Teach your Robot an LL(1)-Jargon

Albert Hoogewijs, Hans Gruyaert, Geert Vernaeve

University of Ghent, Pure Mathematics & Computer Algebra
Galglaan 2, B-9000 Ghent
fax: 32+09 264 49 93 - bh@cage.rug.ac.be - http://cage.rug.ac.be/~bh

Abstract

When we talk about teaching a robot an LL(1)-jargon, we mean specifying a language processor, i.e. an acceptor and transducer that is able to translate robot instructions into machine code for the control unit of that robot. A *syntax-directed development* is proposed, i.e. a software method in which the syntax of the input of the application plays a central role. The syntax forms a frame on which semantic actions, attributes, local and global information can be attached. More specifically we use the ELL(1) format for the description of the language. From this syntax-directed description an executable program is automatically produced using the Visual MIRA™ tool.
We illustrate the usage of this LL(1)-jargons, in the description of an anticipating process of a robot.

Keywords: Domain-Specific Language, Jargon, LL(1)-Language, Domain Engineering, Robot Control

1 Introduction

Small languages, tailored towards the specific needs of a particular domain, can significantly ease building software systems for that domain (Bentley, 1986). We illustrate this idea through the construction of the LL(1)-jargons CDL and RDL for a Robot controller. According to Nakatani and Jones (1997) we use *jargon* in the sense of an easy-to-make Domain-Specific Language (DSL) that domain engineers who are not language experts can easily make themselves. We use LL(1)-descriptions (Hoogewijs, 1997) for the specification of the jargons CDL and RDL and apply Visual MIRA (1993) to produce the deliverable software.

2 Robot control through an LL(1)-Domain-Specific Language

2.1 An LL(1)-jargon

For the non language specialists we recall that "*LL*" stands for a parsing technique where the first "*L*" refers to the fact that the *syntax analyser*, or *parser* scans a given sentence of the language from left to right, and the second "*L*" means that the syntax tree is built-

up (top-down) from left to right (see Hoogewijs, 1995). According to the "1" we need only one look-ahead symbol in order to decide on the action to be taken. Applying this technique to the input 2*(3+4) we see that while scanning the "*" the look-ahead symbol "(" instructs the parser to postpone the call for a multiplication until the "(3+4)" has been parsed. The MIRA™ tool as described by Huybrechts (1995) and the examples presented by Pauwels (1995) and Huybrechts (1999) show that this technique can be used efficiently to produce complex software tools from a description in a high-level specification language.

To meet the drawback that *LL(1)* does not deal with left recursivity, extended context-free (ECF) instead of context-free (CF) syntax is used. In this context a LL(1) ECF language is also called an ELL(1) language. It can be shown that the class of ELL(1) languages coincides with the class of deterministic CF languages. For a full description of the generation of the automatic transducers for a given LL(1) specification, we refer to Lewi (1979, 1982, 1992).

We disagree with Marvin Minsky when he describes Noam Chomsky as a disaster for the development of learning robots (in Van Peteghem, 1999) since we believe that CF languages and more specifically ELL(1) languages are very useful in the control of robots. But we do agree with Minsky when he says that there is more than syntax and that semiotics count as well, if you want a machine to do a task and study the performance of that machine. More specifically we refer to one such basic semiotic concept namely Saussure's (Culler 1986) distinction between the two inseparable components of a sign: the signifier, which in language is a set of speech sounds or marks on a page, and the signified, which is the concept or idea behind the sign. Saussure also distinguished "*parole*", or actual individual utterances, from "*langue*", the underlying system of conventions that makes such utterances understandable; it is this underlying *langue* that interests semioticians most, and where also domain-specific language generators come in.

2.2 A Domain-Specific jargon

When we talk about teaching a robot an LL(1)-jargon, we talk about writing a language processor, i.e. an acceptor and transducer that is able to translate our robot instructions into machine code for the control unit of that robot. As it is the case for software development in general, an adequate methodology is the basis for the construction of high quality programs. By this we mean programs that in the first place are simple, reliable, adaptable and well structured. Other requirements such as efficiency and portability are important but have less priority.

