Fast scanners and self-parsing in TEX

Pedro Palao Gostanza
Universidad Complutense de Madrid
Edificio de CC. Matemáticas,
Ciudad Universitaria s/n,
Madrid 28040,
Spain
ecceso@sip.ucm.es

Abstract

In this paper we will explain how to build fast scanners for regular languages in TEX. The resulting scanners can be composed with different parsers if they are written as *self parsers*. This parsing technique allows us to choose the syntactic rigour: from loose parsers with almost no syntax checks, good for pretty-printers, to the strict parser needed to write a compiler.

Introduction

It is customary to embed little languages in TEX; LATEX picture environment, XY-pic [9] and TESLA [2] are illustrative examples. TEX can process these languages in a reasonable fast manner because their syntaxes are chosen with TEX's abilities in mind; for instance, arguments are surrounded with braces or delimited with fixed characters. Also, programs written with these little languages are usually short.

Sometimes, TEX has to process programs written in *bigger* languages, or in languages designed without paying attention to TEX. Pretty-printing structured programming languages is the ubiquitous example [6, 11], but we also meet the same problems when trying to typeset HTML directly or when doing some *Cronopio* [3] activities like executing programs written in high level languages [5] within TEX. Programs to cope with these problems are written around a slow character by character processing engine. The `\FIND` macro ([4, 10]), possible in a modified or subtle variant, is the kernel of this processing engine. This is the method adopted in Doumont's pretty-printer [6] and Woliński's scanners [11]. Slow processing is particularly striking here because programs in these languages can be really long.

We became interested in this problem several years ago, while doing just another Pascal pretty-printer [7]. Since we found no trick to build a fast scanner, we simply put the burden on the user who was forced to write a backslash before every identifier. In this way, every identifier was a control word and we could take advantage of TEX's scanner. Obviously, this was a poor design decision. Fortunately, one year ago we found a trick to write fast scanners in TEX; to our knowledge, it seems to be an unused TEXnique.

The idea to make a fast scanner in TEX is *not to check* every character. But, since a scanner must *see* every character, the implementation must use *burst like processing*: the scanner checks some characters, but TEX internal machinery eats the rest. In this way, the final complexity will be a small constant (due to TEX internals) multiplying a linear factor, plus a big constant (due to scanner checks) multiplying a sub-linear factor. We will call every scanner that works without checking every character in its input a *fast scanner*.

Our trick to make a fast scanner is based on active characters and `\edef`. We will illustrate it with Pascal in the next section. The scanner engine is spread along all the active characters; each character knows how to keep the scanner alive. This organization, where every token knows what to do, and there is no centralized set of processing macros, is what we call *self-parsing*. Self-parsing can be used to build scanners and parsers. Its two main advantages are: firstly, it allows to split scanners and parsers like in a traditional compiler, and secondly, the syntactic rigour can be chosen. The first advantage allows us to write a definitive Pascal scanner that can be used to feed a pretty-printer parser, a compiler parser, etc. The second allows us to write a strict parser for a compiler and a loose parser for a pretty-printer with this same technique. Of course, each parser will have to be developed from scratch.

To illustrate all these ideas we will use, as a running example, a small Pascal subset that we

Program ::= `program` Id `;` Block `.`
Block ::= Decls `begin` Stats `end`
Decls ::= ⟨*empty*⟩ | Vars Decls
Vars ::= `var` SeqVar SeqsVar
SeqsVar ::= ⟨*empty*⟩ | SeqVar SeqsVar
SeqVar ::= Ids `:` Id `;`
Ids ::= Id IdsExt
IdsExt ::= ⟨*empty*⟩ | `,` Ids
Stats ::= Stat StatsExt
StatsExt ::= ⟨*empty*⟩ | `;` Stat StatsExt
Stat ::= Id `:=` Expr
    | Id OptArgs | `if` Expr `then` Stat Else
    | `while` Expr `do` Stat | `begin` Stats `end`
Else ::= ⟨*empty*⟩ | `else` Stat
OptArgs ::= ⟨*empty*⟩ | `(` Args `)`
Args ::= Expr ArgsExt
ArgsExt ::= ⟨*empty*⟩ | `,` Args
Expr ::= Rel
Rel ::= Term RelExt
RelExt ::= ⟨*empty*⟩ | `=` Term | `<>` Term
    | `<` Term | `<=` Term | `>` Term | `>=` Term
Term ::= Factor TermExt
TermExt ::= ⟨*empty*⟩
    | `+` Factor TermExt | `-` Factor TermExt
Factor ::= Atom FactorExt
FactorExt ::= ⟨*empty*⟩ | `*` Atom FactorExt
    | `div` Atom FactorExt | `mod` Atom FactorExt
Atom ::= Int | Id | `(` Expr `)` | `-` Atom

