编译原理报告——实验1

刘志刚
匡亚明学院
141242022
njuallen@foxmail.com

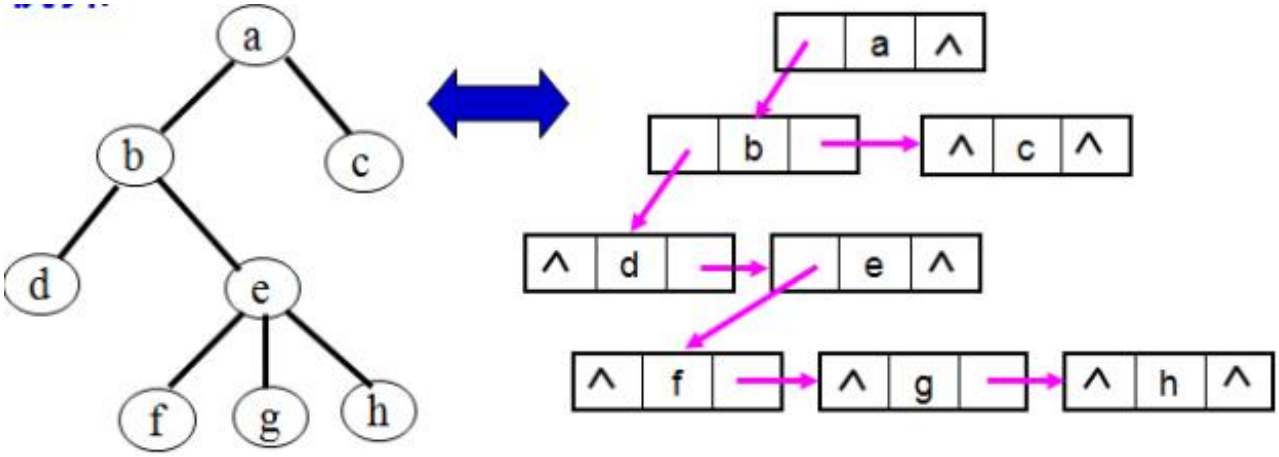
实验进度

- [x] 词法分析
- [x] 语法分析
- [x] 识别八进制、十六进制整数
- [x] 识别指数形式浮点数
- [x] 识别两种风格的注释

语法分析

语法树的结构

语法树是典型的多叉树。其结构如下：



bison

在每个产生式的action部分，主要要干的事情就是为产生式目标创建一个树节点，将产生式体中的各个节点串成链表，并将整个结构插入到语法树中去。action部分的内容，高度重复，且容易错误。如果我们手动写串链表的操作的话，错误就无法避免了。

对此我将create_node，以及串起node的过程全部用函数实现。这样子，每个action部分主要动作就是调用create_node为产生式目标创造节点，然后再调用connect_node将产生式体中的节点串起来。这样子每个action部分的代码量减少了不少，降低了出错的可能性。
示例代码如下：

```
Program : ExtDefList {
    $$ = create_node(Program, 0);
    $$->child = connect_node(1, $1);
    root = $$;
}
;

ExtDefList : {
    $$ = create_node(ExtDefList, 1);
}
|ExtDef ExtDefList {
    $$ = create_node(ExtDefList, 0);
    $$->child = connect_node(2, $1, $2);
}
;
```

其中create_node的第一个参数是节点类型，第二个0/1表明该节点是否生成到空。
connect_node使用了C语言的变参宏，这让我们要connect任意多个节点变得非常容易。当然，比较麻烦的一点是变参宏必须有一个固定参数，因此connect_node的第一个参数是整数，表示其后有多少个node。另外，即使这样子，我们的代码还是比较繁琐的。对于终结符号，我是在产生式体中为他们生成节点的，这就增加了出错的可能性。听欧先飞说，他采取了在lex中为终结符生成语法树节点，产生式部分直接手写个脚本进行代码生成的方式。这个方法就很机智了。出错的可能性也大大降低了。

出错恢复

要实现讲义中所说的每一行的错误都给发现并报出来，就必须要实现错误恢复。但是出错恢复在很大程度上一种guess，很难准确。为此，我主要是将error token加在了分号，逗号，右括号，右中括号，右花括号之前，在一个短语句出错后，尽量进行同步。

另外地，为了能处理好常见的八进制，16进制及浮点数的词法错误，我在lex中检查出错误并报错后，也会返回一个token，以让语法分析继续进行。

在错误恢复的实现中，还要注意的是，bison为了防止一个错误导致的连续错误，它一般在处理一个error之后，要等待正确移入三个正确的token之后，才开始正常的报错。即连续的错误会被supress住。bison的这个特性会导致对连续错误的行只报第一个错误，后面的错误都不报。为此我们使用yyerrok，让错误不会被supress住。例子如下：

```
error SEMI {
    yyerrok;
    is_successful = 0;
    $$ = create_node(ExtDef, 1);
}
```

