编译器设计文档

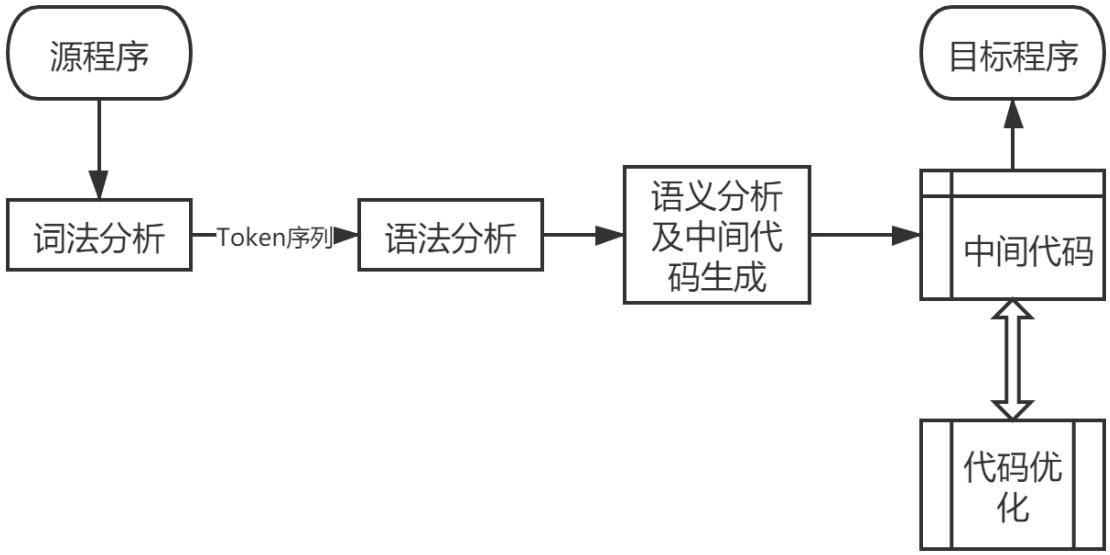
需求分析

编译器功能为将**类C语言高级程序设计语言的源程序**转化为**MIPS体系结构下的汇编目标程序。**

架构设计

数据流

此编译器采用多遍的设计思想，尽可能地将编译器五大阶段分离，其中优化部分也计划采用多遍优化地方案，整体架构如下

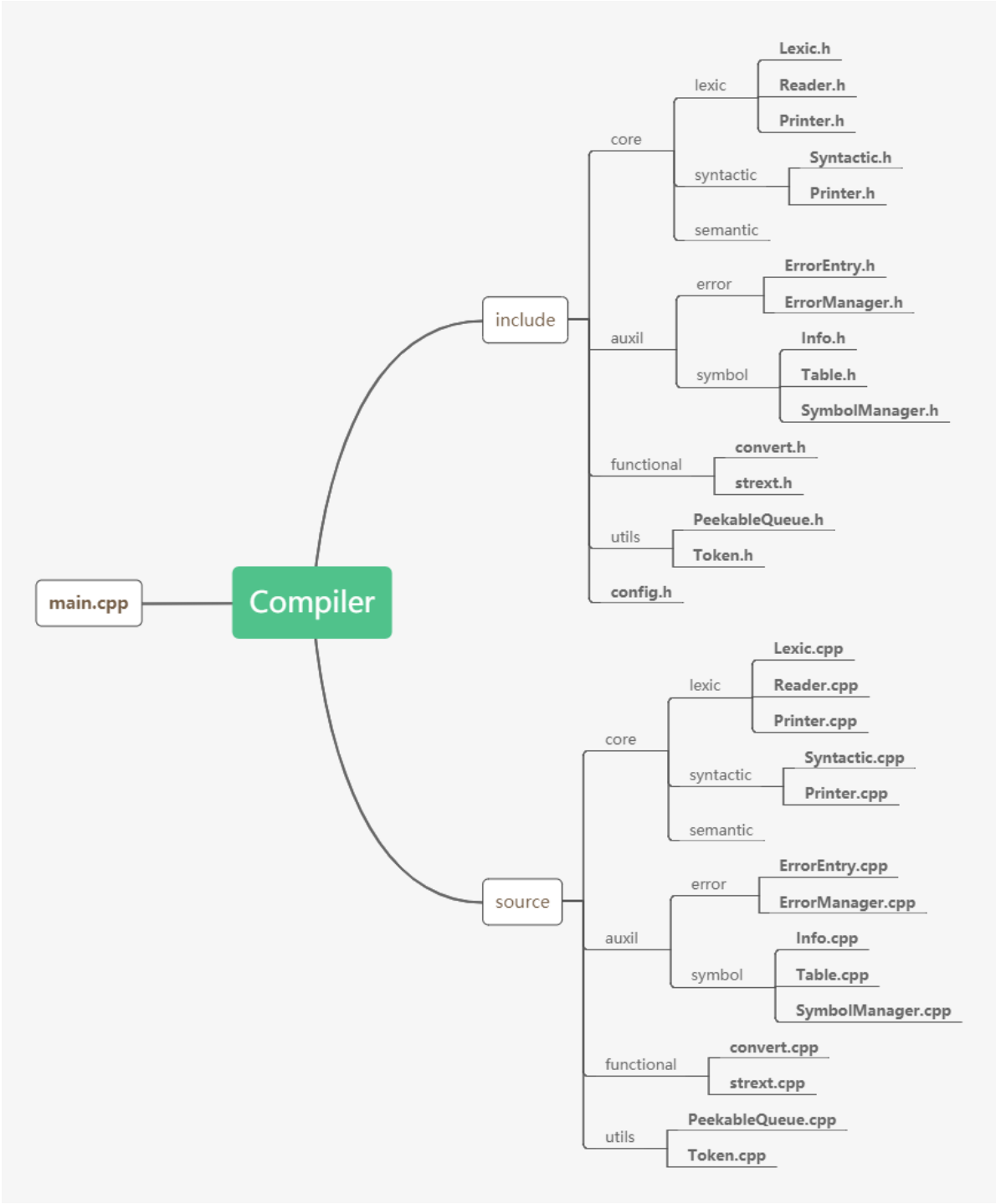


项目结构

将工程文件结构主要分为include和source部分，分别代表头文件及源代码文件，将两者分离便于编译器软件的发布和整理；具体地来看，将整个流程分为三大部分core auxil utils，分别代表编译器核心部分、辅助结构、工具函数文件，具体用加黑粗体表示文件。

此外，整个工程被分为多个命名空间，以便于相似结构部分的内容设计能够采用相同的命名，更加简洁清晰。

特别地，**整个工程项目的参数控制通过include-config.h进行控制，便于管理和部分功能的开关。**



各部分设计

词法分析部分

输入输出

该部分的输入文件为高级语言程序，输出文件为如下定义的二元组列表

$$(TokenCode, TokenValue)$$

二元组设计

TokenCode	TokenValue(默认为大小写不敏感)
IDENFR	＜字母＞ ｛＜字母＞｜＜数字＞｝
INTCON	＜数字＞ ｛＜数字＞｝
CHARCON	'＜加法运算符＞'|'＜乘法运算符＞'|'＜字母＞'|'＜数字＞' **（大小写敏感）**
STRCON	"｛十进制编码为32,33,35-126的ASCII字符｝" **（大小写敏感）**
CONSTTK	const
INTTK	int
CHARTK	char
VOIDTK	void
MAINTK	main
IFTK	if
ELSETK	else
SWITCHTK	switch
CASETK	case
DEFAULTTK	default
WHILETK	while
FORTK	for
SCANFTK	scanf
PRINTFTK	printf
RETURNTK	return
PLUS	+
MINU	-
MULT	*
DIV	/
LSS	<
LEQ	<=
GRE	>
GEQ	>=
EQL	==
NEQ	!=
COLON	:

TokenCode	TokenValue(默认为大小写不敏感)
ASSIGN	=
SEMICN	;
COMMA	,
LPARENT	(
RPARENT	)
LBRACK	[
RBRACK	]
LBRACE	(
RBRACE	)