The proposed methodology is based on the idea that generative devices, which involve definitions of languages and translations, are more natural, compact and readable than recognizing devices. To gain more insight into the problem to be solved, one must start with a good problem definition. Thus before we start writing a language processor, we must be able to define the language to be implemented. Therefore a number of syntax and translation formalisms must be studied in detail. In view of the complexity of the

tasks and the particular possibilities and construction of the robot, we will have to apply different technologies and algorithms for different situations.
In this sense the jargon that we use to describe the shop floor and the robot actions, depends on the environment that the robot is going to act in and the tasks he will have to perform. For our example, we will assume a robot to be designed to get parts from a rack and deliver them to an assembly line or to unload a truck and arrange the parts on the racks. The emphasis is on the aspects of the modular composition by which acceptors and transducers are produced. The syntax-directed descriptions of Visual MIRA™ are based on the ELL(1) parsing techniques and allow systematic refinement of the specifications. To illustrate these ideas, we start with the specification of control system for a virtual robot, and refine this specification to a control system for the Lego® Mindstorms™ robot.

3 Environment of the Robot

We consider a robot as an autonomous unmanned ground vehicle that can perform some tasks with a minimum of human interaction. A working definition is that an intelligent robot is a machine that can extract information from its environment and use knowledge about its world to move safely in a meaningful and purposeful manner and to perform a series of instructions. In order to perform its task, the robot must be able to interact with its environment. Note that our robot is not a conscious machine, as presented by Marvin Minsky (1998). This means that an expert will have to provide the robot with the knowledge and description of the working environment. So we are looking in the first place for a formalism that is able to cope with the problem of describing the shop floor.

In order to visualise this situation, we start with a virtual robot and we consider a two dimensional rectangular grid as the representation of the shop-floor (see also Wide and Schellwat 1997). Each point on the grid is labelled with an element from the set $\{R,T,X,.\}$ where R represents the position of the robot, T the target of the robot, X the location of an obstacle and "." a free location.

3.1 Configuration Description Language

For the description of the shop floor, we use a domain-specific specification language CDL (Configuration Description Language) that is generated from an LL(1)-specification in the Mira environment.
Advantages of using this LL(1)-jargon:

- the implementation of the jargon mainly depends on its syntax description, easing the adaptation to structural modifications of the shop-floor;
- high-level of abstraction and surveyable code;
- automatic generation of the parser, reducing the error rate;
- making syntactic changes easy;

- easily expandable;
- easy to move to another platform.

In a first approach, for the virtual environment we only need a limited number of statements *Workspace(x,y)* (dimension of the grid), *Robot(x,y)* (initial position of the robot), *Target(x,y)*, *Obstacles[(x,y)*$_{11}$*,(x,y)*$_{12}$*, ..., (x,y)*$_{n1}$*,(x,y)*$_{n2}$*]* (left upper corners, right lower corners of the obstacles).

3.2 A two-level LL(1) description of CDL

The application generator Visual MIRA takes as input a description of the lexical and syntax analysers of the language.

```
#RULE
<WorkSpace Part> = "WORKSPACE" <Position> SetWorkSpace

#SetWorkSpace
Err_Code = wSpace->SetSize(X,Y);
#END

#RULE
<Robot Part> = "ROBOT" <Position> PutRobot
#PutRobot
Err_Code = wSpace->PutRobot(X,Y);
#END

#RULE
<Obstacle Part> = "OBSTACLES" "["
<Position> SetFirstPos <Position> PutFirstObstacle
( "," <Position> SetFirstPos <Position> PutObstacle )*
"\]"
#DECLARE
int X1, Y1;
#SetFirstPos
X1 = X; Y1 = Y;
#PutFirstObstacle
Err_Code = wSpace->PutObstacle(X1, Y1, X, Y);
#PutObstacle
Err_Code = wSpace->PutObstacle(X1, Y1, X, Y);
#END

#RULE
<Position> = "(" "NUMBER:n1" "," "NUMBER:n2" ")"
SetPosition
#SetPosition
X = @"n1";
Y = @"n2";
#END
```

Fig. 1: grammar rules for CDLpars

A first Mira-specification "CDLscan.mir" generates a lexical scanner. It will read from standard input and separate characters of the source language into groups that belong together according to our specification rules; these groups or keywords of the language (i.e. *WORKSPACE, ROBOT, TARGET, OBSTACLES*) are coded into "*tokens*". The output of the lexical analyser is a stream of tokens, which is passed to the next phase, the *syntax analyser* or *parser*. A second Mira-specification "CDLpars.mir" generates this syntax analyser.

