Developing an Interpreter for the Expert Systems Language, Flex

05017742 - Thomas Cowell

05017742@glam.ac.uk

September 4, 2010

Abstract

The aim of this project is to create a simple interpreter for the language Flex. Flex is an expert systems language that focuses on using plain English to make it accessible for non-technical users.

Contents

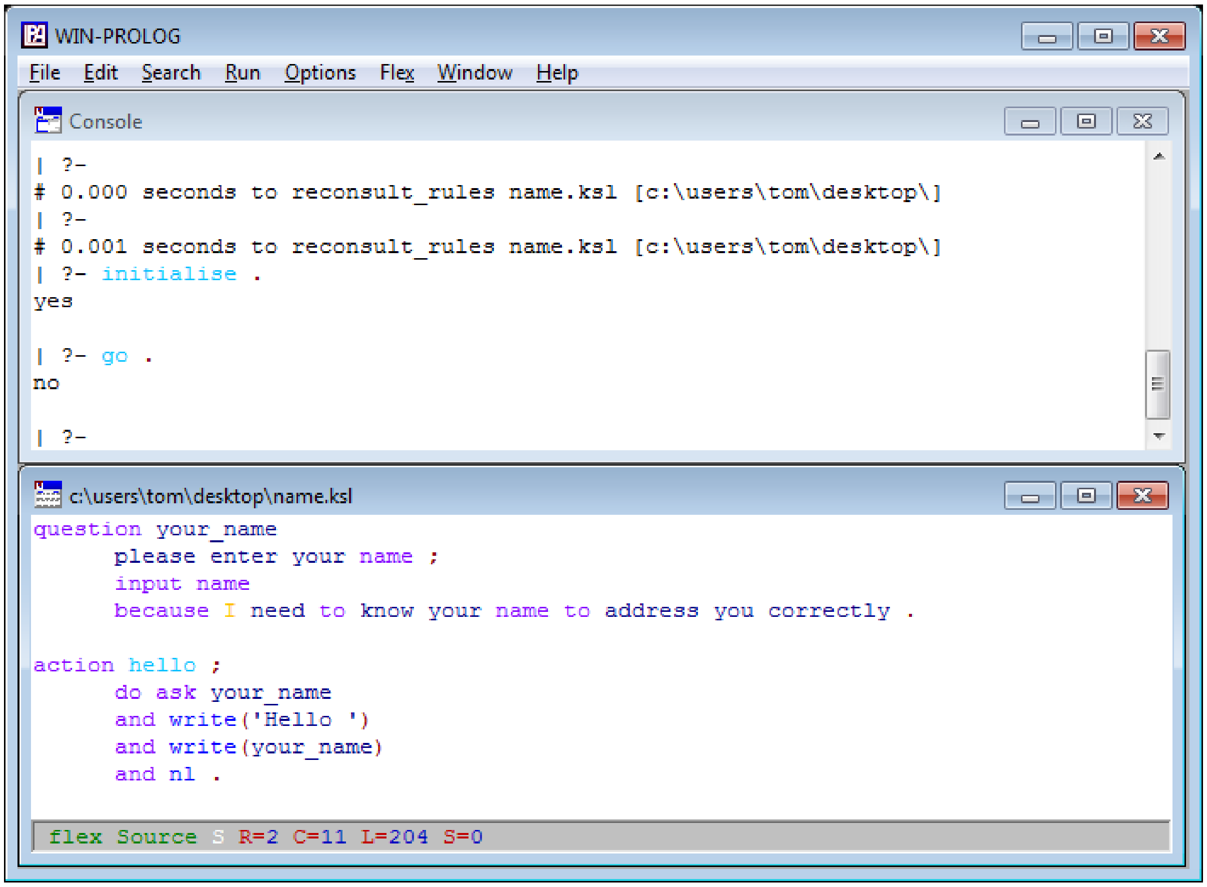
Chapter 1

Introduction

1.1 Why This Project?

Before this paper begins, it is worth noting the motivation behind this project. Certain modules on the 'Intelligent Computer Systems' course required the use of some propietary software, WinProlog. The application can do a wealth of tasks, however, the modules only focused on a niche feature of the software, flex. Flex allows professionals, engineers and students to write a rule-based system quickly using a language that is very close to the English language.

