Taking a closer look at parsing

As Unix tools, `bison` and `flex` shine most strongly perhaps when used for the very common task of parsing configuration files.

Taking a closer look at parsing

Another benefit of using the `bison` is that, while `bison` itself only supports C and C++, it is possible to use the same production rules to drive other systems such as the Perl module "Yapp" (Yet Another Perl Parser compiler).

Let's look at using `bison` for parsing a specific configuration file.

An example problem

Let's say that you want to create a program that moves data from one database to another on an ongoing basis (this is often called "replication.") In particular, it would be desirable to have such replication available between different types of databases, such MySQL, Postgresql, Oracle, and SQLite.

An example problem

You want this program to be flexible in how it works, so it makes sense to have a detailed configuration file associated with the data movement.

An example problem

The three main things you want to define for each replication activity are: data identifying the source of information to be replicated, data identifying the target of where this data is to be replicated, and perhaps some configuration information specifying such things as the frequency of the transfer.

Creating a configuration specification syntax

Let's start with the overall idea of replication. This type of definition often works well with a stanza format, so let's propose a stanza system something like:

First iteration

```
replication "ab-to-xy"
{
   # some sort of general information

   # now information about our source of data
   source "ab"
   {
   }

   # and information about our target
   target "xy"
   {
   }
}
```

Adding more detail

First, let's look at the general information.

It might be worth having such things as having some type of timing information; also, specifying logging activity might be a good idea. Let's try this syntax:

```
interval = 5 ;          # should we just assume seconds?
do_logging = 1 ;        # 1 = yes, 0 = no
log_identifier = "ab to xy transfer" ;
```

How does that look?

```
replication "ab-to-xy"
{
   # some sort of general information
   interval = 5 ;          # should we just assume seconds?
   do_logging = 1 ;        # 1 = yes, 0 = no
   log_identifier = "ab to xy transfer" ;

   # now information about our source of data
   source "ab"
   {
   }

   # and information about our target
   target "xy"
```

```
        {
        }
}
```

Source information

Okay, let's think about the source of data. There are some basic things that we need to know. What's the database type? Do we need to authenticate? If so, what's the authentication information?

Source information

```
type = postgresql ;
authentication_needed = 1 ;
authentication_type = password ;
username = username ;
password = "password" ;
database = "databasename" ;
all_tables = 0 ;
tables = [ table1 table2 table3 table4 ] ;
```

Target information

Now, what kind of information do we on the target side? How about something along the lines of:

```
type = sqlite ;
authentication_needed = 0 ;
database = "databasename" ;
```

Putting it all together

```
replication "ab-to-xy"
{
   # some sort of general information
   interval = 5 ;          # should we just assume seconds?
   do_logging = 1 ;        # 1 = yes, 0 = no
   log_identifier = "ab to xy transfer" ;

   # now information about our source of data
   source "ab"
   {
     type = postgresql ;
     authentication_needed = 1 ;
     authentication_type = password ;
     username = username ;
     password = "password" ;
     database = "databasename" ;
     all_tables = 0 ;
```

```
      tables = [ table1 table2 table3 table4 ] ;
    }

    # and information about our target
    target "xy"
    {
      type = sqlite ;
      authentication_needed = 0 ;
      database = "databasename" ;
    }
}
```

Now specifying a grammar for all of that

So what would we like to reflect? I think we should keep the idea of multiple replication definitions in a single configuration file open, so let's allow that possibility.

```
replication_declarations : | replication_declarations replication_declaration
                           ;

replication_declaration : REPLICATION
                          STRING
                          LEFTBRACE
                          variable_declarations
                          source_declaration
```

```
target_declaration
RIGHTBRACE
;
```

... variable declarations ...

```
variable_declarations : | variable_declarations simple_declaration
                        ;
```

... source declarations ...

```
source_declaration : SOURCE
                     STRING
                     LEFTBRACE
                     variable_declarations
                     RIGHTBRACE
                     ;
```

... target declarations ...

```
target_declaration : SOURCE
                     STRING
                     LEFTBRACE
                     variable_declarations
                     RIGHTBRACE
                     ;
```

... simple variable declarations ...

```
simple_declaration : NAME
                     EQUAL
                     value
                     SEMICOLON
                     ;

value : STRING | NUMBER | NAME | list
        ;
```

... and for some final bits ...

```
list : LEFTBRACKET elements RIGHTBRACKET
       ;

elements : element additional ;

additional : | element additional ;

element : STRING | NAME ;
```

Finally... !

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <linux/unistd.h>
%}
%token NAME STRING EQUAL SEMICOLON REPLICATION LEFTBRACE RIGHTBRACE SOURCE TARGET RIGHTB
%%
replication_declarations : |
                           replication_declarations replication_declaration
                           ;
replication_declaration : REPLICATION
                          STRING
                          LEFTBRACE
                          variable_declarations
                          source_declaration
```

```
                          target_declaration
                          RIGHTBRACE
                          ;
variable_declarations : | variable_declarations simple_declaration
                        ;
source_declaration : SOURCE
                     STRING
                     LEFTBRACE
                     variable_declarations
                     RIGHTBRACE
                     ;
target_declaration : SOURCE
                     STRING
                     LEFTBRACE
                     variable_declarations
                     RIGHTBRACE
                     ;
simple_declaration : NAME
                     EQUAL
                     value
                     SEMICOLON
                     ;
value : STRING | NAME | list
      ;
```

```
list : LEFTBRACKET elements RIGHTBRACKET
       ;
elements : element additional ;
additional : | element additional ;
element : STRING | NAME ;
%%
char *configuration_file = "replication.conf";
int main(int argc, char **argv)
{
  printf("Found %d arguments...\n",argc);

  // parse config file
  char opt;
  extern char *optarg;

  while((opt = getopt(argc,argv,"c:C:")) != -1)
    {
      switch(opt)
      {
      case 'c':
      case 'C':
        configuration_file = optarg;
        break;
      }
```

```
    }

  // read in configuration
  FILE *f = fopen(configuration_file,"r");
  if(f)
    {
      yyrestart(f);
      yyparse();
    }
  else
    {
      printf("Couldn't open %s!\n",configuration_file);
      exit(1);
    }

}
```

Additional ideas

In addition to the actual work behind parsing, I like to add a few bits to help: (1) add some provision to tell the user where the parser has failed and (2) do something to allow at least some version of comments.

Additional ideas

```
int yyerror(){printf("Configuration file error at line %d -- %s.\n",
                     yy_private_lines,
                     yy_private_error_guess); exit(1);}

    [ .  .  . ]

#.*\n                          { /* consume comments */ }
\n                             { yy_private_lines++; }
\/\/.*\n                       { yy_private_lines++; /* consume comments */ }
.                              { yy_private_error_guess = "syntax error"; yyerror(); }
```

Additional ideas

Then I like to add lines like these, so that the user can easily find syntactical errors:

```
replication_declaration :
                          {yy_private_error_guess = "Expecting keyword 'replication'";}
                          REPLICATION
                          {yy_private_error_guess = "Expecting string";}
                          STRING
                          {yy_private_error_guess = "Expecting '{'";}
                          LEFTBRACE
```