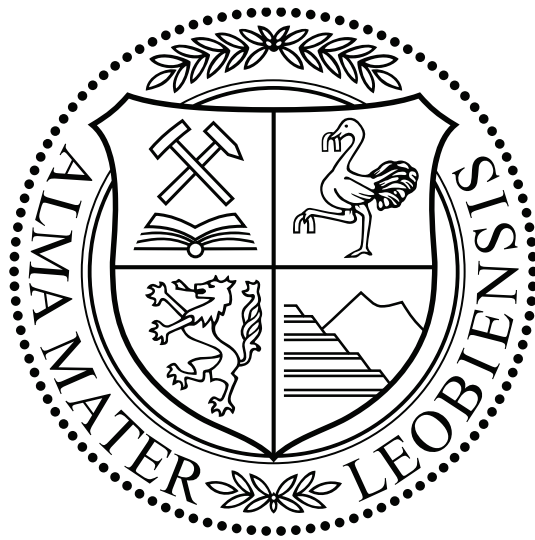


Domain Specific Languages in Automation



Diplomarbeit

Wang Song

Betreuer

Ass.Prof. Dipl.-Ing. Dr.mont. Gerhard Rath

O.Univ.-Prof. Dipl.-Ing. Dr.techn. Paul O'Leary

Montanuniversität Leoben

Institut für Automation

November 2016

Abstract

As technology develops, much of the industrial work has been replaced or simplified by computational devices (e.g. robots), and automation has become of great importance. Computer programming languages, as human and computer interaction tools, are usually difficult to understand by non-programmers. Thus, the most difficult part is the communication between the developers and software users in the software development process. In order to solve this problem, domain specific languages (DSL) were created. This thesis first describes the classification of modern DSL and the role of syntax and semantics of a language. The Backus-Naur form (BNF) is explained as an important formalism to define the syntax of a computer language. Two programming languages, which are well-known in automation technology, are described as successful examples of early DSL, G-code for controlling of manufacturing tools and a modern language for robot control. Finally the development of a specific DSL for processing finite state machines is presented. The finite state machine is an important pattern to implement sequential behaviour for industrial machinery. This is done with two tools, at first with ANTLR (Another tool for language recognition), then with PYPARSING as parser generators.

Keywords:

DSL; Finite State Machine; BNF; Lexer/Parser; Abstract Syntax Tree; Pyparsing

Kurzfassung

Mit der modernen Automatisierungstechnik (z.B. Roboter) wurden viele Tätigkeiten in der Industrie vereinfacht oder durch Maschinen ersetzt. Programmiersprachen für Computer und Steuerungen als interaktive Werkzeuge zur Kommunikation zwischen Mensch und Maschine sind für Nicht-Programmierer nur schwer verständlich. In der Software-Entwicklung wird die Kommunikation zwischen Entwicklern und Anwendern immer wichtiger. Zur Verbesserung wurden daher domain-spezifische Sprachen (Domain specific languages, DSL) entwickelt. In dieser Arbeit wird ein Überblick über die Eigenschaften von DSL und deren Klassifizierung gegeben, sowie auf Grammatik und Semantik einer Sprache eingegangen. Die Backus-Naur-Form zur Beschreibung der Syntax einer Programmiersprache wird beschrieben. Zwei bekannte Beispiele von DSL in der Automatisierung, nämlich G-Code zur Steuerung von Fertigungsmaschinen und eine Sprache zur Robotersteuerung, werden präsentiert. Die Entwicklung einer DSL wird am Beispiel eines endlichen Zustandsautomaten (Finite state machine, FSM) demonstriert. Die FSM ist ein wichtiges Werkzeug in der künstlichen Intelligenz und zur sequentiellen Steuerung von Automaten. Das wird mit Hilfe zweier Tools demonstriert, mit ANTLR (Another tool for language recognition) und mit Python/PYPARSING als Parser-Generatoren.

Schlaegerwörter:

DSL ; Finite State Machine; BNF; Lexer/Parser; Abstract Syntax Tree; Pyparsing

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich diese Arbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfsmittel bedient habe.

AFFIDAVIT

I declare in lieu of oath, that i wrote this thesis and performend the associated research myself, using only literature cited in this volume.

Leoben, October 27,2016

Wang Song

Acknowledgment

- Firstly, I would like to express my gratitude to my co-supervisor, Prof. Dipl.-Ing. Dr. mont. Gerhard Rath, a respectable, responsible and knowledgeable scholar, for his continuous help, patience and support. His guidance helped me in all the time of research and writing of this thesis. Without his support it would not be possible to finish this thesis.
- I appreciate my supervisor, O.Univ.-Prof. Dipl.-Ing. Dr.techn. Paul O'Leary who gives me the chance complete my thesis in automation institute.
- Besides my supervisor,I would like to thank Ms. Petra Hirtenlehner and Mr. Gerold Probst for all their kindness and help.
- In my daily work I have been blessed with a friendly and cheerful group of colleagues.
- I deeply thank my parents Wang Shitai and Li Feng for their supporting, patience and understanding. And thank them to give me a chance to study in Austria.
- Last but not least, I want to thank all my friends in Leoben, Dipl.-Ing. Chen Yuanfei, Dipl.-Ing. Lu Zihua, Dipl.-Ing. Zhuang yu, Dipl.-Ing. Li Yingchi, Mr.Xu Shuang and Ms.Zhou Ru for their encouragement, support and help.

Contents

1	Introduction	7
1.1	Introduction to DSL	7
1.2	The Syntax and Semantics of a Computer Language	10
2	Examples of DSL	13
2.1	The Programming Language of Industrial Robot	13
2.2	A Simple Program of MOVEMASTER RV-M2 Robot	17
2.3	G-Code as Example for DSL	23
3	Finite State Machine	26
3.1	Basic Theory of Finite State Machine	26
3.2	State Diagram in UML Notation	30
3.3	Finite State Machine in Matlab [®] Notation	33
4	BNF	36
4.1	Introduction to BNF	36
4.2	Several important symbols	36
4.3	Where can we use BNF	37
4.4	LL() Recursive-Descent	37
5	The Syntax Definition with ANTLR4	39
5.1	Introducing ANTLR	39
5.2	Syntaxtree Construction	40

5.2.1	Tokenizing	42
5.2.2	Parsing	43
5.2.3	AST	45
6	DSL Pyparsing with Python	52
6.1	Introducing Python	52
6.2	Pyparsing	53
6.2.1	Basic ParserElement, Expression	54
6.2.2	Define Grammar	55
6.3	Pyparsing of a DSL Example	56
6.4	DSL's Structured Analysis with Data-Flow-Diagram	60
6.5	Example for Parsing, Interpreting and Running a FSM	61
7	Summary and Outlook	68
A	Grammar of DSL Compiling with ANTLR4	75
B	DSL Compiling with Pyparsing	77

Chapter 1

Introduction

1.1 Introduction to DSL

What is DSL (Domain Specific Language)? In simple terms, DSL is a specific language at a specific problem area. It is just like that you speak your local accent in your hometown, you do not have to explain the meaning of each term that you said, though to others what you say might be unable to understand. So this is the important difference of GPL (General Purpose Language).

Why do we use the DSL? We can find the solution for a specific problem in GPL for sure, but a lot of complicated code will be generated and a lot of important information will be hiding in structure of the GPL. That why we use the DSL, which is not complicate and easy to read.

We want to construct a DSL, usually with Unix Style to do that: at first to define the syntax, and then through the coding technique to DSL change GPL, or write a compiler about this DSL. We called this DSL is " External DSL ". XML files is another style of the external DSL.

Developing an external DSL is similar to implementing a new language from scratch with its own syntax and semantics. The usually tools we used are YACC and LEX, ANTLR, python and so on.

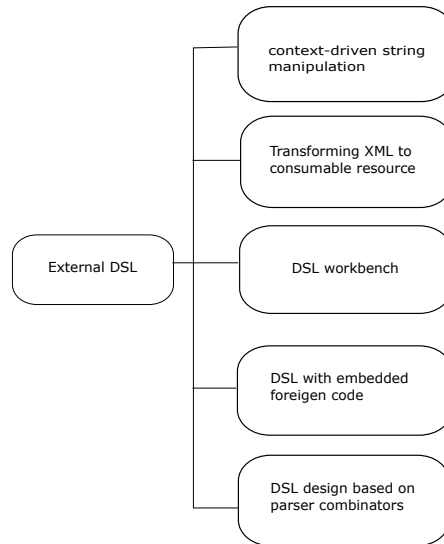


Figure 1.1: An informal micro-classification of common patterns and techniques of implementing external DSLs [5]

Context-driven string manipulation: The string is converted to the host language through a tokenization process, using techniques like regular expression matching and dynamic code evaluation. The resultant code snippet is the integration point with the application [5].

Transforming XML to a consumable resource: You are working on it usually with Spring DI framework. One of the ways you can configure the DI containers through an XML-based specification file.

DSL workbench: It is not a technique, but is a tool to help you to write the External DSL.

Mixing DSL with Embedded foreign Code: Let Language parse the extension syntax using EBNF (Extended Backus-Naur Form)

DSL design based on parser combinators: Another way to define grammar.

An Internal DSL (often called an Embedded DSL) is a Domain Specific Language that is written in an existing host language, i.e. we don't need to create a new language, we just change a GPPL (General - Purpose Programming Language) to DSL, and this DSL with its own syntax structure. The internal DSL is a specific application of a GPPL. This program, that with DSL to be written, has a custom language style. It differs from its host language.

One of the most popular internal DSLs used today is Rails, which is implemented on top of the Ruby programming Language.

Internal DSLs manifest primarily in two forms [5].

1. Generative: transformed to generate code.
2. Embedded: embedded within the type system of the host language.

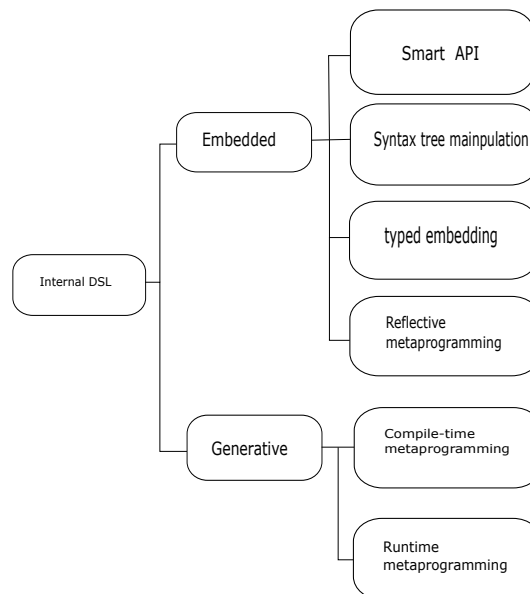


Figure 1.2: An informal micro-classification of patterns used in implementing internal DSLs [5]

Compile-time metaprogramming	Runtime metaprogramming
You define syntax that gets processed before runtime, during the compilation phase. No runtime overhead because the language runtime has to deal only with valid forms.	You define syntax that gets processed through the MOP of the Language during runtime. Some runtime overhead because meta-objects are processed and code is generated during runtime.

Table 1.1: Comparison of compile-time and runtime metaprogramming [5]

Smart API: is based on chaining method, also called fluent interface. Groovy or Ruby offer *arguments* to help build Smart API.

Syntax tree manipulation: We can generate code by manipulating the AST (Abstract Syntax Tree).

Typed Embedding: Some Statically typed language (like Haskell, Scala) offer *types* to abstract domain semantics and make syntax more concisely, without using technique of generating codes.

Reflective Metaprogramming: A language (usually like Ruby) discover methods at runtime, of which the name isn't known yet. We'll use the metaprogramming abilities of Ruby to do a dynamic dispatch on the object, instead of the usual dot notation of invoking methods statically. This coding technique is reflective metaprogramming.

Metaprogramming: Another form of metaprogramming can generate code dynamically during runtime. Runtime metaprogramming is another way by which you can achieve small surface syntax for your DSL [5].

1.2 The Syntax and Semantics of a Computer Language

Now that we want to write our own DSL, the syntax and semantic is very important. A language, whether natural (such as English) or artificial (such as Java), is a set of strings of characters from some alphabet. The strings of a language are called sentences or statements. The syntax rules of a language specify which strings of characters from the language's alphabet are in the language. English, for example, has a large and complex collection of rules for specifying the syntax of its sentences. By comparison, even the largest and most complex programming languages are syntactically very simple [18].

In simple terms, the syntax of the computer language is a set of rules, which define the combinations of symbols, let's look at a very simple example (Java statement):

```
Exp = 5*ID+3;
```

The lexemes and tokens of this statement are:

<i>lexemes</i>	<i>tokens</i>
Exp	identifier
=	equal_sign
5	integer
*	mult_op
ID	identifier
+	plus_op
3	integer

Table 1.2: The meaning of this statement

Then we will make a detailed discussion of a method of describing syntax, BNF (Backus-Naur Form, also known as context-free grammars) in chapter 3.

The definition of semantics is giving a categorical meaning of a syntactically legal program. In other words, for programming languages, semantics describe the behavior that a computer follows when executing a program in the language. We might disclose this behavior by describing the relationship between the input and output of a program or by a step-by-step explanation of how a program will execute on a real or an abstract machine [3].

The biggest difference between syntax and semantics of the computer is that the former is a meaning of expressions, statements, and program units, which is a program of those expressions, statements, and program units. For example: 'the capital of Austria is Vienna', the syntax and semantics of this sentence is both correct. 'the capital of Austria is Berlin', the syntax of this sentence is correct, but the semantics is incorrect.

Due to the limitations of BNF, the semantics of the programming language consists of static semantics and dynamic semantics. The static semantics of a language is only indirectly related to the meaning of programs during execution; rather, it has to do with the legal forms of programs (syntax rather than semantics) [18]. For instance, all tokens of a program must be declared before using, the type of tokens and integer in expressions and the type of operator must compatible, these mandate belong to the static semantics restrictions. The meaning of a program involves also what a program is doing and to prove, that usually expressed through program execution [6]. Whether it is static or dynamic semantics, we need

a method to describe it. Formal semantics describe semantics in - well, a formal way - using notation which expresses the meaning of things in an unambiguous way. It is the opposite of informal semantics, which is essentially just describing everything in plain English. This may be easier to read and understand, but it creates the potential for misinterpretation, which could lead to bugs because someone may not read a paragraph the way you intended them to read it. There are many approaches to formal semantics, these belong to three major classes: Denotational semantics, Operational semantics and Axiomatic semantics.

Chapter 2

Examples of DSL

In the previous chapter we have already talked about definition and classification of DSL. In this chapter we will continue to discuss it with two examples.

2.1 The Programming Language of Industrial Robot

Typical application of DSL is the robot language. With the development of the robot, the robot language has been developed and perfected. The robot language has become an important part of robot technology. The main function of a robot has been achieved with robot language. In the early stage, the robot is able to control with a simple action or a single function, which can be controlled by fixed program. With the diversification of the robot's action and the complication of the working environment, the fixed program can not be satisfied. The robot programming language is generated.

