record type definition }

```

VAR

```

ROBTARGET: p1;          { variable of record type ROBTARGET }
position_type: pos_array;
    { variable of type position_type }
spot: door;             { variable door of record type spot }

```

In this case the following record components are correct:

```

p1.PSE.POS.X            { x component of p1 }

pos_array[1].PSE.POS.X  { x component of the first element of pos_array }

```