Id ::= Letter LettersOrDigits
LetterOrDigits ::= ⟨*empty*⟩ | Letter LetterOrDigits
    | Digit LettersOrDigits
Int ::= Digit Digits
Digits ::= ⟨*empty*⟩ | Digit Digits
Digit ::= `0` | ⋯ | `9`
Letters ::= `a` | ⋯ | `z` | `A` | ⋯ | `Z`

Figure 1: Mini-Pascal syntax

will call 'mini-Pascal.' Its syntax can be found in Figure 1.

The rest of this paper is organized as follows. In the next section, we explain how to write a fast scanner for mini-Pascal. Although its main idea can be reused, each language has its own tricks that speed the scanner even more; part of this section is devoted to explore useful tricks for Pascal and other structured programming languages. Next we will devote two sections to building a pretty-printer and a compiler for mini-Pascal. Both parsers work on top of the same scanner. Since pretty-printing is a widely studied area, the goal of our pretty-printer is not how to pretty-print a programming language but how to build a loose self-parser. Obviously, the goal of the compiler is how to built a strict self-parser. The section 'Other uses' reviews other projects where we have used fast scanners and self-parsers. Finally, we conclude and suggest some future work.

A Pascal scanner

By far, identifiers and keywords account for most of the characters in a Pascal program. Letters are used exclusively for this purpose.[1] Digits can also appear in identifiers, but do so rather seldom; they almost exclusively appear in numbers. Every other character, apart from white space, is seldom used. White space has a strange occurrence pattern: every line starts with a long white sequence (just after an end of line), and then single spaces split other tokens.

In order to get really good sub-linear behavior, a scanner should operate checking no character in any identifier (including keywords), and no space in every start-of-line white sequence.

The scanner will change each Pascal token into a TeX control word: the identifier `Foo` to `\Id{Foo}`, the number `123` to `\Int{123}`, the keyword `begin` to `\Begin`, `:=` to `\Assign`, etc.

Every character but roman letters will be active; that is, `,`, `.`, `:`, `;`, `(`, `)`, `+`, `-`, `*`, `=`, `<`, `>`, `0`, . . . , `9`, and blanks are active characters. Forget for a moment digits and symbols composed with more than one active character, like `<>`. So, every identifier is composed only with letters (non-active characters) and is surrounded with active characters. To catch these identifiers without an explicit character-by-character analysis, it is enough to start a capture at the end of each active character and to finish this capture at the beginning of each active character. The macro `\catchId` starts the capture:

```
\def\catchId{\edef\mayId{\iffalse}\fi}
```

This macro store every following characters in `\mayId`, while expanding active characters. To finish this capture an active: character uses

```
\def\endId{\iffalse{\else}\fi}
```

This capture does not always success; for example, there can be two active characters one after the other. Since every character that cannot take part in an identifier is active, the capture will be successful if and only if `\mayId` is not `\empty`.

[1] Not exactly: `e` and `E` are used in floating point literal numbers.

```
\def\empty{}
\def\flush{\ifx\mayId\empty\else
  \expandafter\Id\expandafter{\mayId}\fi}
```

These three macros are all a white space must do:

```
\def\blank{\endId\flush\catchId}
```

This is also needed at the beginning and the end of the scanner

```
\def\beginScanner{\activeChars\defActive
  \catchId}
\def\endScanner{\endId\flush}
```

Other active characters will produce, in addition, its own Pascal token:

```
\def+{\endId\flush\Plus\catchId}
\def.{\endId\flush\Dot\catchId}
...
```

Of course, some identifiers are keywords, which introduces two problems. First, it is necessary to check every identifier in order to test whether it is a keyword or a regular identifier. Several TEXniques to implement sets may be used, for example:

- For each keyword k there is a control word `\kw@`k trivially defined (but not `\relax`). Checking whether an identifier is a keyword is simple and fast:

  ```
  \def\ifIsNotKW#1{\expandafter\ifx
    \csname kw@#1\endcsname\relax}
  ```

  But each new identifier introduces a new entry in TEX control sequences table. Since TEX never deletes entries from this table, we can exhaust TEX limited hash memory.
- All keywords can be stored in a macro:

  ```
  \def\kws{|begin|end|if|...|while|}
  ```

  Checking an identifier is tricky and long

  ```
  \def\ifIsNotKW#1{\def\aux
    ##1|#1|##2\relax{\ifx\relax##2\relax}%
    \expandafter\aux\kws#1|\relax}
  ```

  but it works in a constant amount of memory.

