yaccviso — a tool for visualizing yacc grammars

Leon Aaron Kaplan
aaron@lo-res.org

October 14, 2006

This work was originaly performed during a "student praktikum" at the Institute for Computer Languages, Technical University of Vienna, Nov. 1997 - Jan. 1998.

Contents

1 Overview

Convention: whenever We refer to "yacc" in this paper, we mean both bison and yacc.
A general knowledge of yacc is assumed.
yaccviso is a tool for visualizing the dependencies of non-terminal and terminal symbols in a yacc grammar. Thus it should help the person developing a compiler in yacc to gain a fast overview over his or her grammar file. The idea we had in mind while developing yaccviso is that the person developing his compiler should be able to quickly generate a huge postscript poster of his or her grammar and have it attached to his wall so that whenever questions arise all he or she has to do is to turn around and take a look at the poster.

2 Syntax

Presently the invocation syntax is simple:
`yaccviso` [`-h`] [*debugoptions*] *inputfile*
where `-h` prints out a usage message and where *debugoptions* are optional and are one of

- `-d DDEBUG`, turns on all debugging messages
- `-d DFATAL`, only fatal error messages are printed
- `-d DSCANDBG`, debug messages while scanning are printed
- `-d DPARSEDBG`, debug messages while parsing
- `-d DSEMANALY`, debug messages during sematic analysis
- `-d DCODEGEN`, debug messages during output code generation
- `-d DSYMTAB`, debug messages concering symbol tables
- `-d DIO`, debug messages during input output
- `-d DWARNING`, warning messages

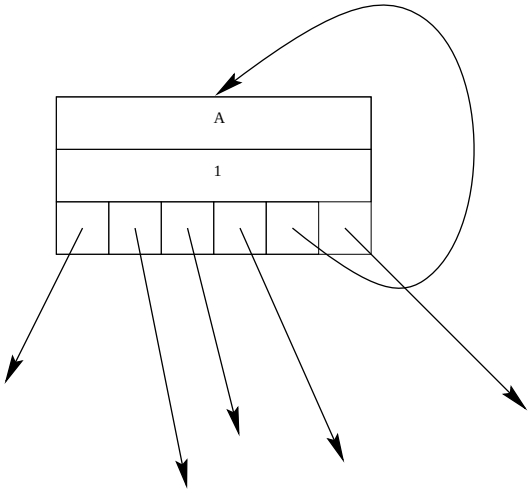
The defaults for debugging are: `-d DFATAL` and `-d DWARNING`. Other debug messages should be used with care as they generate a lot of output on stderr. If no input file is given, yaccviso will try to read from stdin.
yaccviso will generate two output files: `depgraph.vcg` and `depgraph.dot` the former is an input file for the VCG tool, the latter an input file for the dot tool.

3 How to interpret the dot output

The file `depgraph.dot` when fed into the dot(1) program via the line `dot -Tps -o` *outputfilename* `depgraph.dot` will be laid out by dot(1) in the following fashion:

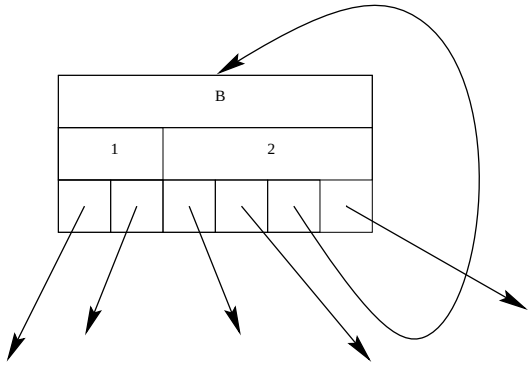
1. terminal symbols will be drawn in color (I used gold, but you can search and replace that with anything you wish of course, or modify the corresponding line in the file `vcg.c` - search for "gold")

2. a production consisting of a single line such as *A : A1 A2 A3 ...* will be printed like



where the "A" means that rule *A* depends on the boxes below and the "1" means that all the boxes below are terminal or non terminal symbols on the right hand side of *line one* as they appear in the order (i.e. A1 is before A2, etc.). Note the self edge in the above example.

3. a rule consisting of multiple lines / productions are drawn similar to



where the "1" and "2" refer to line one and line two. Note that it is possible that a rule can reference the same non terminal or terminal symbol on the right hand side *in different lines*. For example:
B : C D E
C F G;
C is used in both lines.

This scheme - every line of a rule spawns a box in which subboxes are used to point to the corresponding first, second, etc. dependency allows for a clear distinction of dependencies.

Here I have to thank Axel Belifante of the University of Twente, Holland for his usefull code examples.

4 Functionality

yaccviso caries out the fillowing steps when given an input file:

1. scanning: at this step comments are eliminated and ANSI C code as well as preprocessor code which can appear within %{ ... %} sections or within actions is passed to the parser as a specific semantic value (yylval). For this purpose a special mini-scanner was written in plain C. Thus the normal scanner interleaves its operation with this mini scanner. The normal scanner (written in flex) will call the mini scanner whenever neccessary. This way we achieve a seperation of languages and we don't have to scan ANSI C, preproccesor cammands and yacc files all in one flex scanner. The scanner builds up a symbol table.

2. parsing: accompanied in the distribution you will find a stripped yacc grammar file for parsing yacc files (new_grammar.txt). This grammar is the base for our parser. In addition the parser performs strict syntax error checking and will print out meaningful error messages (cf. follow set, [2]). Finally it controls the construction of a parse tree.

3. semantic analysis: the parse tree is traversed and neccessary information is extracted and inserted into a hash table with external extensions which in turn holds the information about the dependencies.

4. code emitter: a collection of a few routines which will search the dependencies hash table and emit the dependency information in two suitable languages: VCG and dot (depgraph.vcg, depgraph.dot).

5 Related work

yacc to dot (yacc2dot) - convert yacc files to dot graph descriptions
Author: Philippe Oechslin (`oechslindi.epfl.ch`)
version: 1.0 date: 3.23.95
This gawk programm produces a graph layout for the Dot program from a yacc grammar. However it has many restrictions that our approach does not have such as only one line per yacc rule. It is much smaller though. yacc2dot is included in the graphviz (i.e. dot) distribution.

6 Future Research

In future versions we want to add various well known algorithms from the field of compiler construction and graph theory such as dominator tree, strongly connected components, finding of cycles (i.e. to answer the question "which rules depend on themselves directly or indirectly") or subgraph starting at a specific rule.
Any suggestions are welcome.

A known bugs

1. `The`following does not work:
   #ifdef blabla
   %}
   #endif
   Neither does:
   #ifdef blabla
   }
   #endif
   Neither does:
   #ifdef blabla
   {
   #endif
   But the following works:
   #ifdef blabla
   { foobar }
   #endif

   In other words: these types of brackets must always be well formed within a #ifdef environment.

2. please report bugs as you see them accompanied with the input grammar.

B References

References

[1] VCG, Visualisation of Compiler Graphs, Georg Sander,
`ftp://ftp.cs.uni-sb.de/pub/graphics/vcg`

[2] Compiler Construction - Prinicples and Design. Aho, Sehti, Ullmann, Addison Wesley