The role of this analyser consists of recognising the commands *WORKSPACE, ROBOT, TARGET* and a tuple of integers *(x,y)* as argument, or the command *OBSTACLES* and a list $[(x,y)_{11},(x,y)_{12}, \ldots, (x,y)_{n1},(x,y)_{n2}]$ of tuples as argument, and generating the corresponding semantic actions. Since these semantic actions, which are respectively: declaring the dimension of the shop floor and describing the position of the robot, the target and the obstacles, are obviously reflected in the syntax of the commands, we get a self-explanatory LL(1)-specification of the parser as shown in figure1.

Figure 2 shows the structure of a typical translator tool.

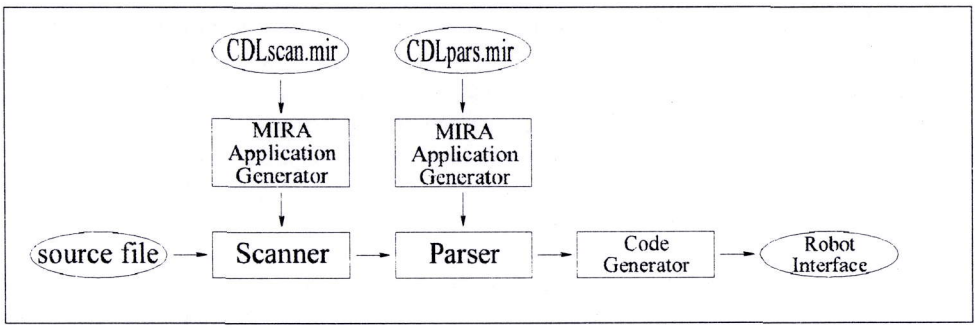

Fig. 2: structure of a typical translator tool

3.3 A CDL-description

Figure 3 illustrates a CDL-description of a virtual shop floor and the resulting "grid". Note that here the robot (*R*) is represented as a small triangle, conform to the turtle representation in Logo, a circle represents the target (*T*) and the obstacles (*X*) are shown as filled squares.

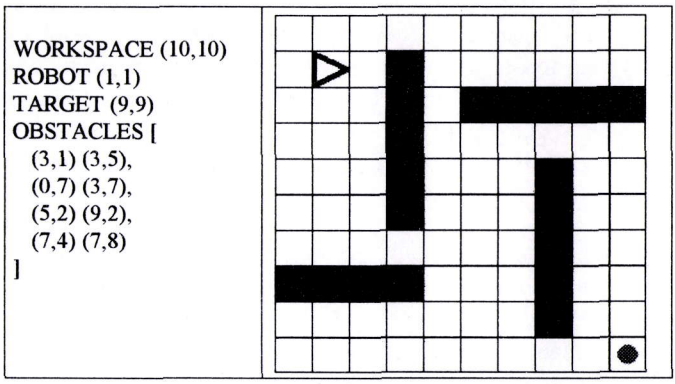

Fig. 3 CDL description and corresponding grid

4. Robot Control Language

Syntax-directed development is a software development method in which the syntax of the input of the application plays a central role. The syntax forms a frame on which the semantic actions, attributes, local and global information (such as variables, types and routines) can be attached. From this syntax-directed description an executable program is mechanically produced in a traditional programming language such as C++ or Java. In (Hoogewijs, 1997) we propose to formalise human-computer interaction through the vocabulary and the grammar of the "interaction language".

The syntax of the language is represented in Backus-Naur Form. This is a highly structured, hierarchical metalanguage that results in a so-called "fan-out" problem. That is, the introduction of so-called non-terminals in an expression that can be replaced by more non-terminals through several successive iterations before a terminal symbol is finally reached. This multilevel tree structure is difficult for human beings to follow, since by the time the terminals are reached, the highest level expression and the language structure may be long forgotten. In the Visual MIRA environment, these descriptions can be visualised in corresponding transition diagrams, which helps in overcoming this problem.
Figure 4 shows the Mira specification for a <Simple Statement> in RCL and the corresponding graphical representation.

