Typesetting TEX documents containing computer code

Włodek Bzyl
matwb@univ.gda.pl

Abstract

I would like to present a preprocessor driven by grammars able to automatically mark-up pieces of computer code immersed in TEX documents. By computer code I mean any language for which a context free grammar exists, which YACC will accept. These include almost any computer language. But my tool could use any such grammar to automatically mark-up text written in the language defined by it.

Introduction

One of the most difficult tasks in technical typesetting is to get computer programs to look right. [. . .] No automatic system can hope to find the best breaks in programs, since an understanding of the semantics will indicate that certain breaks make the program clearer and reveal its symmetries better.

— D. E. KNUTH, Digital Typography

Computer books and journals do not look as beautiful as they used to. It is not their content that is unsatisfactory, rather the typography is strange. The example below illustrates that. It is a really disgusting piece of typography taken from the Polish translation of an English book.

Program JavaScript przedstawiony w listingu 6.2 stanowi przyklad zastosowania cookie.

```
//=========================================
// Here are our standard Cookie routines
//=========================================
//-----------------------------------------
// SetCookieEZ - Quickly sets a cookie
//    which which will last until the user
//    shuts down his browser
//-----------------------------------------
function SetCookieEZ(name, value) {
  document.cookie=name+"="+escape(value);
}
//-----------------------------------------
// GetCookie - Returns the value of the
//    specified cookie or null if the
//    cookie doesn't exist
```

It seems that typesetters think that a typewriter is the best tool to prepare readable and clear programs. This situation reminds us of an old Polish proverb: "The shoemaker does not wear shoes". Why? Because it is inexplicable that the typesetter does not typeset programs. Why do they use verbatim mode of typesetting — a kind of 'ASCII typography'? Do they have nothing better? Although typography is well developed, that of computer code lags far behind. However, there are rules for typographic formatting of computer code. Developing excellent computer code typography was pioneered by two people: Peter Naur and Myrtle Kellington, who set the standards that were adopted by many computer journals [8, 6]. But it seems that they are no longer used at all.

Editor's note: What does the citation 6 in the above paragraph signify? — it's a citation of Knuth

In the next section I will try to analyze why, and what makes code typesetting so difficult. In the following one, I will present my idea of a prettyprinting tool, which combines Knuth's [3] and Oppen's [9] approaches. One example is worth a thousand words, so the last section presents two longer examples typeset by my tool.

The term 'prettyprinting', which goes back to 1975 book *Programming Proverbs* by Henry Ledgard, is nowadays used instead of 'code typesetting'.

Prettyprinting HOWTO

Obviously ASCII typography is what programmers are familiar with and see most often. Look at something less ugly.

The following code uses the extended features of BC to implement a simple program for calculating checkbook balances.

```
print "Check book program!\n";
print "  Exit by a 0 transaction.\n\n";
print "Initial balance? ";
bal = read();
```

```
  bal /= 1;
  print "\n";
  while (1) {
    "current balance = "; bal
    "transaction? "; trans = read()
    if (trans == 0) break;
    bal -= trans
    bal /= 1
```

Although the structure of the code is clearly laid out, the beginner will have problems with recognizing the elements which are predefined part of the language. Adding some typography to this example solves this problem and makes the program clearer.

The following code uses the extended features of BC to implement a simple program for calculating checkbook balances.

```
    print "Check book program!\n";
    print "  Exit by a 0 transaction.\n\n";
    print "Initial balance? ";
    bal = read( );
    bal /= 1;
    print "\n";
    while (1) {
      "current balance = "; bal
      "transaction? "; trans = read( )
      if (trans ≡ 0)  break;
      bal −= trans
      bal /= 1
    }
```

A simply bit of typography makes the difference. So the question is: why is it not used? Adding typography means adding mark-up to the source code. This makes the source for the example look like

```
The following code uses the extended
features of |\acro{BC}| to implement a
simple program for calculating checkbook
balances.
\startPP[BC]
\PPK{print}\PPbreakspace \PPS{"Check\ book\
\PPK{print}\PPbreakspace \PPS{"\ \ Exit\ by
\PPK{print}\PPbreakspace \PPS{"Initial\ bal
\PPV{bal}\PPequal \PPK{read}\PPbraceleft \P
\PPV{bal}\PPslasheq \PPN{1}\PPsemicolon \PP
\PPK{print}\PPbreakspace \PPS{"\\n"}\PPsemi
\PPK{while}\PPspace \PPbraceleft \PPN{1}\PP
  \PPS{"current\ balance\ =\ "}\PPsemicolon
  \PPS{"transaction?\ "}\PPsemicolon \PPspa
  \PPK{if}\PPspace \PPbraceleft \PPV{trans}
  \PPV{bal}\PPminuseq \PPV{trans}\PPforce
  \PPV{bal}\PPslasheq \PPN{1}\PPforce
  \PPbackspace \PPparenright
\stopPP[BC]
```

