MiniC 编译器程序
项目详细设计文档

[V1.0(版本号)]

拟 制 人吴佩华 20172131059

侯玉婷 20172131134

审 核 人_____

批 准 人_____

2020 年 4 月 29 日

目录

详细设计说明书

一、引言

1.1 编写目的

此概要设计说明书对《MiniC 编译器程序》开发做了全面细致的功能需求分析，明确所要开发的平台应具有的功能、性能与界面，使系统分析人员及软件开发人员能清楚地了解用户的需求，并在此基础上进行先一步完成后续设计与开发工作。此文档的编写旨在对《MiniC 编译器程序》的词法分析、语法分析、语义分析、代码生成以及迅疾解释功能模块进行研究。

本文档的预期读者是：

- 需求分析人员
- 开发人员
- 项目管理人员
- 测试人员
- 用户

1.2 背景

- 开发软件的名称：MiniC 编译器程序
- 项目任务提出者：黄煜廉老师
- 项目开发者：吴佩华、侯玉婷
- 用户：使用 MiniC 语言的开发人员
- 实现软件单位：本小组

1.3 定义

1.3.1 Qt

Qt 是一个由 Qt Company 开发的跨平台 C++图形用户界面应用程序开发框架。它既可以开发 GUI 程序，也可用于开发非 GUI 程序，比如控制台工具和服务器。Qt 是面向对象的框架，使用特殊的代码生成扩展以及一些宏，Qt 很容易扩展，并且允许真正地组件编程。2014 年 4 月，跨平台集成开发环境 Qt Creator 3.1.0 正式发布，实现了全面支持 iOS、Android、WP，它提供给应用程序开发者建立艺术级的图形用户界面所需的所有功能。基本上，Qt 同 X Window 上的 Motif，Openwin，GTK 等图形界面库和 Windows 平台上的 MFC，OWL，VCL，ATL 是同类型的东西。

1.3.2 Mini C 语言

此程序定义的编程语言称为 MiniC，这是一种适合编译器设计方案的语言，它比 TIYN 语言更复杂，包括函数和数组。本质上它是 C 的一个子集，但省去了一些重要的部分，因此得名。

1.3.3 TINY 语言

描述真实的编译器非常困难。“真正的”编译器——也就是希望在每天编程中用到的内容太复杂，另一方面，一种很小的语言（其列表包括 10 页左右的文本）的编译也不可能准确地描述出“真正的”编译器所需的所有特征。

为了解决上述问题，人们在（ANSI）C 中为小型语言提供了完整的源代码，这种语言称作 TINY。

1.3.4 编译器

简单讲，编译器就是将“一种语言（通常为高级语言）”翻译为“另一种语言（通常为低级语言）”的程序。一个现代编译器的主要工作流程：源代码 (source code) → 预处理器 (preprocessor) → 编译器 (compiler) → 目标代码 (object code) → 链接器 (Linker) → 可执行程序 (executables)

高级计算机语言便于人编写，阅读交流，维护。机器语言是计算机能直接解读、运行的。编译器将汇编或高级计算机语言源程序作为输入，翻译成目标语言机器代码的等价程序。源代码一般为高级语言， 如 Pascal、C、C++、Java、汉语编程等或汇编语言，而目标则是机器语言的目标代码，有时也称作机器代码。

对于 C#、VB 等高级语言而言，此时编译器完成的功能是把源码编译成通用中间语言的字节码。最后运行的时候通过通用语言运行库的转换，编程最终可以被 CPU 直接计算的机器码。

1.3.5 词法分析

词法分析是计算机科学中将字符序列转换为单词（Token）序列的过程。进行词法分析的程序或者函数叫作词法分析器（Lexical analyzer，简称 Lexer），也叫扫描器。词法分析器一般以函数的形式存在，供语法分析器调用。完成词法分析任务的程序称为词法分析程序或词法分析器或扫描器。

完成词法分析任务的程序称为词法分析程序或词法分析器或扫描器。从左至右地对源程序进行扫描，按照语言的词法规则识别各类单词，并产生相应单词的属性字。

1.3.6 语法分析

语法分析是编译过程的一个逻辑阶段。语法分析的任务是在词法分析的基础上将单词序列组合成各类语法短语，如“程序”，“语句”，“表达式”等等。语法分析程序判断源程序在结构上是否正确。源程序的结构由上下文无关文法描述。语法分析程序可以用 YACC 等工

具自动生成。

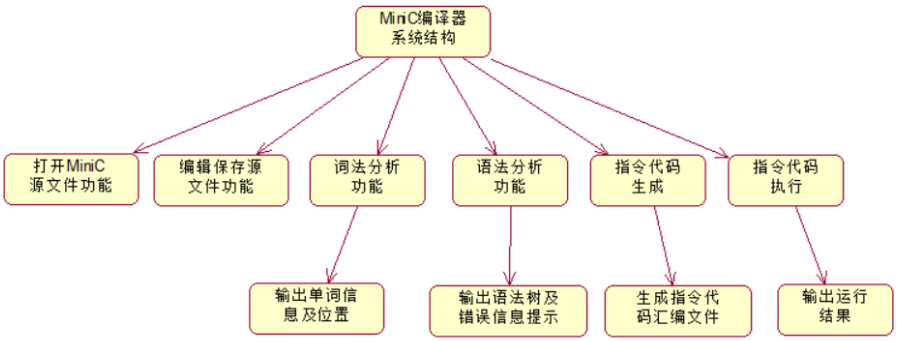
完成语法分析任务的程序称为语法分析器，或语法分析程序。按照源语言的语法规则，从词法分析的结果中识别出相应的语法范畴，同时进行语法检查。