Since we want to build fast scanners, the first technique is preferable.

The second problem arising with keywords concerns capitalization. Pascal is case insensitive, so we should be capable of recognizing an identifier or keyword without regard to its capitalization. This can be easily solved invoking a `\lowercase` over those characters stored in `\mayId`. Since some parsers care about capitalization, but others do not, it is better to change the identifier `Foo` into `\Id{foo}{Foo}` instead of only to `\Id{Foo}`.

Part of the speed has already been achieved. The other acceleration source is to deal with spaces at the beginning of lines. The trick is to recover the original category codes of spaces at end of lines. Then we look for the next TEX token so that TEX eats all the intermediate spaces.

```
\def\eoln{\endId\flush\catcode`\ =10 \eolnB}
\def\eolnB#1{\catcode`\ =\active\catchId#1}
```

Category code 9 (ignored character) also works.

To consider numbers and composed symbols, an state needs to be added to the scanner; we will call it ‘scanner state’ because, latter in the paper, some other states will come into play. The scanner state due to numbers is a macro `\mayInt` where its digits will be stored. Composed symbols need two macros: `\maySym` and `\symCode`. `\symCode` is a number that uniquely determines what characters are in the current symbol; it is 0 if there is no character, or a non-zero integer for each of the three characters that can start composed symbols:

```
\chardef\noCode=0
\chardef\colonCode=1
\chardef\lessCode=2
\chardef\greaterCode=3
```

`\maySym` stores which token will be generated if there is no character extending the current symbol. For instance, a colon does not generate a `\Colon` token directly; instead, it is stored, so that, if there is an equal immediately after it, an `\Assign` will be generated.

```
\def:{\endId\flush
  \gdef\maySym{\Colon}\glet\symCode\colonCode
  \catchId}
```

But what if there is an identifier followed by an space; the `\Id` token will be generated before the `\Colon`; it is even possible that the `\Colon` get lost. The solution is simple: `\flush` must not only take care of captured identifiers but also of delayed symbols and stored numbers.

```
\def\flush{\flushSym\flushInt\flushId}
\def\genSym{\maySym\glet\symCode\noCode}
\def\flushSym{\ifnum\symCode=\noCode
  \else\genSym\fi}
\def\genInt{\expandafter\Int\expandafter
  {\mayInt}\glet\mayInt\empty}
\def\flushInt{\ifx\mayInt\empty\else\genInt\fi}
\def\genId{\expandafter\Id\expandafter{\mayId}}
\def\flushId{\ifx\mayId\empty\else\genId\fi}
```

Characters that can be in the second place of a composed symbol cannot simply `\flush`; they should flush integers and identifiers, but can only flush delayed symbols if there is something intertwined:

```
\def\flushInter{%
  \ifx\mayInt\empty\else\flushSym\genInt\fi
  \ifx\mayId\empty\else\flushSym\genId\fi}
```

The definition of ‘>’ is illustrative because it can be the first and the second character in a composed symbol:

```
\def>{\endId\flushInter
  \ifnum\symCode=\lessCode
```

```
    \glet\symCode\noCode \NotEqual
  \else\flushSym\gdef\maySym{\Greater}%
    \glet\symCode\greaterCode
  \fi\catchId}
```

Digits do important work to keep the scanner alive, but they do not need to flush because a character will follow that will cause flushing. The main digit task is to know whether it is part of a number or an identifier.

```
  \def\digit#1{\endId
    \ifx\mayInt\empty
      \ifx\mayId\empty \gdef\mayInt{#1}\catchId
      \else \catchId\mayId#1\fi
    \else\ifx\mayId\empty
      \xdef\mayInt{\mayInt#1}\catchId
    \else\errmessage{An identifier cannot
      start with digits}\catchId
    \fi\fi}
```

