目录

1.C-语言的语法图描述

1. 关键字:
else if int return void while所有的关键字都是保留字。
2. 专用符号:
+ - * / < <= > >= == != = ; , ( ) [ ] { } /* */
3. 其它标记:
ID = letter letter*
NUM = digit digit*
Letter = a|..|z|A|..|Z
Digit = 0|..|9
4. 空格有空白、换行符和制表符组成。空格通常被忽略，除了它必须分开ID、NUM关键字
5. 注释用通常的C语言符号/*…*/围起来。注释可放在任何空白出现的位置(即注释不能放在标记内)上，且可以超过一行，注释不能嵌套。

以下为C-语法的BNF:

1. program -> declaration-list
2. declaration-list -> declaration-list declaration | declaration
3. declaration -> var-declaration | fun-declaration
4. var-declaration -> type-specifier ID ; | type-specifier ID [NUM] ;
5. type-specifier -> int | void
6. fun-declaration -> type-specifier ID (params) | compound-stmt
7. params -> params-list | void
8. params-list -> params-list , param | param
9. param -> type-specifier ID | type-specifier ID [ ]
10. compound-stmt -> { local-declarations statement-list }
11. local-declarations -> local-declarations var-declaration | empty
12. statement-list -> statement-list statement | empty
13. statement -> expression-stmt | compound-stmt | selection-stmt | iteration-stmt |

return-stmt

14. expression-stmt -> expression ; | ;
15. selection-stmt -> if (expression) statement | if (expression) statement else statement
16. iteration-stmt -> while (expression) statement
17. return-stmt -> return ; | return expression ;
18. expression -> var = expression | simple-expression
19. var -> ID | ID [expression]
20. simple-expression -> additive-expression relop additive-expression | additive-expression
21. relop -> <= | < | > | >= | == | !=
22. additive-expression -> additive-expression addop term | term
23. addop -> + | -
24. term -> term mulop factor | factor
25. mulop -> * | /
26. factor -> (expression) | var | call | NUM
27. call -> ID (args)
28. args -> arg-list | empty
29. arg-list -> arg-list , expression | expression

因为 LL（1）文法无法处理 S->AB,A->a…,B->a…的问题，所以对原文法进行了一些修改：

1. S -> program
2. program -> declaration-list
3. declaration-list -> declaration-list declaration | declaration
4. declaration -> var-declaration | fun-declaration
5. var-declaration -> type-specifier ID ; | type-specifier ID [NUM] ;
6. type-specifier -> int | void
7. fun-declaration -> type-specifier IDF (params) | compound-stmt
8. params -> params-list | void
9. params-list -> params-list , param | param
10. param -> type-specifier ID | type-specifier ID [ ]
11. compound-stmt -> { local-declarations statement-list }
12. local-declarations -> local-declarations var-declaration | empty
13. statement-list -> statement-list statement | empty
14. statement -> expression-stmt | compound-stmt | selection-stmt | iteration-stmt | return-stmt
15. expression-stmt -> expression ; | ;
16. selection-stmt -> if (expression) statement | if (expression) statement else statement
17. iteration-stmt -> while (expression) statement
18. return-stmt -> return ; | return expression ;
19. expression -> var1 = expression | simple-expression
20. var -> ID | ID [expression]
21. var1 -> ID1 | ID1 [expression]

22. simple-expression -> additive-expression relop additive-expression | additive-expression
23. relop -> <= | < | > | >= | == | !=
24. additive-expression -> additive-expression addop term | term
25. addop -> + | -
26. term -> term mulop factor | factor
27. mulop -> * | /
28. factor -> (expression) | var | call | NUM
29. call -> IDF (args)
30. args -> arg-list | empty
31. arg-list -> arg-list , expression | expression

2. 系统设计

2.1 系统的总体结构

① 基于 LL（1）文法的编译器（手写实现）;
② 基于 LALR 文法的编译器——使用 Flex 和 Bison 工具实现;

2.2 主要功能模块的设计

①基于 LL（1）文法的编译器（主要功能）

(1) 程序读入两个文件后，首先进行词法分析，在这之前，先预处理一下，对源程序去除注释，去除多余空格，随后对处理过后的源程序进行，得到一个 token 流，供语法分析时使用，生成栈以及语法分析树。
(2) 对语法规则消除左递归以及公共左因子，并且根据|拆分语法推导式，对处理后的语法规则，首先计算出所有的终结符以及非终结符，并求出非终结符的 first,follow 集，生成预测分析表，根据后序语法分析发现的问题后回过来对语法做的一些细微的修改也在这一步完成。
(3) 根据 token 流计算出符号表
(4) 根据 token 流以及预测分析表完成整个程序的出入栈以及语法树的建立。
(5) 使用深度遍历语法树遇到相应节点，即进行语义动作，产生三地址代码。
(6) 最后对产生额三地址代码进行优化。

②基于 LALR 文法的编译器

(1) 编写 lex.l 文件。该文件用于 flex 词法分析，文件内容包含 C/C++语言声明与和定义正则表达式，匹配模式和事件。
(2) 编写 Syntax.y 文件，用于 bison 语法分析和语义分析，内容对文法的终结符和非终结符做一些相关声明，并段定义了文法的非终结符及产生式集合，以及当归约整个产生式时应执行的操作
(3) 输入命令 g++ -o Syntax.tab.exe Syntax.tab.c lex.yy.c -w 编译得到 C-语法分析器 Syntax.tab.exe，设计测试用例 test.txt，输入命令 Syntax.tab.exe < test.txt 即可运行并输出相应文件。

(4) 使用 Makefile 以简化编译流程。

3.系统实现

3.1 主要函数说明

基于 LL（1）文法的编译器——词法分析

基于 LL（1）文法的编译器——语法分析

基于 LL（1）文法的编译器——语义分析

基于 LL(1)文法的编译器——代码优化

基于 LALR 文法的编译器——词法分析

基于 LALR 文法的编译器——语法分析

基于 LALR 文法的编译器——语义分析

3.3 测试用例