4.1 The Turtle Concept

The Robot Control Language RCL is based on the "*turtle graphics*" as introduced by Seymour Papert. The goal of the project is to develop a simple but powerful interactive system that is able to communicate with a robot.

```
#RULE
 <Simple Statement> = "GO"   ( & MkGo1 | <Expression:e1> MkGo2 )
                    | "TEST" ( & MkTest1 | <Expression:e2> MkTest2 )
                    | "TURN"  MkTurn
                    | "FLIP"  MkFlip
 : STMT
#END
```

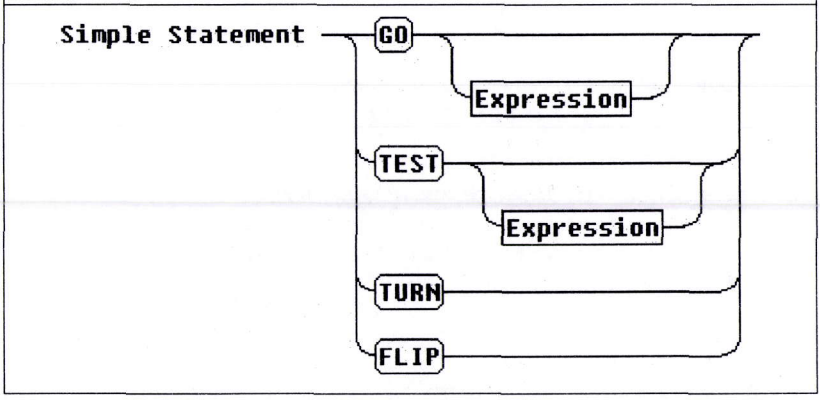

Fig. 4 Rule for <Simple Statement> and corresponding railroad diagram

4.1.1 Motion commands

"Turtle graphics" means that the movement of the robot can be mastered through elementary commands. For a start we consider *go(west|north|east|south)* for the movement actions, *flip* (180° turn) and *turn* (90° turn right) for changing the direction of the robot and *test(west|north|east|south)* to see if there is an obstacle.

4.1.2 Control statements, basic operations

Besides the movement commands, the language has some elementary control statements such as *if ... then ...*, *loop ... times ...*, *while ... repeat ...* and some basic operations +, -, *, /, *abs*, <, <=, ==, >= ,> ,!= ,*not*, *and*, *or* on the three data-types *int*, *bool* and *dir* (possible directions: *west, north, east, south*).

4.2 LL(1)-grammar for RCL

The generation of the lexical and syntax analysers for the language RCL follows the same two-level strategy as explained above. An LL(1) description RCLscan.mir specifies the lexical scanner, which generates the tokens for the defined keywords that will be passed to the parser.

4.2.1 The RCL-parser

```
<RCL input> = (<variable declaration> | <statement> | <procedure declaration>) ;
<variable declaration> = (int | bool | dir) (& | [ number ]) identifier
<statement> = <simple statement>
              | <conditional>
              | <iteration>
              | <procedure call – assignment>

<simple statement> = go (& ( <expression>)
                   | test (& ( <expression>)
                   | turn
                   | flip
<conditional> = if <expression> then <statement list>
<iteration> = loop <expression> times <statement list>
              | while <expression> repeat <statement list>
<procedure call – assignment> = identifier
                                (<actual arguments> |
                                (& | [ <expression> ]) = <expression>)
<statement list> = [ <statement> (; <statement>)* ]
<actual arguments> = & | (<expression> (, <expression>)*)
<expression> = <comparison> ((and | or) <comparison>)*
<comparison> = <integer> ((< | <= | == | >= | > | !=) <integer>)*
<integer> = <term> (( + | -) <term>)*
<term> = <factor> ((* | /) <factor>)*
<factor> = not <factor>
         | - <factor>
         | abs <factor>
         | number
         | true
         | false
         | west
         | north
         | east
         | south
         | (<expression>)
         | identifier ([ <expression> ] | &)
<procedure declaration> = procedure identifier <formal arguments>
                          <local variables>
                          <body definition>
<formal arguments> = & | (<variable declaration> (, <variable declaration>)*)
<local variables> = & | var <variable declaration> (, <variable declaration>)* ;
<body definition> = begin
                          <statement> (; <statement>)*
                    end
```

Fig. 5: LL(1) grammar for RCL

The LL(1) specification for the RCL-parser, is given in figure 5.

