CGM: A context-free grammar manipulator*

André Almeida José Alves
{bernarduh,c0507049}@alunos.dcc.fc.up.pt

Nelma Moreira Rogério Reis
{nam,rvr}@ncc.up.pt

DCC-FC & LIACC, Universidade do Porto
R. do Campo Alegre 1021/1055, 4169-007 Porto, Portugal

Abstract. We present an interactive graphical environment for the manipulation of context-free languages. The graphical environment allows the editing of a context-free grammar, the conversion to Chomsky normal form, word parsing and the (interactive) construction of parse trees. It is possible to store positive and negative datasets of words to be tested by a given grammar. The main goal of this system is to provide a pedagogical tool for studying grammar construction and parse trees.

1 Introduction

Context-free languages are fundamental computer science structures and efficient software tools are available for their representation and manipulation. But for experimenting, studying and teaching their formal and computational models it is useful to have tools for manipulating them as first-class objects. Automata theory and formal languages courses are mathematical in essence, and traditionally are taught without computers. Well known advantages of the use of computers in education are: interactive manipulation, concepts visualisation and feedback to the students.

In this paper, we describe an interactive graphical environment, implemented in Python [vR05], for manipulation of context-free languages. The use of Python, a high-level object-oriented language with high-level data types and dynamic typing, allows us to have a system which is modular, extensible, clear, easy to implement, and portable. The Python interface to the `Wxwidgets` graphical toolkit [SRZD] gives a good platform to build a graphical environment. This work is part of the **FAdo** project which aims the development of interactive environments for symbolic manipulation of formal languages [MR,MR05], that includes already a toolkit for the manipulation of finite automata and regular expressions [MR05], a Turing machine simulator and an LR parser generator, `Yappy` [RM03].

The context-free grammar manipulator now presented allows the editing of a context-free grammar, the conversion to Chomsky normal form, word parsing

* This work was partially funded by Fundação para a Ciência e Tecnologia (FCT) and Program POSI.

and the (interactive) construction of parse trees. It is possible to store positive and negative datasets of words to be tested by the grammar, respectively. Grammar ambiguity will the detected if an inputed word has several parse trees, and all those parse trees will be shown.

The paper is organized as follows. In the next section we review the basic concepts of context-free languages and context-free grammars. In Section 3 we describe the parsing algorithm that was implemented and discuss how it can be used to allow the interactive construction of parse trees. Section 4 describes the **CGM** interactive graphical environment. Section 5 concludes with a short discussion of related work and some future work.

2 Context-free languages

We assume basic knowledge of formal languages and automata theory [HMU00]. An alphabet Σ is a nonempty set of symbols. A word over Σ is a finite sequence of symbols of Σ. The empty word is denoted by ϵ. The set $\Sigma^\star$ is the set of all words over Σ. A language L is a subset of $\Sigma^\star$.

A *context-free grammar* (or just *grammar*, from now on) is a tuple $G = (V, \Sigma, P, S)$ where V is a non empty set of *non-terminals*, Σ is the alphabet or set of *terminals* $\Sigma \cap V = \emptyset$, $S \in V$ is the *start symbol* and $P \in V \times (V \bigcup \Sigma)^*$ a set of *rules*. If $(X, w) \in P$ we write $X \to w$. If there are several rules with the same non-terminal X, we can write $X \to \alpha_1 \mid \cdots \mid \alpha_n$.

Given α, $\beta \in (V \bigcup \Sigma)^*$, β is *derivable* from α on one step, $\alpha \Rightarrow_G^1 \beta$, if there exist α_1, $\alpha_2 \in (V \bigcup \Sigma)^*$ such that $\alpha = \alpha_1 X \alpha_2$, $\beta = \alpha_1 \gamma \alpha_2$ and $X \to \gamma \in P$. The relation derivation on n steps, $\Rightarrow_G^n$, is defined by induction in $n \geq 0$, and $\Rightarrow_G^*$ (*derivation*) is the reflexive and transitive closure of $\Rightarrow^1$.