1. VAL : Variable Assembly Language (VAL) is a computer-based control system and language designed by U.S.A Unimation company in 1979, mainly used in PUMA and Unimation's robot.

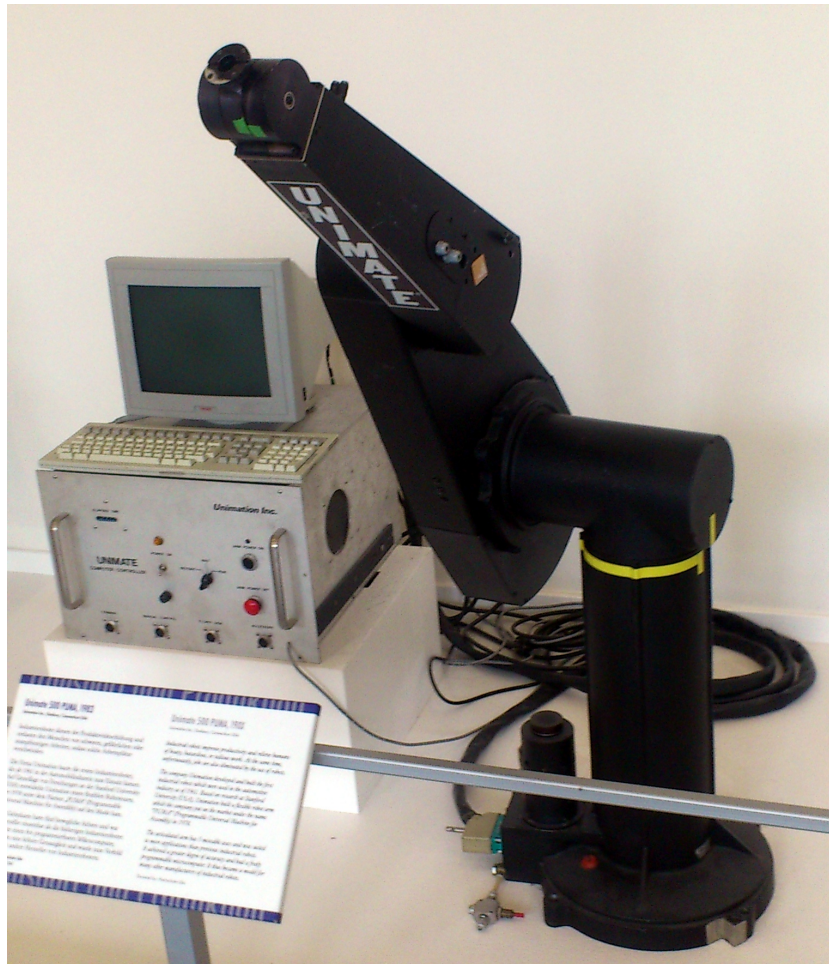


Figure 2.1: Unimate 500 PUMA (1983), control unit and computer terminal at Deutsches Museum, Munich

VAL language is based on BASIC language. VAL language command is simple, clear and easy to understand, which describes the action of the robot and communication of the host computer. The VAL language consists of monitor commands and program instructions. The monitor commands are used to prepare the system for execution of user-written programs. Program instructions provide the repertoire necessity to create VAL programs for controlling robot actions.

The monitor commands consists consists of position and attitude definition instructions, program editing instructions, list instructions, store instruction, control program

execution instructions and system status control commands. For example, the BASE commands belongs to the position and attitude definition instructions, which is used for setting a reference coordinate system. The form is :

```
BASE [<dX>], [<dY>], [<dZ>], [<rotation of Z-direction>]
```

For example:

```
BASE 300, 0, -50, 30
```

means that redefined the position of the reference coordinate system, which moves from the initial position to the x-direction 300 mm, to the Z-direction -50 mm, and rotate 30 degrees clockwise around Z-direction.

2. AL : Assembly Language. Designed in 1974 at the Stanford Artificial Intelligence Laboratory as a frame-oriented language with openings to artificial intelligence concepts. Its basic concepts have influenced LM and SRL. The language is used to program a set of four robots of different types and a two-dimensional vision system [8]. The structure of the AL is similar to the PASCAL language, also can be compiled into machine language to run on the real time control. The AL system includes a big mainframe computer, and it generally was running on PDP 11/45. The PDP 11/45 implements one terminal, 128 KB RAM memory, and floating point processor. This language has got the capability to control two Stanford Scheinman and two PUMA 600 arms simultaneously.

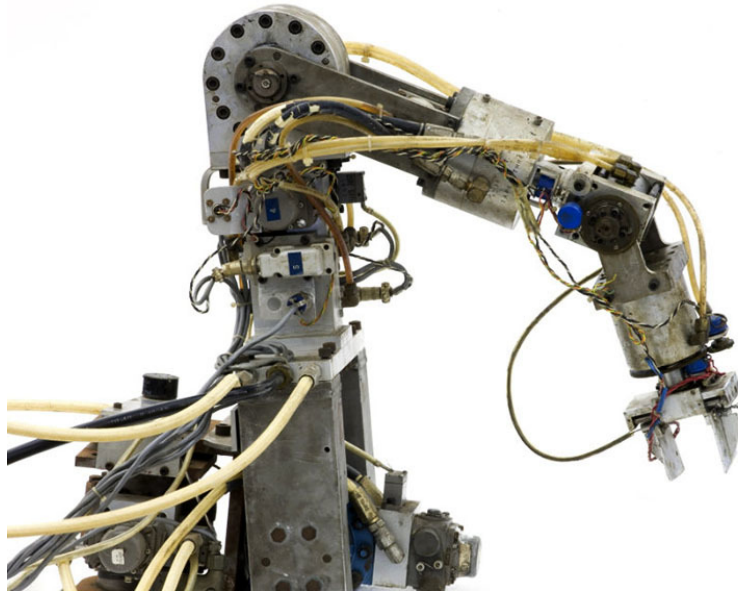


Figure 2.2: Victor Scheinman's stanford arm in 1969

3. RAIL : RAIL is a high-level robot programming language developed by Automatix Inc in 1981 for controlling their Cybervision, Autovision, and Robovision systems. Cybervision system is designed for performing assembly operation, Autovision (Machine Vision) system for identification and inspection process, and Robovision system for Robot arc welding process. RAIL language includes three data types like Paths, Points, and Reference Frames for robot locations. It has several special-purpose commands for interfacing a robot with other equipments. Apart from these functions, this robot language also provides many programming features [1].
4. SIGLA : SIGma LAnguage. The language for programming Olivetti SIGMA robots. Now quite obsolete and under replacement, it has been available since 1975. SIGLA is a complete software system which includes: a supervisor, which interprets a job control language, a teaching module which allows teaching-by-guiding features, an execution module, editing and saving of program and data. SIGLA has been in use for years at the Olivetti plant in Crema (Italy). Its applications span from assembly to riveting, drilling, milling. All the system and the application program run in 4K of memory, and this compactness was necessary at the time SIGMA was delivered because memory was still expensive [17].



Figure 2.3: The Olivetti "SIGMA" Cartesian coordinate robot, is one of the first used in assembly applications in 1969

2.2 A Simple Program of MOVEMASTER RV-M2 Robot

The industrial micro-robot "MOVEMASTER RV-M2" was developed by Mitsubishi Electric Corporation, which can be utilized in a wider range of applications such as educational and research purposes and the automation of handling works in production lines and inspection works at laboratories.

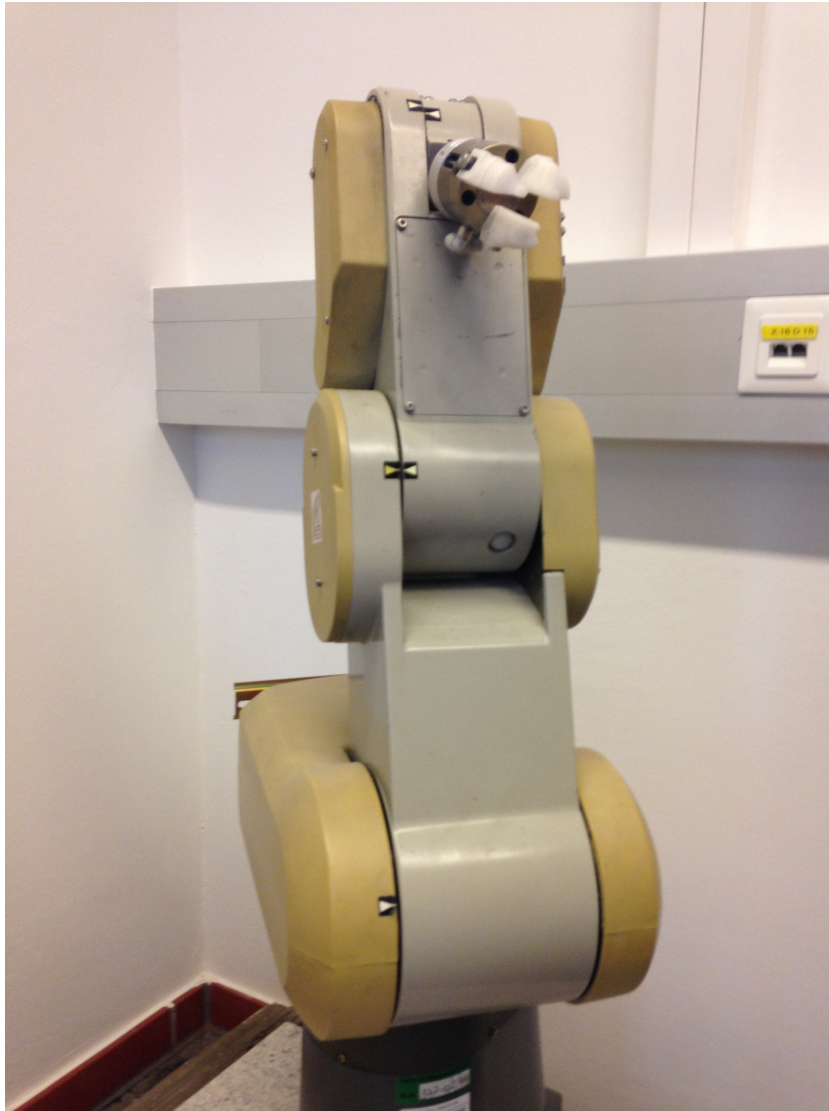


Figure 2.4: The Mitsubishi Electric "MOVEMASTER RV-M2" industrial micro-robot

The "MOVEMASTER RV-M2" has excellent position (5 degrees of freedom) repeatability, combined with a high velocity but reliably constant speed, It has also high lifting capacity (2kg). In addition to many other potential tasks can be achieved. The typical command:

1. MA: Move Approach: Moves the end of the hand form the current position to a position away from the specified position in increments as specified for another position.

MA <Position number (a)>, <Position number (b)> [, <O or C>]

Sample Input:

MA, 2, 3, C

Explanation:

- (a) Moves the end of the hand from the current position to a position away from position (a) in increments as specified for position (b). It does not change the coordinates of positions (a) and (b).
- (b) If the open/close state of the hand has been specified (O: open; C: closed), the robot moves after executing the hand control instruction. If it has not been specified, the hand state in position (remains) valid.

2. MO: Move: Moves the end of the hand to the specified position.(Articulated interpolation)

MO <Position number> [, <O or C>]

Sample Input:

MO, 2, C

Explanation:

- (a) Moves the end of the hand to the coordinates of the specified position.
- (b) O/C(open/close) command is the same definition as the last command MA.

3. GO: Grip Open: opens the hand(to release workpiece)

Sample Input:

GO

4. GC: Grip Close: closes the hand(to hold workpiece)

Sample Input:

GC

5. PD: Position Define: Defines the coordinates(position and angle) of the specified position.

PD <Position number>, <X-axis coordinate>, <Y-axis coordinate>, <Z-axis coordinate>, <Pitch angle>, <Roll angle>, [, <O or C>]

Sample Input:

PD 20, 200, 350, 300, -60, -30, C

Explanation :


```
80 MA 2, 20, C      / Moves the robot to a location 30 mm above  
                    position 2.  
90 MO 2, C          / Moves the robot to position 2.  
100 GO              / Opens the hand to release the workpiece.  
110 MA 2, 20, O     / Moves the robot above position 2 (distance 30  
                    mm) with the workpiece grasped.
```

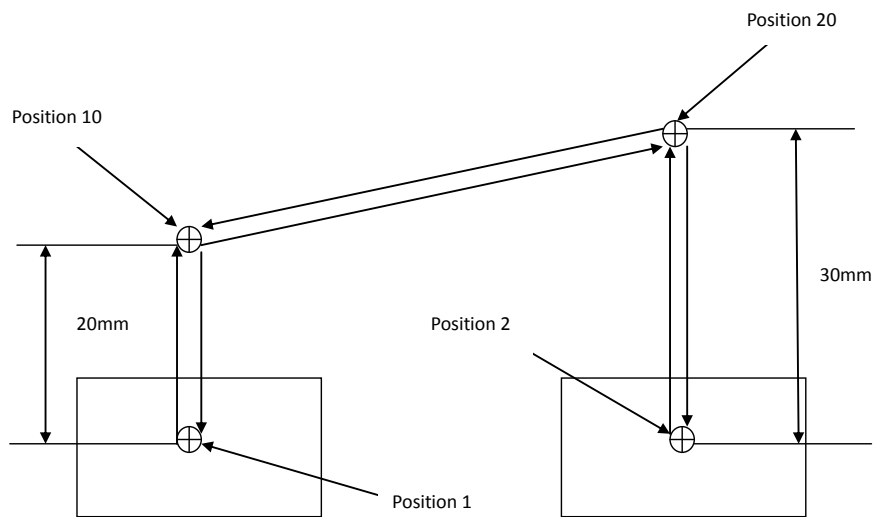


Figure 2.5: Pick-and-place work

where the numbers q_1, q_2, q_3, q_4, q_5 and q_6 are computed via inverse transformation from the position $[X, Y, Z, A, B, C] = [10, 15, 20, 25, 30, 35]$.

4. G90/G91: Absolute/Relative Position Coordinates. A movement to a position in the coordinate system can be specified using absolute or relative coordinates.

G90: Absolute Position Coordinates The entry of the coordinates is absolute, that means the given values refer to the current zero point.

G91: Relative Position Coordinates The entry of the coordinates is relative, that means the given values refer to the current position.