1.4 参考资料

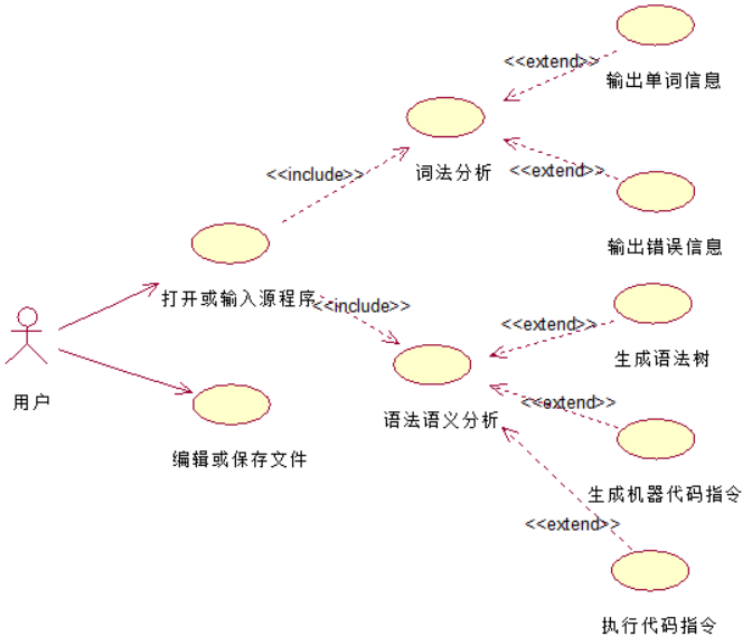
[1] 编译原理课程电子书. [DB]中文版.
[2] Alfred V.Aho/Monica S.Lam/Ravi Sethi/Jeffrey D.Ullman.编译原理[M].2008-12.