This completes our fast scanner. But a little problem remains. Sometimes, the program we want to parse is not embedded in the document sources, but stored in a separate file. Invoking `\input` inside an scanner context (`\beginScanner\endScanner`) has no use because backslash will loose its usual meaning inside this context. The usual roundabout

```
\expandafter\beginScanner\input p.pas \endScanner
```

does not work either (read `\@@input` instead of `\input` if thinking in LaTeX) because the last active character in `p.pas` launches a `\catchId` that will be closed in `\endScanner` after the end-of-file, i.e., the last active character starts a definition that will be closed pass the end of the file. Since TeX does not allow a definition to span across files, we will get the error "File ended while scanning definition of `\mayId`." Fortunately, TeX appends an end-of-line character to the last line of every file, if absent; so the last character in every file processed with TeX is an end-of-line. We are going to put on EOLN character the burden of crossing the end-of-file boundary before to launch `\catchId`. Of course, if not at the end of a file, an EOLN should behave as before. The cheapest manner to cross the end-of-line boundary is with `\futurelet`:

```
\def\eoln{\endId\flush\catcode`\ =9
  \futurelet\aux\eolnB}
\def\eolnB{\catcode`\ =\active\catchId}
```

Incidentally, `\futurelet` makes leading spaces (ignored characters) in the next line disappear.

A Pascal pretty-printing

Now, we have our scanner ready. Let us use it to build a pretty-printer. The pretty-printer is responsible for choosing a correct definition for the tokens that the scanner generates. These definitions cannot look forward following tokens because the scanner may not have produced them yet and because there can be TeX control words (that remain to be evaluated) before the next token. So the pretty-printer must conform to the self-parsing technique. Of course, each token can change the pretty-printer state, in order to produce a visible effect, to prepare the environment for the next tokens, and to communicate to future tokens its previous occurrence.

Self-parsing is such a natural technique to use in a pretty-printer that it has been discovered and used in several pretty-printers before (at least in [6] and in [7]). But it has never been used in a pure manner, neither recognized as a useful general parsing technique. So, our emphasis will be to explain how self-parsing can be used to build a loose parser. Pretty-printer output will be very simple, just the raw style in [7]: every statement in a line; keywords are in bold face; identifiers are in italics; every expression, assignment and procedure call is typeset in TeX math mode. The formatted program to compute x^n, for $x = 3$ and $n = 9$, follows:

```
program power;
  var x, n: integer;
      x1, n1, pow: integer;
begin
  $x \leftarrow 3$;
  $n \leftarrow 9$;
  $x1 \leftarrow x$;
  $n1 \leftarrow n$;
  $pow \leftarrow 1$;
  while $n1 \geq 1$ do begin
    if $n1$ mod $2 = 1$ then $pow \leftarrow pow \times x1$;
    $x1 \leftarrow x1 \times x1$;
    $n1 \leftarrow n1$ div 2
  end;
  write(pow)
end.
```

To keep the pretty-printer alive, every token must do some work. Some, like parenthesis, do a really simple work, without bothering about where it is used.

```
\def\OpenPar{(}
```

Others, like assignments, do a simple work too, but require that other tokens have already opened a math mode.

```
\def\Assign{\leftarrow}
```

An assignment in an incorrect place will cause a "Missing $ inserted" error; this is a syntactic check with a bad error message.

Identifiers behave differently if placed at the beginning of an statement or inside an expression. In the first case, they must open a math mode; in the second case, they only have to write themselves. The best agreement is to use a `\ifinsideexpr` condition

```
\let\ifinsideexpr\ifmmode
```

```
\def\openExpr{\ifinsideexpr\else$\fi}
\def\closeExpr{\ifinsideexpr$\fi}
\def\Id#1{\openExpr\hbox{\it#1}}
```

If a token can appear after an expression it should ensure that the expression is closed; for example:

```
\def\Semicolon{\closeExpr;\par}
```

But not every semicolon behaves in this way; the semicolon after the program name must indent following constructions a bit. There are three techniques to solve this problem:

Conditions technique in which we have a global condition `\ifafterprogram`, which the definition of `\Program` sets true. Other definitions set it false; since there is only one semicolon after a program name, the definition of semicolon is the best place to set it false.

Redefinition technique in which we have one definition for each possible behaviour. Other tokens redefine `\Semicolon` according to the expected behaviour in an immediate future.