Example :

```
G90 G01 X40 Y20 / The X axis is moved to the absolute
                  position 40 and the Y axis is moved
                  to the absolute position 20.
```

```
G91 G01 X40 Y20 / From the current position the X axis
                  is moved 40 units in the positive
                  direction and the Y axis is moved 20
                  units in the positive direction.
```

5. G02/G03: Circular Interpolation Clockwise/Counter Clockwise. Circular interpolation is programmed using the path information G02 or G03, and the entry of the following parameters: target position <coordinate>, center of the circle <interpolation parameter> and the path feed <path feed>. The target position is approached with the programmed speed. The interpolation parameter is defined as follows :

- (a) I : Position of the center of the circle in the X direction (absolute or relative to the start position).
- (b) J : Position of the center of the circle in the Y direction (absolute or relative to the start position).
- (c) K : Position of the center of the circle in the Z direction (absolute or relative to the start position).
- (d) H : Rotation angle of the circle.

(e) R : Radius of the circle.

Example :

```
G03 I30 H1440 Z300 F1000 / From the recent position move
                           to the targeted position via
                           4 full counter clockwise to
                           the targeted position (x=30,
                           z=100) with with the programmed
                           speed.
```

Let's look at a simply program about movement of the robot with G code:

```
N010 G90
N020 F5000
N030 G101 Q1=0 Q2=45 Q3=90 Q4=0 Q5=45 Q6=0
N040 G91
N045 F10000
N050 G01 Y-100
N060 G01 X100
N070 G01 Y100
N080 G01 X-100
N081 F10000
N085 G03 I30 H1440 Z300
N090 G90
N100 G101 Q1=0 Q2=0 Q3=0 Q4=0 Q5=0 Q6=0
```

Chapter 3

Finite State Machine

3.1 Basic Theory of Finite State Machine

A finite-state machine (FSM), or simply a state machine is a mathematical model of computation, which exists a finite number of states and transfer or action between these states. A model of computation consists of a set of states, a start state, an input alphabet and a transition function. Computation begins in the start state with an input string. It changes to new states depending on the transition function [20].

If *output*, i.e. entry action depends only on the states, we call this machine a *Moore* model. The advantage of the Moore model is a simplification of the behaviour.

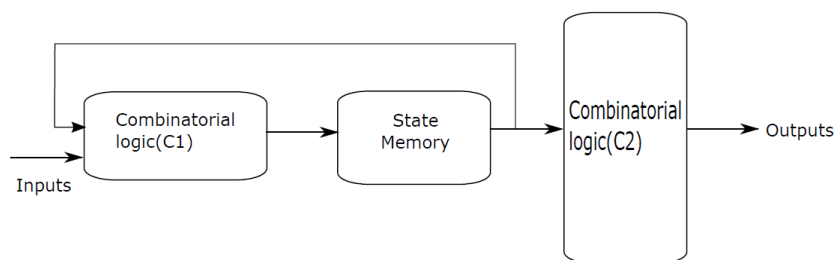


Figure 3.1: A Moore model for a generalised non-clocked sequential machine

If *output* values depends on its current state and the current *inputs*, we call this FSM a *Mealy* model.

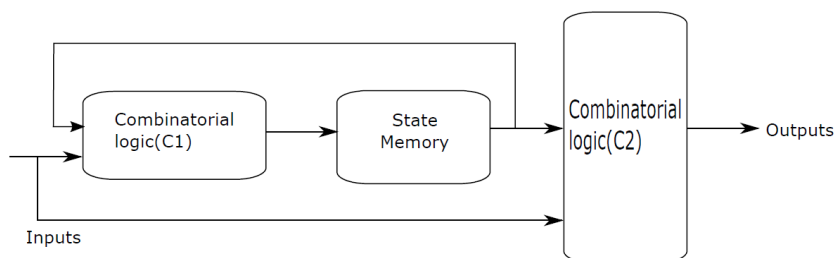


Figure 3.2: A mealy model for a generalised non-clocked sequential machine

Comparison between Mealy model and Moore model:

1. The output sequence of Mealy circuits is one clock cycle earlier than Moore circuits' [9]. Because the input of Mealy machine action immediately in present circuit, the input of Moore effects the next state, and the output depends on only the state. So by this point we can find out the reason, which the Mealy machine is 'faster' than Moore machine.
2. Moore circuit is more complex than the Mealy circuit because there is more than one state.

We will discuss the FSM further with a simple Example. Consider a very simple elevator: There is a up-down elevator, which has only three buttons: UP, DOWN and STOP. If a person press UP, the elevator will rise until the End-Up switch is activated. And the elevator will descend until the End-Down switch is activated, if you press DOWN. The elevator will stop for waiting the user to press another button with someone to press STOP.

Consider the states of elevator:

There are five states: The elevator is at the top position; the elevator is going down; the elevator is in the bottom position; the elevator is going up and the elevator is waiting.

There are five triggers, i.e.transition condition: UP; DOWN; STOP; End-Up and End-Down.

State	Comment
S0	The elevator is at the top
S1	The elevator is going down
S2	The elevator is at the bottom
S3	The elevator is going up
S4	The elevator is waiting

Table 3.1: state table of the elevator

We can model the system graphically, using cycle to represent states and arrows to represent transitions between states, as shown in Figure 3.3 below. And the definition of states is shown in above Table.

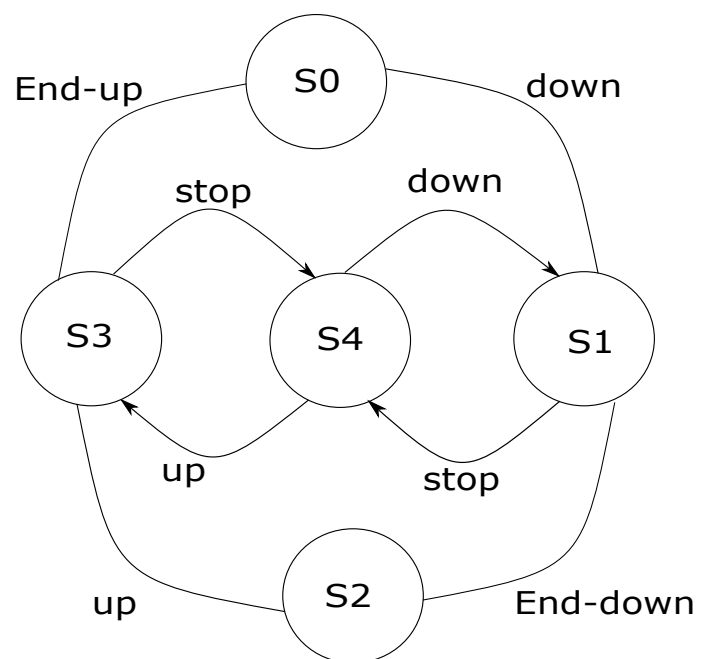


Figure 3.3: A simple state machine model of a elevator

3.2 State Diagram in UML Notation

The Unified Modeling Language (UML) is a general-purpose modeling language in the field of software engineering, which is designed to provide a standard way to visualize the design of a system. It was created and developed by Grady Booch, Ivar Jacobson and James Rumbaugh at Rational Software during 1994-95, with further development led by them through 1996. [7]

A state machine diagram of UML models the behaviour, which explain the series of a object during its lifetime, and respond to event [23].

Consider the previous example of elevator, a state machine diagram of UML is shown in Figure 3.4 below.

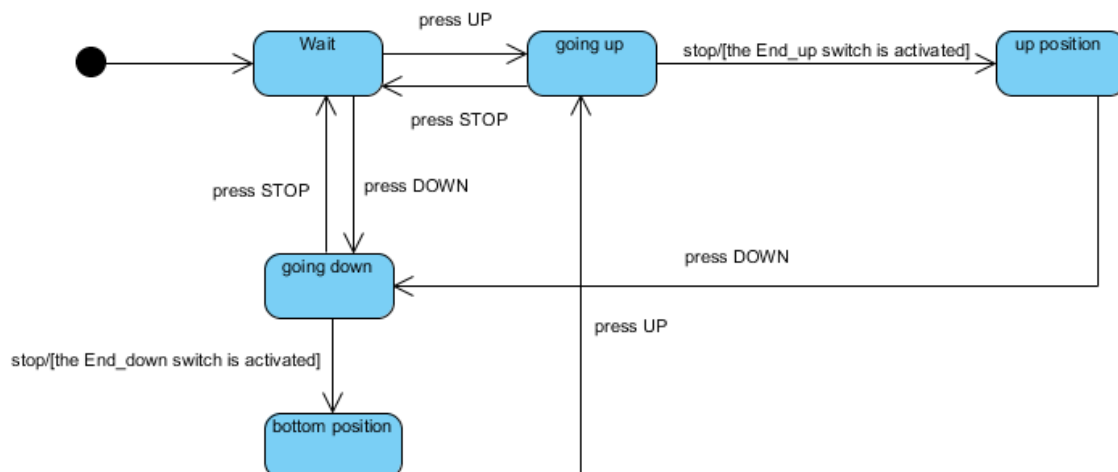


Figure 3.4: UML state diagram

A state diagram consists of State, Transition, Event, Activity, Action and so on:

1. **State** is a state of a model during its lifetime. A state is denoted by a round-cornered rectangle with the name of the state written inside it.



Figure 3.5: a state in state diagram

2. **Initial State**, which is a pseudo state, is denoted by a filled black circle and may be labeled with a name.
3. **Final State** is an endpoint of a state diagram, which is denoted by a circle with a dot inside and may also be labeled with a name.

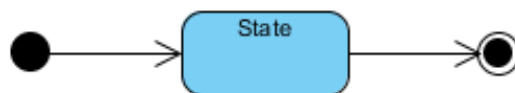


Figure 3.6: Initial state and final State in state diagram

4. **Transition** from one state to the next are denoted by lines with arrowheads. A transition may have a trigger, a guard and an effect, as shown in Figure 3.7 below.



Figure 3.7: Transition in state diagram

Trigger is the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time. *Guard* is a condition which must be true in order for the trigger to cause the transition. *Effect* is an action which will be invoked directly on the object that owns the state machine as a result of the transition.

5. **Decision** flows the results according to different conditions of Guard, which is denoted by a hollow diamond, as shown in Figure 3.8 blow.

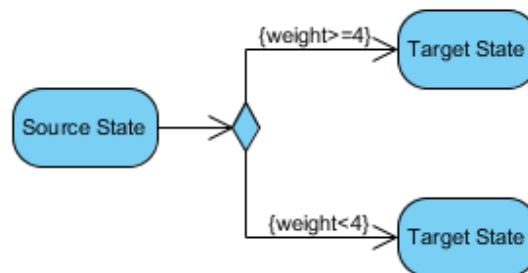


Figure 3.8: Decision in state diagram

6. **Synchronization** is defined a Fork or Join of a work flow, which is denoted by a short and thick line.

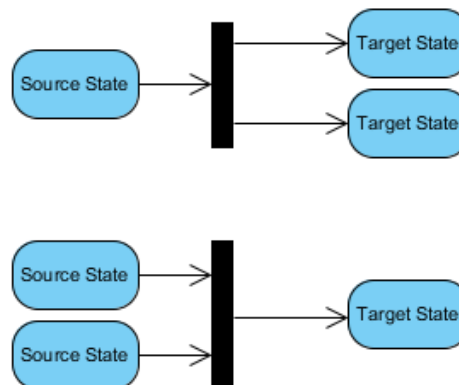


Figure 3.9: Fork and Join in state diagram

3.3 Finite State Machine in Matlab® Notation

With Matlab/Simulink to compile a state diagram, we can use *Stateflow*® charts.

Stateflow is an environment for modeling and simulating combinatorial and sequential decision logic based on state machines and flow charts. Stateflow lets you combine graphical and tabular representations, including state transition diagrams, flow charts, state transition tables, and truth tables, to model how your system reacts to events, time-based conditions, and external input signals.

We create a new model in Simulink and add a Chart object from the Stateflow, as shown in Figure 3.10 below.

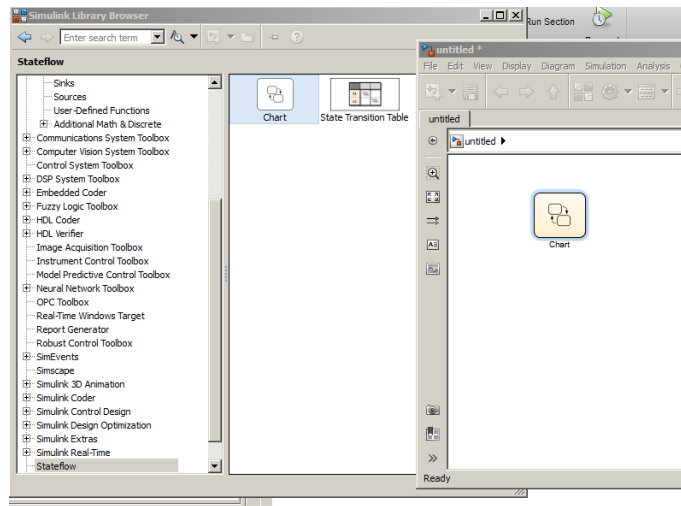


Figure 3.10: Add a Chart object into new model

And then we can define this Chart as wish.(Figure 3.11)

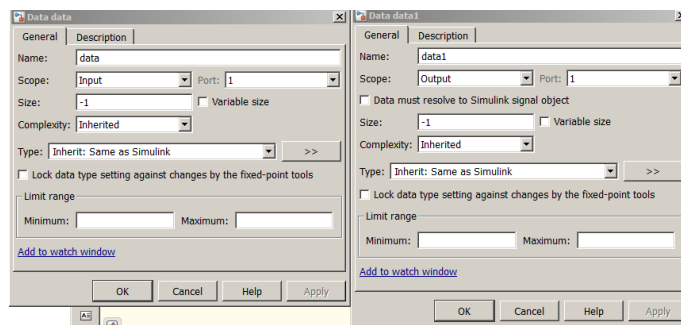


Figure 3.11: Define inputs and outputs

For definition of a state, Matlab allows more than one entry/exit action, and UML can not do this.(Figure 3.12)

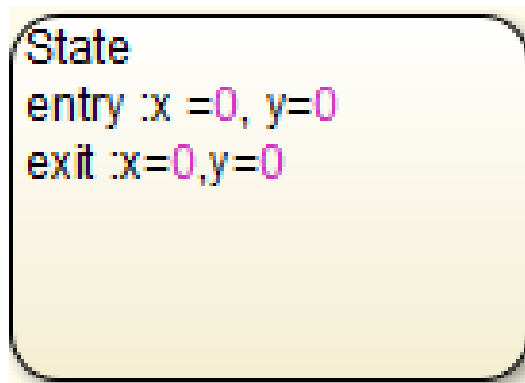


Figure 3.12: Define a state

Transition is very similar to UML. Condition action is done before condition is checked.