二、程序系统的结构

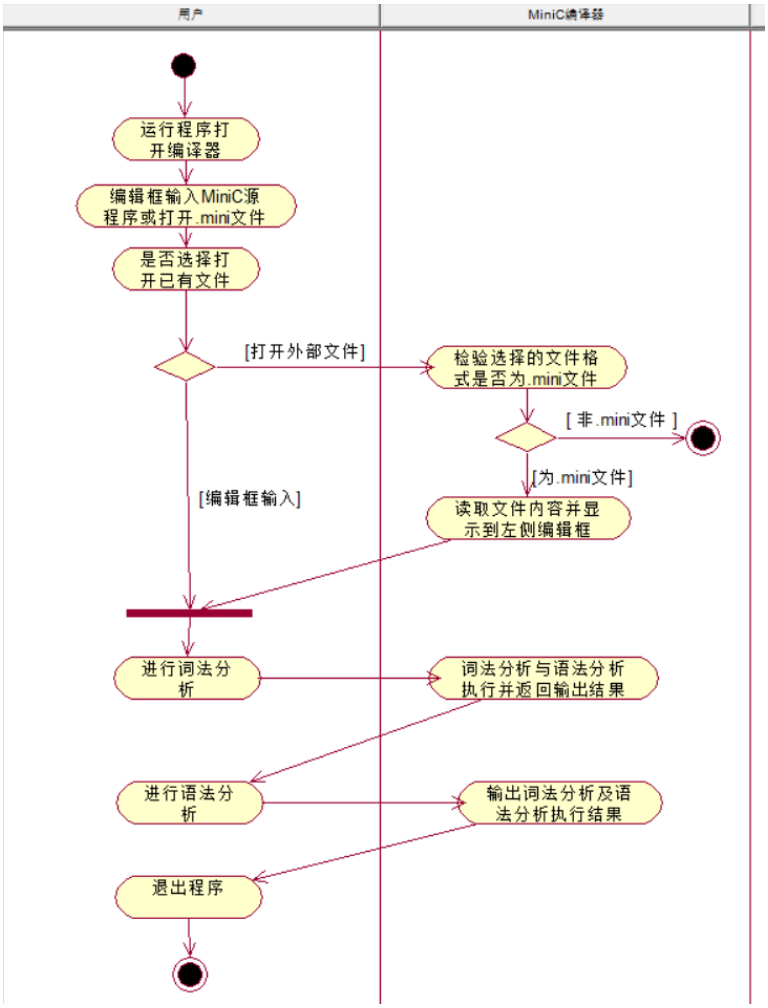
2.1 系统总体结构图



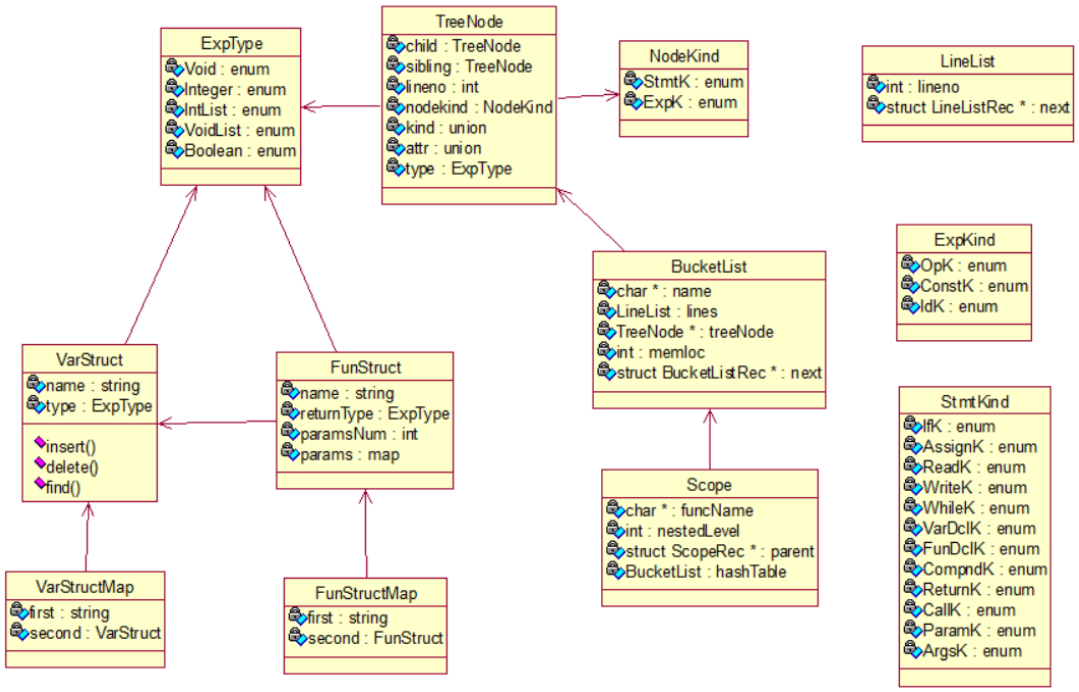
系统总体结构图



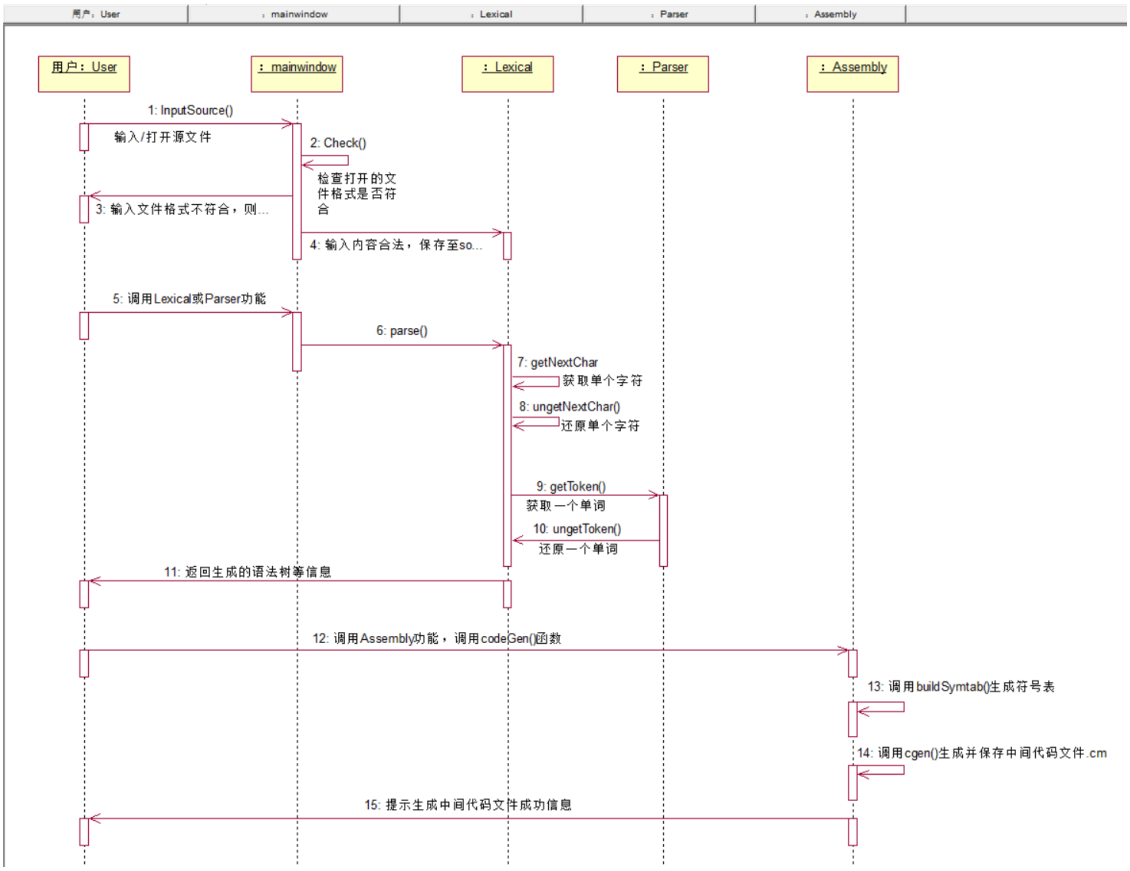
系统总体用例图


用户
MiniC编译器
运行程序打开编译器
编辑框输入MiniC源程序或打开.mini文件
是否选择打开已有文件
[打开外部文件]
检验选择的文件格式是否为.mini文件
[非.mini文件]
[为.mini文件]
[编辑框输入]
读取文件内容并显示到左侧编辑框
进行词法分析
词法分析与语法分析执行并返回输出结果
进行语法分析
输出词法分析及语法分析执行结果
退出程序

系统总体活动图



系统定义数据结构



系统总体时序图

三、词法分析模块设计说明

3.1 程序描述

词法分析阶段是编译过程的第一个阶段，是编译的基础。词法分析模块是本实验项目第一个功能模块，核心任务是对用户给出的 MiniC 源程序转换为字符串形式，根据 MiniC 语言的此法规则将源程序中的字符扫描、识别出具有独立意义的单词，输出与源程序等价的 token 流。本模块根据 MiniC 语言的构词规则，定义了关键字、标识符、常数、运算符及界符等单词的识别。

3.2 功能

对于字符串表示的源程序，从左到右依次进行扫描和分解，根据词法规则，识别出具有独立意义的单词符号，以供后续语法分析使用。对于 MiniC 词法中的关键字和界符，转换为对应类型 token 返回；对于 MiniC 中的常数数值，转换为 NUM 类型 token 返回，并将对应数值字符串保存；对于其他符合 MiniC 词法规则的变量名，装换为 ID 类型的 token 返回，并将对应变量名字符串保存；对于扫描到的词法错误，记录并输出至语法错误输出框。扫描结束是，将词法分析结果显示在词法分析输出框。

3.3 输入项

词法分析模块输入字符串形式的单个 MiniC 源程序。以下为 MiniC 中能够识别的单个字符：