Steps technique in which we have a number, a step holder, that records the syntactic construction where the present token occurs. Each token can adjust its behaviour according to the value in the step holder, and change it as necessary.

These three techniques are equivalent, but depending on the problem one is easier than the others. As the syntactic checks become stricter, one should move from the first to the third. These three techniques can be used simultaneously; for instance, in the above fast scanner we have mixed conditions and steps in order to build symbols composed with more than one character.

Usually a stack is needed in order to store values (before changing a condition, making a redefinition or assigning to the step holder) that will be restored when a nested construction ends. For a pretty-printer, TEX grouping is enough, but for stricter parsers it is better to maintain an explicit stack.

The following macros illustrate a pretty-printer built with redefinition technique.

```
\def\Program{{\bf program}\
  \let\Begin\BeginBlock
  \let\Semicolon\SemicolonProgram}
\def\SemicolonProgram{\closeExpr;\par\indent
  \let\Semicolon\SemicolonBlock}
\def\SemicolonBlock{\closeExpr;\par}
\def\BeginBlock{\par\outdent
  {\bf begin}\par\indent
  \let\Begin\BeginStat}
\def\BeginStat{\ {bf begin}\par\indent}
```

A Pascal compiler

Implementing a pretty-printer with self-parsing is an easy task, easier than doing it with a classical and strict parser. So, we wonder how the effort needed to write a self-parser evolves when increasing syntactic checks. We thought that the best test was to write a mini-Pascal compiler.

We envisaged the following organization. To compile a program, it must be surrounded with the pair `\beginPC`/`\endPC`. A program called *power*, for instance, will be translated to a TEX macro called `\power`, that comprises an instruction sequence for a virtual stack machine. Whenever this macro is called, its instructions get executed, and everything written (with *write*) appears inserted in the text.

Since a compiler needs to ensure a complete syntactic conformance, we will use the step technique to produce its parser. But which are the correct steps? This question has already been answered: classical parsing techniques, like LL and LR, rewrite a context free grammar as an automaton. This automaton states are the steps we were looking for.

Here we will work out how to build an LL self-parser because it is simpler than LR parsing and Pascal has an (almost) LL grammar. Nevertheless, the main idea and many details can be reused in an LR self-parser.

The construction of an LL self-parser for a given grammar has been automated with a simple program (written in Haskell [8]) called parTEX. Here we are explaining how to do by hand what this program already does alone. This program expects an LL grammar annotated with semantic actions and semantic checks. Figure 2 shows the production for mini-Pascal statements. Semantic actions are surrounded with braces and semantic checks are also preceded with a question mark. Both semantic actions and checks use several auxiliary macros that read and modify the compiler state; an explanation of their implementation and behaviour is beyond the scope of this paper, but their names are chosen to evoke its meaning (sequences without spaces or end-of-lines are just one macro call with its arguments). Semantic actions (checks) immediately following a terminal that carries information, like an `Id`, get this information through parameters. So, in the semantic actions (checks) following `Id`, `#1` is the identifier down-cased string and `#2` (not used) is the identifier string.

Then, following [1], we add state numbers between every symbol that appears in each production right hand side. The state numbers for the production in Figure 2 (without semantic actions) are:

```
Stat ::=
  Id ?{\isVarQ{#1}}
      {\memDir{#1}\aMemDir
       \emitCode{\lit{\aMemDir}}}
  ":=" Expr {\emitCode{\put}}
| Id {\def\procToCall{#1} \noa=0 }
  OptArgs {\iftrue \isProcQ\procToCall\noa
       \procDir\procToCall\noa\aProcDir
       \emitCode{\call{\aProcDir}}%
     \else \errmessage{No procedure
       "\procToCall" with \number\noa\space
       arguments}\fi}
| "if" Expr {\newLabel\labelse \newLabel\labend
             \emitCode{\jzero{\number\labelse}}}
  "then" {\bgroup}
  Stat {\egroup \emitCode{\jump{\number\labend}}%
        \emitCode{\label{\number\labelse}}}
  Else {\emitCode{\label{\number\labend}}}
| "while" {\newLabel\loopL \newLabel\endloopL
           \emitCode{\label{\number\loopL}}}
  Expr {\addCode{\jzero{\number\endloopL}}}
  "do" {\bgroup}
  Stat {\egroup \addCode{\jump{\number\loopL}}
        \emitCode{\label{\number\endloopL}}}
| "begin" Stats "end"
.
```