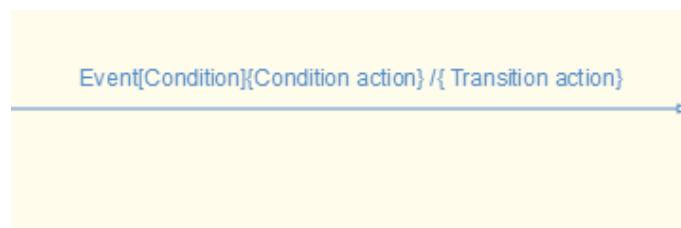


Figure 3.13: Define transition

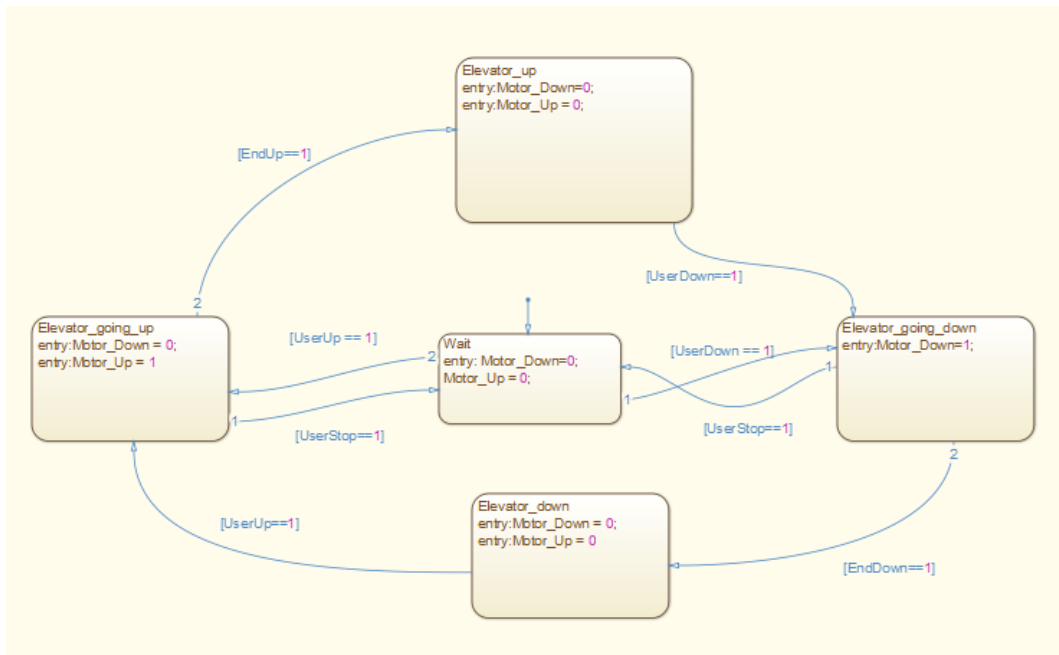


Figure 3.14: The previous example of elevator with Matlab/Simulink

Chapter 4

BNF

4.1 Introduction to BNF

When we learn or write a language, we must come across the BNF. So what is BNF? BNF (Backus Normal Form or Backus-Naur Form) and EBNF (Extended Backus-Naur Form) is a way of writing grammars to define the syntax of a language. It was invented to describe the Algol language in the 60s. Since then, BNF grammars have been widely used both for explanation and to drive Syntax-Directed Translation [10]. BNF is way of a language for defining syntax, it does not have a itself have a standard syntax. This syntax can be achieved during multiplicity symbols.

4.2 Several important symbols

At first, we must know the most important concept of BNF: terminal and nonterminal.

Let's talk about them with a simple example:

```
symbol ::= expression
```

where expression is a terminal, that never appears on the left side, consists of one or more sequences of symbols; on the other hand, symbol that appears on the left side is a nonterminal; and '::<=' means that the symbol on the left must be replaced with the expression on the right.

We extend the previous the example:

```
S ::= D | '-' D*
```

Symbol	meaning	example
	indicates choice between elements	D D*
*	none or more	D*
+	one or more	D+
?	optional	D?
..	range	'0'..'9'

Table 4.1: BNF symbols

```
D ::= '0'..'9'
```

In this example, "D" is a nonterminal instead of terminal. We can get a positive integer within ten or a negative integer.

4.3 Where can we use BNF

I have already mentioned in the previous section, the most important application of BNF is defining syntax (Grammar) for a language. In another way, you'll need to use BNF whenever you are working with a parser generator, as these tools use BNF grammars to define how to parse. It's also very useful as an informal thinking tool to help visualize the structure of your DSL, or to communicate the syntactic rules of your language to other human [10].

4.4 LL() Recursive-Descent

LL() recursive-descent parser is a top-down parser. To implement an LL() recursive-descent parser, we can find the lookahead expression (the first nonterminal sets) for each row. To make parsing decisions, the parser tests the current lookahead token against the alternatives' lookahead sets. A lookahead set is the set of tokens that can begin a particular alternative. The parser should attempt the alternative that can start with the current lookahead token [15].

Let's look at a simple example:

```
state : 'pressUp' // lookahead set is pressUp
      | 'pressGoingDown' // lookahead set is pressGoingDown
      | 'pressDown' // lookahead set is pressDown
```

An alternative that begins with a token reference. Its lookahead set is just that token. In this

case, i define the state with a token reference: `pressUp`, `pressGoingDown` and `pressDown`, which each alternative begins with a single token reference.

If we begin with a rule reference instead of a token reference, that the lookahead set is whatever begins with any alternative of the rule. It is just like this:

```
Identifier : 'pressUp' | 'pressGoingDown' | 'pressDown'
FSM-keyword : 'FSM'
state : Identifier
END-keyword : 'END'
FSM : FSM-keyword // lookahead set is FSM_keyword
    | state+ // lookahead set is {Identifier}
    | END-keyword // lookahead set is END_keyword
```

The second alternative of the lookahead set is a union of the lookahead sets from *state*. Each token has two basis properties: type and payload. The type is the kind of token we have, for example: `FSM-keyword` or `state`. The payload is the text that was matched as part of the lexer: `pressUp` or `pressDown`. For keywords, the payload is pretty much irrelevant; all that matters is the type. For identifiers, the payload does matter, as that's the data that will be important later on in the parse. Lexing is separated out for a few reasons. One is that this makes the parser simpler, because it can now be written in terms of tokens rather than raw characters.[10] Now let's consider what happens when the same token predicts more than one alternative:

```
expr : ID '++' //match "D++"
     | ID '--' //match "D--"
```

The two alternatives begin with the same token: *ID*. The token beyond dictates which alternative phrase is approaching. in other words, *expr* is LL(2). An LL(1) parser can't see past the left common prefix with only one symbol of lookahead. Without seeing the suffix operator after the *ID*, the parser cannot predict which alternative will succeed [15].

Chapter 5

The Syntax Definition with ANTLR4

5.1 Introducing ANTLR

ANTLR is a tool of the computer language which provides a framework to the programmer. It is a powerful parser generator that you can use to read, process, execute, or translate structured text or binary files. It's widely used in academia and industry to build all sorts of languages, tools and frameworks [16].

Recognizing and processing of the programming language is the primary task of ANTLR. The compiling of programming language consists of two parts (Front-End and Back-End), the Front-End includes lexical analysis, syntax analysis, semantic analysis and intermediate code generation; the Back-End includes code generation, code optimization, etc. The problem of the Front-End can be solved with ANTLR.

In simple terms, the task of lexical analysis is tokens identification. The process of tokens identification which is called lexer. The lexer can group related tokens into token classes, or token types, such as INT (integers), ID (identifiers), FLOAT (floating-point numbers), and so on [16].

Grade = 100;  'Grade', ' ', '=', '100', ';'

Figure 5.1: Language recognizer

A simple expression is shown in Figure 5.1, we found five tokens: Grade, whitespace character, equal character, 100 and semicolon character. Each token can be defined by BNF rules.

Programs which recognize language are called parsers or syntax analysis. Construction of a grammar generally follows the rules of BNF. A parser checks sentences for membership in a specific language by checking the sentence's structure against the rules of a grammar. The best analogy for parsing is traversing a maze, comparing words of sentence to words written along the floor to go from entrance to exit. ANTLR generates top-down parsers that can use all remaining input symbols to make decisions. Top-down parsers are goal-oriented and start matching at the rule associated with the coarsest construct, such as program or inputFile [16].

5.2 Syntaxtree Construction

In the following section, an example for a DSL is constructed. A language that implements a finite state machine efficiently is developed. Later it will be parsed and interpreted to verify the design process. The following elements are used for the language.

FSM / ENDFSM This keywords delimit a finite state machine and will enable to define more than one FSM in a single program file. The statement FSM should be followed by a unique name.

ENABLE Program components for industrial controls usually have an enable input, which prevents running the code, when not necessary. This is useful for saving computing time and to enhance reliability of the software.

STATE States are the main parts of a FSM. The keyword is followed by a unique name for the state.

INIT The system has to know, which state is active initially. Here this is indicated clearly with a separate keyword for the initial state.

ERROR An automated system can have several error states. Basically those are states like the others also. In automation practice a special case is often useful. Resolving an error condition can have the same procedure for failure in different situations. For example, a motor can fail moving up or moving down. For this case, a RETURN statement like in a subroutine is required to move back to the previous state. This is not a typical part of a FSM, where the system state is memorised in the states only. But for practical work this pattern is quite useful.

Commands, which consist of setting or resetting digital outputs, are executed in three different situations: During each cycle, when the system is in the actual state. Once, when the system is entering the actual state. This is indicated by the keyword ENT. Finally, also once, when a transition is going on to change the state. It was decided not to implement exit actions, like they are defined in UML.

TRANS Transitions are indicated by this keyword, a condition follows. If true, the transition gets active and the state changes.

TIMEOUT The operation of many process are monitored by the time they require. A dedicated timeout transition saves the necessity to program a timer.

At first, let's define a DSL for programming the FSM:

```

FSM pressControl
ENABLE enableInput
STATE pressGoingDown
    ENT motorDown = TRUE
    warnLamp = TRUE
    TRANS pressDown ON endDown == TRUE
    motorDown = FALSE
    TIMEOUT stop 60s
    motorDown = FALSE
STATE pressDown
    TRANS pressGoingUp ON commandUp == TRUE
STATE pressGoingUp
    ENT motorUp = TRUE
    LOG message
    warnLamp = TRUE
    TRANS pressUp ON endUp == TRUE
    motorUP = FALSE

```

```

    TIMEOUT stop 60s
    motorUp = FALSE
INIT pressUP
    TRANS pressGoingDown ON commandDown == TRUE
ERROR stop
    ENT motorUp = FALSE
    ENT motorDn = FALSE
    warnLamp = TRUE
    RETURN ON commandQuit == TRUE
ENDFSM

```

There are several keywords in this program, such as : FSM, ENABLE, STATE, INIT, ERROR and ENDFSM. This program will start with keyword "FSM", and end with keyword "ENDFSM", in which every status can be defined. The keyword "ENABLE" represents input variants. Moreover, the program consists of several states and two special states (INIT and ERROR).

Let's see the details of a state:

```

STATE pressGoingDown
    ENT motorDown = TRUE // command definition with output variants
    warnLamp = TRUE // during action definition
    TRANS pressDown ON endDown == TRUE // transition action
    motorDown = FALSE // command activation after transition action
    TIMEOUT stop 60s // definition a warning time
    motorDown = FALSE // command activation after warning time

```

In this small program, commands are able to be activated by manual (with keyword 'ENT') or automatically. The action during every state will be continued in the whole process.

5.2.1 Tokenizing

Tokenizing of this program is not complex. There are a few keywords, operators ("=" and "=="), digits and identifiers. ANTLR allows us to put the keywords as literal text in the grammar rules, which is generally easier to read. So we only need lexer rules for identifiers and space.

```

ID : [a-zA-Z]+ ; // match lower-case identifiers
INT : [0-9]+ ;
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines

```

All upper case of letters will be shown as keywords, such as: STATE, ENT, TRUE, TRANS and so on. The upper case and lower case combination will be shown as identifiers, such as: pressGoingDown, warnLamp, endDown and so on.

In this case, WS(whitespace), including the line endings and newlines, is removed. ANTLR allows you to do this by sending whitespace tokens on a different channel, with syntax like:

```
WS : [ \t\r\n]+ -> skip
```

.

5.2.2 Parsing

Parsing is the process of determining how a string of terminals can be generated by a grammar [2].

Parsers that handle specific bits of input, such as floating-point numbers, integers, etc., are combined together to form parsers for larger expressions. A good parser library supports sequential and alternative cases, repetition, optional terms, etc [21].

Let's look at the grammar of the above DSL:

```
grammar FSMTEST;

machine: fsm_keyword enable_keyword state* init_state error_state '
    ENDFSM' ;
fsm_keyword : 'FSM' ID;
enable_keyword : 'ENABLE' i_var;
i_var : ID;//input_variante:'enableInput' | 'commandDown' | '
    commandUp' | 'commandQuit' ;
o_var :ID;//output_variante: 'motorDown' | 'motorUp' | 'motorDn' | '
    warnLamp' ;
sna: ID;//state_name
eq: '=' ;
ceq: '==';

state: 'STATE' sna
    com*
    dac*
    lcom*
    dac*
```

```

    tr*
    com*
    wt*
    com*;

init_state : 'INIT' sna
    tr*;
error_state : 'ERROR' sna
    com*
    dac*
    rcom*;
com : 'ENT'? o_var eq le
    //command:| output_variante '=' logic_element;
dac : o_var eq le;//duringaction
tr : 'TRANS' sna 'ON' i_var ceq le ;//transition
lcom : 'LOG' ID ;//logcommand
wt : 'TIMEOUT' sna INT time_units ;//waringtime
time_units : 'ms' | 's' | 'min' | 'h' | 'd';
le : 'TRUE' | 'FALSE';//logic_element
rcom : 'RETURN' 'ON' i_var ceq le;//returncommand

ID : [a-zA-Z]+ ; // match lower-case identifiers
INT : [0-9]+ ;
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines

```

This grammar, FSMTEST, which defines a domain specific language for finite state machine, is composed of some rules.

Following the definition:

1. As i said above, this program (machine) consists of three keywords (fsm_ keyword, ENDFSM and enable_ keyword), one or more states, one or more special states (init_ state and error_ state).
2. Each state(special state) consists of several commands (logcommand, returncommand), several actions(duringaction transitionaction) and a few of special commands(warningtime).
3. A command could be expressed as a optional keyword (ENT) and a value of a logic output (like: motorDown = TRUE, motorDown is a output variable). Logcommand

consists of a keyword (LOG) and the message of circulation. Returncommand could be represented as a value of the input variable.

4. Action in this program consists of duringaction and transitionaction, duringaction is very similar to command, which a logical value assigned to a output variable.