ASCII 值	控制字符	ASCII 值	控制字符	ASCII 值	控制字符
0	NUL	62	>	96	`
9	HT	63	?	97	a
10	LF	64	@	98	b
13	CR	65	A	99	c
32	(sapce)	66	B	100	d
33	!	67	C	101	e
34	“	68	D	102	f
35	#	69	E	103	g
36	$	70	F	104	h
37	%	71	G	105	i
38	&	72	H	106	j
39	‘	73	I	107	k
40	(	74	J	108	l
41	)	75	K	109	m
42	*	76	L	110	n
43	+	77	M	111	o

44	,	78	N	112	p
45	-	79	O	113	q
46	.	80	P	114	r
47	/	81	Q	115	s
48	0	82	R	116	t
49	1	83	S	117	u
50	2	84	T	118	v
51	3	85	U	119	w
52	4	86	V	120	x
53	5	87	W	121	y
54	6	88	X	122	z
55	7	89	Y	123	{
56	8	90	Z	124	|
57	9	91	[	125	}
58	:	92	\	126	~
59	;	93	]		
60	<	94	^		
61	=	95	—		

3.3 输出项

词法分析模块输出类型为枚举 TypeToken 的属性单词 token，以下为省略分隔符扫描到字符串对应的 token 类型：

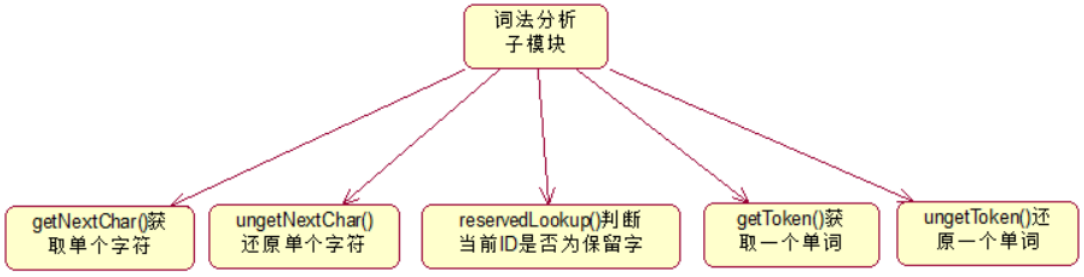
扫描到字符串	token 类型	备注
EOF	ENDFILE	关键字
else	ELSE	关键字
if	IF	关键字
return	RETURN	关键字
while	WHILE	关键字
int	INT	变量类型关键字
void	VOID	变量类型关键字
<	LT	后一个字符不为’=’
<=	LTEQ	
>	GT	后一个字符不为’=’
>=	GTEQ	
=	ASSIGN	后一个字符不为’=’
==	EQ	
!=	NEQ	
+	PLUS	
-	MINUS	
*	TIMES	
/	OVER	