Whilst flex achieved results, it was felt that the debugging features were sub-par and were often inaccurate, whilst giving no helpful messages for a beginner to use to correct their mistakes. Also, because of the restrictions of having a licence, it meant that the software could only be installed on certain machines, in a different department within the University campus.


WIN-PROLOG
File Edit Search Run Options Flex Window Help
Console
c:\users\tom\desktop\name.ksl

```
| ?-
# 0.000 seconds to reconsult_rules name.ksl [c:\users\tom\desktop\]
| ?-
# 0.001 seconds to reconsult_rules name.ksl [c:\users\tom\desktop\]
| ?- initialise .
yes

| ?- go .
no

| ?-
```

```
question your_name
      please enter your name ;
      input name
      because I need to know your name to address you correctly .

action hello ;
      do ask your_name
      and write('Hello ')
      and write(your_name)
      and nl .
```

```
flex Source S R=2 C=11 L=204 S=0
```

Figure 1.1: A simple flex program that is syntatically wrong, plus the resulting error.

In figure 1.1, the line `input name` should have a semi-colon at the end, like `input name ;`, to let the compiler know that the line is finsihed, but the question isn't over. Whilst this is a simple mistake, the flex compiler hasn't picked it up, and is only being discovered when trying to run the action `go`, where flex returns an error message `no`, with no indiciation what the error is, where it is or how to fix it. If this were a large set of rules, finding such a basic mistake could take a long time to fix.

The examples below show flex encountering a gramatically incorrect script, with the compiler displaying errors. What we see here is some random text talking about nothing much at all!

Whilst the compiler in WinProlog does try to track down the error, it's not very friendly to new users, as it doesn't let them know where the error is, or what is wrong.