The parsing result is shown in Figure 5.2:

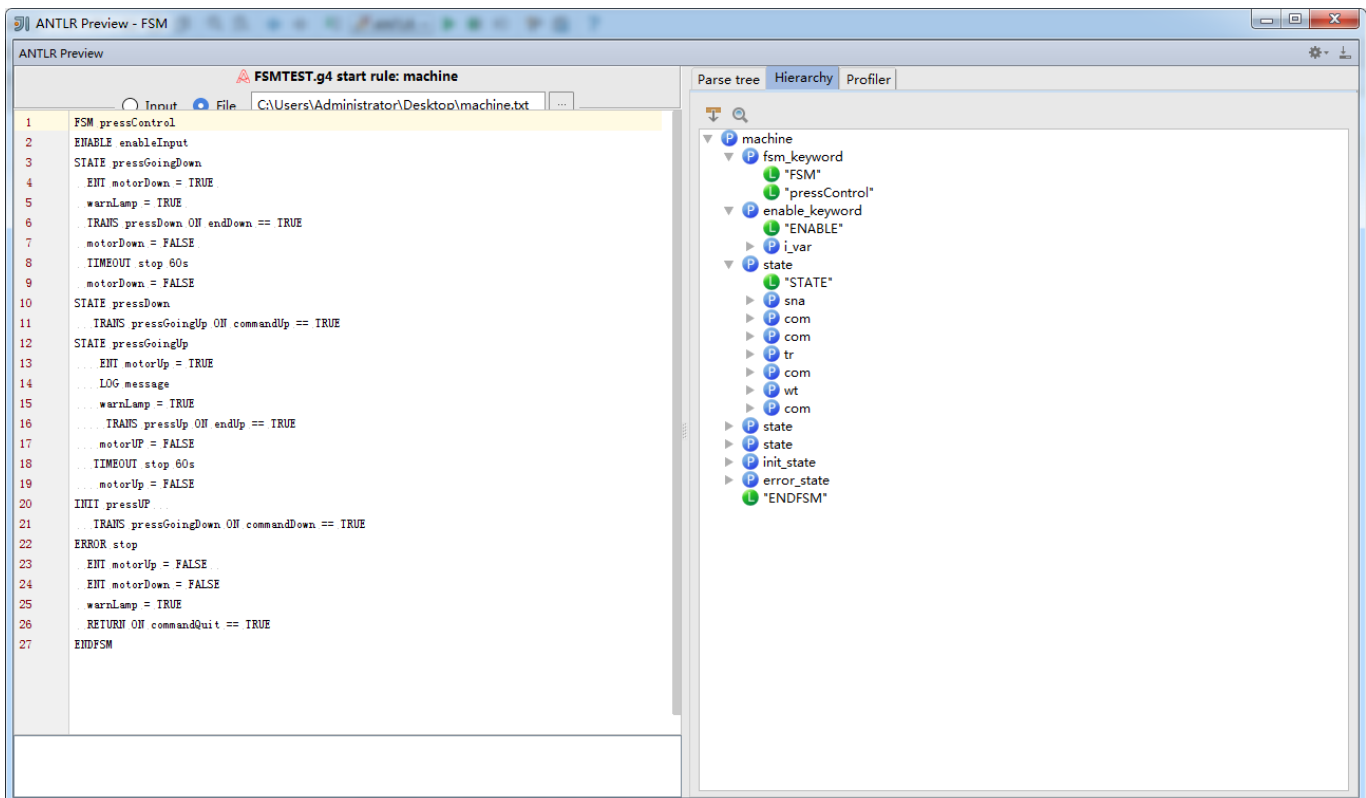


Figure 5.2: The parsing result

5.2.3 AST

In computer science, an abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code [14].

The parser creates and returns a syntax tree representation of the source text that is manipulated later by tree-walking tree [10].

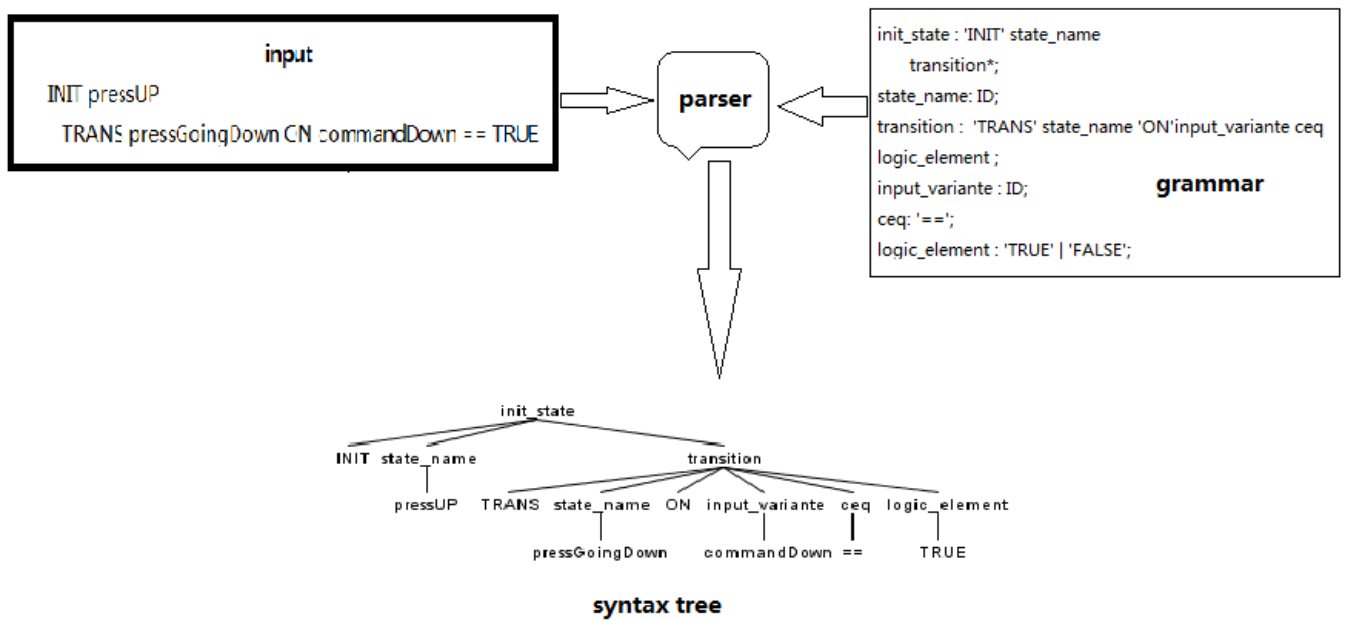


Figure 5.3: Tree construction

Any parser using Syntax-Directed Translation builds up a syntax tree while it's doing the parsing. It builds the tree up on the stack, pruning the branches when it's done with them. With Tree Construction, we create parser actions that build up a syntax tree in memory during the parse. Once the parse is complete, we have a syntax tree for the DSL script. We can then carry out further manipulations based on that syntax tree. If we are using a Semantic Model, we run code that walks our syntax tree and populates the Semantic Model [10].

In the end, i mentioned 'Syntax-Directed Translation', but what does Syntax-Directed Translation mean? In simplest terms, 'Syntax Directed Translation' means driving the entire compilation (translation) process with the syntax recognizer (the parser).

Conceptually, the process of compiling a program (translating it from source code to machine code) starts with a parser that produces a parse tree, and then transforms the parse tree through a sequence of tree or graph transformations, each of which is largely independent, resulting in a final simplified tree or graph which is traversed to produce machine code.

At first, a 'machine' can be consisted of a 'fsm_keyword', a 'enable_keyword', several 'state' (including 'init_state' and 'error_state') and 'ENDFSM'.

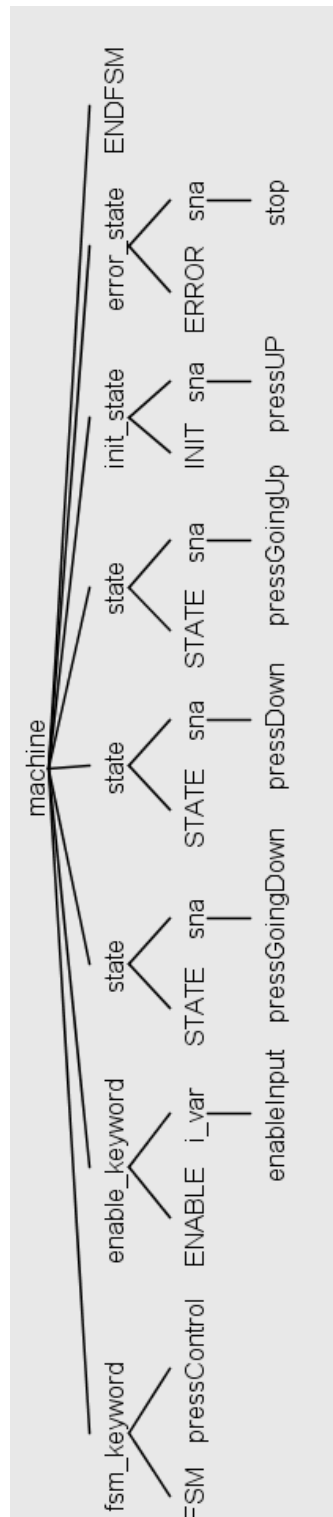


Figure 5.4: The parsing tree of the whole code

And each 'state' will be defined by commands (logcommand), actions (duringactions), tran-

sitions and warningtime, and the parsing trees of each state (including 'init_state' and 'error_state') are shown in Figure 5.5, Figure 5.6, Figure 5.7, Figure 5.8 and Figure 5.9:

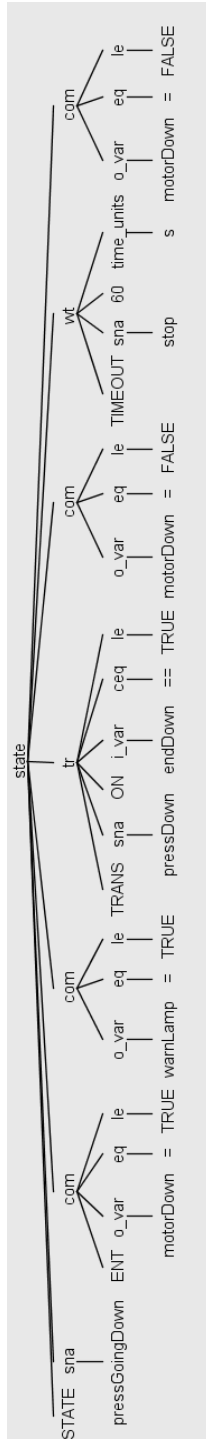


Figure 5.5: The parsing tree of state (pressGoingDown)

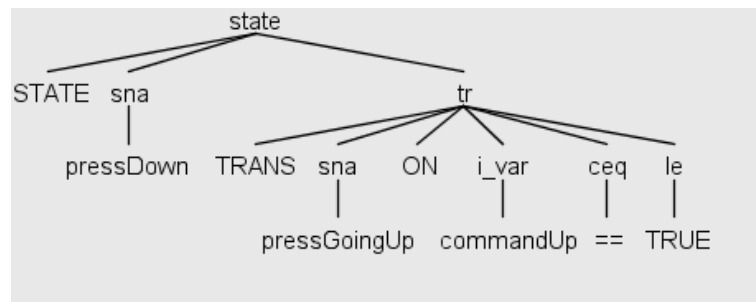


Figure 5.6: The parsing tree of state (pressDown)

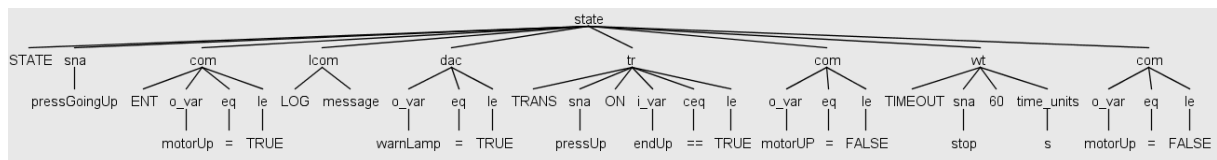


Figure 5.7: The parsing tree of state (pressGoingUp)

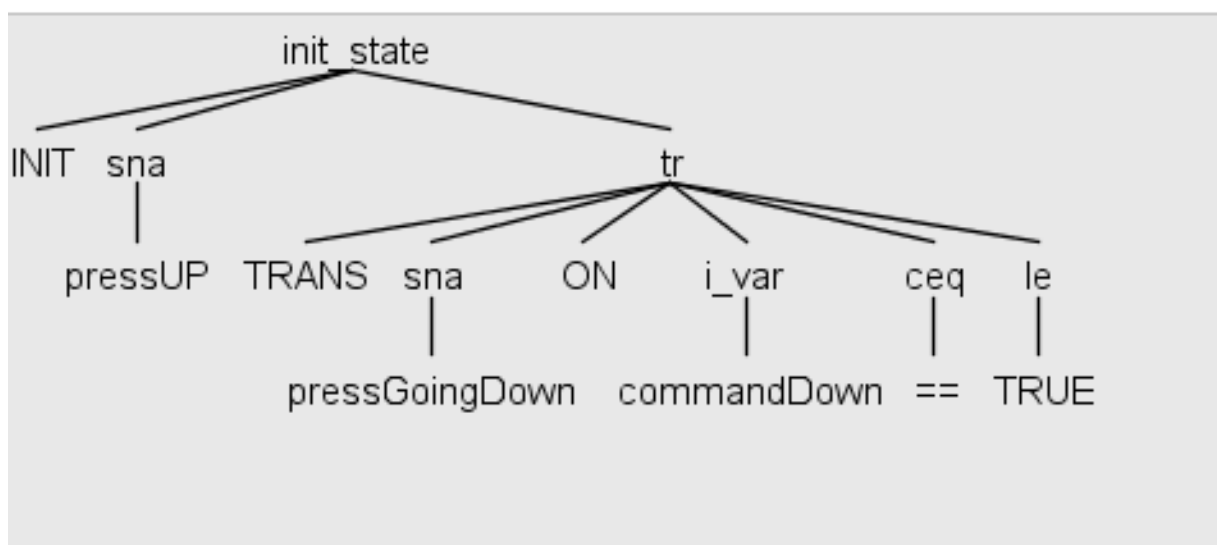


Figure 5.8: The parsing tree of init_state (pressUp)

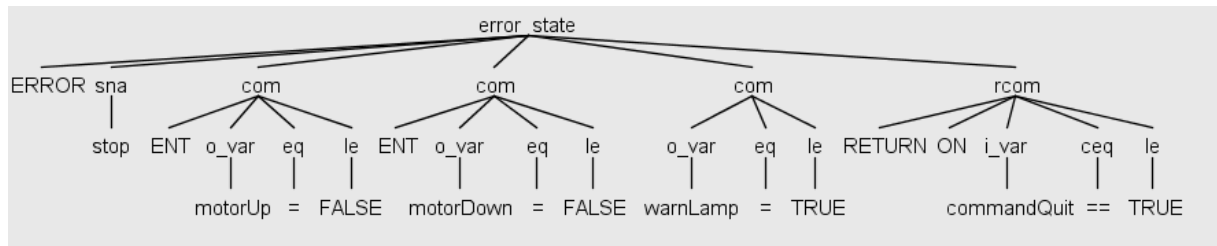


Figure 5.9: The parsing tree of error_state (stop)

Chapter 6

DSL Pyparsing with Python

In the previous chapter, we use ANTLR to compile a DSL example, and generate AST. Now we are using Python to achieve this simple DSL.

