2022秋compiler–lab1

学号: 201220154　姓名: 王紫萁*

Department of Computer Science, Nanjing University

2022 年 9 月 25 日

1　实验目标

1.1　词法分析器

作为NJU-compiler制作之旅的开始，我们首先需要将输入文件中的所有词法单元提取出来，c–的所有词法见Appendix_A[1]。词法单元的提取利用DFA的思想实现，而正则文法描述的语言正是他所对应的DFA能够接受的语言。

1.2　语法分析器

得到词法单元后，下一步就需要实现语法分析，通俗地，每个词素应该如何摆放到合适的位置上。为了实现这一点，最强大的工具就是巴科斯范式表达的语法产生式，以及他所生成的语法树（AST）

2　实验内容

2.1　实验环境

- Ubuntu 18.04.6 LTS on WSL2
- GCC version 7.5.0
- GNU Flex version 2.6.4
- GNU Bison version 3.0.4

*E-mail: ziqiwang.wayne@gmail.com

2.2　构建词法分析器

不同的词素有不同的词法，我们主要介绍int型，float型以及块级注释的正则表达式。int型有十进制，十六进制和八进制的写法。十进制是0或不以0开头的数字串，八进制是以0开头、长度不小于2（包括0）且数字字面值为0-7以内的数字串，十六进制是以0x开头长度不小于3（包括0x）的数字串。写作正则表达式为

```
HEX 0x[0−9a−fA−F]+                    1
OCT 0[0−7]+                           2
DEC 0|([1−9][0−9]∗)                   3
```

小数有科学计数法和常规表示法两种。常规表示法需要注意小数点两侧至少出现一个数字，且整数部分可以有任意多的0。科学计数表示法是有如下格式的数字串

```
(({digit}+\.{digit}∗)|\.{digit}+)     1
    ([Ee][+−]?{digit}+)?
```

其中e—E之前的是基数部分，之后是指数部分，且指数部分包括e—E可以不显示，基数部分就是常规表示法的格式。

最困难的当属块级注释表达式的书写了。先给出正确的写法：

```
\/\∗(?:[^\∗]|\∗+[^\/\∗])∗\∗+\/        1
```

首先需要熟知块级注释不允许嵌套，即“/*/**/*/”的形式是不被允许的。表达式需要由/*开头，一个以上的*以及一个/结束。来到最复杂的注释体部分，?:是为支持该语法的解析器加

速匹配的控制符。主体部分由除*以外的字符或一个* + 一个过多个* + 除/*以外的字符三部分组成。主体部分重复一次或多次就能得到完整的注释内容。

2.3 构建语法分析器

在词法分析后我们得到了每一个词素所在的词法类型，他们作为终结符用在语法分析中。 Appendix_A中有关于每个非终结符的产生式，产生式的书写不做赘述。值得一提的是有关 unary minus和binary minus的优先级确定问题。在教程中介绍了如何指定产生式的优先级来避免二义性，这给了我们启发，即让uminus的优先级高于bminus即可，另外还需要说明uminus是左结合的（与终结符一致）。

其次，在int和float的产生式中并没有关于+-符号的匹配处理（即我们的int和float实际上是无符号的），expression的产生式中也仅有uminus的匹配项。这给我们后期处理字符串转数值型减少了判断条件，但也让编译器缺少处理uplus的能力。解决方法和uminus的处理方法类似。

最后有必要说明一下错误恢复的过程。在flex中，error是一个特殊的标记用来指明一个出错的非终结符。这样在规约时如果遇到错误就能用error所在的产生式进行匹配。从而保证语法分析的继续执行。在实验过程中，最易出错的点是在函数体和if-else的错误恢复中。即以下两句产生式：

$$ExtDef \rightarrow Specifier\ error\ CompSt$$

$$Stmt \rightarrow IF\ error\ ELSE\ Stmt$$

错误原因在于，原先位置上的非终结符$FunDec$以及Exp均处于AST的较低层。因此他们首先检测到错误后将错误向上传递。如果不添加上述两句产生式，缺失前者会在形如`int main{}`的输入串fail掉从而无法读取函数体内的内容，而缺失后者可能会产生多余的报错信息，原因在于上文的错误未被跳过，影响到了下文错误的推断。总而言之，错误恢复需要自顶向下分析，将易出现问题的非终结符所在的产生式之后加入在相同位置用error替换的新产生式，经过不断尝试最终在合适的位置报告出合适数量的语法错误。

2.4 FLEX 与 BISON综合构建Parser

我们在前面讲到，FLEX将提取到的词素归类后传送给Bison，而Bison需要用其作为终结符作为产生式的一部分进行匹配。实验手册告诉我们可以借助yyval全局变量传递参数。利用好这一性质，就可以很方便地构建AST了。一个AST是一棵多叉树，其节点声明如下：

```
typedef struct tree_t {                                    1
    int no_line;          // line number of                2
    current node
    ValueType value_t;  // value type of the               3
    scanned node
    union {                                                4
        int v_int;                                         5
        float v_float;                                     6
        char *v_string;                                    7
    } value;                                               8
    struct tree_t *father;                                 9
    struct tree_t * first_child ;                          10
    struct tree_t *next_sibling;                           11
    struct tree_t *prev_sibling;                           12
} TNode;                                                   13
```

采用子女-兄弟表示法建立多叉树，用联合类型表示节点的真实值。只有INT, FLOAT, TYPE, ID 类型的节点具有有效的真实值，其中TYPE的真实值是int—float，ID的真实值是ID的字面量,均存放在 v_string字段中。

我们为AST建立了new_node, add_child, free_node, show_tree四个接口，分别表示建立节点，为父节点添加子节点，释放节点，以及打印语法树。头文件中有详尽的用法注释，在此不多介绍。

在flex源文件中，每识别到一个非空格非注释的词法单元后，就新建节点指明属性，同时如果有字面值就将yytext加入节点（深拷贝），否则传入空指针。Bison源文件在每个非空串产生式的语义动作中先创建左侧非终结符的节点，再将右侧所有节点加入到其子节点中。由于flex采用自底向上分析，因此右侧的所有节点在加入时已经创建成功。

2.5 编译与链接

parser.y文件中的定义部分包含了对lex.yy.c文件的引用，parser.l 包含了syntex.tab.h的引用。前者是flex生成的c文件，后者是bison -d选项拆分出的头文件，包含tokens的声明。因此最后只需要将main.c, parser.y, tree.c链接编译即可。即最终的编译指令如下：

```
$(CC) ${CMAIN} ${YFC} ./src/tree.c −lfl −ly
    −o parser
```

其中CC CMAIN YFC分别是gcc, main.c, .y文件的预定义。

3 总结

本次实验实现了编译器中最基础的词法分析和语法分析部分，并为语义分析构建出语法树。实现了识别科学计数、十六进制，八进制的附加功能，后续考虑加入对#include的识别来实现多文件分析。

参考文献

[1] 《编译原理实践与指导教程》 许畅等

[2] http://gnu.ist.utl.pt/software/flex/flex.html

[3] https://en.wikipedia.org/wiki/Backus-Naur_form