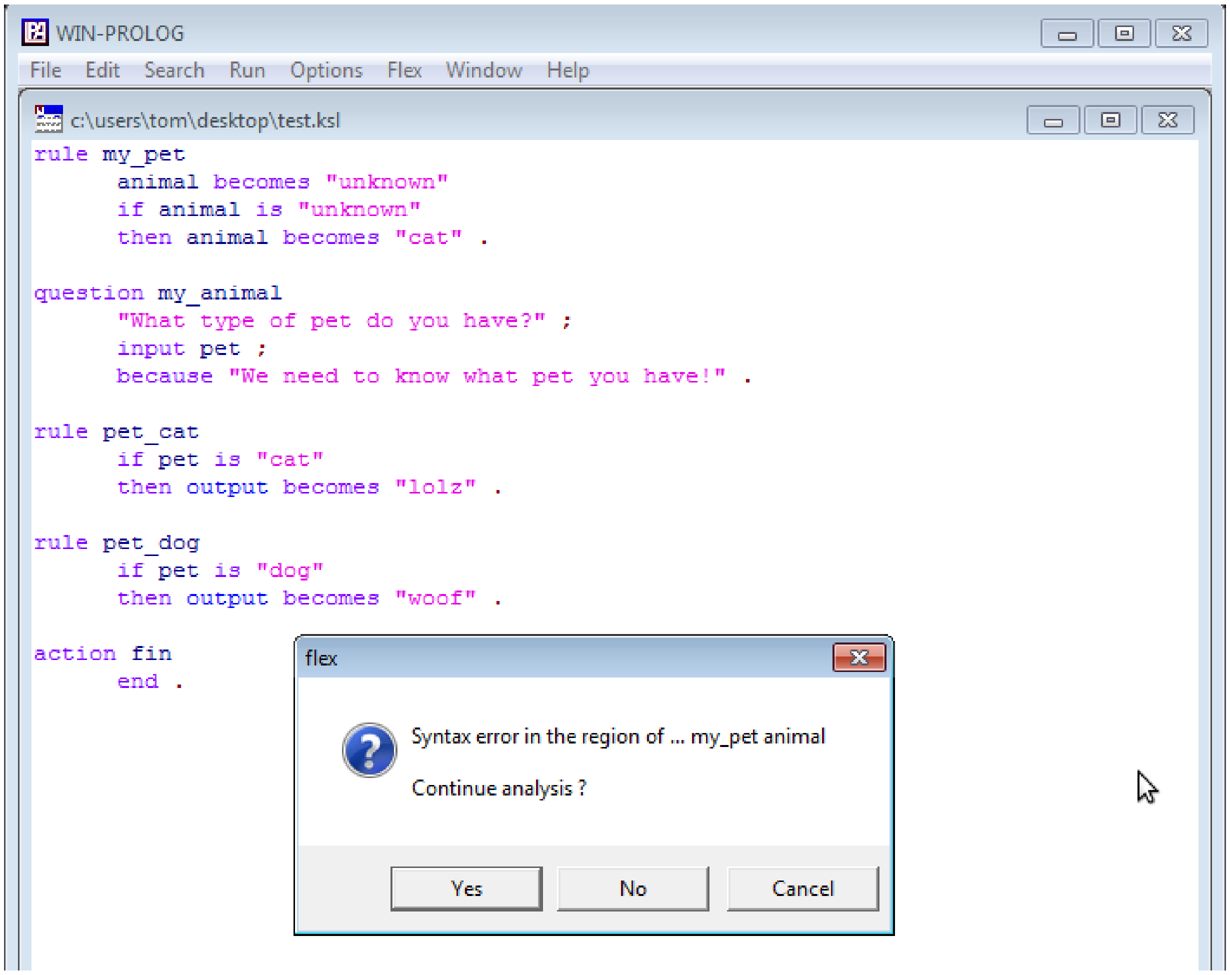
1.2 Aim

This project aims to create a basic implementation of the flex toolkit, using open source software. The software would be able to be installed on any linux machine, whilst providing the basic features required to teach students how to write expert systems.

1.3 Objectives

The objectives of this project are to:

- Create a simple flex interpreter that can work on the simple tutorials from a tutorial.
  - Recognise rules, questions and actions;
  - Recognise variables, assignments and comparisons;
  - Work with simple if-statements
  - Output variables and text using the write() statement.
- Create it for the Linux platform.
  - Command line driven - feed the application a flex source file.
  - Not be coupled to a Desktop Environment (Gnome, KDE etc).
- Keep the project open source.
  - Keep the project available on a source-control service so it can be ammended to once the project is finished.