(	LPAREN	
)	RPAREN	
[	LBRACKET	
]	RBRACKET	
{	LBRACE	
}	RBRECE	
,	COMMA	
;	SEMI	
数字集字符串	NUM	每个字符 ASCII 码不小于 48 且不大于 57
非关键字字母字符集	ID	每个字符 ASCII 码不小于 65 且不大于 90 或每个字符 ASCII 码不小于 97 且不大于 122，且不为关键字
其他	ERRO	不符合 MiniC 词法规则

3.5 算法

调用 getNextChar()函数获取单个字符后，按照词法规则 DNF 扫描并返回识别的 token。

3.6 流程逻辑



词法分析子模块

3.7 接口

3.7.1 内部接口

方法名称	返回类型	参数	备注
getToken	TypeToken	void	依次扫描字符串，当扫描得到独立意义的完整单词时，返回扫

			描到单词的 token 类型
ungetToken	void	void	getToken 的逆方法，回退到开始本次单词扫描的前一个状态

3.7.2 内部方法

方法名称	返回类型	参数	备注
getNextChar	void	int	扫描识别到下一个非空白非换行非结束符号，并
ungetNextChar	void	void	getNextChar 的逆方法，回退到开始扫描本个字符的前一个状态
reservedLookup	TypeToken	char *s	识别字符串 s 为关键字字符串还是变量字符串，返回相应的 token 类型

3.8 存储分配

3.8.1 数据结构

名称	标识	定义	备注
单词类型	TypeToken	typedef enum { ENDFILE, ERRO, ELSE, IF, RETURN, WHILE, INT, VOID, ID, NUM, ASSIGN, EQ, NEQ, LT, LTEQ, GT, GTEQ, PLUS, MINUS, TIMES, OVER, LPAREN, RPAREN, LBRACKET, RBRACKET, LBRACE, RBRACE, SEMI, COMMA, ADDITIVE,POWER } TypeToken;	由于标识单词类型

扫描状态	StateType	typedef enum
		{
		START,
		INCOMMENT, INNUM,
		INID, DONE ,
		INPOWER
		}
		StateType;
			记录当前扫描状态

3.8.2 变量

名称	标识	数据类型	备注
最大缓冲区长度	BUFLEN	宏定义	数值 256
扫描行字符串	lineBuf	char[BUFLEN]	记录当前扫描行的字符串信息
扫描位置	linepos	int	记录当前行字符串扫描的位置
缓冲长度	bufsize	int	记录当前扫描行字符串长度
单词长度	tokenLength	int	记录前一个扫描单词已扫描长度
单词	token	TypeToken	记录前一个扫描单词类型
单词字符串	tokenString	char *	记录前一个扫描单词的字符串内容
前单词	lastToken	TypeToken	记录前一个单词之前扫描的单词
前单词字符串	lastTokenString	char *	记录前一个单词之前扫描的单词的字符串内容

3.9 限制条件

软件中必须先打开 MiniC 源程序。

3.10 测试计划

程序测试及报告\BUG+MiniC 编译器+语法分析模块+00001.doc
程序测试及报告\MiniC 编译器测试用例.xls

3.11 尚未解决的问题

无

四、语法分析模块设计说明

4.1 程序描述

语法分析阶段是编译过程的第二个阶段，也是继词法分析模块之后的功能模块，核心任务是根据已进行文法规则，对词法分析模块中的输出项判断结构上是否符合文法规则，符合时组合成各类语法短语，并以语法树的形式返回，否则返回错误信息。本功能模块使用自顶向下分析，进行语法分析前，将文法规则进行左递归消除和合并左公因子。考虑到 MiniC 语法复杂性，使用 LL(2)文法。

4.2 功能

对于词法分析获得的 token，反复使用生产式对句型中的非终结符进行替换推导，将扫描的 token 装换成对应语法短语的树节点，生成语法树，并将语法树结果显示语法分析输出框。当扫描的 token 不符合语法语句结构时，记录语法错误信息并输出至语法错误输出框。

4.3 输入项

语法分析过程的输入项即为词法分析中识别到的单词（token），词法分析后识别到的 token 有以下情况：

token 名称	token 类型	输入方式	备注
EOF	ENDFILE	getToken()函数调用	关键字
else	ELSE	getToken()函数调用	关键字
if	IF	getToken()函数调用	关键字
return	RETURN	getToken()函数调用	关键字
while	WHILE	getToken()函数调用	关键字
int	INT	getToken()函数调用	变量类型关键字
void	VOID	getToken()函数调用	变量类型关键字
<	LT	getToken()函数调用	后一个字符不为’=’
<=	LTEQ	getToken()函数调用	
>	GT	getToken()函数调用	后一个字符不为’=’
>=	GTEQ	getToken()函数调用	
=	ASSIGN	getToken()函数调用	后一个字符不