4.2.2 Semantic actions

By adding the C++ code for the corresponding semantic actions we get the Visual MIRA input. Once more, since the semantic actions are immediately induced from the syntax description, we get a self-explanatory LL(1)-specification as shown in Figure 6. Then MIRA automatically generates the C++ source file, which can be compiled to produce an interactive robot controller.

```
#RULE
 <Simple Statement> = "GO"   ( & MkGo1 | <Expression:e1>
MkGo2 )
              | "TEST" ( & MkTest1 | <Expression:e2> MkTest2 )
              | "TURN"  MkTurn
              | "FLIP"  MkFlip
 : STMT

 #MkGo1
  @<Simple Statement>->Make0ArgStmt(GO);
 #MkGo2
  @<Simple Statement>->Make1ArgStmt(GO, *@<e1>);
 #MkTest1
  @<Simple Statement>->Make0ArgStmt(TEST);
 #MkTest2
  @<Simple Statement>->Make1ArgStmt(TEST, *@<e2>);
 #MkTurn
  @<Simple Statement>->Make0ArgStmt(TURN);
 #MkFlip
  @<Simple Statement>->Make0ArgStmt(FLIP);
#END
```

Fig. 6: Semantic actions

4.2.3 Object oriented function calls

The statements

@<Simple Statement>->Make0ArgStmt(GO);

@<Simple Statement>->Make1ArgStmt (GO, *@<e1>);

are calls to methods, described in a class "Statement" (see figure 7), that make calls to methods described in a class "Robot" corresponding to the interface that is specific for the robot we want to "teach our jargon".

Note that this paradigm facilitates the changeover from one robot controller to another.

In a first approach, we consider the robot as a turtle, moving around on the screen. The corresponding class "Robot" contains the method "go" as shown in figure 8, that controls the motion of the turtle. Later on we will replace this class, with an appropriate Robot class, which contains calls to methods that control the motor motion of the Lego® Mindstorms™ robot.

```
// GO, TEST, TURN, FLIP, WHERE
void Statement::Make0ArgStmt(StmtKind kind) {
  if (s != 0) delete s;
  s = new RepStmt;
  s->kind = kind;
}
// ( GO | TEST ) [ west | east | north | south ]
void Statement::Make1ArgStmt(StmtKind kind, const Expression& expr)
{
  if (s != 0) delete s;
  s = new RepStmt;
  s->kind = kind;
  s->args.AddItem(expr);
}
```

Fig 7: class Statement

```
Bool Robot::Go(int d) {
  if (mySpace != 0) {
    if (d == W || d == N || d == E || d == S)
      direction = d;
      switch (direction) {
       case W :
          if (mySpace->WhatAt(position.X-1, position.Y) == OBSTACLE)
                return false;
          position.X--;
          break;
       case N :
          if (mySpace->WhatAt(position.X, position.Y-1) == OBSTACLE)
                return false;
          position.Y--;
          break;
       case E :
          if (mySpace->WhatAt(position.X+1, position.Y) == OBSTACLE)
                return false;
          position.X++;
          break;
       case S :
          if (mySpace->WhatAt(position.X, position.Y+1) == OBSTACLE)
                return false;
           position.Y++;
           break;
       default : return false;
    }
    mySpace->SetRobotPos(position.X, position.Y);
    myTrace.AddItem(position);
    return true;
  }
  return false;
```

Fig 8: method go

4.3 Single-movers Problem and the Robot Controller

As a test for the language we discuss two solutions for the unconstrained single-movers problem. Single-mover refers to the fact that there is only one robot moving around, which has to avoid fixed obstacles. Note that in the proposed solutions, the robot acts as an anticipatory system, i.e. at each point of the decision process (described as an RCL-algorithm), the robot computes its next movement based on anticipated states of its environment, which is modelled through a CDL-description.