6.1 Introducing Python

Python is an easy to learn, powerful programming language, invented by Guido van Rossum in 1989, the first public release was released in 1991. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms [19].

Python language and its many extensions posed by the development environment is very suitable for engineering, researchers dealing with experimental data, charting, and even the development of scientific computing applications.

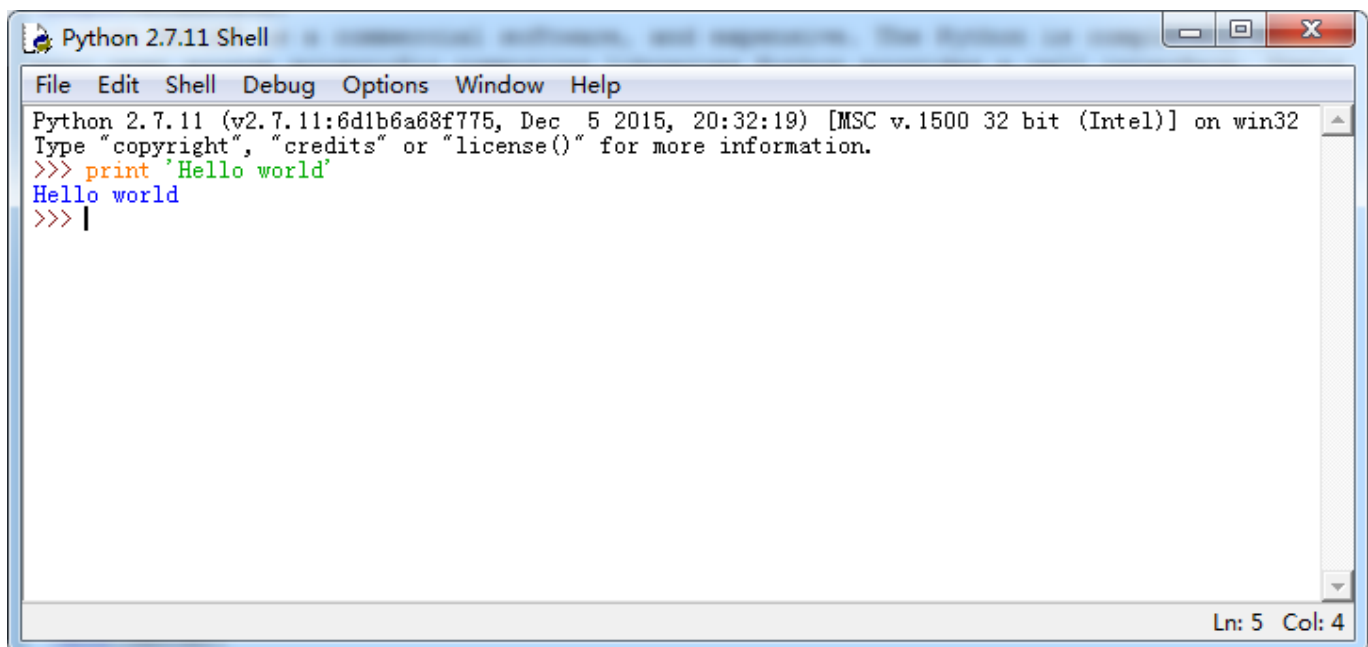
Mentioning of scientific computing, MATLAB may be the first mentioned herein. However, in addition to a number of highly specialized MATLAB toolbox can not be substituted outside, the most common functions of MATLAB can be found in the corresponding extensions Python world. Comparing with MATLAB, Python in scientific computing has the following advantages:

1. MATLAB is a commercial and expensive software. The Python is completely free, can provide many open sources in scientific computing libraries. Users are free to install Python and the vast majority of extensions on any computer.
2. Compared with MATLAB, Python is an easier to learn, more rigorous programming

language. It allows users to write more readable, maintainable code.

3. MATLAB primarily focuses on engineering and scientific computing. However, MATLAB is not only used in the field of computing, but also used in encounter file management, interface design, network communications and other needs. The Python has a lot of wealthy extensions, you can easily complete a variety of advanced tasks. Developers can implement a complete application by using required Python functions .

Now i will use a very simple program "Hello world" to learn how to write, save, and run Python programs. First of all, we have to choose a suitable editor, usually the editor which has syntax highlighting function is our first choice. Here i choose to use IDLE (GUI Python).



```
Python 2.7.11 Shell
File Edit Shell Debug Options Window Help
Python 2.7.11 (v2.7.11:6d1b6a68f775, Dec 5 2015, 20:32:19) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print 'Hello world'
Hello world
>>> |
Ln: 5 Col: 4
```

Figure 6.1: "Hello world" with Python (IDLE editor)

Python will immediately give us the output (Hello world) in the next line. We use 'print' to print our offers to its value.

6.2 Pyparsing

Pyparsing is a Python class library that helps you to quickly and easily create recursive-descent parsers. Pyparsing's class library provides a set of classes for building up a parser

from individual expression elements, up to complex, variable-syntax expressions. Expressions are combined using intuitive operators, such as `+` for sequentially adding one expression after another, and `|` and `^` for defining parsing alternatives (meaning "match first alternative" or "match longest alternative"). Replication of expressions is added using classes such as `OneOrMore`, `ZeroOrMore`, and `Optional` [13].

Basic Form of a pyparsing Program

The prototypical pyparsing program has the following structure [13]:

1. Import names from pyparsing module
2. Define grammar using pyparsing classes and helper methods
3. Use the grammar to parse the input text
4. Process the results from parsing the input text

6.2.1 Basic ParserElement, Expression

ParserElement:

1. Literal: Construct with a complete matching string; for example: `comma = literal(",")`
2. Word: one or more contiguous characters; construct with a string containing the set of allowed initial characters, and an optional second string of allowed body characters; if only one string given, it specifies that the same character set defined for the initial character is used for the word body; a `Word` may also be constructed with any of the following optional parameters [11]:
 - (a) indicating a minimum length of matching characters
 - (b) indicating a maximum length of matching characters
 - (c) indicating an exact length of matching characters
3. Suppress: match, but suppress matching results; useful for punctuation, delimiters [12].

Expression:

1. And: construct with a list of `ParserElements`, all of which must match for `And` to match; can also be created using the `'+'` operator

2. Or: construct with a list of ParserElements, all of which must match for Or to match; can also be created using the '^' operator
3. OneOrMore: one or more strings
4. ZeroOrMore: none or more strings

6.2.2 Define Grammar

Based on the previous section, the pyparsing is actually one module of python, and the edited pyparsing program (.py files) could be stored into the installation directory, after that the module will be imported by using the key words such as 'import', 'import as', 'from import' and so on.

In chapter 4, we have discussed the definition of BNF. Let's put this theory into practice, and write some basic parsers in Python, using pyparsing.

Pyparsing allows a pretty one-to-one mapping of BNF to Python code: we can define sets and combinations, then parse any text fragment corresponding to it. This is something very important to notice: one basic BNF definition can (and should) be reused: if we once wrote a BNF definition for an integer value, we can easily reuse this definition in, eg, a basic integer math expression.

Although pyparsing is defined according to the BNF grammar, we don't need to very strictly follow it. We distinguish them through a table:

BNF	Pyparsing	Meaning
	^	indicates choice between elements
*	ZeroOrMore	none or more
+	OneOrMore	one or more
::=	=	is defined as

Table 6.1: The difference of symbol definition between BNF and Pyparsing

Let's define a simple grammar of the example in chapter 5.1 (Grade=100;). In this simple sentence, 4 types elements could be found: identifiers (Grade), "=", integers (100) and ";". I used the pyparsing Word class to define a typical programming variable name consisting of a leading alphabetic character with a body of alphanumeric characters or underscores [13]:

```
identifier = p.Word(p.alphas, p.alphanums+'_')
```


I might also want to parse numeric constants, either integer or floating point. A simplistic definition uses another `Word` instance, defining our number as a "word" composed of numeric digits, possibly including a decimal point [13], and the punctuation marks will be defined by "Literal":

```
number = p.Word(nums+".")
eq = p.Literal("=")
semi = p.Literal(";")
```

So this simple sentence will be represented in Python:

```
assignmentExpr = identifier + eq + number + semi
```

Now let's run this program:

```
import pyparsing as p
# define grammar
identifier = p.Word(p.alphas, p.alphanums + '_')
number = p.Word(p.nums+".")
eq = p.Literal("=")
semi = p.Literal(";")
assignmentExpr = identifier + eq + number + semi
assignmentTokens = assignmentExpr.parseString("Grade=100;")

# parse input string
print assignmentTokens
```

Figure 6.2: The program of the example(Grade=100)

And the result is shown in Figure 6.3.

```
===== RESTART: C:\Python27\Lib\Grade.py =====
['Grade', '=', '100', ';']
>>>
```

Figure 6.3: The result of this example

6.3 Pyparsing of a DSL Example

In previous section, we discussed the definition and how to define pyparsing grammar. In this section, we will parse the program of the previous DSL:

```

ivar = p.Word (p.alphas, p.alphanums + '_').setParseAction(gotInput
)
ovar = p.Word (p.alphas, p.alphanums + '_')
num = p.Word (p.nums + '.' + 'TRUE' + 'FALSE')
eq = p.Literal('=')

com = (ovar + eq + num).setParseAction(gotOutput)
entryact = 'ENT' + com
duract = com

ceq = p.Literal('=='); cne = p.Literal('!=') ; cg = p.Literal('>');
cl = p.Literal('<'); cge = p.Literal('>='); cle = p.Literal
('<=')

condi = (ivar + (ceq ^ cne ^ cl ^ cg ^ cge ^ cle) +num)
log = ('LOG' + p.restOfLine).setParseAction(gotUserLog)

#condi = ivar + ceq + num
statename = p.Word (p.alphas, p.alphanums + '_')
targetname = p.Word (p.alphas, p.alphanums + '_')
fsname = p.Word (p.alphas, p.alphanums + '_')
enable = 'ENABLE' + ivar
progname = p.Word (p.alphas, p.alphanums + '_')
timechunk = p.Word (p.nums + '.') + p.Or ([p.Literal('s'),
p.Literal('ms'), p.Literal('h'), p.Literal('m'), p.Literal('d')])
time = p.OneOrMore(timechunk)
comlog = p.Or([com, log])
trans = 'TRANS' + targetname + 'ON' + condi + p.lineEnd + p.
ZeroOrMore(comlog)
timeout = ('TIMEOUT' + targetname + time + p.lineEnd + p.ZeroOrMore(
comlog)).setParseAction(gotTimer)

ret = 'RETURN ON' + condi
#statecore ::= (entryact* log* duringact*) & log*) trans*
#state ::= 'STATE' statename statecore

statecore = p.ZeroOrMore(entryact) + p.ZeroOrMore(log) + p.
ZeroOrMore(duract) + p.ZeroOrMore(trans) + p.ZeroOrMore(timeout)

```

```
state = 'STATE' + statename + statecore
initstate = 'INIT' + statename + statecore
errorstate = 'ERROR' + statename + p.ZeroOrMore(entryact) + p.
    ZeroOrMore(log) + p.ZeroOrMore(duract) + ret

fsm = 'FSM' + fsname + enable + p.Each([initstate, p.ZeroOrMore(p.
    Or([state, errorstate]) ) ]) + 'ENDFSM'
prog = p.OneOrMore(fsm) + p.StringEnd()

comment = '#' + p.restOfLine
code = p.ZeroOrMore(p.CharsNotIn('#\n'))
line = code + p.Optional(comment)

tokentext = prog.parseFile(progfile)
```

The complete Python code for parsing, interpreting and running this FSM can be found in Appendix B.

The pyparsing class diagram of this example is shown in Figure 6.4

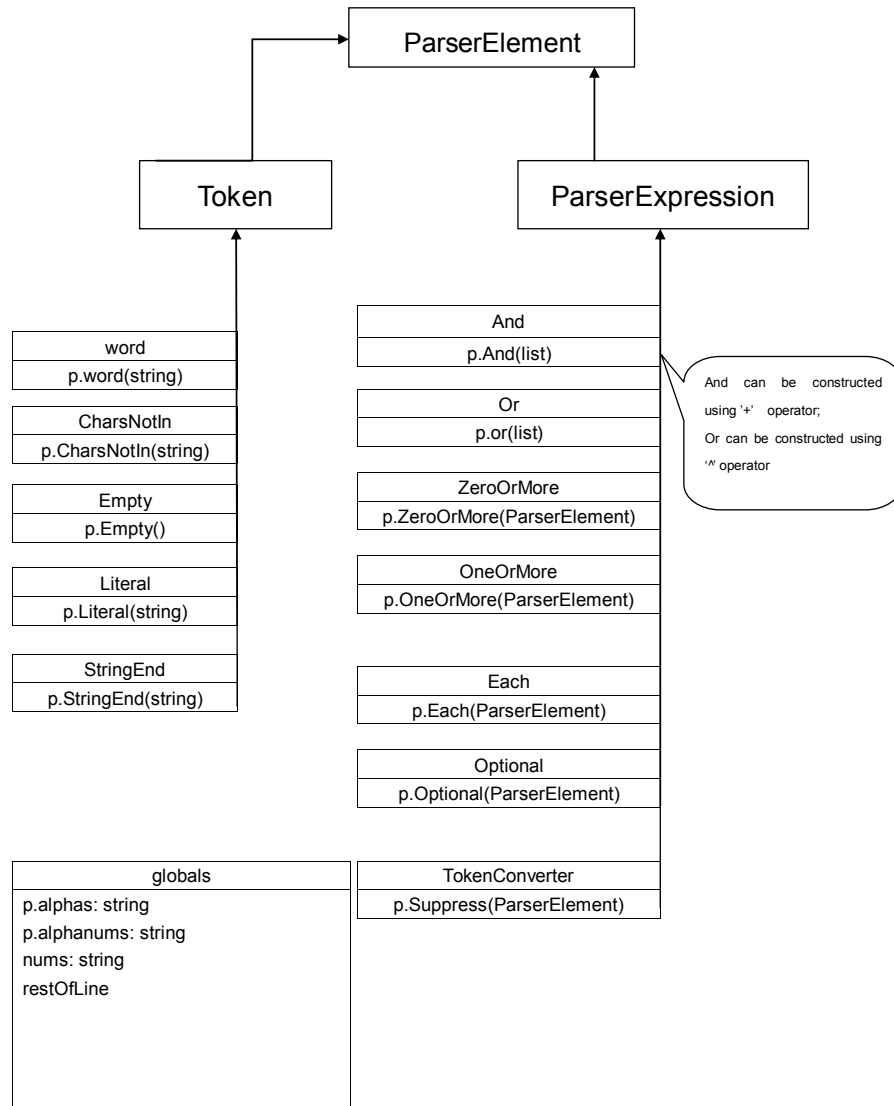


Figure 6.4: The pyparsing class diagram

6.4 DSL's Structured Analysis with Data-Flow-Diagram

Structured Analysis (SA) is a method of software engineering. Structured analysis and structured design are able to analyse requirements, convert it to specification files, and finally produce computer software, hardware configuration and related manuals.