结构说明

Reader

专注于从输入文件读取文本，设立专用缓冲区，并进行$(row, column)$统计，便于后续错误处理程序的需要；除此之外，Reader类还需要对文件结束进行判断，停止文件读入操作并设置下一个返回字符为EOF，便于词法分析主程序判断读入结束。

```
class Reader
    {
    private:
        std::ifstream fsIn;
        queue<char> buffer;
        int row;
        int column;

    public:
        Reader(const string& fIn);
        ~Reader();

    public:
        char next();
        int getRow() const;
        int getColumn() const;
    };
```

Printer

专注于输出文件的操作，被调用时接受二元组$(TokenCode, TokenValue)$，输出到词法分析阶段的调试输出文件；其中通过$enabled$开关控制是否进行中间二元组序列调试文件的输出与否。

```
class Printer
    {
    private:
        const bool enabled = config::PRINT_LEXIC;
        std::ofstream fsOut;

    public:
        Printer(const string& fOut);
        ~Printer();

    public:
        void print(const config::TokenCode tkcode, const string& tkvalue);
    };
```

Lexic

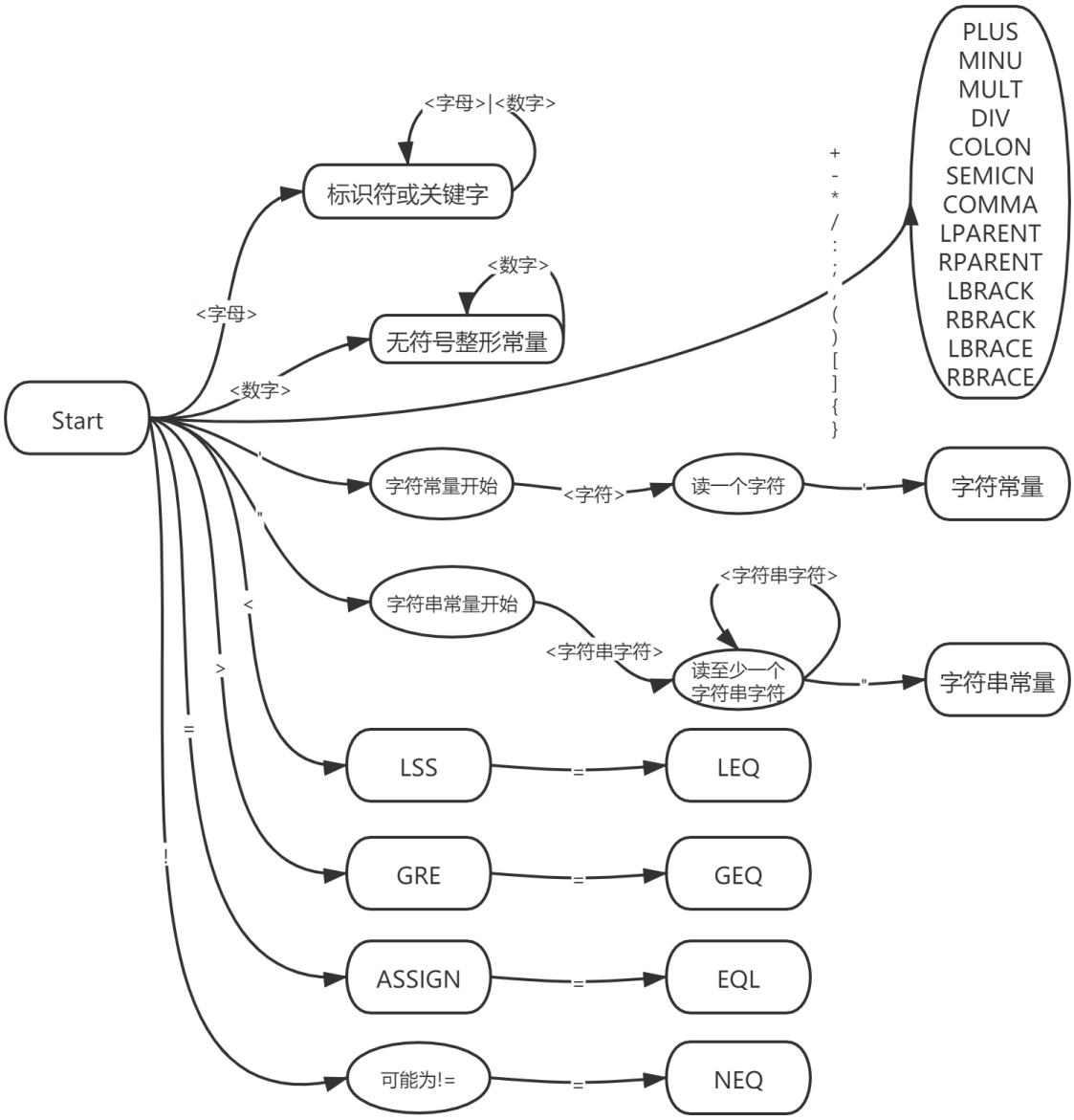
此为词法分析阶段的主类，其中设置全局的当前字符ch用于保存当前处理到的字符；整体结构如下所示，$parseTk()$返回值用于控制是否还能继续读入文本。

```
class Lexic
    {
    private:
        Reader* reader;
        Printer* printer;
        char ch;

    public:
        Lexic(const string& fIn, const string& fOut);
        ~Lexic();

    private:
        static bool _isBlank(const char& c);
        static bool _isLetter(const char& c);
        static bool _isDigit(const char& c);
        static bool _isCharLetter(const char& c);
        static bool _isStringLetter(const char& c);
        static bool _isReservedToken(const char* buffer);
        void _readNext();
        void _skipBlank();
        void _logtoken(const config::TokenCode& tkcode, const string& value);
        bool _parseTk();

    public:
        void run();
    };
```