			为’=’
==	EQ	getToken()函数调用	
!=	NEQ	getToken()函数调用	
+	PLUS	getToken()函数调用	
-	MINUS	getToken()函数调用	
*	TIMES	getToken()函数调用	
/	OVER	getToken()函数调用	
(	LPAREN	getToken()函数调用	
)	RPAREN	getToken()函数调用	
[	LBRACKET	getToken()函数调用	
]	RBRACKET	getToken()函数调用	
{	LBRACE	getToken()函数调用	
}	RBRECE	getToken()函数调用	
,	COMMA	getToken()函数调用	
;	SEMI	getToken()函数调用	
数字集字符串	NUM	getToken()函数调用	
非关键字字母字符集	ID	getToken()函数调用	
其他	ERRO	getToken()函数调用	

4.4 输出项

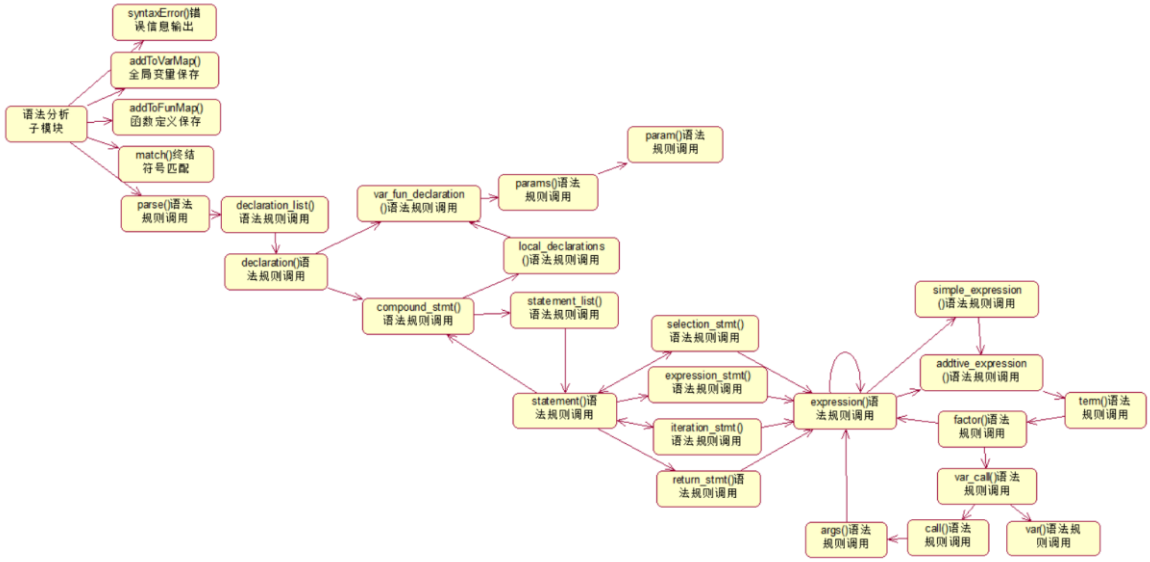
语法分析过程中根据扩充的 BNF 在识别过程中构建语法树，每次函数调用返回子树的根节点，直至最后返回整颗语法树的根节点 TreeNode*类型 syntaxTree。

名称	取值类型	输出形式	备注
syntaxTree	TreeNode*	调用 parse()函数返回值	
errorMessage	string	以全局变量形式记录	

4.5 算法

自顶向下分析，按照扩充 BNF 文法规则进行递归下降扫描。调用 parse()返回语法树根节点后，调用 printTree()深度优先输出语法树。

4.6 流程逻辑



语法分析子模块

4.7 接口

4.7.1 内部接口

方法名称	返回类型	参数	备注
parse	TreeNode*	void	依次扫描 token，将单词识别成文法短句，并返回语法树

4.7.2 内部方法

方法名称	返回类型	参数	备注
syntaxError	char*	char * message	语法错误输出，返回输出内容
match	void	TokenType expected	当识别 token 匹配预期时，获取下一个 token，否则追加语法错误信息
declaration_list	TreeNode *	void	declaration-list -> declaration {declaration}
declaration	TreeNode *	void	declaration ->

			type-specifier ID (; | [NUM]; | (params)) | compound-stmt
var_fun_declaration	TreeNode *	VarFunDclType decType	使用 LL(2) 匹配 type-specifier ID (; | [NUM]; | (params)) | compound-stmt
compound_stmt	TreeNode *	void	compound-stmt -> { local-declarations statement-list }
params	TreeNode *	void	params -> param{,param} | void
param	TreeNode *	void	param -> type-specifier ID (empty | [ ])
local_declaration	TreeNode *	void	local-declarations -> var-declaration{var-declaration} | empty
statement_list	TreeNode *	void	statement-list -> statement { statement }| empty
statement	TreeNode *	void	statement -> expression-stmt | compound-stmt | selection-stmt | iteration-stmt | return-stmt
selection_stmt	TreeNode *	void	selection-stmt -> if(expression) statement [elsestatement]
expression_stmt	TreeNode *	void	expression-stmt -> expression ; | ;
var	TreeNode *	void	var -> ID (empty | [expression])
var_call	TreeNode *	void	使用 LL(2)匹配 var -> ID (empty | [expression]) 和