The language generated by a grammar G is defined by

$$L(G) = \{x \in \Sigma^* \mid S \Rightarrow_G^* x\}.$$

A parse tree for $x \in L(G)$ is a tree such that:

- The root is S.
- Each node is labeled by a symbol of $V \bigcup \Sigma \bigcup \{\varepsilon\}$
- A label of a internal node is a non-terminal
- The label of the leaves are terminals or ε
- If an internal node has label X and has n sons with labels $X_1, X_2, ..., X_n$, respectively, then $X \to X_1 X_2 \cdots X_n \in P$
- x is the concatenation of the leaves labels (ordered from left to right) (also called the *yield* of the tree).

To each word derivation corresponds a parse tree. A grammar G is said *ambiguous* if there exists more than one parse tree for some $x \in L(G)$. The grammar $G = (S, (,), S \to (S) \mid SS \mid \varepsilon, S)$, that generates a language of balanced parenthesis, is ambiguous, as is illustrated in Figure 1.

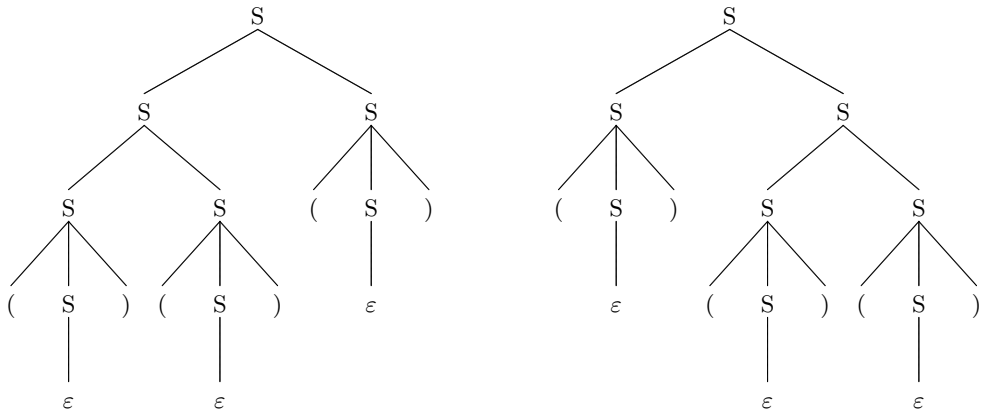

Fig. 1. Two distinct parse trees for the word "()()()".

3 Parsing

There are several parsing methods for determining if a word x is generated by a grammar, and, if so, to produce the parse trees [GJ90]. Applications of grammars for programming languages development, namely compilers, normally require that grammars are non-ambiguous (otherwise several semantics can be given to a word). Efficient (linear) parsing algorithms are well-known for sub-classes of context-free languages as the ones generated by LR-grammars [ASU86].

Here we wanted a parser for a generic context-free grammar that would allow the interactive construction of parse trees.

There are two main approaches to parse a word:

top-down building the parse tree from the start symbol down to the *yield* (the word)

bottom-up building the parse tree from the leaves (the word) up to the start symbol

We choose a bottom-up parsing method: the CYK algorithm due to J. Cocke, D. H Younger and T. Kasami [HMU00].

This algorithm implies that the grammar is in Chomsky Normal Form (CNF), i.e, all rules either have the form $A \to a$ or $A \to BC$, where A, B and C are non-terminals and a is a terminal.

Every grammar G can be transformed into a CNF grammar that generates $L(G) \setminus \{\epsilon\}$. For that the following steps must be taken:

- Eliminate ϵ-rules ($A \to \epsilon$);
- Eliminate unitary rules ($A \to B$);
- Eliminate useless symbols, i.e., non-terminals that do not contribute to the derivation of any word;
- Ensure that all rules has right-hand sides with two non-terminals or one terminal.

All these steps can be automatically implemented.

Given a CNF grammar G and an input word x, the CYK algorithm constructs a table that indicates which non-terminals derive which subwords of the input word. This is the recognition phase. In a second phase, the table and the grammar are used to construct all possible parse trees (derivations).

The algorithm decides if $x \in L(G)$ by analysing all its subsequences (subwords) of $x = x_1 x_2 \dots x_n$. For all $1 \leq i \leq i+s \leq n = |x|$, let $N[i, i+s]$ be the set of non-terminals that derive the subword $x_i \dots x_{i+s}$, i. e.

$$N[i, i+s] = \{A \mid A \Rightarrow^\star x_i \dots x_{i+s}\}$$

CYK Recognition Algorithm:

```
For i=1 to n:
      N[i,i]={A| A→ x_i}
For s=1 to n-1:
      For i=1 to n-s:
          N[i,i+s]=∅
          For k=i to (i+s)-1:
              N[i,i+s] = N[i,i+s]∪
                     {A| A → BC, B ∈ N[i,k] and C ∈ N[k+1,i+s]}
```

The word x is in $L(G)$ iff $S \in N[1, n]$.