Figure 2: Statement production

```
Stat ::=
  Id 42 ":=" 43 Expr 44
| Id 45 ArgsOpt 46
| "if" 48 Expr 49 "then" 50 Stat 51 Else 52
| "while" 54 Expr 55 "do" 56 Stat 57
| "begin" 59 Stats 60 "end" 61
```

There is an obvious map between semantic actions (checks) and automaton states: every semantic action (check) occurs in an automaton state and an automaton state may have one semantic action (check). The semantic action (check) occurring in state n is stored in macro `\sa@`n (`\sc@`n). This macro has as many parameters as information bundles carried by the token preceding the semantic action. For example:

```
\defx{sc@42}#1#2{\isVarQ{#1}}
\defx{sa@42}#1#2{\memDir{#1}\aMemDir
   \emitCode{\lit{\aMemDir}}}
\defx{sa@49}{\newLabel\labelse \newLabel\labend
   \emitCode{\jzero{\number\labelse}}}
```

We will call these macros through

```
\def\semaction#1{\csname sa@#1\endcsame}
```

Due to the behaviour of `\csname`, an action can be called even if it does not exist. To exploit this circumstance, those states after a token that carries information but has no semantic action will have an explicit empty action with enough parameters.

A semantic action is executed when entering its state. A semantic check is executed before entering its state; if it returns true, its state will be entered; if it returns false, another possible next state will be tried.

Traditional parsers encode the automaton and semantic actions in a table. A loop uses the current automaton state and the next token to index this table, to perform some action, and to change to the next automaton state, and a stack is needed to store return states when entering a non-terminal. In a self-parsing implementation the current automaton state and the stack are in the global parser state: `\state` and `\stack`. The table becomes code; the action performed in the loop when looking at the token `\Tok` in state n is stored in the macro `\Tok@`n. Therefore, token `\Tok` behaviour is

```
\csname Tok@\state\endcsname
```

Since most entries in the table are just errors, memory can be saved not defining them. Checks for errors can be factored in the following macro:

```
\def\exe#1{\expandafter\let\expandadfter
  \aux\csname #1@\state\endcsname
  \ifx\aux\relax
    \errmessage{Unexpected token "#1"}\fi
  \aux}
```

So, the definition of `\Tok` can be simplified to

```
\def\Tok{\exe{Tok}}
```

Changing to another state, and pushing and popping states from the stack will be abstracted with `\toState`, `\pushState` and `\popState`. These macros are the best place to call semantic actions.

```
\def\toState#1{\def\state{#1}\semantic{#1}}
```

With all these helper macros, encoding the automaton table with a set of macros is a simple but boring task. For example, a **while** in state 53 only have to change to state 54

```
\defx{while@53}{\toState{54}}
```

But **while** can appear in other states; for example, nested inside another **while**, that is, after a **do** (state 56); in this case, it must push state 57 in the stack, change to state 54

```
\defx{while@56}{\pushState{57}{54}}
```

When nested **while** parsing ends, the state 57 will be restored from the stack and its semantic action executed so that the last instructions of the outer loop were generated. In a self-parser, tokens that may appear after a while statement are responsible for doing this. For example, **end** may appear after every statement type:

```
\defx{end@44}{\popState\exe{end}}
\defx{end@46}{\popState\exe{end}}
\defx{end@51}{\toState{52}\exe{end}}
```

```
\defx{end@52}{\popState\exe{end}}
\defx{end@57}{\popState\exe{end}}
\defx{end@60}{\toState{61}}
```

Notice, that **end** keeps rescheduling itself until reaching state 60.

Encoding entries in the automaton table that need a semantic check to be disambiguated is a bit more complex. For example, `Id` may occur in state 56, just after a **do**; changing to state 42 or 45 depends on the semantic check in state 42:

```
\defx{Id@56}#1#2{%
  \iftrue\semcheck{42}{#1}{#2}%
    \toState{42}{#1}{#2}\else
    \toState{45}{#1}{#2}\fi}
```

As can be seen, there is nothing radically new in this code. The classical parser engine with a monolithic table is split into a lot of little macros. Deciding what to do next does not rely on the parser lookahead but each token checks the correctness of its present occurrence, changes to the next state and invokes a semantic action. Therefore, TEX is a nice source language for a compiler-compiler. Before implementing parTEX, we checked all these ideas by hand; having done this boring work, we are eager to use it wherever possible. So now, we will input again the program pretty-printed on page 238, this time surrounded with `\beginPC`/`\endPC`, just here, only with the intention to execute `\power` to ensure that $3^9 = 19683$.