词法分析部分的=入口函数定义为$run()$，具体实现和结束控制如下所示

```
void lexic::Lexic::run()
{
    while (_parseTk());
}
```

核心部分为字符串识别自动机，其中椭圆形状为单圈节点，圆矩形为双圈节点，双圈节点代表接受该字符串，得到识别。



语法分析部分

语法分析部分整体采用非回溯的递归下降子程序法进行，通过预读的方式进行，**LL(k)文法**

Token序列多遍传递数据结构

经过词法分析部分后，得到由Token及其值组成的序列，观察文法能够更发现其已经消除了左递归文法，但可能出现需要回溯递归下降，考虑通过预读多个Token进行分支的选择，避免回溯。

在词法分析和语法分析之间，通过一种数据结构进行中间表达形式的传递，定义抽象数据结构**PeekableQueue**，能够支持

- push(Token), 向队列尾部添加一个元素
- peek(k), 查看队列首部第k个元素，但并不对原队列进行任何修改，并且如果元素个数不足k个则直接以空Token EMPTY作为结果
- pop(k), 弹出队列首部的k个元素，不足k个则以实际个数为准

数据结构的代码形式声明如下：

```
class PeekableQueue
{
private:
    vector<Token> queue;

public:
    PeekableQueue();

public:
    void push(const Token& tkpair);
    Token peek(const int& k = 1) const;
    void pop(const int& k = 1);
};
```

基于预读的递归下降子程序法

预读情况分支选择表

为方便表述，这里用$token[i]$表示当前预读队列首开始的第i个Token, 在设计层面做如下分析

层次	可选项	进入条件
< 程序 >	[< 常量说明 >]	token[1]==CONSTTK
	[< 变量说明 >]	token[3]!=LPARENT
	{ < 有返回值函数定义 > | < 无返回值函数定义 > }	token[2]!=MAINTK
	< 有返回值函数定义 >	token[1]!=VOIDTK
	< 无返回值函数定义 >	token[1]==VOIDTK
< 常量说明 >	const < 常量定义 > ;	循环条件token[1]==CONSTTK
< 变量说明 >	< 变量定义 > ;	循环条件 token[1]==INTTK|CHARTK and token[2]==IDENFR and token[3]!=LPARENT
< 主函数 >	void main‘(‘’)’ ‘{’ < 复合语句 > ‘}’	TRUE
< 有返回值函数定义 >	< 声明头部 > '(' < 参数表 > ')' '{' < 复合语句 > '}'	TRUE
< 无返回值函数定义 >	void < 标识符 > '(' < 参数表 > ')''{' < 复合语句 > '}'	TRUE
< 有返回值函数调用语句 >	< 标识符 > '(' < 值参数表 > ')'	TRUE
< 无返回值函数调用语句 >	< 标识符 > '(' < 值参数表 > ')'	TRUE
< 声明头部 >	int < 标识符 >	token[1]==INTTK
	char < 标识符 >	token[1]==CHARTK
< 复合语句	[< 常量说明 >]	token[1]==CONSTTK
	[< 变量说明 >]	token[1]==INTTK|CHARTK
< 语句列 >	< 语句 >	循环条件token[1]!=RBRACE
< 语句 >	< 空 > ;	token[1]==SEMICN
	'{' < 语句列 > '}'	token[1]==LBRACE