			call -> ID(args)
iteration_stmt	TreeNode *	void	iterstions-stmt -> while(expression) statement
return_stmt	TreeNode *	void	reutrn-stmt -> return (;|expression ;)
expression	TreeNode *	void	expression -> var = expresion | simple-expression
simple_epression	TreeNode *	void	simple-expression -> additive-expression [relop addtive-expression]
addtive_expression	TreeNode *	void	addtive-expression -> term {addop term}
term	TreeNode *	void	term -> factor{ mulop factor }
factor	TreeNode *	void	factor -> (expression) | var | call | NUM
call	TreeNode *	void	call -> ID(args)
args	TreeNode *	void	args ->expression{, expression } | empty
addToVarMap	void	VarStruct v	将识别到的变量声明加入对应的变量空间，及全局变量空间和函数变量空间，当同一变量空间存在相同名称变量名时，追加错误信息
addToFunMap	void	FunStruct f	将识别到的函数声明加入函数空间，当同一存在函数名称时，追加错误信息

4.8 存储分配

4.8.1 数据结构

名称	标识	定义	备注
语法树节点	TreeNode	typedef struct treeNode { struct treeNode *child[MAXCHILDREN]; struct treeNode *sibling; int lineno; NodeKind nodekind; union { StmtKind stmt; ExpKind exp; } kind; union { TypeToken op; int val; * name;} attr; ExpType type; } TreeNode;	语法树数据结构
语法树节点类型	NodeKind	typedef enum { StmtK, ExpK } NodeKind;	根据文法规则，分为状态语句和表达式语句
状态语句类型	StmtKind	typedef enum { IfK, AssignK, ReadK, WriteK , WhileK, VarDclK, FunDclK, CompndK, ReturnK, CallK , ParamK , ArgsK} StmtKind;	文法规则中语句状态语句类型
表达式变量类型	ExpType	typedef enum { Void, Integer, IntList, VoidList, Boolean } ExpType;	文法规则中变量类型
表达式语句类型	ExpKind	typedef enum { OpK, ConstK, IdK } ExpKind;	文法规则中表达式语句类型
变量定义数据	VarStruct	typedef struct varStruct { ExpType type; std::string name; } VarStruct;	存储源代码中识别出的变量名称、变量类型
函数定义数据	FunStuct	typedef struct	存储源代码中识别出

		funStruct｛ExpType returnType; std::string name; int paramsNum; std::map<std::string, VarStruct> params; } FunStruct;	的函数名称、返回类型及函数内定义参数和变量
变量函数声明标识	VarFunDclType	typedef enum{ VarDcl, FunDcl, VarFunDcl }VarFunDclType;	由于 LL(2)文法标识变量声明和函数声明
变量声明状态	ManageMapState	typedef enum{ InCompound, InFunDcl, GlobalVarDcl }ManageMapState;	由于变量声明类型标识，即函数内变量声明、函数参数声明、全局变量声明

4.8.2 变量

名称	标识	数据类型	备注
单词	token	TypeToken	记录前一个扫描单词类型
单词字符串	tokenString	char *	记录前一个扫描单词的字符串内容
全局变量集	VarStructMap	std::map<std::string, VarStruct>	记录已经声明全局变量
函数集	FunStructMap	std::map<std::string, FunStruct>	记录已经声明的函数信息
变量声明状态标识	manageMapState	ManageMapState	
最新声明函数名	lastDeclaredFunName	std::string	

4.9 限制条件

软件中必须先打开 MiniC 源程序。未选择执行词法分析功能而执行语法分析功能时，将先进行源程序的词法分析。

4.10 测试计划

程序测试及报告\BUG+MiniC 编译器+语法分析模块+00001.doc
程序测试及报告\MiniC 编译器测试用例.xls

4.11 尚未解决的问题

暂未对数组类型的变量进行下标非负检查。

五、代码产生模块设计说明

5.1 程序描述

中间代码（Intermediate Representation）是复杂性介于源程序语言和机器语言的一种表示形式。 编译 程序中使用的中间代码有多种形式，常见的有逆波兰记号，三元式，四元式，和树形表示。四元式是一 种普遍采用的中间代码形式，很类似于三地址指令，有时把这类中间表示称为“三地址代码”，这种表示 可以看作是一种虚拟三地址机的通用汇编码，每条：“指令”包含操作符和三个地址，两个是为运算对象，一个是为结果。

在本项目中，中间代码使用三元组表示，并将输入的.minic 文件翻译为.cm 文件。在生成代码中，默认开启中间代码注解功能，及生成的.cm 文件中以注释形式说明每段代码指令的功能。本项目不内嵌中间代码的运行功能，需用户自行调用 TINY 虚拟机执行程序运行生成的.cm 文件并的得出运行结果。