4.3.1 Straightforward solution

A first solution implements the following principle. In "*normal mode*", i.e. as long as the robot meets no obstacle, the choice strategy as presented in figure 9 is used. The tuple (x,y) represents the position of the target and (x_r,y_r) refers to the position of the robot.

```
Δx = x-xr;
Δy = y-yr;
if |Δx| >= |Δy| then [
        if Δx >= 0 then [ go east ] ;
        if Δx < 0 then [ go west ] ; ];
if |Δx| < |Δy| then [
        if Δy >= 0 then [ go south ] ;
        if Δy < 0 then [ go north ] ; ];
```

Fig 9: Choice strategy in normal mode

Whenever an obstacle is met, the algorithm switches to "*avoid mode*". As long as the chosen direction is not free, the robot keeps searching for a free "neighbour" taking it closer to the solution. This choice is also based on anticipatory calculations of Δ_x and Δ_y. We get the RCL-algorithm as presented in figure 10.

4.3.2 Using a potential

A second algorithm uses a so-called potential function $V(x,y) = |\Delta_x + \Delta_y|$. Since the strategy is based on the observations of the robot, i.e. information about the free locations around the current position of the robot, we try to get a local optimisation of the potential along the path of the robot, based on anticipatory description of the environment.

Both algorithms result in a description (implementation) of a *goto(X,Y)* function in RCL. As a result we get the robot moving on the screen, and finding its way to the destination, avoiding the obstacles.

```
PROCEDURE GoTo1 (int X, int Y)

VAR int MODE,
      int DeltaX,
      int DeltaY,
      dir way,
      dir OBSTRUCTION,
      int stop;
BEGIN
  MODE = 0;
  DeltaX = X – Xr;
  DeltaY = Y – Yr;
  Stop = 0;
  while ( ((DeltaX != 0) OR (DeltaY != 0)) AND (stop < 4)) repeat [
    if (MODE == 0) then [
      if (ABS(DeltaX) >= ABS(DeltaY)) then [
        if (DeltaX >= 0) then [ way = EAST ];
        if (DeltaX < 0) then [ way = WEST ] ];
      if (ABS(DeltaX) < ABS(DeltaY)) then [
        if (DeltaY >= 0) then [ way = SOUTH ];
        if (DeltaY < 0) then [ way = NORTH ] ];
      test way;
      if (NOT CONDITION) then [
        OBSTRUCTION = way;
        MODE = 1;
        if (ABS(DeltaX) >= ABS(DeltaY)) then [
          if (DeltaY >= 0) then [ way = SOUTH ];
          if (DeltaY < 0) then [ way = NORTH ] ];
        if (ABS(DeltaX) < ABS(DeltaY)) then [
          if (DeltaX >= 0) then [ way = EAST ];
          if (DeltaX < 0) then [ way = WEST ] ] ] ];
    if (MODE == 1) then [
      test OBSTRUCTION;
      if (CONDITION) then [
        way = OBSTRUCTION;
        MODE = 0        ];
if (NOT CONDITION) then [
        test way;
        if (NOT CONDITION) then [ way = way + 2; stop = stop + 1 ] ] ];
    go way;
    DeltaX = X - Xr;
    DeltaY = Y - Yr
  ]
END;
```

Fig 10: RCL implementation of GoTo1

```
Void Robot::moveto(int dir) {
      FILE *program;

      program = fopen("gridmove.nqc","w");
      fprintf(program,"#include \"roverbot.h\"\n");
      fprintf(program,"task main() {\n"
            "initmotors();\n");

      switch(dir) {
      case N:     /* North */
            fprintf(program,"initforward();\n"
                  "Wait(%d);\n"
                  "Off(OUT_A + OUT_C);\n",GRIDTIME);
            break;

      case E:     /* East */
            fprintf(program,"right(90);\n"
                  "initforward();\n"
                  "Wait(%d);\n"
                  "Off(OUT_A + OUT_C);\n"
                  "Wait(%d);\n"
                  "left(90);\n",GRIDTIME,RELAXTIME);
            break;

      case S:     /* South */
            fprintf(program,"initbackward();\n"
                  "Wait(%d);\n"
                  "Off(OUT_A + OUT_C);\n",GRIDTIME);
            break;


      case W:     /* West */
            fprintf(program,"left(90);\n"
                  "initforward();\n"
                  "Wait(%d);\n"
                  "Off(OUT_A + OUT_C);\n"
                  "Wait(%d);\n"
                  "right(90);\n",GRIDTIME,RELAXTIME);
            break;
      }
      fprintf(program,"Wait(%d);\n}\n",RELAXTIME);
      fclose(program);

      system("nqc -d gridmove.nqc -run");

      sleep(5);
}
```

