Advanced Topics in Databases
Troels Arvin <troels@itu.dk>
Mandatory assignment, Code A2:

Implementation of the simplification procedure

Version 1, April 19, 2004.
Code, Makefile, and executables are available at http://troels.arvin.dk/itu/adb/simp/

Objective

The objective of this assignment is to implement the **simplification** procedure, as described in [1]:

> *Make an implementation in Prolog or Java of (part of) the simplification algorithm described during the course; test it on more complicated examples than those we used in the course. It will end up being a non-trivial piece of code and you should keep contact with one of the teachers to get some advice while working with this assignment. Subsumption is probably the most difficult part and could be left as an option if the group consists of only one person.*

I changed the implementation language to C++. Why C++? - I hoped to obtain some code which could – at least for experimental reasons – be incorporated into an existing relational database system with open source. The open source DBMSes that I know are written in C or C++, so it seemed most relevant to try a C or C++ implementation. I have no C experience; I do, however, have some C++ experience. Hence, the choice of C++.

Implementation choices

I envisioned a command line system where a constraint theory could be entered, along with an update expression, resulting in a simplified set of integrity constraints. The input would – for this assignment – be prolog-like predicates, connected by AND or ORs, followed by a ";", and an update expression. The predicates and dis/con-junctions may be grouped by parenthesis, and predicates and parenthesized groups may be negated by prefixing with "~". Constants should be written in lower case letters, variables in upper case letters, and parameters as a "?", followed by lower case letters. (In)equalities should be prifixed with "=" or "<>". White space may be placed between any element. An expression which the parser should be able to handle:

```
~(~p ( X , Y ,Z) AND x(b)) AND q(?a) OR v(j) ; p(?a,?b,?c)
```

First step would, therefore, be to implement a parser. I found Boost::Spirit[2] which looked like a suitable parser generation tool: I thought that learning Boost::Spirit would be easier than to manually parse the input, and easier than having to learn lexers like Yacc[3]. Boost::Spirit takes advantage of C++'s operator overloading features, and uses modern, template-driven "meta-programming" to enable Backus-Naur Form (BNF)-like syntax directly in the program code. The parser should transform the input into a parse tree, and further into a tree of relevant objects

(“object tree”).

The object tree would then be put through the three overall steps of the simplification procedure described in [1]: *after*, *norm*, *rsub*. The program should finally print the result.

The parser is described through a number of *rules* (see “grammer.h”):

- parameter
- constant
- variable
- term: parameter | constant | variable
- negation: ~
- double negation: ~~
- predicate: p (term)
- (in)equality: = or <>, term
- atom: optional negation, predicate or (in)equality
- literal: atom, or atom followed by a con/dis-junction
- literalgroup: parenthesized literal or literalgroup, optinally negated

The (in)equality is actually a special case of a predicate: X=Y is another way of writing a =(X,Y) predicate.

The rules are combined into two overall rules, one for the constraint theory *(literalgroup),* and one for the update expression. Those two overall rules are parsed by a root-rule, *entry*:
literalgroup:

- literalgroup: parenthesized literal or literalgroup, optinally negated
- update: predicate
- entry: literalgroup or literal, “;”, update

The parse tree is run though a set of functions that map rules to a (smaller) set of classes. The classes are defined in grammar_classes.h. The super-class of these is the abstract *pt_element* (parse-tree element). Terms, constants, and parameters have a class each. The *pred* class covers the *atom* parse-rule, i.e. it may be negated. The *pred* class has a *tailed_by* attribute which can be used to chain predicates. The *tail_conj* attribute tells if the chained predicate is to be interpreted as a conjunction (otherwise, it should be interpreted as a disjunction). The *pred_group* class is used to map parenthesized expressions. *pred_base* is an abstract base class of *pred* and *pred_base*.

A helper-class, *object_tree*, is used as a container for chained predicates/predicate groups. It keeps track of the head and the tail of the chain of predicates and predicate groups.

When an object tree has been constructed, it will be passed through a number of functions, corresponding to the steps of the simplification procedure.
Unfortunately, learning how to implement the parser, took several days, because I didn't have any prior experience with lexers or lexer-like parser generation. Hence, the initial objective of the program was not met.

The program could also use more comments, some restructuring, object

deletion (or use of *smart pointers*).

The resulting program

The program is written in ISO C++, and compiled on Linux, including a cross-compilation to a Windows-target. The compilation is governed by a Makefile.

The program, called `simp`, asks for the constraint theory and the update when started. Alternatively, the input may be passed through standard input. If the program is given any argument, then it runs in *debug*-mode, printing parse trees, object trees, etc., before the result. The result is printed on standard output, in much the same way as the input format. An example of running the program in debug mode:

```
./simp debug
Please enter denials and update, separated by ;
... or enter 'h' for help.
 ~(~p ( X , Y ,Z) AND x(b)) AND q(?a) OR v(j) ; p(?a,?b,?c)
Parse tree:
----------
Node with value '' has type 'entry', and has 2 children.
 Children nodes:
   Node with value '' has type 'literalgrouping', and has 3 children.
    Children nodes:
      Node with value '~' has type 'neg', and has 0 children.
      Node with value '' has type 'literal', and has 2 children.
       Children nodes:
         Node with value '' has type 'atom', and has 2 children.
          Children nodes:
            Node with value '~' has type 'neg', and has 0 children.
            Node with value 'p' has type 'pred', and has 3 children.
             Children nodes:
               Node with value 'X' has type 'variable', and has 0 children.
               Node with value 'Y' has type 'variable', and has 0 children.
               Node with value 'Z' has type 'variable', and has 0 children.
         Node with value 'AND' has type 'junc', and has 1 children.
          Children nodes:
            Node with value 'x' has type 'pred', and has 1 children.
             Children nodes:
               Node with value 'b' has type 'constant', and has 0 children.
      Node with value 'AND' has type 'junc', and has 1 children.
       Children nodes:
         Node with value '' has type 'literal', and has 2 children.
          Children nodes:
            Node with value 'q' has type 'pred', and has 1 children.
             Children nodes:
               Node with value 'a' has type 'parameter', and has 0 children.
            Node with value 'OR' has type 'junc', and has 1 children.
             Children nodes:
               Node with value 'v' has type 'pred', and has 1 children.
                Children nodes:
                  Node with value 'j' has type 'constant', and has 0
children.
   Node with value 'p' has type 'pred', and has 3 children.
    Children nodes:
      Node with value 'a' has type 'parameter', and has 0 children.
      Node with value 'b' has type 'parameter', and has 0 children.
      Node with value 'c' has type 'parameter', and has 0 children.

Result:
```

References

1. Christiansen H, Martinenghi, D:

Simplification of database integrity constraints revisited: a transformational approach.
2. http://www.boost.org/libs/spirit/
3. One of the many Yacc implementations is Berkeley Yacc (byacc): http://invisible-island.net/byacc/