Other uses

We have used these TEXniques in other projects.

AGL is a small graphic library that our students (and we) write and improve year after year. It is written with the literate paradigm, using `noweb`. Moreover, there is a separate document “getting started and reference”. In order to keep the reference up to date, and to allow concurrent development, the description of each element (function, type, constant, etc.) is embedded in the implementation; typesetting the implementation produces an up to date file to be input in the reference. To ensure full agreement and to save some typing, there is no place in a description to put its definition (the head of a function, the structure of a type, etc.); instead, while processing the reference, TEX opens source code files (built with tangle (`notangle`)) and looks for the definition of each described identifier. A fast scanner splits Pascal programs into tokens; then a search engine, organized like a self-parser, stores the tokens constituting each requested definition in a macro; finally, when typesetting an element description, these tokens feed a pretty-printer. So, the same scanner is composed both with a search engine and a pretty-printer. TEX process hundreds definitions in a few seconds, thanks to the fast scanner.

EXercita is a hierarchical, human-readable database of exercises. Every exercise has, in addition to the wording of the exercise itself, an author (or source), its objective and difficulty, and several solutions. A set of macros helps in extracting exercises to be included in a document. The macros to search databases use a self-parser.

HTEXML (HTML in TEX) is a work in progress to make TEX capable of type setting HTML directly. Almost all the processing work comes from HTML tags. It is important to do it as fast as possible because, although hand written HTML has few tags with few parameters, machine generated HTML has large numbers of tags with lots of (usually unnecessary) parameters. The simple syntax of HTML tags makes really ease to write a fast scanner. With the arrival of CSS a lot of new parsing capabilities are needed.

Conclusions

Fast scanners are clearly fast. We have only collected simple figures. For example, in my old Intel 80486, more than 1000 lines of Pascal code are scanned in 5.4 seconds, and pretty-printed in 4.6 seconds more. TEX typesets this same code (thinking that it is plain text) in 1.7 seconds and needs 0.9 seconds to process a file that only loads the scanner and the pretty-printer — so, parsing and pretty printing is only one order of magnitude slower. In general, it is astonishing to see TEX working so fast in every project were we have tried a fast scanner.

Self-parsing is a nice TEXnique to organize reusable parsers. It also allows an adaptable syntactic rigour. Its main drawback is the effort to build a strict self-parser by hand. Fortunately, parTEX removes this burden. Fast scanners, being an application of self-parser, should inherit this complexity; but writing a fast-scanner generator for TEX is a daunting task because each language has its own tricks. Fortunately, writing a fast scanner by hand is affordable because the lexical part of a programming language is simpler than its syntax. In addition, each processing kind needs a new parser, but the same scanner can be used once and for all.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Henry Baragar and Gail E. Harris. An example of a special purpose input language to LATEX. In

Proceedings of the TUG Annual Meeting, 1994.

[3] Julio Cortazar. *Historias de Cronopios y Famas*.

[4] Jonathan Fine. The `\CASE` and `\FIND` macros. *TUGBoat*, 1(14):35–39, 1993.

[5] Andrew Marc Greene. BASIX—an interpreter written in TEX. In *Proceedings of the TUG Annual Meeting*, 1994.

[6] Jean luc Doumont. Pascal pretty-printing: an example of "preprocessing with TEX". In *Proceedings of the TUG Annual Meeting*, 1994.

[7] Pedro Palao Gostanza and Manuel Núñez García. pa sc al : Formating pascal using TEX. In *EuroTEX*, 1995.

[8] Simon Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudack, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98. a non-strict, purely functional language. 1999.

[9] Kristoffer H. Rose. XY-pic user's guide. 1998.

[10] C.G. van der Laan. `\FIFO` and `\LIFO` sign the BLUes. *TUGBoat*, 1(14):54–60, 1993.

[11] Marcin Woliński. Pretprin—a LATEX 2ε package for pretty-printing texts in formal languages. In *Proceedings of the TUG Annual Meeting*, 1998.

Pedro Palao Gostanza