Fig. 11: method *moveto*

5. RCL for Lego® Mindstorms™

5.1 Adapting the "robot" interface

The considered presentation, assumes that the robot "understands" the basic commands *go, test* ... . In order to adapt the LL(1) jargon to the Lego® Mindstorms™ robot, we extend the class "Robot" with a method *moveto* that generates intermediate NQC code (NQC = Not Quite C by D. Baum 2000) that will be compiled to proper RCX™-code to control the motor movements of the robot. The *go* method is adapted by adding a *moveto(N|E|S|W)* call depending on the case. As an illustration we present in figure 11 an implementation of the *moveto* method. The ease of this adaptation is due both to the use of the LL(1) and the Object Oriented paradigm.

5.2 Tailoring the language

The preceding example illustrates that the considered methodology allows easy customisation to a new environment. The following example refers to the extension of the jargon, to allow the description of more complex tasks. Assume that we want to ask the robot to find us a specific part from the shelves in a stockroom. Then, in the first place we will have to extend CDL with some database elements to allow the description of the stock and the position of each part on the racks. Secondly we extend RCL with a *get* command, which accepts the references of the part we want. This *get* command might call a *lookup* command to locate the position of that part, and then calls an appropriate *goto* command.

6. Conclusions

We have shown that a simple specific language can be rich enough to describe complex tasks for a feasible robot acting as an anticipatory system. Such a "*jargon*" can be specified in a LL(1)-format that allows easy adaptation to a new specific situation. The considered specification is clear enough to allow tailoring by example. The most important is that a lot of irrelevant details can be hidden to the operator of the control system. This allows the user to concentrate on solving the main problems.
We illustrated the use of the LL(1)-jargons, in two implementations of generic algorithms for routing an autonomous robot. The resulting anticipating process is described in the RDL jargon, and relies on an anticipatory CDL description of the environment.

References

Baum Dave (2000). Not Quite C. http://www.eneract.com/~dbaum/nqc/index.html.

Bentley Jon (1986). Programming pearls: Little languages. *Communications of the ACM, 29 (8), 711-721.*

Culler Jonathan (1986). *Ferdinand de Saussure,* rev. ed. Cornell Univ Pr.

Hoogewijs Albert (1997). LL(1) Descriptions for Robots. *Robotica vol 15, 105-110.*

Huybrechts Michel (1995). Visual Mira. *CC-AI vol 12, 365-381.*

Huybrechts Michel (1999). Ventilation selection program. http://www.e2s.be/Customsoftware/Industry/vamsel/

Huybrechts Michel (1999). Managing an air conditioning system. http://www.e2s.be/Customsoftware/Industry/dacms/

Huybrechts Michel (1999). Air conditioning selection program. http://www.e2s.be/Customsoftware/Industry/hi-vrv/

Lewi J., De Vlaminck K., Huens J., Huybrechts M., (1979). A Programming Methodology in Compiler Construction: Part 1 Concepts. North-Holland.

Lewi J., De Vlaminck K., Huens J., Steegmans E., (1982). A Programming Methodology in Compiler Construction: Part 2 Implementation. North-Holland.

Lewi J., De Vlaminck K., Steegmans E., Van Horebeek L. (1992). Software Development by LL(1) syntax description. Wiley.

Nakatani Lloyd and Jones Mark (1997). Jargons and Infocentrism. *Proceedings of DSL '97 (First ACM SIGPLAN Workshop on Domain-Specific Languages)* Paris, January 18, 1997, 59-74. Published as University of Illinois Computer Science Report, http://www-sal.cs.uiuc.edu/~kamin/dsl.

Pauwels Guy (1995). Process Control and Domain-Specific Languages, *CC-AI vol 12, 425-434.*

Van Peteghem Luc (1999). Marvin Minsky: "Het menselijk brein is een machine". De Financieel-Economische Tijd, 30-11-1999, p15.

Wide Peter and Schellwat Holger (1997). Implementation of a generic algorithm for routing an autonomous robot. *Robotica vol 15, 207-211.*