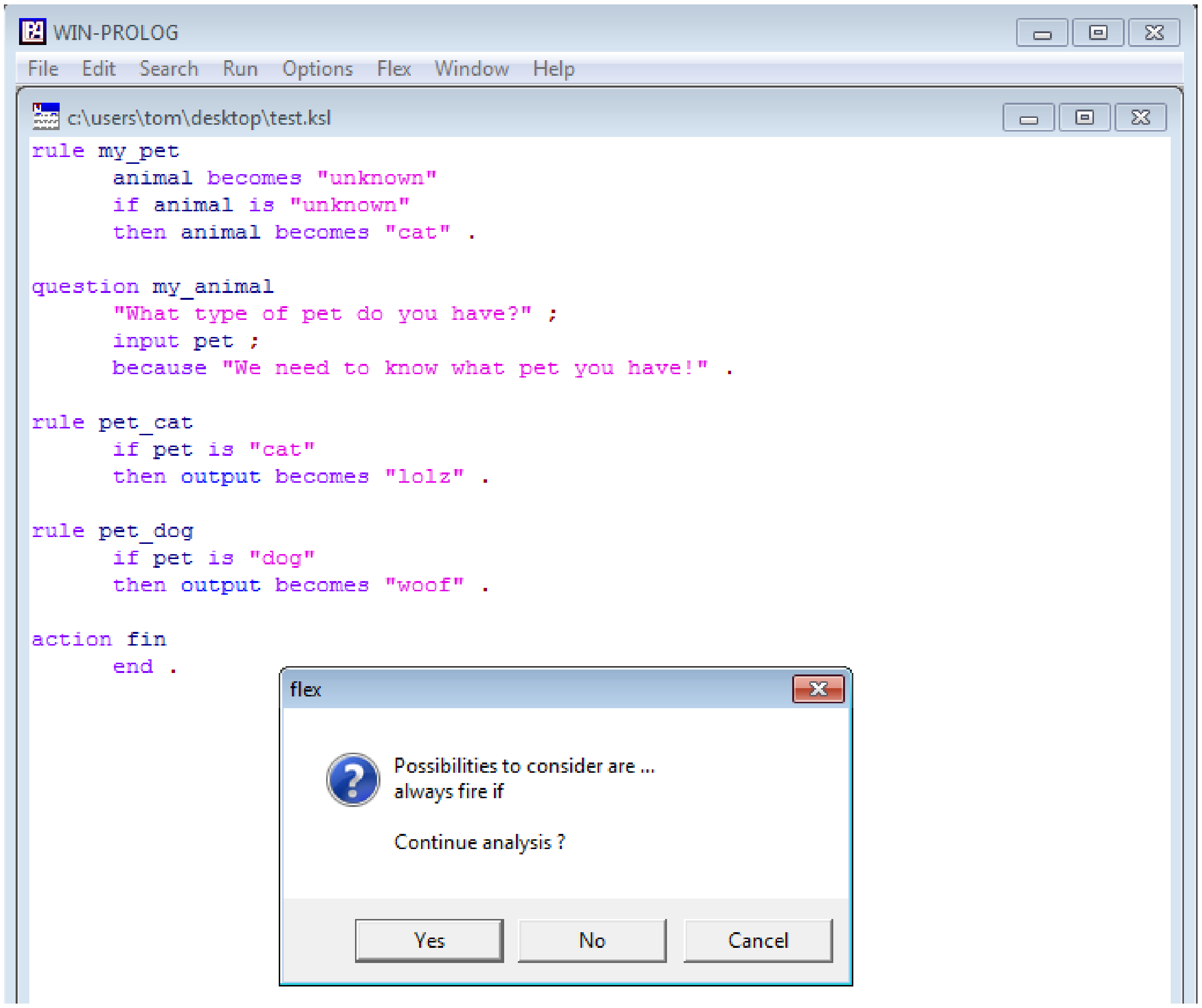

WIN-PROLOG
File Edit Search Run Options Flex Window Help
c:\users\tom\desktop\test.ksl

```
rule my_pet
      animal becomes "unknown"
      if animal is "unknown"
      then animal becomes "cat" .

question my_animal
      "What type of pet do you have?" ;
      input pet ;
      because "We need to know what pet you have!" .

rule pet_cat
      if pet is "cat"
      then output becomes "lolz" .

rule pet_dog
      if pet is "dog"
      then output becomes "woof" .

action fin
      end .
```

Syntax error in the region of ... my_pet animal

Continue analysis ?

Figure 1.2: The compiler has detected an error in the grammar.


WIN-PROLOG
File Edit Search Run Options Flex Window Help
c:\users\tom\desktop\test.ksl
flex
Yes
No
Cancel

```
rule my_pet
      animal becomes "unknown"
      if animal is "unknown"
      then animal becomes "cat" .

question my_animal
      "What type of pet do you have?" ;
      input pet ;
      because "We need to know what pet you have!" .

rule pet_cat
      if pet is "cat"
      then output becomes "lolz" .

rule pet_dog
      if pet is "dog"
      then output becomes "woof" .

action fin
      end .
```

```
Possibilities to consider are ...
always fire if

Continue analysis ?
```

Figure 1.3: The compiler trying to detect where the error is.

Chapter 2

Literature Review

2.1 Outline

As described in the previous chapter, the outline of this project is to create a compiler for the expert systems language, flex. Whilst Flex is based on the Artificial Intelligence langauge Prolog, this basic compiler will be written in C, emulating the basic features of flex, mainly as a teaching tool, rather than a fully fledged toolkit. However, because Flex is built on top of Prolog, there are many papers on writing Prolog compilers

2.2 Previous Work

Because the Flex toolkit is unique, there aren't many examples or previous work done on creating a compiler soley for Flex. However, there are countless items of literature dedicated to writing compilers, all for different languages, but constructed in the same manor - using Flex and Bison.

Chapter 3

Research

3.1 Research

Before any implementation can begin, research needs to be conducted in order to gain better knowledge about expert systems, the Flex language and various toolkits for developing interpreters.

3.2 Expert Systems

In the modern world where information is highly valuable and where time is of the esence, many experts have turned to computing to help provide answers for them that may otherwise take a while to reach through their own expertese. Expert Systems provide an easier way for professionals to reach conclusions through a series of questions, usually linking to a knowledge base. For example, a doctor may use an expert system to reach a diagnosis for a patient, where the illness or problem may not be readily obvious. The system would ask a question, where the doctor would answer with the symptoms, and then the expert system would continue asking more questions, based on the previous answers, until a conclusion is reached.

Expert system development entials the conversion of information about a bounded problem domain into knowledge, which is then represented in a format suitable for computer manipulation. The created depository of knowledge, known as the knowledge base, can then by used by various deductive reasoning techniques to derive solutions [Nikolopoulos, 1997].