	<循环语句>	token[1]==FORTK|WHILETK
	<条件语句>	token[1]==IFTK
	<读语句>;	token[1]==SCANFTK
	<写语句>;	token[1]==PRINTFTK
	<返回语句>;	token[1]==RETURNTK
	<情况语句>	token[1]==SWITCHTK
	<有返回值函数调用语句>; | <无返回值函数调用语句>; | <赋值语句>;	token[1]==IDENFR
	<有返回值函数调用语句> | <无返回值函数调用语句>	token[2]==LPARENT
	<赋值语句>	token[2]!=LPARENT
	<有返回值函数调用语句>	符号表记录标识符为有返回值类型
	<无返回值函数调用语句>	符号表记录标识符为无返回值类型
<赋值语句>	<标识符> = <表达式>	token[2]!=LBRACK
	<标识符>'['<表达式>']'=<表达式>	token[2]==LBRACK and token[5]!=LBRACK
	<标识符>'['<表达式>']''['<表达式>']' = <表达式>	token[2]==LBRACK and token[5]==LBRACK
<条件语句>	[else <语句>]	token[1]==ELSETK
<循环语句>	while '('<条件>')'<语句>	token[1]==WHILETK
	for'('<标识符> = <表达式>;<条件>;<标识符> = <标识符>(+|-)<步长>')'<语句>	token[1]==FORTK
<读语句>	scanf '('<标识符>')'	token[1]==SCANFTK
<写语句>	printf '(' <字符串>,<表达式> ')'	token[3]==STRCON and token[4]==COMMA
	printf '('<字符串> ')'	token[3]==STRCON and token[4]!=COMMA
	printf '('<表达式>')'	token[3]!=STRCON
<情况语句>	switch '('<表达式>')' '{'<情况表><缺省>'}	TRUE
<返回语句>	return['('<表达式>')']	token[2]==LPARENT
	return	token[2]!=LPARENT
<表达式>	[+ | -]	token[1]==PLUS|MINU

	{＜加法运算符＞ ＜项＞}	循环条件token[1]==PLUS|MINU
＜项＞	＜乘法运算符＞ ＜因子＞	循环条件token[1]==MULT|DIV
＜因子＞	＜字符＞	token[1]==CHARCON
	＜整数＞	token[1]==PLUS|MINU|INTCON
	'('＜表达式＞')'	token[1]==LPARENT
	＜标识符＞ ｜ ＜标识符＞'['＜表达式＞']'｜＜标识符＞'['＜表达式＞']''['＜表达式＞']'｜＜有返回值函数调用语句＞	token[1]==IDENFR
	＜有返回值函数调用语句＞	token[2]==LPARENT
	＜标识符＞ ｜ ＜标识符＞'['＜表达式＞']'｜＜标识符＞'['＜表达式＞']''['＜表达式＞']'	token[1]!=IDENFR
	＜标识符＞	token[2]!=LBRACK
	＜标识符＞'['＜表达式＞']'	token[2]==LBRACK and token[5]!=LBRACK
	＜标识符＞'['＜表达式＞']''['＜表达式＞']'	token[2]==LBRACK and token[5]==LBRACK
＜参数表＞	＜空＞	token[1]!=INTTK and token[1]!=CHARTK
	＜类型标识符＞ ＜标识符＞{,＜类型标识符＞ ＜标识符＞}	token[1]==INTTK|CHARTK
	{,＜类型标识符＞ ＜标识符＞}	循环条件token[1]==COMMA
＜值参数表＞	＜空＞	token[1]==RPARENT
	＜表达式＞{,＜表达式＞}	token[1]!=RPARENT
	{,＜表达式＞}	循环条件token[1]==COMMA
＜情况表＞	{＜情况子语句＞}	循环条件token[1]==CASETK
＜情况子语句＞	case＜常量＞： ＜语句＞	TRUE
＜缺省＞	default : ＜语句＞	TRUE
＜条件＞	＜表达式＞ ＜关系运算符＞ ＜表达式＞	TRUE
＜步长＞	＜无符号整数＞	TRUE
＜常量定义＞	int＜标识符＞ = ＜整数＞{,＜标识符＞ = ＜整数＞}	token[1]==INTTK