We can see that it is not that simple. Markup requires consistency and time which people do not have, whereas computers have both. So, why not use computers? — it seems possible, because computer languages, unlike natural languages, are unambiguously described by grammars. So we can try to use grammars to control the process of marking-up code. For example, in C or PERL, one grammar rule for *statement* says that *statement* is build of the opening brace '{' followed by a *statement_list* and the closing brace '}', which could be concisely written as:

$$\textit{statement} \ \rightarrow\ \texttt{\'\{\'}\ \textit{statement\_list}\ \texttt{\'\}\'}$$

Now, assume that we want to typeset braces on separate lines with *statement_list* typeset indented between them

```
{
  statement_list
}
```

This could be done in the following way: after recognizing the statement components, we typeset the opening brace followed by the newline; next we typeset indented each statement from the statement list; next we typeset the newline followed by the closing brace.

Editor's note: Propose above paragraph be replaced, to save enough space that the paper doesn't run to 6pp; wording would be (note that the operations described also appear later on): This could be done very simply once the syntactic element has been recognized.

However, prettyprinting is not that simple, because programmers use both syntax and semantics to make their programs clearer and more readable. This makes the task of building a wholly automatic prettyprinting tool impossible. Fortunately, the conflicts between syntax and semantics are sufficiently rare that it is acceptable to require the user to override syntax-based decisions when necessary.

Building a tool

Many people have tried to formalize and implement the idea of prettyprinting. William McKeeman [11] was the first to present a prettyprinting algorithm. Oppen [9] came up with the idea of using the original grammars to control the marking-up process.

Knuth's [3, 12] approach is based on his so-called 'format primitives' listed below.

indent	indent the next line one more notch
outdent	indent the next line one less notch
optbreak	optional line break
backspace	backspace one notch
breakspace	optional break or space
force	force line break
bigforce	force line break and a little extra vertical space
noindent	no indentation

Spacing in expressions is inherited from the TEX mathematical mode.

Knuth does not use the original PASCAL or C grammars to derive prettyprinting grammars. For each language, a specially designed and optimized context sensitive grammar is created. These grammars do not describe reference languages, which means that some pieces of correct code would not be prettyprinted and incorrect code would be prettyprinted. Markup is done by manually built parsers. Using parsers for building prettyprinters is a double-edged sword: comments are lost during parsing, and while prettyprinting they have to be put back. Knuth's tools recover comments too.

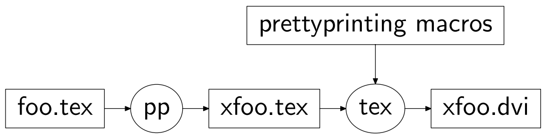
The main difficulty with Knuth's approach lies in creating prettyprinting grammars. So far only two such nontrivial grammars have been created. It took 10 years for the prettyprinting C grammar to evolve. Therefore, I think that the Oppen's idea of enhancing the original grammars is easier to implement.

For example, look at the rule for *statement* above and assume, that we want to typeset statement list indented within braces. The following rule will do:

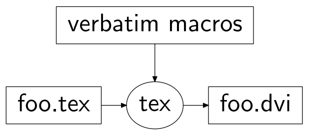
statement → ´`{`´ *indent force statement_list*
force backspace ´`}`´ *outdent force*

This reads as follows: print the opening brace `{` and prepare to *indent* following lines; next break the line with *force* and typeset *statement_list*; now, break the line again with *force*, *backspace* and print the closing brace `}`; finally, remove with *outdent* the indentation for the following lines and *force* the line break again.

Combining the ideas above I designed a prettyprinting tool named pp which works as a preprocessor for TEX.



It is also possible to typeset code verbatim — simply typesetting as it used to be.



