Bison parser expanding rule instead of reducing rule

Ask Question

0

Is it possible for a Bison rule to expand instead of reducing so that it turns into more tokens? Asked a different way: is it possible to insert extra tokens to be parsed before the next token in the parser input?

Here is an example where I might want this:

Suppose I want a parser that understands three token types. Numbers (just positive integers for the sake of simplicity - INT), words (any number of letters, upper or lower case STRING) and some kind of other symbol (lets use an exclamation mark for no good reason - EXC)

Suppose I have a rule that reduces a word followed by a number followed by an exclamation mark. This rule results in an integer type, let's say for now that it simply doubles its input. This rule also allows itself to be the integer that it parses.

I also have a rule to accept any number of these in a row (the start rule).

The Bison parser look like this: (quicktest.y)

```
%{
#include <stdio.h>
%}

%union {
    int INT_VAL;
}

%token STRING EXC
%token <INT_VAL> INT
%type <INT_VAL> somenumber

%%

    start: somenumber           {printf ("Result: %d\n", $1);}
         | start somenumber     {printf ("Result: %d\n", $2);}
         ;

    somenumber: STRING INT EXC           {$$ = $2 *2;}
              | STRING somenumber EXC    {$$ = $2 *2;}
              ;

%%

main(int argc, char ** argv){

    yyparse();

}

yyerror(char* s){
    fprintf(stderr, "%s\n", s);
}
```

The tokens can be generated with a flex lexer like so: (quicktest.l)

```
%{
    #include "quicktest.tab.h"
%}

%%

[A-Za-z]+               {return STRING;}
[1-9]+                  {yylval.INT_VAL = atoi(yytext); return INT;}
"!"                     {return EXC;}
.                       {}
```

This can be built with the following commands:

```
bison -d quicktest.y
flex quicktest.l
gcc -o quicktest quicktest.tab.c lex.yy.c -lfl -ggdb
```

I can now input something like this:

```
double double 2 ! !
```

and get the result 8

Now if I want the user to be able to avoid having lots of exclamation marks on one line, like this:

```
a b c d e f 2 ! ! ! ! ! !
```

I'd like to be able to allow them to input something like this:

```
a b c d e f 2 !*6
```

So I can add a flex expression for such a token that simply extracts the number of exclamations needed:

```
!\*[1-9]+               {
                            char *number = malloc(sizeof(char) * (strlen(yytext)-1))
                            strcpy(number, yytext+2);
                            yylval.INT_VAL = atoi(number);
                            free(number);
                            printf("Multiple exclamations: %d\n", yylval.INT_VAL);
                            return REPEAT_EXC;
                        }
```

But how would I implement the bison side of things?

I can add the token type like so:

```
%token <INT_VAL> REPEAT_EXC
```

And then a rule of some kind perhaps?

```
    repeat_exc: REPEAT_EXC      {/*expand into n exclamation marks (EXC tokens)*/}
              ;
```

Does Bison support this in any way?

If not how should I implement this?

Should I somehow have the lexer return the EXC token n times when it receives the repeat exc expression? (I'd rather avoid this if possible as this requires the flex code to keep record of some

kind of state, it could be in the repeat exclamation state or in a normal state. The lexer is then not as simple to maintain.)

parsing insert token bison

share improve this question

edited Aug 3 '14 at 17:02

Timofei Davydik
5,972 5 26 53

asked Aug 3 '14 at 14:35

DanJAB
361 4 19

add a comment

1 Answer

active oldest votes

1

That's really not possible in a context-free grammar.

It's not that difficult to do in a traditional lexer, but as you say it requires that the lexer maintain state. An easier approach is to use a push parser, where the parser is called from the lexer rather than the other way around. [Note 1]

The bison manual doesn't explain the API very well; if you declare a pure push parser, the interface you get is:

```
int yypush_parse(yypstate*, int, const YYSTYPE*);
```

or, if position-tracking is enabled:

```
int yypush_parse(yypstate*, int, const YYSTYPE*, YYLTYPE*);
```