Constructing the parse trees of the original grammar

The transformation of the grammar in CNF can eliminate some non-terminals of the original grammar, because either they are not useful or as result of the elimination of ϵ-rules. But it is possible to recover the removed non-terminals and add that information when building the CYK recognition table. In our implementation we build a list of tuples that relates the original non-terminals with the ones that appear in the CNF grammar.

When building a parse tree, each occurrence of a new symbol in the recognition table is substituted by the respective original symbol.

To verify if a word belongs to the language, the program checks if the start symbol occurs $N[1, n]$. If so it begins to build the possible parse trees (as tuples of trees). If not, the program will try to find trees for the subwords, that in the worst case, will be the trees for each symbol of the word.

4 Interactive visualization

The interactive graphical environment **CGM** has the following components and functionalities:

- Console and grammar information panel
- Grammar Editor

- Edition of a grammar
- Compilation of a grammar
- Load and save a grammar
- Transformation into CNF

– Words Lists
- Input or remove a word
- List of words generated by the grammar
- List of words not generated by the grammar

– Parse Tree Editor
- View all parse trees associated with a word
- Interactive construction of a parse tree
- Save and load parse trees

Several grammars can be manipulated simultaneously by using different windows (`tabs`). In Figure 2 we present the general view of the **CGM** application, with the main five working areas identified. The **CGM** graphical interface was implemented using the `wxWidgets` toolkit for Python.

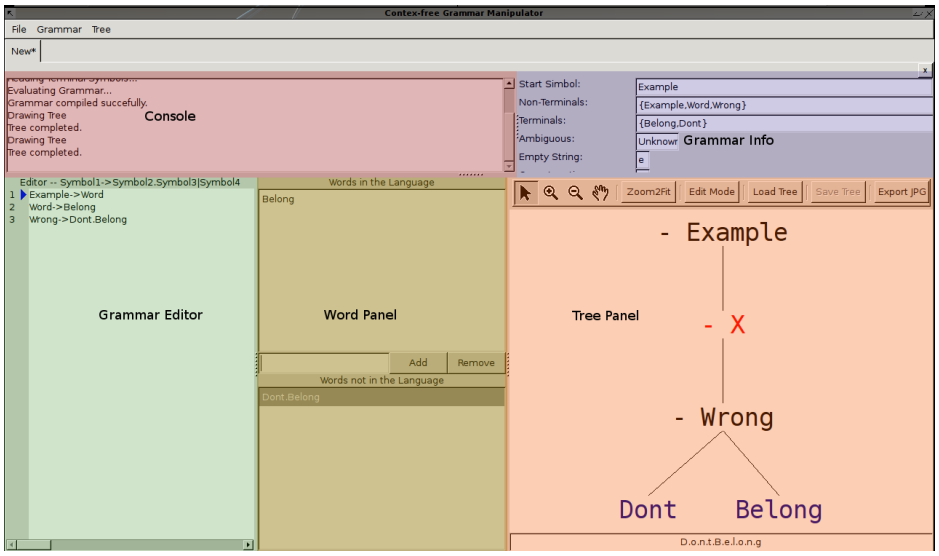

Fig. 2. The CGM interactive graphical environment.

In the following we will describe each of the components and some implementation details.

4.1 Console and grammar info panel

The **console** logs the events that are performed by the user and shows some extra information.

The **grammar info panel** provides information about a compiled grammar: the start symbol, the set of terminals and the set of non-terminals. Each grammar has also associated special characters that indicates the empty word and the concatenation symbol that allows the separation of grammar symbols.

4.2 Grammar editor

The **grammar editor** allows the introduction of the grammar rules. Each rule is of the form $X \rightarrow \alpha_1 | \cdots | \alpha_n$, where the rule separator is | and each α_i is a sequence of symbols (terminals or non-terminals) separated by a special concatenation symbol (by default, '.'), that do not belong to the grammar symbols.