5.2 功能

生成程序的中间代码并保存到与.minic 文件同路径下的.cm 文件。使用 TINY 虚拟机程序执行后能在控制台上显示程序运行结果。该模块实际上对已生成的语法树进行遍历，识别语法书中各个节点的类型，并根据已有的符号表及文法规则，对识别到的节点生成三元组指令，而使用到的指令为 TINY 虚拟机所提供的指令。

5.3 输入项

该模块的输入项为语法分析输出的语法树，语法树结构参见 4.8.1。

5.4 输出项

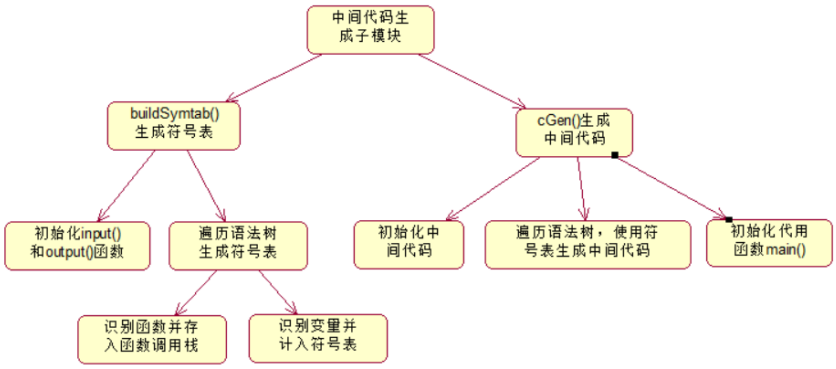
该模块输出语义分析后的符号表以及生成的中间代码，中间代码以三元组形式输出到本地文件中。以下为该模块的输出项：

名称	取值类型	输出形式	备注
scopes	Scope 数组	调用 buildSymtab()函数后保存在全局变量中	
scopeStack	Scope 数组	调用 buildSymtab()函数后保存在全局变量中	

5.5 算法

使用 DFS 算法遍历语法树并使用栈数据结构存储函数调用过程。

5.6 流程逻辑



5.7 接口

5.7.1 内部接口

方法名称	返回类型	参数	备注
buildSymtab	void	TreeNode * syntaxTree	扫描语法树生成符号表
codeGen	void	TreeNode * syntaxTree, const char * codefile	通过得出的符号表生成中间代码并保存到本地文件中

5.7.2 内部方法

方法名称	返回类型	参数	备注
Hash	int	char *	对传入变量名称 name 进行哈希，并返回得出的哈希值，哈希值在 0-211 范围内
st_insert	void	char *, int, int, TreeNode *	通过传入的变量信息，将符号插入符号表
st_lookup	int	char *	返回变量的相对内存的偏移量
st_lookup_top	int	char *	在函数调用栈栈顶函

			数中查找变量
st_add_lineno	int	char *, int	
sc_top	Scope	void	获取函数调用栈栈顶函数
sc_pop	void	void	推出一个函数调用栈栈顶函数
sc_push	void	Scope	向函数调用栈中插入函数信息
sc_create	Scope	char *	生成函数信息并加入到函数调用栈中
st_bucket	BucketList	char *	返回变量的信息数据结构
traverse	void	TreeNode * , void (* preProc) , void (* postProc)	生成符号表调用入口
insertIOFunc	void	void	初始化 input() 和 output()
insertNode	void	TreeNode *	遍历语法树生成符号表
afterInsertNode	void	TreeNode *	生成符号后还原函数调用栈
getBlockOffset	int	TreeNode *	获得变量的偏移量
genStmt	void	TreeNode *	对 StmtK 类型的节点生成中间代码
genExp	void	TreeNode *,int	对 ExpK 类型的节点生成中间代码
cGen	void	TreeNode *	中间代码生成调用入口
genMainCall	void	void	对 main() 函数初始化入口

5.8 存储分配

5.8.1 数据结构

名称	标识	定义	备注
变量引用行数结构体	LineList	typedef struct LineListRec { int lineno; struct LineListRec * next;	

		} *LineList;	
变量信息结构体	BucketList	typedef struct BucketListRec { char * name; LineList lines; TreeNode *treeNode; int memloc; struct BucketListRec * next; } *BucketList;	
函数信息结构体	Scope	typedef struct ScopeRec { char * funcName; int nestedLevel; struct ScopeRec * parent; BucketList hashTable[SIZE]; } * Scope;	

5.8.2 变量

名称	标识	数据类型	备注
函数定义数组	scopes	Scope 数组	记录语法树中的
函数调用数组	scopeStack	Scope 数组	存储调用的函数信息，使用数组实现栈结构

5.9 限制条件

软件中必须先打开 MiniC 源程序。未选择执行词法分析和语法分析功能而执行中间代码生成功能时，将先进行源程序的上述两个功能。

5.10 测试计划

程序测试及报告\MiniC 编译器测试用例.xls

5.11 尚未解决的问题

暂未对数组类型的变量进行下标非负检查；测试用例较少，生成中间代码模块未进行一定程序运行测试，对某些程序可能无法正确生成代码。