Examples

There are many computer languages in use nowadays. Some of them I use on a day to day basis, some I use occasionally, some I want to look at, and still others are used by my colleagues. Therefore I want my tool to be able to learn as many languages as possible. The current version knows how to typeset only two languages: EBNF — extended Backus-Naur formalism and BC — the language of the binary calculator (a tool available on every UNIX system), because it is still being developed.

Someone said "One picture is worth one thousand of words," so I want to end up with two pieces of prettyprinted code. Because there is no general agreement how to typeset computer documents, as opposed to mathematical ones, some readers may like these examples and others not. Moreover, mathematicians, have a well developed notation which is supported by suitable fonts. This may explain why these examples are not as clear and readable as they could be.

Source: Ken Arnold, James Gosling.
"The Java Programming Language."
Addison Wesley Longman, Inc. 1996.

Consider the two groups of productions:

FieldDeclaration → {*FieldModifiers*}∗ *Type*
VariableDeclarators
;
FieldModifiers → *FieldModifier*
| *FieldModifiers FieldModifier*
;
FieldModifier → keywords common for field and method
| `transient` | `volatile`
;

and:

MethodHeader → {*MethodModifiers*}∗
ResultType MethodDeclarator {*Throws*}∗
;
MethodModifiers → *MethodModifier*
| *MethodModifiers MethodModifier*
;

```
  MethodModifier → keywords common for field
        and method
    | abstract | native | synchronized
    ;
```

where common keywords consists of: `public`, `protected`, `private`, `final`, `static`.

Source: Philip A. Nelson.
LIBMATH.B.
The arbitrary precision math library for the BC calculator.

To compute exponential we use the fact that $e^x = (e^{x/2})^2$. When x is small enough, we use the series: $e^x = 1 + x + x^2/2! + x^3/3! + \dots$.

```
scale = 20;
define e(x) {
  auto a, d, e, f, i, m, n, v, z
  ▸ a — holds $x^y$ of $x^y/y!$
  ▸ d — holds $y!$
  ▸ e — is the value $x^y/y!$
  ▸ v — is the sum of the e's
  ▸ f — number of times x was divided by 2.
  ▸ m — is 1 if x was minus.
  ▸ i — iteration count.
  ▸ n — the scale to compute the sum.
  ▸ z — orignal scale.
  ▸ Check the sign of x.
  if (x < 0) {
    m = 1; x = −x
  }
  ▸ Precondition x.
  z = scale;
  n = 6 + z + .44 ∗ x;
  scale = scale(x) + 1;
  while (x > 1) {
    f += 1; x /= 2; scale += 1;
  }
  ▸ Initialize the variables.
  scale = n;
  v = 1 + x
  a = x
  d = 1
  for (i = 2; 1; i++) {
    e = (a ∗= x)/(d ∗= i)
    if (e ≡ 0) {
      if (f > 0) while (f −−) v = v ∗ v;
      scale = z
      if (m) return (1/v);
      return (v/1);
    }
    v += e
  }
}
```

Define the logarithm function.

```
define l(x) {
  auto e, f, i, m, n, v, z
  ▸ Return something for the special case.
  if (x ≤ 0) return ((1 − 10ˆscale)/1)
  ▸ Precondition x to make .5 < x < 2.0.
  z = scale; scale = 6 + scale;
  f = 2; i = 0;
  ▸ For large numbers.
  while (x ≥ 2) {
    f ∗= 2; x = sqrt(x);
  }
  ▸ For small numbers.
  while (x ≤ .5) {
    f ∗= 2; x = sqrt(x);
  }
  ▸ Set up the loop.
  v = n = (x − 1)/(x + 1)
  m = n ∗ n
  ▸ Sum the series.
  while (x < 2) {
    e = (n ∗= m)/i
    if (e ≡ 0) {
      v = f ∗ v
      scale = z
      return (v/1)
    }
    v += e
  }
}
```

The sin function uses the standard series: $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \dots$

```
define s(x) {
  auto e, i, m, n, s, v, z
  ▸ Precondition x.
  z = scale
  scale = 1.1 ∗ z + 2;
  v = a(1)
  if (x < 0) {
    m = 1;
    x = −x;
  }
  scale = 0
  n = (x/v + 2)/4
  x = x − 4 ∗ n ∗ v
  if (n % 2) x = −x
  ▸ Do the loop.
  scale = z + 2;
  v = e = x
  s = −x ∗ x
```

```
    for ($i = 3$; 1; $i$ += 2) {
      $e *= s/(i * (i - 1))$
      if ($e \equiv 0$) {
        $scale = z$
        if ($m$) return ($-v/1$);
        return ($v/1$);
      }
      $v$ += $e$
    }
  }
```