I made fairly minimal changes to your example, in order to show the push_parser interface. First, the parser; the only differences are the `%define` directives to declare a push parser; the elimination of `main` (the lexer is now top-level), and the declaration of `yyerror` with an explicit `void` return type. [Note 2]

```
%{
    #include <stdio.h>
    void yyerror(char* msg);
%}

%define api.pure full
%define api.push-pull push
%union {
    int INT_VAL;
}

%token STRING EXC
%token <INT_VAL> INT
%type <INT_VAL> somenumber

%%

    start: somenumber           {printf ("Result: %d\n", $1);}
         | start somenumber     {printf ("Result: %d\n", $2);}
         ;

    somenumber: STRING INT EXC           {$$ = $2 *2;}
              | STRING somenumber EXC    {$$ = $2 *2;}
              ;

%%
```

```
void yyerror(char* s){
    fprintf(stderr, "%s\n", s);
}
```

The lexer has some more substantial changes, but I don't think the end result is any harder to read or maintain. It might even be easier.

- The macro `PARSE` sends a token with a specified type tag and value to `yyparse`; the macro `PARSE_TOKEN` sends a token without a semantic value.
- The `%options` line removes several warnings from the compile step
- The initialization of the parser state was added. (Indented lines after the `%%` and before any rule are inserted at the top of the lexer function, in this case `yypush_parse`, so they can be used to declare and initialize local variables.)
- The `INT` rule was changed to allow `10` to be a valid integer.
- The `!*<int>` rule was added.
- The `<<EOF>>` rule was added. (It's pretty well boiler-plate for lexer-driven push-parsing.)
- A `main` function was added, which calls `yylex`.

(Oh, and I changed a rule to avoid echoing new lines.)

```
%{
  #include "push.tab.h"
  #define PARSE(tok,tag,val) do { \
    YYSTYPE yylval = {.tag=val};  \
    int status = yypush_parse(ps, tok, &yylval); \
    if (status != YYPUSH_MORE) return status; \
  } while(0)
  #define PARSE_TOKEN(tok) do { \
    int status = yypush_parse(ps, tok, 0); \
    if (status != YYPUSH_MORE) return status; \
  } while(0)
%}
%option noyywrap nounput noinput

%%
                         yypstate *ps = yypstate_new ();

[A-Za-z]+               {PARSE_TOKEN(STRING);}
[1-9][0-9]*             {PARSE(INT,INT_VAL,atoi(yytext));}
"!*"[1-9][0-9]*         {int r = atoi(yytext+2);
                         while (r--) PARSE_TOKEN(EXC);
                        }
"!"                     {PARSE_TOKEN(EXC);}
.|\n                    {}
<<EOF>>                 {int status = yypush_parse(ps, 0, 0);
                         yypstate_delete(ps);
                         return status;
                        }

%%

int main(int argc, char** argv) {
  return yylex();
}
```

Notes

1. This is the style of the `lemon` parser generator. `lemon` was originally written to create the `sqlite` SQL parser but is used in various projects precisely for the convenience of the "push" interface. `bison`'s push-parser support is more recent, and very welcome.
2. I'm not crazy about `INT_VAL`; I prefer lower-case for union tags, but I was trying to minimize the diff.

share improve this answer

answered Aug 3 '14 at 20:09

rici
123k 12 100 163

This all looks great but... What version of Bison is the first to support this? With version 2.5 I get a compiler error saying invalid value for api.pure. The machine I am using is a little old and awkward to update, I will try it on a later machine as soon as I get the chance. Also, yes, I don't know why I made INT_VAL upper case. I usually do this lower too. – DanJAB Aug 4 '14 at 7:14

@DanJAB: I believe it was implemented in 2.4.1, but it is possible that the syntax for the declarations was different in that release. Consult the documentation included in the 2.5 version (hopefully it was installed). It's likely that the `%define api.pure full` declaration was added later; in this simple case, it should be sufficient to just `%define api.pure`. – rici Aug 5 '14 at 1:18

@DanJAB: To be more specific, the `full` declaration only affects the prototype for `yyerror`; the original `%define api.pure` declaration did not change `yyerror`'s prototype, which was later recognized to be a bad idea but could not be changed without losing backwards compatibility. – rici Aug 5 '14 at 1:38

Aha! Using a more up to date machine (and Bison 3.0.2) this compiles and runs without issue. Perfect. Thanks. – DanJAB Aug 5 '14 at 17:43

add a comment