Structured analysis and design techniques are the basis of systems analysis, evolved from the 1960s to 1970s systems analysis technology [24].

Data-Flow-Diagram (DFD), is a method of SA, which is used to represent the system logic model of a tool. It describe the flow of data in the system and processing process with graphical representation. A DFD is often used as a preliminary step to create an overview of the system [4]. DFDs can also be used for the visualization of data processing (structured design). With a data flow diagram, users are able to visualize how the system will operate, what the system will accomplish, and how the system will be implemented.

A few notations of DFD:

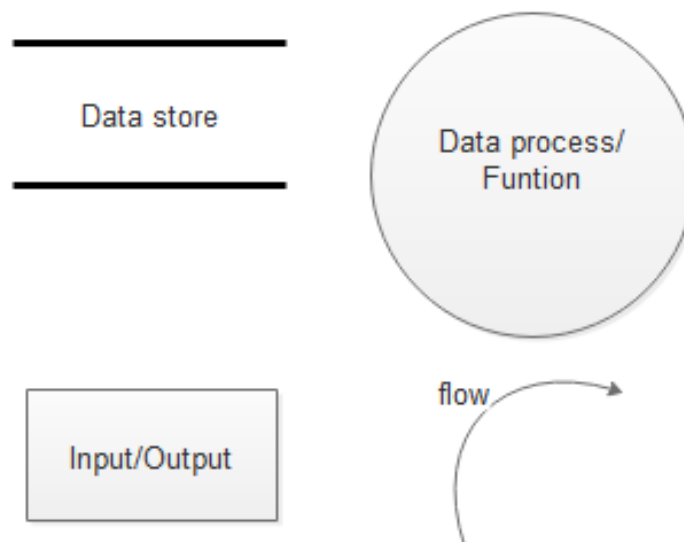


Figure 6.5: Notations of the DFD

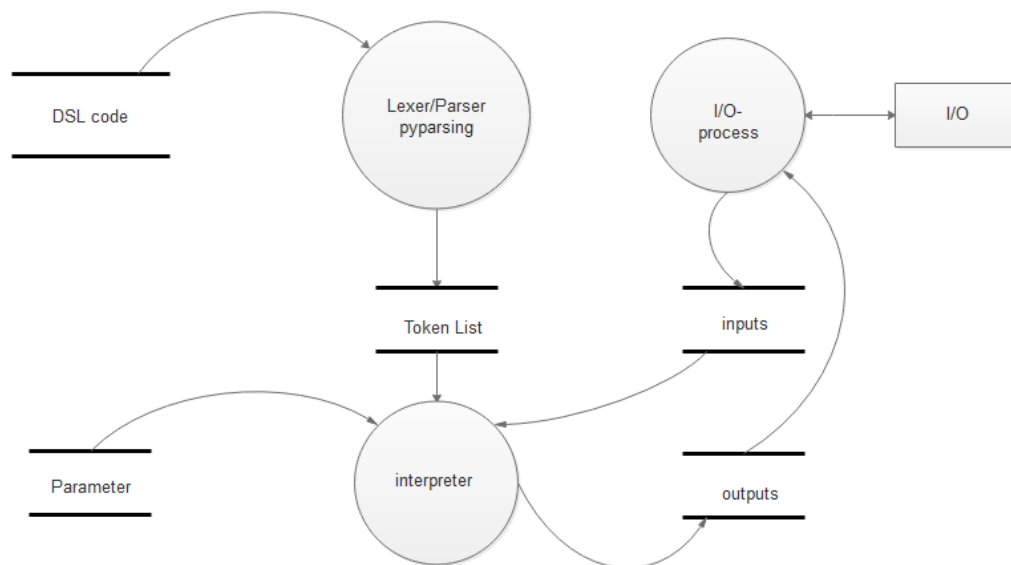


Figure 6.6: The data-flow-diagram of our DSL

DSL code is the database of the DFD. By using the Pyparsing compiler, we can send out the Tokenlist. Then, the Tokenlist combining with other parameters flow into the interpreter, the output data are able to be generated. Meanwhile, at the right side of the figure (shown above), through I/O-Processing function, the input data can be distinguished. After that, the input data, as a parameter, also flow into the interpreter. The output data return into the I/O-Processing function.

6.5 Example for Parsing, Interpreting and Running a FSM

In this section, we will parse, interpret and run a FSM through a verified example.

A barrier is to be controlled with a finite state machine. It has outputs to activate the motor upwards or downwards.

In addition, a warning lamp is to be activated during moving down. Limit switches for upper and lower position are used to indicate the position and to turn the drive off. Command inputs for motion up and down must be managed.

As a protection element, a timeout should stop the motion, if it cannot be accomplished in time. The advantage of the actual FSM implementation allows to continue after repairing and quitting the error condition without an additional code element. The barrier should close automatically after being open for a certain time. The code for the FSM is the following.

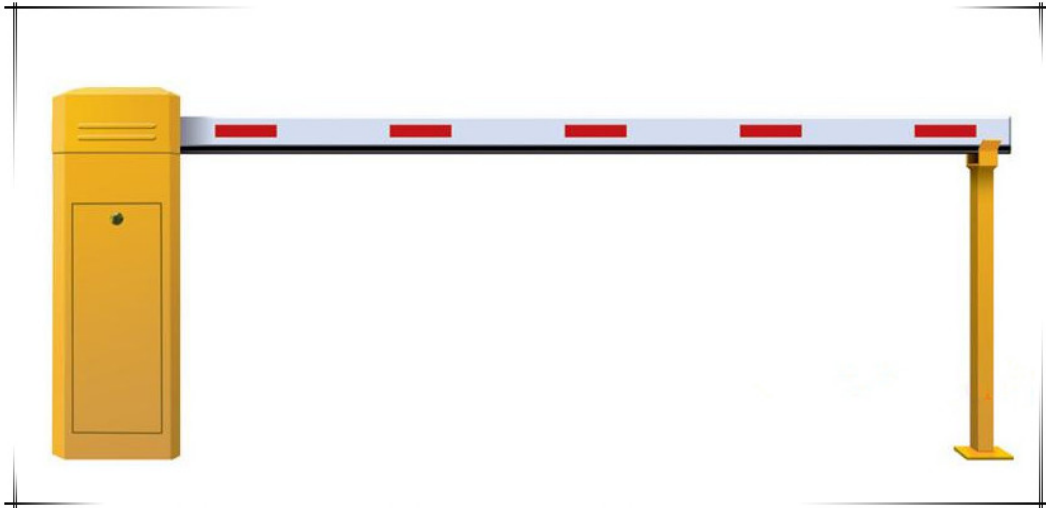


Figure 6.7: A parking barrier control system

```

FSM barrierControl
ENABLE enableInput
STATE goingDown
    ENT motorDown = TRUE
    LOG Going Down
    warnLamp = TRUE
    motorUp = FALSE
    TRANS down ON endDown == TRUE
        motorDown = FALSE
    TRANS goingUp ON commandUp == TRUE
    TIMEOUT stop 10s
        LOG Timout from downwards
INIT goingUp
    ENT motorUp = TRUE
    LOG Going Up
    warnLamp = FALSE
    motorDown = FALSE
    TRANS goingDown ON commandDown == TRUE
    TRANS up ON endUp == TRUE

```

```

    TIMEOUT stop 10s
    LOG Timeout from upwards
STATE down
    ENT motorDown = FALSE
    LOG Is down!
    warnLamp = FALSE
    TRANS goingUp ON commandUp == TRUE
STATE up
    ENT motorUp = FALSE
    ENT warnLamp = FALSE
    LOG Is up!
    TRANS goingDown ON commandDown == TRUE
    TIMEOUT goingDown 20s
        warnLamp = TRUE
        LOG Going down after open time
ERROR stop
    ENT motorUp = FALSE
    ENT motorDown = FALSE
    LOG Error stop
    warnLamp = TRUE
    RETURN ON commandQuit == TRUE
ENDFSM

```

The complete Python code for parsing, interpreting and running this FSM can be found in Appendix B.

The proper operation is verified in a simulation, where the inputs are controlled by changing the input file in parallel to running the FSM. The documentation in the following displays first the inputs to the FSM, then the LOG message to show the reaction and then the outputs of the FSM.

In the INIT state, this means, after powering up the controller, the barrier is going up, even without an active command input.

```

enableInput 1
endDown 0
commandDown 0
commandUp 0
endUp 0
commandQuit 0

```



```
----> Log: Going Up
```

```
motorDown 0  
warnLamp 0  
motorUp 1
```

If, after a timeout of ten seconds, the upper limit switch does not respond, obviously an error occurred.

The motor could be tripped, or the limit switch might be bad. The system goes to an error state.

```
enableInput 1  
endDown 0  
commandDown 0  
commandUp 1  
endUp 0  
commandQuit 0
```

```
----> Log: Timeout from upwards  
----> Log: Error stop
```

```
motorDown 0  
warnLamp 1  
motorUp 0
```

After repair and quitting the alarm, the controller resumes to go up again (still in the initial state without an active command).

```
enableInput 1  
endDown 0  
commandDown 0  
commandUp 0  
endUp 0  
commandQuit 1
```

```
----> Log: Going Up
```

```
motorDown 0  
warnLamp 0
```

```
motorUp 1
```

If we grant now the upper limit switch, the barrier finally is open and the motor stops.

```
enableInput 1
endDown 0
commandDown 0
commandUp 0
endUp 1
commandQuit 0
```

```
----> Log: Is up!
```

```
motorDown 0
warnLamp 0
motorUp 0
```

If the command input 'commandDown' is activated, the barrier goes to down-state again.

```
enableInput 1
endDown 0
commandDown 1
commandUp 0
endUp 0
commandQuit 0
```

```
----> Log: Going Down
```

```
motorDown 1
warnLamp 1
motorUp 0
```

When the lower limit switch is active, the barrier stops in down position.

```
enableInput 1
endDown 1
commandDown 1
commandUp 0
endUp 0
commandQuit 0
```

```
----> Log: Is down!
```

```
motorDown 0
warnLamp 0
motorUp 0
```

Now the input command 'commandUp' lets go up the barrier again.

```
enableInput 1
endDown 0
commandDown 0
commandUp 1
endUp 0
commandQuit 0
```

```
----> Log: Going Up
```

```
motorDown 0
warnLamp 0
motorUp 1
```

In normal operation, the upper limit switch is activated, and the barrier is open now.

```
enableInput 1
endDown 0
commandDown 0
commandUp 1
endUp 1
commandQuit 0
```

```
----> Log: Is up!
```

```
motorDown 0
warnLamp 0
motorUp 0
```

A timeout condition lets close the barrier automatically. In this case, the timeout is not an error condition!

```
enableInput 1
endDown 0
```

```
commandDown 0
commandUp 0
endUp 0
commandQuit 0
```

```
----> Log: Going down after open time
----> Log: Going Down
```

```
motorDown 1
warnLamp 1
motorUp 0
```

Now the barrier goes down until the closed position is reached.

```
enableInput 1
endDown 1
commandDown 0
commandUp 0
endUp 0
commandQuit 0
```

```
----> Log: Is down!
```

```
motorDown 0
warnLamp 0
motorUp 0
```

Chapter 7

Summary and Outlook

As our civilization develops, much of the industrial work had been replaced or simplified by computational devices, and automation had become of great import. Programming languages have been widely applied to many fields of interest as the basis of automation. However, programming languages have an inherent limitation, as people of non-programming professions have great difficulty in understanding them. In order to solve this problem, and to let management and experts to utilize the programming tools at their disposal, DSL has been designed. Through DSL the application of programming logic had for the first time approach a resemblance of natural languages; the maintenance of the systems had also been greatly simplified. It has not only helped the personnel to develop clear business protocols, and also give these protocols a more descriptive nature. Business personnel can become qualified DSL programmers after even simple training [22]. This essay has discussed several robotic languages, the basic application of BNF, the construction of a DSL program for FSM in an example, and the compilation of the said DSL using ANTLR4 and Pyparsing. We can surmise that DSL has the following advantages in comparison to traditional computer languages:

1. DSL is easier to understand and read.
2. It is highly expressive in special circumstances.
3. It improves development efficiency.
4. When encountering different problems in the same field, it is easier for modular correction.
5. Different traditional languages could be translated through a DSL translator.

Bibliography

- [1] RAIL Programming Language. <http://www.roboticsbible.com/rail-robot-programming-language.html>. accessed Apr. 2, 2016, 2012.
- [2] V. Aho, S. Lam, R. Sethi, and D. Ullman. *Compilers: Principles, Techniques*. Pearson Education, New York, second edition, 2008.
- [3] L. K. Barry and S. Kenneth. *Formal Syntax and Semantics of Programming Languages*. Addison-Wesley Publishing Company, Inc., Bonn, 1995.
- [4] P. D. Bruza and Th.P. van der Weide. The semantics of data flow diagrams. In *In Proceedings of the International Conference on Management of Data*, pages 66–78. McGraw-Hill Publishing Company, 1993.
- [5] G. Debasish. *DSLs in action*. Manning Publications Co., New York, 2011.
- [6] L. Gansheng and W. Huamin. *Compiler theory and technology*. Tsinghua University Publications, Beijing, 1997.
- [7] B. Grady, R. James, and J. Ivar. *Unified Modeling Language User Guide*. Addison-Wesley Professional, Indianapolis, second edition, May 2005.
- [8] A. Kent and J.G. Williams. *Encyclopedia of Computer Science and Technology: Volume 21 - Supplement 6: ADA and Distributed Systems to Visual Languages*. Encyclopedia of Computer Science and Technology. Taylor & Francis, London, 1989.
- [9] L. Lan. Comparison between mealy circuits and moore circuits. *Measurement and Control Technique*, 2006.
- [10] F. Martin. *Domain-Specific Languages*. Addison-Wesley Professional, Indianapolis, September 2010.
- [11] P. McGuire. Using the pyparsing module. <http://www.ptmcg.com/geo/python/how-touseyparsing.html>. accessed Sep. 20, 2016, 2004.

