THE COMPUTER SCIENCE OF TEX AND LATEX; COMPUTER SCIENCE COURSE 594, FALL 2004

VICTOR EIJKHOUT
DEPARTMENT OF COMPUTER SCIENCE,
UNIVERSITY OF TENNESSEE, KNOXVILLE TN 37996

Chapter 1

TEX and LATEX

In this chapter we will learn

- The use of LATEX for document preparation,
- LATEX style file programming,
- TEX programming.

Handouts and further reading for this chapter

For LATEX use the ‘Not so short introduction to LATEX’ by Oetiker *et al.*. For further reading and future reference, it is highly recommended that you get ‘A guide to LATEX’ by Kopka and Daly [4]. The original reference is the book by Lamport [5]. While it is a fine book, it has not kept up with developments around LATEX, such as contributed graphics and other packages. A book that does discuss extensions to LATEX in great detail is the ‘LATEX Companion’ by Mittelbach *et al.* [6].

For the TEX system itself, consult ‘TEX by Topic’. The original reference is the book by Knuth [3], and the ultimate reference is the published source [2].

LATEX.

1.1 Document markup

If you are used to ‘wysiwyg’ (what you see is what you get) text processors, LATEX may seem like a strange beast, primitive, and probably out-dated. While it is true that there is a long history behind TEX and LATEX, and the ideas are indeed based on much more primitive technology than what we have these days, these ideas have regained surprising validity in recent times.

1.1.1 A little bit of history

Document markup dates back to the earliest days of computer typesetting. In those days, terminals were strictly character-based: they could only render mono-spaced built-in fonts. Graphics terminals were very expensive. As a result, compositors had to key in text on a terminal, and only saw the result when it would come out of the printer.

Any control of the layout, therefore, also had to be through character sequences. To set text in bold face, you may have had to surround it with `<B> .. the text .. </B>`. Doesn’t that look like something we still encounter every day?

Such ‘control sequences’ had a second use: they could serve a template function, expanding to often used bits of text. For instance, you could imagine `<ADAM>` expanding to ‘From our correspondent in Amsterdam:’.

TEX works exactly the same: I have to type `\LaTeX` to get the string ‘LATEX’ plus the control codes for all that shifting up and down and changes in font size. Also, you get bold type by specifying `\bf`, et cetera.

1.1.2 Logical markup

TEX.

1.2 The TEX typesetting system

Exercises and projects.

1. Write macros for homework typesetting.
2. Write a style for homework typesetting.

Chapter 2

Parsing

The programming language part of TEX is rather unusual. In this chapter we will learn the basics of language theory and parsing, and apply this to parsing TEX and LATEX. Although TEXcan not be treated like other programming languages, it is interesting to see how far we can get with existing tools.

Handouts and further reading for this chapter

The theory of languages and automata is discussed in any number of books, such as the Hopcroft and Ulman one. For a discussion that is more specific to compilers, see the compilers book by Aho and Ulman or Aho, Seti, and Ulman.

The tutorials on *lex* and *yacc* should suffice you for most applications. The O'Reilly book is probably the best reference on *lex* and *yacc*. There are some other tutorials to be found on the web, but none that impressed me.

Parsing theory.

2.1 Introduction

2.1.1 Levels of parsing

A compiler, or other translating software, has two main tasks: checking the input for validity, and if it is valid, understanding its meaning and transforming it into an executable that realizes this meaning. We will not go into the generation of the executable code here, but focus on the validity check and the analysis of the meaning, both of which are parsing tasks.

A parser needs to look at the input on all sorts of levels:

- Are all characters valid – no 8-bit ascii?
- Are names, or identifiers, well-formed? In most programming languages `a1` is a valid name, but `1a` is not. By contrast, in TEX a name can only have letters, while in certain Lisp dialects `!!important_name!!` is allowed.
- Are expressions well-formed? An arithmetic expression like `5/*6-` does not make sense, nor does `CALL )FOO(` in Fortran.
- If the input is well-formed, are constraints satisfied such as that every name that is used is defined first?

These different levels are best handled by a several different software components. In this chapter we will look at the two initial stages of most translators[1].

1. First of all there is the lexical analysis. Here a file of characters is turned into a stream of tokens. The software that performs this task is called a tokenizer, and it can be formalized. The theoretical construct on which the tokenizer is based is called a 'Finite State Automaton'.
2. Next, we need to check if the tokens produced by the tokenizer come in a legal sequence. For instance, opening and closing parentheses need to come in matched pairs. This stage is called the syntactical analysis, and the software doing this is called a parser.

2.1.2 Very short introduction to formal languages and automata

A language is a set of words that are constructed from an alphabet. The alphabet is finite in size, and words are finite in length, but languages can have an infinite number of words. The alphabet is often not specified explicitly.

Languages are often described with set notation and regular expressions, for example '$L = \{a^nb^*c^n|n > 0\}$', which says that the language is all strings of equal number of as and cs with an arbitrary number of bs in between.

Regular expressions are built up from the following ingredients:

$\alpha|\beta$ either the expression α or β
α^* zero or more occurrences of α
$\alpha+$ one or more occurrences of α
$\alpha?$ zero or one occurrences of α

1. I will use the terms 'translating' and 'translater' as informal concepts that cover both compilers and interpreters and all sorts of mix forms. This is not the place to get philosophical about the differences.

We will see more complicated expressions in the *lex* utility.