From this, it is obvious that the usefulness of the expert system is only as good as the underlying data - the knowledge base. Using the doctors example from the previous paragraph, the system would be of no use if the knowledge base was minimal, whilst in the reverse situation, it would be extremely useful to have a large knowledge base. Doctors from around the world could contribute answers as new illnesses and cures are discovered, which in turn would improve the accuracy and usefulness of the expert system.

Expert systems can be found in many areas of industry, including engineering, medical, financial forecasting, or even just personal information systems for helping customers at a shopping centre.

Disadvantages

There are some notable disadvantages to expert systems. One is that if the rules are of sub-par quality, then the answer may not be useful. Also, if the person being questioned gives an answer that they're uncertain of, then the result may not be accurate, and the expert system would be non-the wiser. A human expert would be able to tell the certainty of someones answer, and possibly re-ask the question in a way that the questionee would understand.

3.2.1 WinProlog

WinProlog is a propietary software package, developed by Logic Programming Associates Ltd, that ¡blah blah blah¿

WIN-PROLOG is the leading Prolog compiler system for Windows-based PCs. Prolog is an established and powerful AI language which provides a high-level and productive environment based on logical inference.[LPA]

3.2.2 Flex

The 'Flex Tutorial' describes Flex as "Flex is a software system specifically designed to aid the development and delivery of Expert Systems."

To appreciate both the power and limitations of Expert System approaches to reaching expert conclusions, it is necessary to construct and experiment with expert systems. In this module, the Flex Expert System Shell will be used. Flex describes knowledge in terms of *production rules* (that is, *if-then* statements), which has proved the most popular approach to encapsulation expert knowledge. Such rules, despite appearing simple, enable relatively complex connections to be made between individual pieces of 'knowledge', thereby solving apparently difficult problems.[Roach, 2009].

3.3 Development

3.3.1 Environment

One of the main aims of this project was to create an open-source alternative to the propietary package "WinProlog", which provides the Flex toolkit. This aim was to provide the project on an open-source platform, such as Linux, which would allow lecturers to tailor the package to their needs, allowing them to fix any bugs or add more advanced features, without the fear of violating any licences. Therefore, Linux was the platform of choice to develop on, where the distribution of choice was Ubuntu, as it has a great wealth of development applications in its repositories, such as `flex`, `bison` and `gcc`. Ubuntu provides a package, `build-essential`, to help developers write and compile applications easily, which contains many useful tools that should please a large majority of software developers.

Installing these packages in Ubuntu simply required the following command:

```
$ sudo apt-get install flex bison build-essential
```

Flex and Bison shall be discussed in the next chapter. `build-essential` is a Debian meta-package, that is, it is simply a list of packages that are essential to building applications in a Debian based distribution. The package satifisies the needs for most developers wishing to develop applications on Linux, and includes packages such as `gcc` and `g++` - a C and C++ compiler, respectively.

Ubuntu provides a wealth of choice when it comes to development tools, however only the simple tools are required, such as a text editor such as `gedit`, or console based text editor, `nano`, whilst the terminal is perfectly acceptable for running commands to compile the tokens, grammar and C program together. `make` comes part of the `build-essential` package, which allows a developer to write a series of commands into a makefile, and then simply run the command:

```
$ make
```

in the directory where the makefile is located.

3.3.2 Source Control

Source control allows the source code to be backed up on a remote repository and ensures that several users are working on the same code, rather than each working on several different versions. Whilst this won't be an issue for this project, it will be a useful time to research and learn how to use source control, and will also provide a useful back up tool. The advantage of using source control, over traditional back-ups, or using removable memory, is that it's unlikely it will get lost, and the same, up to date work can be accessed from other machines if the situation requires it. For example, a developer has a desktop PC and a laptop, and requires to work on the code using the laptop for several days. Source control allows the developer to easily pull the latest version of the code from the repository, code away, and then push the changes back to the repository, so that when the developer comes to use their PC, they can pull the latest code and be up to date, without having to work out which files are more recent on a USB stick.

There are several different source control services available for free, that do roughly the same job.

This chapter will cover the research into the technical research involved into the implementation process.