An example of a rule is:

```
Num -> Int.Digit | Digit | Int.Frac
```

Note that the concatenation symbol will be also used for the input words. For instance `1.2.,.3.0` will represent the number $12, 30$. By default the empty word will be represented by `e`, but both of the special symbols (concatenation and empty word) are configurable. If a rule is not well-formed the editor will detect and a red bullet will appear in the correspondent line.

There is no need to introduce the start symbol and the sets of terminals and of non-terminals. Whenever the grammar is compiled those symbols are inferred by the set of rules. By default, the start symbol corresponds to the symbol in the left-hand side of the first rule. But the user can specify the start symbol by setting the cursor on the desired rule and pressing `ctrl+i` (or using the menu `Grammar>Set Start Symbol`). The line containing the rule with the start symbol will be marked with a blue arrow.

If option `auto-compile` is enabled, the grammar will be automatically compiled when the editor loses focus. If `auto-compiled` is disabled, the grammar can be compiled by selecting the menu `Grammar>Compile`. If a grammar is successfully compiled, some information appears in the **grammar info panel** and words can now be inputed for parsing.

The text editor is based on the `wxStyledTextCrl` class, that implements the editing component `Scintilla` [Gro07,Sas03]. This component provides all the usual editing facilities (select, copy, cut, paste, undo, etc) that are available by pressing the left mouse button.

A grammar can be transformed into Chomsky Normal Form (CNF) by selecting the menu `Grammar>Normal Form`. A new window (`tab`) will be opened with the new grammar.

A grammar can be saved or loaded (by selecting the menu `File`). The grammar filename will have the extension `.cgm`, and it is saved as a persistent Python object (`shelve`). Besides the grammar information, those files will also store the datasets of words and the parse trees associated with the grammar.

In Figure 3 we present a grammar for simple arithmetic expressions, some datasets of words and the parse tree of the current word.

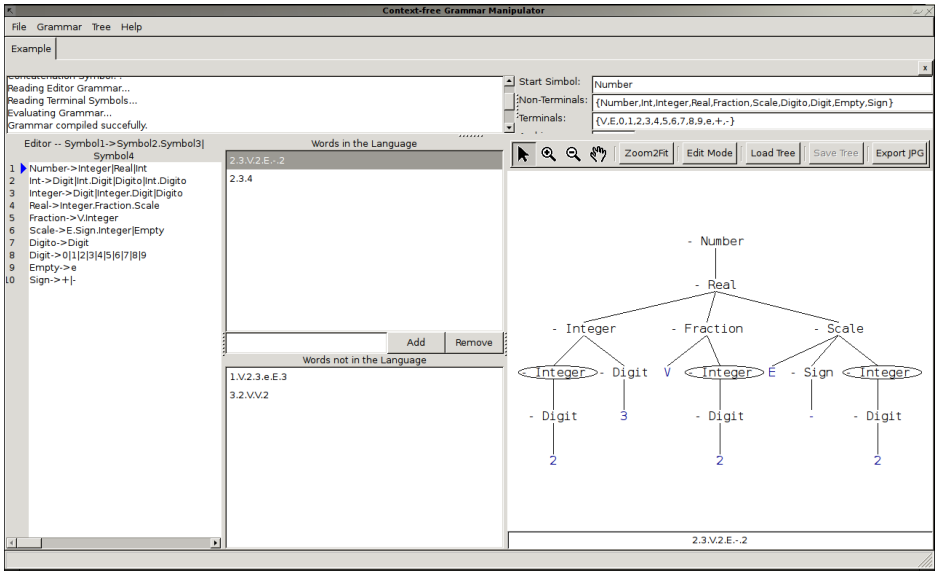

Fig. 3. An example of a grammar and the parsing of words.

4.3 Words lists

In order to write a correct grammar for a given language it is helpful to have positive and negative datasets of words that should be generated by the grammar or not, respectively.

A word (sequence of symbols) can be inputed and by pressing the `add` button (or `enter` key) the word will be parsed by the grammar. An error is reported if any symbol of word is not a terminal of the grammar. If the word is generated by the grammar it will be added to the positive dataset, otherwise to the negative one. If the word is generated by the grammar then a parse tree will be drawn in the **parse tree editor** area, as is exemplified in Figure 3.