- [12] P. McGuire. Introduction to pyparsing: An object-oriented easy-to-use toolkit for building recursive descent parsers. 2006.
- [13] P. McGuire. *Getting Started with Pyparsing*. O'Reilly shortcuts. O'Reilly Media, Boston, 2007.
- [14] F.P. Miller, A.F. Vandome, and M.B. John. *Abstract Syntax Tree*. VDM Publishing, Saarbrücken, 2010.
- [15] T. Parr. *Language Implementation Patterns*. The Pragmatic Bookshelf, North Carolina, 2010.
- [16] T. Parr. *The Definitive ANTLR 4 Reference*. O'Reilly and Associate Series. Pragmatic Bookshelf, Frisco, TX, 2012.
- [17] K. Rathmill, P. MacConaill, S. O'Leary, and J. Browne. *Robot Technology and Applications: Proceedings of the 1st Robotics Europe Conference Brussels*. Springer Berlin Heidelberg, Berlin, 2013.
- [18] W. S. Robert. *Concepts of Programming Languages*. Pearson, New York, tenth edition, January 2012.
- [19] C. H. Swaroop. *A Byte of Python*. CreateSpace Independent Publishing Platform, South Carolina, 2015.
- [20] P. Vreda and eds Paul E. B. Dictionary of algorithms and data structures. September 2014.
- [21] D. Wampler and A. Payne. *Programming Scala: Scalability = Functional Programming + Objects*. O'Reilly Media, Boston, 2014.
- [22] D. Wei. Research on domain specific language based on drools. 2011.
- [23] W. Xianguo. *Unifide Modeling Language Guide*. Tsinghua university Publications, Beijing, 2009.
- [24] E. Yourdon. *Managing the structured techniques: strategies for software development in the 1990's*. Yourdon Press, New York, 1986.

List of Figures

1.1	An informal micro-classification of common patterns and techniques of implementing external DSLs [5]	8
1.2	An informal micro-classification of patterns used in implementing internal DSLs [5]	9
2.1	Unimate 500 PUMA (1983), control unit and computer terminal at Deutsches Museum, Munich	14
2.2	Victor Scheinman's stanford arm in 1969	16
2.3	The olivetti "SIGMA" a cartesiancoordinate robot, is one of the first used in assembly applications in 1969	17
2.4	The Mitsubishi Electric "MOVEMASTER RV-M2" industrial micro-robot	18
2.5	Pick-and-place work	22
3.1	A Moore model for a generalised non-clocked sequential machine	26
3.2	A mealy model for a generalised non-clocked sequential machine	27
3.3	A simple state machine model of a elevator	29
3.4	UML state diagram	30
3.5	a state in state diagram	31
3.6	Initial state and final State in state diagram	31
3.7	Transition in state diagram	31
3.8	Decision in state diagram	32
3.9	Fork and Join in state diagram	32
3.10	Add a Chart object into new model	33

3.11	Define inputs and outputs	33
3.12	Define a state	34
3.13	Define transition	34
3.14	The previous example of elevator with Matlab/Simulink	35
5.1	Language recognizer	40
5.2	The parsing result	45
5.3	Tree construction	46
5.4	The parsing tree of the whole code	48
5.5	The parsing tree of state (pressGoingDown)	49
5.6	The parsing tree of state (pressDown)	50
5.7	The parsing tree of state (pressGoingUp)	50
5.8	The parsing tree of init_state (pressUp)	50
5.9	The parsing tree of error_state (stop)	51
6.1	”Hello world” with Python (IDLE editor)	53
6.2	The program of the example(Grade=100)	56
6.3	The result of this example	56
6.4	The pyparsing class diagram	59
6.5	Notations of the DFD	60
6.6	The data-flow-diagram of our DSL	61
6.7	A parking barrier control system	62

List of Tables

1.1	Comparison of compile-time and runtime metaprogramming [5]	10
1.2	The meaning of this statement	11
3.1	state table of the elevator	28
4.1	BNF symbols	37
6.1	The difference of symbol definition between BNF and Pyparsing	55

Listings

Appendix A

Grammar of DSL Compiling with ANTLR4

```
grammar FSMTEST;

machine: fsm_keyword enable_keyword state* init_state error_state '
    ENDFSM' ;
fsm_keyword : 'FSM' ID;
enable_keyword : 'ENABLE' i_var;
i_var : ID;//input_variante:'enableInput' | 'commandDown' | '
    commandUp' | 'commandQuit' ;
o_var :ID;//output_variante: 'motorDown' | 'motorUp' | 'motorDn' | '
    warnLamp' ;
sna: ID;//state_name
eq: '=' ;
ceq: '==';

state: 'STATE' sna
    com*
    dac*
    lcom*
    dac*
    tr*
    com*
```

```
wt*
com*;

init_state : 'INIT' sna
            tr*;
error_state : 'ERROR' sna
            com*
            dac*
            rcom*;
com : 'ENT'? o_var eq le
     //command:| output_variante '=' logic_element;
dac : o_var eq le; //duringaction
tr : 'TRANS' sna 'ON' i_var ceq le ; //transition
lcom : 'LOG' ID ; //logcommand
wt : 'TIMEOUT' sna INT time_units ; //waringtime
time_units : 'ms' | 's' | 'min' | 'h' | 'd';
le : 'TRUE' | 'FALSE'; //logic_element
rcom : 'RETURN' 'ON' i_var ceq le; //returncommand

ID : [a-zA-Z]+ ; // match lower-case identifiers
INT : [0-9]+ ;
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

Appendix B

DSL Compiling with Pyparsing

```
import pyparsing as p
import sys
import os
import time as tt

#===== check cmd line arguments
=====
if sys.argv.__len__() != 2:
    print ('Needs one argument: prog name!')
    sys.exit ()

file = p.Word(p.alphas, p.alphanums + '_')
progfile = file.parseString(sys.argv[1])[0]+'.prg'
parfile = file.parseString(sys.argv[1])[0]+'.par'
infile = file.parseString(sys.argv[1])[0]+'.in'
outfile = file.parseString(sys.argv[1])[0]+'.out'
print (progfile, parfile)

#===== check files =====
if os.path.isfile(progfile) == False:
```

```

#filePrg = open (progfile, 'r')
print ('Prog file not found')
sys.exit()
try:
    filePar = open (parfile, 'r')
except:
    print ('Paramter file not found')
    sys.exit()
try:
    fileIn = open(infile, 'r')
except:
    print ('Input file not found')
    #sys.exit
try:
    fileOut = open(outfile, 'r')
except:
    print ('Output file not found')
    #sys.exit

#===== check inputs and outputs
=====
innames = p.OneOrMore(p.Word (p.alphas, p.alphanums + '_' ) + p.
    Suppress((p.Literal('1') ^ p.Literal('0'))))
varvalues = p.OneOrMore(p.Suppress (p.Word(p.alphas, p.alphanums +
    '_')) + (p.Literal('1') ^ p.Literal('0')))
inputs = fileIn.read()
fileIn.close()
ins = innames.parseString(inputs)
inv = varvalues.parseString(inputs)
print (ins)
print ()
inset = set(ins)
if inset.__len__() != ins.__len__():
    print ('Multiple inputs defined!')
    sys.exit()

outnames = p.OneOrMore(p.Word (p.alphas, p.alphanums + '_' ) + p.
    Suppress((p.Literal('1') ^ p.Literal('0'))))
outputs = fileOut.read()

```

```

fileOut.close()
outs = outnames.parseString(outputs)
outv = varvalues.parseString(outputs)
print (outs)
print ()
outset = set(outs)
if outset.__len__() != outs.__len__():
    print ('Multiple outputs defined!')
    sys.exit()

#=====
def gotInput (t):
    print ('einput: ', t[0])
    if t[0] not in inset:
        print ('Undefined Input!')
        sys.exit()
    gotInput.inputs.append(t[0])
def gotOutput (t):
    print ('ausput: ', t[0])
    if t[0] not in outset:
        print ('Undefined Output!')
        sys.exit()
    gotOutput.outputs.append(t[0])
def gotUserLog (t):
    pass
    #print ('Logentry ', t[1])
def gotTimer(t):
    print ('Timer ', t)

gotInput.inputs = []
gotOutput.outputs = []

#===== Parse the program =====
ivar = p.Word (p.alphas, p.alphanums + '_').setParseAction(gotInput
)
ovar = p.Word (p.alphas, p.alphanums + '_')
num = p.Word (p.nums + '.' + 'TRUE' + 'FALSE')
eq = p.Literal('=')

```



```

com = (ovar + eq + num).setParseAction(gotOutput)
entryact = 'ENT' + com
duract = com

ceq = p.Literal('=='); cne = p.Literal('!=') ; cg = p.Literal('>');
    cl = p.Literal('<'); cge = p.Literal('>='); cle = p.Literal
    ('<=')

condi = (ivar + (ceq ^ cne ^ cl ^ cg ^ cge ^ cle) + num)
log = ('LOG' + p.restOfLine).setParseAction(gotUserLog)

#condi = ivar + ceq + num
statename = p.Word (p.alphas, p.alphanums + '_' )
targetname = p.Word (p.alphas, p.alphanums + '_' )
fsname = p.Word (p.alphas, p.alphanums + '_' )
enable = 'ENABLE' + ivar
progname = p.Word (p.alphas, p.alphanums + '_' )
timechunk = p.Word (p.nums + '.' ) + p.Or ([p.Literal('s'),
    p.Literal('ms'), p.Literal('h'), p.Literal('m'), p.Literal('d')])
time = p.OneOrMore(timechunk)
comlog = p.Or([com, log])
trans = 'TRANS' + targetname + 'ON' + condi + p.lineEnd + p.
    ZeroOrMore(comlog)
timeout = ('TIMEOUT' + targetname + time + p.lineEnd + p.ZeroOrMore(
    comlog)).setParseAction(gotTimer)

ret = 'RETURN ON' + condi
#statecore ::= (entryact* log* duringact*) & log* trans*
#state ::= 'STATE' statename statecore

statecore = p.ZeroOrMore(entryact) + p.ZeroOrMore(log) + p.
    ZeroOrMore(duract) + p.ZeroOrMore(trans) + p.ZeroOrMore(timeout)
state = 'STATE' + statename + statecore
initstate = 'INIT' + statename + statecore
errorstate = 'ERROR' + statename + p.ZeroOrMore(entryact) + p.
    ZeroOrMore(log) + p.ZeroOrMore(duract) + ret

```

```

fsm = 'FSM' + fsname + enable + p.Each([initstate, p.ZeroOrMore(p.
    Or([state, errorstate]) ) ]) + 'ENDFSM'
prog = p.OneOrMore(fsm) + p.StringEnd()

comment = '#' + p.restOfLine
code = p.ZeroOrMore(p.CharsNotIn('#\n'))
line = code + p.Optional(comment)

tokentext = prog.parseFile(progfile)

def endOfState (n):
    if (n == 'STATE') | (n == 'INIT') | (n == 'ERROR') | (n == 'ENDFSM') :
        return True
    else:
        return False

def doCommand(n):
    if tokentext[n + 2] == 'TRUE':
        outv[outs[:].index(tokentext[n])] = 1
    else:
        outv[outs[:].index(tokentext[n])] = 0

def checkCondition(n):
    inp = inv[ins[:].index(tokentext[n])]
    if inp == '1':
        return True
    else:
        return False

def doLog (n):
    print ('----> Log: ', tokentext[i+1])

def findState(fsm, n):
    for l in range(len(tokentext[fsm:])):

```

```

        if ((tokentext[l] == 'STATE')|(tokentext[l] == 'INIT')|(
            tokentext[l] == 'ERROR'))&( tokentext[l+1]==n) :
            return l+1

#find all FSMs:
fsmIndices = [i for i,x in enumerate(tokentext) if x=='FSM']

#----- init variables for FSMs
fsmvars={}
iActualState = 0; iEndOfState = 1; iFirstEntry = 2; iSkipEntry = 3;
    iEntryTime = 4; iRecentState = 5;

#iterate over FSMs, find INIT states
for fsm in fsmIndices:
    firststate = tokentext[fsm:].index('INIT')
    fsmvars[fsm.__str__()] = [firststate+fsm+1,0,0,0,0,0]

#----- endless real-time loop:
while True:
    fileIn = open(infile, 'r');
    inputs = fileIn.read()
    fileIn.close()
    ins = innames.parseString(inputs)
    inv = varvalues.parseString(inputs)
    for i in range(len(ins)):
        pass #print (ins[i], inv[i])
    #----- iterate over FSMs:
    for fsm in fsmIndices:
        tt.sleep (1)
        actualState = fsmvars[fsm.__str__()][iActualState]
        i = actualState +1
        i_enable = tokentext[fsm:].index('ENABLE')+fsm+1
        if inv[ins[:].index(tokentext[i_enable])]=='1':
            #first run in state:
            if fsmvars[fsm.__str__()][iFirstEntry] == 0:
                fsmvars[fsm.__str__()][iFirstEntry] = 1
                fsmvars[fsm.__str__()][iEntryTime] = tt.time()

```

```

i = 0
for n in tokentext[actualState:]: #firststate+fsm+1:]:
    i += 1
    if endOfState(n) == True:
        break
fsmvars[fsm.__str__()][iEndOfState]=actualState+i
#process entry commands and logs:
i = actualState +1
while tokentext[i] == 'ENT':
    doCommand(i+1); i += 4
while tokentext[i] == 'LOG':
    doLog(i); i += 2
fsmvars[fsm.__str__()][iSkipEntry] = i
else: #not first run, remember offset to cyclic commands
    i = fsmvars[fsm.__str__()][iSkipEntry]
#----- do cyclic commands:
while not ((tokentext[i] == 'TRANS') | (tokentext[i] == 'TIMEOUT
    ') | (tokentext[i] == 'RETURN ON')):
    doCommand(i); i += 3
#----- do transitions:
leave = False
oldState = fsmvars[fsm.__str__()][iActualState]
while (endOfState(tokentext[i]) == False): #and not (tokentext[i
    ] == '\n'):
    if tokentext[i] == 'TRANS':
        if leave == True:
            break
        if checkCondition (i+3) == True:
            fsmvars[fsm.__str__()][iActualState] = findState(fsm,
                tokentext[i+1])
            fsmvars[fsm.__str__()][iFirstEntry] = 0
            leave = True
        i += 7
    if tokentext[i] == 'TIMEOUT':
        if leave == True:
            break
        if fsmvars[fsm.__str__()][iEntryTime] < (tt.time() - float(
            tokentext[i+2]))):

```

```

        fsmvars[fsm.__str__()][iActualState] = findState(fsm,
            tokentext[i+1])
        fsmvars[fsm.__str__()][iFirstEntry] = 0
        leave = True
    i += 5
if tokentext[i] == 'RETURN ON':
    if leave == True:
        break
    if checkCondition (i+1) == True:
        fsmvars[fsm.__str__()][iActualState] = fsmvars[fsm.__str__
            ()][iRecentState]
        fsmvars[fsm.__str__()][iFirstEntry] = 0
        leave = True
    i += 4
if (tokentext[i] == 'LOG'):
    if leave == True:
        doLog(i)
        i += 2
if (tokentext[i] in outset):
    if leave == True:
        doCommand(i)
        i += 3
if leave == True:
    fsmvars[fsm.__str__()][iRecentState] = oldState
tt.sleep(1)
#end fsm loop

fileOut = open(outfile, 'w')
for i in range(len(outs)):
    #print (outs[i], outv[i])
    fileOut.write(outs[i]+ ' ' + str(outv[i]) + '\n')
fileOut.close()

```