A description of a language is not very constructive. To know how to generate a language we need a grammar. A grammar is a set of rules or production $\alpha \rightarrow \beta$ that state that, in deriving a word in the language, the intermediate string α can be replaced by β. These strings can be a combination of

- A start symbol S,
- 'Terminal' symbols, which are letters from the alphabet; these are traditionally rendered with lowercase letters.
- 'Non-terminal' symbols, which are not in the alphabet, and which have to be replaced at some point in the derivation; these are traditionally rendered with uppercase letters.
- the empty symbol ϵ.

Languages can be categorized according to the types of rules in their grammar:

type 0 These are called 'recursive languages', and their grammar rules can be of any form: both the left and right side can have any combination of terminals, non-terminals, and ϵ.

type 1 'Context-sensitive languages' are limited in that ϵ can not appear in the left side of a production. A typical type 1 rule would look like

$$\alpha A\beta \rightarrow \gamma$$

which states that A, in the context of $\alpha A\beta$, is replaced by γ. Hence the name of this class of languages.

type 2 'Context-free languages' are limited in that the left side of a production can only consist of single non-terminal, as in $A \rightarrow \gamma$. This means that replacement of the non-terminal is done regardless of context; hence the name.

type 3 'Regular languages' can additionally only have a single non-terminal in each right-hand side.

2.1.3 Lexical analysis

The lexical analysis phase of program translation takes a stream of characters and outputs a stream of tokens.

It might be tempting to consider the input stream to consist of lines, each of which consist of characters, but this does not always make sense. Programming languages such as Fortran do look at the source, one line at a time; C does not. TEX is even more complicated: the interpretation of the line end is programmable.[2]

A token is a way of recognizing that certain characters belong together, and form an object that we can classify somehow. In some cases all that is necessary is knowing the class, for instance if the class has only one member. However, in general a token is a pair consisting of *its type and its value.* For instance, in `1/234` the lexical analysis recognizes that `234` is a number, with the value 234. In an assignment `abc = 456`, the characters `abc` are recognized as a variable. In this case the value is not the numeric value, but rather something like the index of where this variable is stored in an internal table.

Lexical analysis is relatively simple; it is performed by software that uses the theory of Finite State Automata and Regular Languages; see section 2.2.

2. Ok, if we want to be precise, TEX does look at the input source on a line-by-line basis. There is something of a preprocessor *before* the lexical analysis. See section 2.2 of TEXby Topic for the full details.

2.2 Finite state automata and regular languages

2.2.1 Definition of regular languages

Recursive definition of regular languages

2.2.2 Finite state automata

Construction of accepting automaton based on the construction of the language

2.2.3 Examples

Examples and counter examples of regular languages

2.2.4 Normal form

Normal form of regular languages, proof of equivalence

2.2.5 Non-deterministic automata

why they are useful for lexical analysis; equivalence to deterministic

2.2.6 Lexical analysis with FSAs

Adding actions to recognition rules

Exercises and projects.

1. Write a FSA that can parse Fortran arithmetic expressions. In Fortran, exponentiation is written like `2**n`.
2. Write a *lex* program that parses TEX code correctly, using the standard category code values.

Project Extend *lex* and *yacc* so that category code changes can be handled.

Project Implement *lex* and *yacc* in a language of your own choice (Perl or Python?) so that they generate code in other languages than C.

Chapter 3

Breaking things into pieces

The line breaking algorithm of TEX is interesting, in that it produces an aesthetically optimal solution in very little time.

Paragraph breaking.

3.1 Paragraph breaking

A simple cost function is

$$\sum_{\mathrm{l}ines} \left(W - \sum_{\mathrm{w}ordiinline} w(word_i) \right)$$

- No penalty for adjacent tight/loose lines.
- Does this try to minimize the number of lines?

Possible extensions

- phonetic breaking
- rivers

Chapter 4

Fonts

Knuth wrote a font program, Metafont, to go with TEX. The font descriptions involve some interesting mathematics.

Chapter 5

TEX's macro language

The programming language of TEX is rather idiosyncratic. One notable feature is the difference between expanded and executed commands. The expansion mechanism is very powerful: it is in fact possible to implement lambda calculus in it.

Chapter 6

Literate programming

Yet another part of the TEX system is the Web system for ‘literate programming’.

Chapter 7

Document markup

Scribe, troff, Lout, TEX.

Exercises and projects.

Project Design a general model for tables and write software that formats tables. You can output TEX macros, or argue why they aren't powerful enough, and design a better language for describing tables.

Chapter 8

Big code development

Handouts and further reading for this chapter

Knuth wrote a history of the TEX project in [1].

Chapter 9

Character encoding

There was life before ascii, even before ebcdic.

Exercises and projects.

Project Dig into history (find the archives of `alt.folklore.computers`!) and write a history character encoding. Highlight design decisions good and bad.

Bibliography

[1] D.E. Knuth. The errors of TEX. *Software Practice and Experience*, 19:pages = 607–681.

[2] D.E. Knuth. *TEX: the Program*. Addison-Wesley, 1986.

[3] D.E. Knuth. *The TEX book*. Addison-Wesley, reprinted with corrections 1989.

[4] Helmut Kopka and Patrick W. Daly. *A Guide to LATEX*. Addison-Wesley, first published 1992.

[5] L. Lamport. *LATEX, a Document Preparation System*. Addison-Wesley, 1986.

[6] Frank Mittelbach, Michel Goossens, Johannes Braams, David Carlisle, and Chris Rowley. *The LATEX Companion, 2nd edition*.

Index