A word selected can be removed from the corresponding dataset, by pressing the `remove` button. To remove all words (from both datasets) the user must select the menu `Grammar>Erase all words`.

4.4 Parse tree editor

The **parse tree editor** allows the visualization and interactive construction of parse trees associated with a currently selected word, that appears in the bottom panel of the editor. It is constituted by a canvas and a toolbar (see Figure 3). It is implemented using the `Floatcanvas` component [Bar07] of the `wxWidgets` toolkit. The `Floatcanvas` is an object canvas that allows high-level manipulation of graphical objects and has built-in zoom.

The toolbar buttons have the following functionalities:

`+:` Zoom in mode, in the canvas
`-:` Zoom out mode, in the canvas
`Zoom2Fit:` Adjusts the view window to the tree
`Hand` Drag mode, the user can drag the view window
`Edit/View mode:` Switch between edit and view mode (see below)
`Load tree:` If the user saved any trees, they can be loaded (requires Edit mode)
`Save tree:` Saves the current parse tree (requires Edit mode)
`Export JPG:` Exports the parse tree to a jpg file

The **parse tree editor** has two modes of visualization and interaction: the **view mode** and the **edit mode**.

View mode

Allows the visualization of the parse trees associated to the current word. The parse trees were previously constructed by the parser, as described in Section 3. If there are more then one, the grammar is *ambiguous* and that fact is indicated in the **grammar info panel**.

In this mode the **parse tree editor** functionalities are the following ones.

- The sub-trees associated to each non-terminal can be expanded or collapsed (by pressing left mouse button). Double-click expands all sub-trees of that node.
- If a node has an ellipse around it, it means that the tree has more than one (valid) subtrees at that node. Right-clicking on the node, shifts between the several (sub-) parse trees.
- A red `X` means that it is impossible to reach the yield from the root (which means that the word is not generated by grammar).

In Figure 4 we show a tree where the nodes labelled by the non-terminal `Integer` have several sub-trees.

Edit mode

Allows the interactive construction of parse trees. This is an innovative and important feature of this application because it can help learning the concept of word parsing. Its functionalities are illustrated in Figure 6 and are as follows:

- Left-clicking on a non-terminal makes a bar pop-up with the possible right-hand symbols of the rules that are associated with that non-terminal. Selecting one symbol (non-terminal or terminal) will allow the construction of the tree (and the associated derivation)
- Right-clicking on a non-terminal removes its sub-tree.

In this mode, it is also possible to save a tree and to load it latter. Figure 6 illustrates how we can load one tree previously saved.

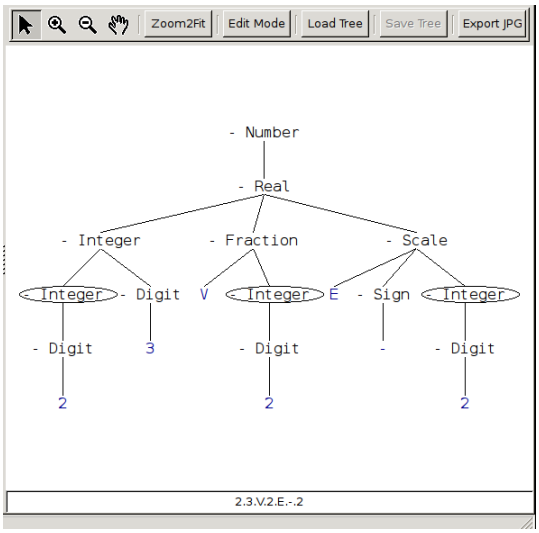

Fig. 4. The Parse Tree Editor in **view mode**.

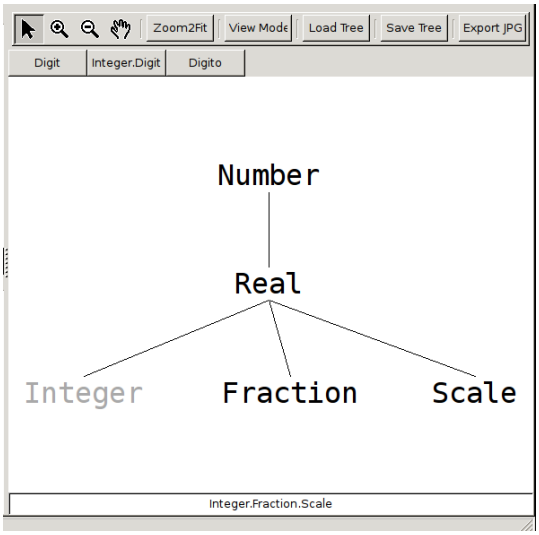

Fig. 5. Interactive building of a parse tree: the second bar has the possible choices for the current node `Integer`.