For arctan we use the formula: $\arctan(x) = \arctan(c) + \arctan((x - c)/(1 + xc))$ for small c (0.2 here). For $x \le 0.2$, use the series: $\arctan(x) = x - x^3/3 + x^5/5 - x^7/7 + \ldots$.

```
  define $a(x)$ {
    auto $a, e, f, i, m, n, s, v, z$
    ▸ $a$ is the value of $a$ (0.2) if it is needed.
    ▸ $f$ is the value to multiply by $a$ in the return.
    ▸ $e$ is the value of the current term in the
        series.
    ▸ $v$ is the accumulated value of the series.
    ▸ $m$ is 1 or $-1$ depending on $x$ ($-x \to -1$);
    ▸ results are divided by $m$.
    ▸ $i$ is the denominator value for series element.
    ▸ $n$ is the numerator value for the series
        element.
    ▸ $s$ is $-x \cdot x$.
    ▸ $z$ is the saved user's scale.
    $m = 1$;
    ▸ Negative $x$?
    if ($x < 0$) {
      $m = -1$; $x = -x$;
    }
    ▸ Special case and for fast answers
    if ($x \equiv 1$) {
      if ($scale \le 25$) return
              ($.7853981633974483096156608/m$)
    }
    if ($x \equiv .2$) {
      if ($scale \le 25$) return
              ($.1973955598498807583700497/m$)
    }
    ▸ Save the scale.
    $z = scale$;
    ▸ Note: $a$ and $f$ are known to be zero due to
        being auto vars. Calculate arctan of a
        known number.
    if ($x > .2$) {
      $scale = z + 5$; $a = a(.2)$;
    }
    ▸ Precondition $x$.
    $scale = z + 3$;
    while ($x > .2$) {
      $f$ += 1; $x = (x - .2)/(1 + x * .2)$;
    }
    ▸ Initialize the series.
    $v = n = x$;
    $s = -x * x$;
    ▸ Calculate the series.
    for ($i = 3$; 1; $i$ += 2) {
      $e = (n *= s)/i$;
      if ($e \equiv 0$) {
        $scale = z$;
        return ($(f * a + v)/m$);
      }
      $v$ += $e$
    }
  }
```

References

[1] Blashek, Günter and Johannes Sametinger. “User-Adaptable prettyprinting.” *Software — Practice & Experience* **19** (July 1989).

[2] Gärtner, Felix. 1998. *The PretzelBook.* Available online as part of the PRETZEL distribution, in directory `~gaertner/pretzel` at `www.iti.informatik.th-darmstadt.de`

[3] Knuth, Donald E. 1983. “The WEB System of Structured Documentation.” Computer Science Report 980. Stanford University. Available online as part of the WEB distribution, in file `~web/doc/webman.tex` at `ftp.dante.de`.

[4] Knuth, Donald E. “Literate Programming”, *The Computer Journal* **27** (1984). pp. 97–111.

[5] Knuth, Donald E. 1986. “How to read a WEB.” Appeared in *Computers & Typesetting*, Volume B. Addison-Wesley.

[6] Knuth, Donald E. 1992. *Literate Programming.* Center for the Study of Language and Information Leleand Stanford Junior University.

[7] Knuth, Donald E. 1999. *Digital Typography.* Center for the Study of Language and Information Leleand Stanford Junior University.

[8] Naur, Peter [ed.] et al. “Report on the algorithmic language ALGOL 60.” *Communications of the ACM* **3** (May 1960). pp. 299–314.

[9] Oppen, Derek C. “Prettyprinting.” *ACM Transactions on Programming Languages & Systems* **2** (1980). pp. 465–483.

[10] Ramsey, Norman. 1988. “A SPIDER user's guide.” Technical Report, Princeton

University. Available online as part of the SPIDER distribution, in file `~web/spiderweb/doc/spiderwebman.tex` at `ftp.dante.de`.

[11] McKeeman, William. "Algorithm 268." Communications of the ACM **8** (1965). pp. 667–668.

[12] Knuth, Donald E. and Silvio Levy. 1990. "The CWEB System of Structured Documentation." Computer Science Report 1336. Stanford University. Available online as part of the CWEB distribution, in file `~web/c_cpp/cweb/cwebman.tex` at `ftp.dante.de`.

Włodek Bzyl