	char < 标识符 > = < 字符 > {, < 标识符 > = < 字符 > }	token[1]==CHARTK
	{, < 标识符 > = < 整数 > }	循环条件token[1]==COMMA
	{, < 标识符 > = < 字符 > }	循环条件token[1]==COMMA
< 变量定义无初始化 >	{,(< 标识符 > | < 标识符 > '[' < 无符号整数 > ']' | < 标识符 > '[' < 无符号整数 > ']''[' < 无符号整数 > ']')}	循环条件token[1]==COMMA
	< 标识符 >	token[2]!=LBRACK
	< 标识符 > '[' < 无符号整数 > ']'	token[2]==LBRACK and token[5]!=LBRACK
	< 标识符 > '[' < 无符号整数 > ']''[' < 无符号整数 > ']'	token[2]==LBRACK and token[5]==LBRACK
< 变量定义及初始化 >	< 类型标识符 > < 标识符 > = < 常量 >	token[3]!=LBRACK
	< 类型标识符 > < 标识符 > '[' < 无符号整数 > ']'='{' < 常量 > {, < 常量 > }'} '	token[3]==LBRACK and token[6]!=LBRACK
	< 类型标识符 > < 标识符 > '[' < 无符号整数 > ']''[' < 无符号整数 > ']'='{''{' < 常量 > {, < 常量 > }'}'{, '{' < 常量 > {, < 常量 > }'}'}'}'	token[3]==LBRACK and token[6]==LBRACK
< 整数 >	[+ | -]	token[1]==PLUS|MINU
< 常量 >	< 字符 >	token[1]==CHARCON
	< 整数 >	token[1]!=CHARCON

递归结构

通过递归调用，每一层次判断应该进入的分支，进行匹配，遇到非终结符号时，通过调用子处理程序进行递归处理。如下为语法分析主要部分：

```
class Syntactic
    {
    private:
        PeekableQueue* queue;
        Printer* printer;
        symbol::SymbolManager* symbolManager;

    public:
        Syntactic(const string& fOut, PeekableQueue* _queue,
symbol::SymbolManager* _symbolManager);
        ~Syntactic();

    private:
        Token _cur();
        void _next();
        void _printAndNext();
        bool _isComeFirstThan(const config::TokenCode& tkcode1, const
config::TokenCode& tkcode2) const;
```

```
public:
    void parseProgram();

private:
    // Illustration
    void parseConstIllustration();
    void parseVarIllustration();
    // Function
    void parseMainFunction();
    void parseFunctionValuedDeclaration();
    void parseFunctionVoidDeclaration();
    void parseFunctionValuedCallStatement();
    void parseFunctionVoidCallStatement();
    void parseDeclarationHead(string& _idenfr);
    // Statement
    void parseCompoundStatement();
    void parseStatementList();
    void parseStatement();
    void parseAssignStatement();
    void parseConditionStatement();
    void parseLoopStatement();
    void parseWhileStatement();
    void parseForStatement();
    void parseReadStatement();
    void parsePrintStatement();
    void parseSwitchStatement();
    void parseReturnStatement();
    // Expression
    void parseExpression();
    void parseTerm();
    void parseFactor();
    // Parameter
    void parseParameterDeclarationList();
    void parseParameterValueList();
    // Conditional component
    void parseCaseList();
    void parseCaseSubStatement();
    void parseDefault();
    void parseCondition();
    void parseStepLength(int& _step);
    // Const & Var
    void parseConstDeclaration();
    void parseVarDeclaration();
    void parseVarDeclarationUninitialized();
    void parseVarDeclarationInitialized();
    // Values
    void parseInteger(int& _integer);
    void parseUnsigned(int& _unsigned);
    void parseString(string& _str);
    void parseConstant(int& _value, bool& _isInteger);
};
```