3.4 Lexical Analysis

There are several incarnations of lexical analysis programs, but the most popular is Flex, which originally dirived from yacc and lex. Lex was written in 1975 by Mike Lesk and Eric Schmidt (now CEO of Google Corporation)

Flex and Bison are tools for building programs that handle structured input. They were originally tools for building compilers, but they have proven to be useful in many other areas.[Levine, 2009].

Lexical analysis is the process of taking an input, and splitting it up into meaningful parts, or *tokens*. For instance, `answer = 5 + 4;` would be split into six tokens: *answer, equals, five, plus, four* and *semi-colon*. These tokens can then be passed onto bison, which can then interpret the context of the tokens by the grammar specified in the EBNF (Extended Backus-Naur Form) .

Using the grammar, the parser can work out that `5 + 4` is an expression, in which the answer is assigned (`=`) to `answer`. By using this approach, it makes it easier to analyse and input sources that will be interpreted.

3.5 Parsing

Parsing is the process as described in the previous paragraph, where expressions are split up into tokens. These tokens are defined through regular expressions, where flex will match a regular expression with the input source, and return a token, which represents a numerical value. For example, the following expression has two parts - first, the regular expression, and then the returning of the token for that expression:

```
"if"     return IF;
```

Here, when the lexical analyser recognises the word ”if”, it will return a token, IF. There are also occasions whereby literal values need to be passed onto bison to be parsed, for instance, a number, or string. In this instance, values need to be stored in a variable, which in turn is then returned, with a token associated to the type, for example:

```
[0-9]+     yylval.d = atof(yytext); return NUMBER;
```

In this case, a number expression, represented by the sequence `[0-9]+` is stored into yylval, and then returned as a NUMBER token. This is usefull when dealing with literal values, such as assigning numbers and strings into variables, and also identifiers, such as variable and function names.

It is possible to create a simple program just by using flex. C code can be appened at the bottom of a lex file, and variables can be defined and used in the body of the token declarations. For instance, a small program could be written that counts the number of words in a file, and outputs it after flex has finished analysing the file.

3.5.1 Regular Expressions

Regular Expressions, or more commonly known as 'regex', allows developers to match patterns of text. It is extremely powerful for recognising patterns in text, and thus, is used in flex for matching expressions to create tokens.

A regular expression is a pattern description using a metalanguage, a language that you can use to describe what you want the pattern to match. Flex's regular expression language is essentially POSIX-extended regular expressions (which is not surprising considering their shared Unix heritage).[Levine, 2009]

For example, numbers can be expressed in several ways, some numbers are negative and some have decimal places. An expression needs to be able to recognise these different formats. A string of digits, which can either be positive or negative, can be represented using the expression below:

```
[-+]?[0-9]+
```

The square brackets `[ ]` represents a character class, matching anything within the brakcets. The first occurance, `[-+]` matches either a postive of a negative symbol at the start of the expression. The `?` character means that the preceding character class can either occur once, or never, which means that the plus or minus characters can be optional. Next is the `[0-9]` expression, which matches any text that is number 0 to 9 - the `+` at the end means that the preceeding expression can be matched one or more times, allowing a number to be either 1 or 12345.

3.5.2 Top-Down Parsing

Parses input string of tokens by tracing out the steps of the left most derivation. Called top down because the implied traversal of the parse tree is a preorder traversal and thus occurs from the root to the leaves (think BST's).

Two Types: Backtracing parsers & predictive parsers. Predictive attempts to predict the next construction in the input string using one or more lookahead tokens, while a background parser will try different possibilities for a parse of the input, backing up an arbitrary amount in the input if one possiblity fails. Backtracing is more powerful, but slower than predictive - useful point to make when running an interpreter.[Louden, 1997]

3.5.3 Bottom-Up Parsing

Bottom up parsing is parsing that goes from bottom to up.

Chapter 4

Implementation

Implementation details, including looking into LLVM etc.

Chapter 5

Conclusion

I conclude that the project was poo.

Bibliography

John R. Levine. *Flex & Bison*. O'Reilly Media, 2009.

Kenneth C. Louden. *Compiler Construction - Principles and Practice*. PWS Publishing Company, 1997.

LPA. Lpa win-prolog 4.9. http://www.lpa.co.uk/win.htm.

Chris Nikolopoulos. *Expert Systems*. CRC Press; 1 edition, 1997.

Paul Roach. Techniques handout 3. Module CS4S22, Lecture Note, 2009.