在错误恢复中，还有一个值得注意的是，对于语法错，我们希望它输出的类似`Missing ";"`之类的具体的，明确的信息。但是根据bison文档上的描述，当发现错误之后，yyparse会先调用yyerror输出错误信息。然后，再移入error，如果恰好匹配上了某个有error的产生式，产生式体才会被执行。这就导致了对于同一条语法错，会输出两行，第一行是由yyerror输出的，第二行是由error的产生式体输出的。为了让一个语法错只输出一行报错，我没有在error的产生式体中输出报错信息，而是使用`%define parse.error verbose`将bison的报错变成了verbose，这样子报错信息也会更具体。

识别八进制、十六进制整数、指数形式的浮点数

识别合法八进制，十六进制，浮点数不是难点，难点在于识别出不合法的这些数。

主要问题是：合法的数的词法形式有定义，但不合法的数的词法形式没有定义。所以只能识别出常见的不合法的八进制，十六进制及浮点数的形式，并进行报错。

具体在识别上，我们可以这样做，即让正确的这些数对应于flex中的一个rule，错误的这些数

是另一个rule。

但我在实现时，是写出了一个笼统的rule，覆盖了正确及错误的数的大致形式，然后直接用strtol/strtof来处理识别出来的串，让这些函数帮助我们识别串合法不合法。另外，在strtol这个函数的帮助下，我们不仅可以识别出正确的数，对于错误形式的数，我们还可以指出第一个错误的字符。此外，对于超出int/float表示范围的数，我也进行检测了并报错。

识别两种风格的注释

单行comment的处理很简单，rule很简单。而多行comment的处理比较困难，主要原因是多行comment中的一些rule与其他rule有冲突。

但是使用flex提供的state这个功能之后，就很简单了。我们可以设置flex这个大的自动机的状态，并制定某些rule只有在某些状态时才有效。这样子，通过state，相当于是在自动机中构建了小的自动机。我们可以利用state，构造一个识别多行comment的小自动机。代码如下：

```
"/*"         { BEGIN(comment); }     /* deal with multi-line comments,
                                        see these two links:

http://dinosaur.compilertools.net/flex/flex_11.html

http://web.eecs.utk.edu/~bvz/teaching/cs461Sp11/notes/flex */
<comment>[^*\n]*  {}                 /* eat anything that's not a '*' */
<comment>"*"+[^*/\n]*  {}            /* eat up '*'s not followed by '/'s
*/
<*>\n     { yycolumn = 1; };         /* reset yycolumn in any state */
<comment>"*"+"/"       { BEGIN(INITIAL); } /* resume to initial state *
/

"//".* { }                           /* deal with one line comment
                                        activate this rule only in INITIA
L state
                                        so that one line comment in multi
-line comments
                                        will not be caught by this rule *
/
```

可以看到，在识别出`/*`之后，我们将flex自动机设置为了comment状态，只有在进入comment状态后，后面的那些以`<comment>`开头的rule才会真正地起作用。当我们识别出`*/`之后，我们将flex自动机设置为INITIAL状态，这是默认状态，这样子我们就退出了comment状态，那些以comment开头的rule就失效了，不会对其他rule产生干扰。

注：识别多行注释的代码来自这里。

测试

对parser进行测试的时候，困难在于编写测试用例。对于没有词法或语法错误的例子，手写出语法树很困难。对于有语法错的例子，如果一行有错误，那报了语法错就退出了，要想在同一个文件里面测试多个错误也比较困难。

因此，对于没有词法或语法错误的例子，我采取了手写语法树的方式，进行比对测试。而对于有错误的例子，由于我的parser已经实现了最基本的基于分号、右括号同步的错误恢复。因此只要保证一行只有一个错，并且每一个语句都以分号正确结束就行了。此外，对于有语法错的例子，具体的报错信息也很难人工确定，因此我只比对错误类型。为了简化测试用例的编写，对于错误的测试用例，我在相应的answer文件中，只需写一个json串，指明哪一行有个什么类型的错误即可。

例如对于如下的测试浮点数识别的用例：

```
int main()
{
    float i = 1.05e;
    float i = 1E-2;
    float i = 01.2E+34;
    float i = 43.e-3;
    float i = .5e02;
    float i = 9.8E7.6;
}
```

对应的answer文件就是：

```
[
```

```
    [3, "A"],
    [8, "A"]
  ]
```

这样子书写测试用例变得相对地简单。测试时，我使用了一个python脚本负责批量化测试，每次做了一个修改之后，输入`make test`，即可将所有测试样例都运行一遍，确保没有引入其他的bug。