符号表的初步设计

语法分析部分本应该为上下文无关文法，但涉及到如下语法时，无法单从预读的方式来判断应该选用哪个分支

```
<有返回值函数调用语句> ::= <标识符>'('<值参数表>')'          /*测试程序需出现有返回值的函数调用语句*/
<无返回值函数调用语句> ::= <标识符>'('<值参数表>')'          /*测试程序需出现无返回值的函数调用语句*/
```

因此一种方便的可扩展的解决方式为通过符号表对相应函数定义时候的标识符进行属性记录，在调用时查表可以知道应选用有返回值还是无返回值函数调用。

具体符号表设计链接到后文

符号表设计

动态子表结构

设计思路

进入子结构（类C语言中目前只有函数和主函数需要）建立新的子表，执行定位和重定位操作，查表按照从里到外的顺序查表，但是为了**实现更普遍的情况**，此**编译器的符号表采用支持分程序结构的符号表**设计思路，按照子表结构进行管理；查找时从里到外的作用域逐渐查找，直到找到标识符或者发现当前查询的标识符未定义为止；插入时总是插入到当前最内层的作用域对应的子表中。

编码实现

通过SymbolManager类对整个符号表进行管理，用变量curTable标出当前所在子表层次，便于进行动态管理；而定位和重定位操作通过$pushNewScope()$和$popCurScope()$实现，分别为建立一个作用域和删除最里层作用域。

如下为符号表的整体接口情况：

```
class SymbolManager
    {
    private:
        vector<Table> tables;
        int curTable;

    public:
        SymbolManager();

    public:
        bool hasSymbolInScope(const string& symbol) const;
        bool hasSymbolInAll(const string& symbol) const;
        Info& getInfoInAll(const string& symbol) const;
        Info& getInfoFromLastScope(const string& symbol) const;
        bool declareSymbol(const string& symbol, const Info& info);
        void pushNewScope();
        void popCurScope();
    };
```

表内元素管理

设计思路

对于每一个子表，使用挂链法的哈希表进行子表内部结构管理，以标识符作为Key，能够找到对应唯一的属性信息Value；每一次查找当前表内有无元素时，通过hash函数映射到bucket编号上， 查询目标bucket对应的链表中有无带查找元素，如果没有便是未找到；插入时同理。

编码实现

在具体实现上，通过Table类进行子表管理，如下为相应接口：

```
class Table
    {
    private:
        unordered_map<string, Info> data;

    public:
        Table();

    public:
        bool hasKey(const string& _key) const;
        Info& getInfo(const string& key) const;
        bool insertRecord(const string& key, const Info& info);
    };
```

属性信息

设计思路

通过名字可以找到唯一的一个属性信息结构，其中记录了如下字段和相应表示的信息：

属性名称	属性意义
$symbolType$	区别CONST，VAR，FUNCTION三种符号类型
$dataType$	区别INT，CHAR，VOID三种取值类型
$arrayDim$	如果是非函数类型，表示数组维数，其中非数组标识为维数dim=0
$dimLimit_{0,1}$	用于表示数组定义时相应维度定义的长度，即数组模板信息
$declareRow$	标识符定义的时候对应的行数
$referRows...$	表示标识符引用的时候对应的行数列表
$funcParamDataTypeList$	如果当前信息记录的是函数，那么还需要记录函数参数表的个数和相应参数类型向量，个数通过向量长度确定
$address$	用于表示分配的内存地址，数组则是内存中的数组首地址