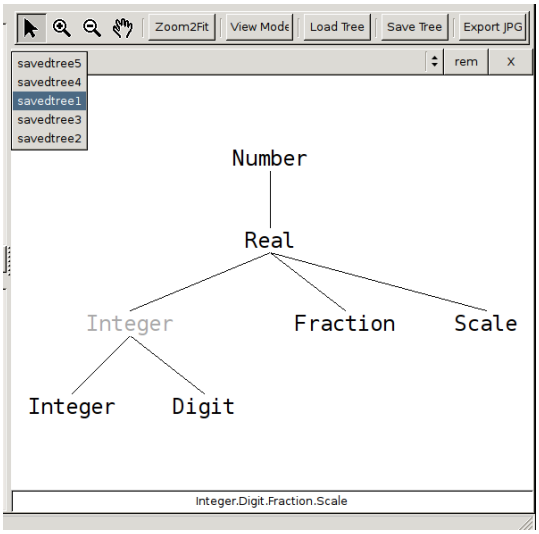

Fig. 6. Parse tree editor with several saved trees.

5 Conclusions

In this paper we presented an interactive graphical environment that allows the edition of context-free grammars and the parsing of words. Its main goal is to provide a pedagogical tool for studying grammar construction and parse trees.

Similar projects are, besides others, the JFLAP[BLP+97] and the Gaminal project[DK00]. Both have graphical symbolic environments to manipulate grammars. Comparing with our approach JFLAP, although more complete, has a less user-friendly grammar editor and lacks some of our functionalities. Gaminal is more oriented to teaching compiler design.

Concerning future work, we would like to study ways of automatic detection of grammar ambiguities. The graphical environment can also be improved and extended in several directions. In particular, by the implementation and visualization of more parsing algorithms and by supporting languages operations such as, union, concatenation and Kleene closure. Currently, we are already implementing a top-down LL1 parser. We also plan to integrate `Yappy` with **CGM** in order to provide a learning environment for LR parsing concepts. New versions of the system will be found its Web page (`www.ncc.up.pt/cgm`).

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[Bar07] Christopher Barker. Floatcanvas. `http://morticia.cs.dal.ca/FloatCanvas/`, 2007.

[BLP+97] Bilska, Leider, Procopiuc, Procopiuc, Rodger, Salemme, and Tsang. A collection of tools for making automata theory and formal languages come alive. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, 29, 1997.

[DK00] Stephan Diehl and Thomas Kunze. Visualizing principles of abstract machines by generating interactive animations. *Future Generation Computer Systems*, 16(7):831–839, 2000.

[GJ90] Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques - A Practical Guide*. Prentice Hall, 1990. http://www.cs.vu.nl/ dick/PTAPG.html.

[Gro07] Scintilla Project Group. Scintilla. `http://www.scintilla.org/`, Date of Access:2007.

[HMU00] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 2nd edition, 2000.

[MR] Nelma Moreira and Rogério Reis. FAdo: tools for formal languages manipulation. http://www.ncc.up.pt/fado.

[MR05] Nelma Moreira and Rogério Reis. Interactive manipulation of regular objects with FAdo. In *Proceedings of 2005 Innovation and Technology in Computer Science Education (ITiCSE 2005)*. ACM, 2005.

[RM03] Rogério Reis and Nelma Moreira. *Yappy:Yet another LR(1) parser generator for Python*. DCC-FC & LIACC, Universidade do Porto, 2003.

[Sas03] Jeff Sasmor. Yellow brain guide to wxpython. `http://www.yellowbrain.com/stc/index.html`, 2003.

[SRZD] Julian Smart, Robert Roebling, Vadim Zeitlin, and Robin Dunn. *wxWidgets 2.6.3: A portable C++ and Python GUI toolkit*.

[vR05] Guido van Rossum. *Python Library Reference*, 2.4.2 edition, 2005. `http://python.org`.