编码实现

由于符号表Entry属性的填充是**动态更新**的，很多信息需要回填，在实现上添加三个控制标签$ctrlDeclared$和$ctrlAddressed$和$ctrlParamListFilled$分别表示该标识符**是否已经完善了定义、该标识符是否已经进行了地址分配、该标识符是否回填了参数表的信息**，并通过手工assert断言进行保护，避免意想不到的情况出现，声明结构如下：

```
class Info
    {
    private:
        config::SymbolType symbolType;
        config::DataType dataType;
        uint arrayDim;
        uint dimLimit[2];
        uint declareRow;
        vector<uint> referRows;
        vector<config::DataType> funcParamDataTypeList;
        uint address;

    private:
        bool ctrlDeclared;
        bool ctrlParamListFilled;
        bool ctrlAddressed;

    public:
        Info();
        Info(const config::SymbolType &_symbolType, const config::DataType
&_dataType, const uint &_declareRow,
             const uint &_arrayDim = 0, const uint &_dim0 = 0, const uint &_dim1
= 0,
             const std::initializer_list<config::DataType> &
_funcParamDataTypeList = {});

    public:
        void logReference(const uint& _row);
        void logAddress(const uint& _address);
        void logFuncParam(const vector<config::DataType> &_paramList);
        bool checkDeclared() const;
        bool checkAddressed() const;
        void assertDeclared() const;
        void assertAddressed() const;
        void assertParamFilled() const;
        bool isFunction() const;
        bool isValuedFunction() const;
        bool isVoidFunction() const;
        bool isSymbolTypeOf(const config::SymbolType & _symbolType) const;
        bool isDataTypeOf(const config::DataType & _dataType) const;
        config::DataType queryDataType() const;
        bool isDimOf(const int & _dims, const int & _dimLim0 = 0, const int &
_dimLim1 = 0) const;
        int queryFuncParamCount() const;
        bool checkFuncParamMatchAt(const int & _index, const config::DataType &
_dataType);
        vector<config::DataType> queryParamDataTypeListOfFunction() const;
    };
```

错误处理设计

错误细分设计表

Code	ErrorType	Description
A	IllegalLetterChar	字符常量中出现非法字符
	IllegalLetterString	字符串中出现非法字符
	EmptyCharOrString	字符常量或字符串中为空
	CharLengthError	字符常量单引号中的字符个数超过1
B	DuplicatedName	标识符重复定义
C	UndefinedName	引用未定义的标识符
D	FunctionParamCountMismatch	函数调用中传入参数的个数与函数模板中的个数不匹配
E	FunctionParamTypeMismatch	函数调用中传入参数的类型存在与模板不匹配的情况
F	IllegalTypeInCondition	条件中比较的两端类型存在非int的情况
G	VoidFunctionWithParents	无返回值函数存在return ();的语句
	VoidFunctionWithValue	无返回值函数存在return(表达式);的语句
H	ValuedFunctionWithoutReturn	有返回值函数中不存在return语句
	ValuedFunctionWithVoid	有返回值函数中存在return;的语句
	ValuedFunctionWithParents	有返回值函数中存在return();的语句
	ValuedFunctionReturnTypeMismatch	有返回值函数中存在返回值类型不匹配的return语句
I	ArraySubIndexTypeNotInt	数组引用时下标不为int类型
J	ModifyConstWithAssign	赋值语句中对常量标识符进行修改
	ModifyConstWithScanf	读语句中对常量标识符进行修改
K	ExpectSemicnInStatementEnd	在七种语句结尾缺少;
	ExpectSemicnInFor	在for语句圆括号内缺少;
	ExpectSemicnAtConstVarDeclarationEnd	在常量定义或变量定义末尾处缺少;
L	ExpectRParentAtFunctionCall	函数调用处缺少)

	ExpectRParentAtFunctionDeclaration	函数定义处缺少)
	ExpectRParentAtMain	Main函数定义处缺少)
	ExpectRParentAtExpression	在带括号表达式缺少中缺少右端的;
	ExpectRParentAtIf	在if括号内缺少右端;
	ExpectRParentAtWhile	在while括号内缺少右端;
	ExpectRParentAtFor	在for括号内缺少右端;
	ExpectRParentAtSwitch	在switch括号内缺少右端;
	ExpectRParentAtScanf	在scanf括号内缺少右端;
	ExpectRParentAtPrintf	在printf括号内缺少右端;
	ExpectRParentAtReturn	在return括号内缺少右端;
M	ExpectRBrackAtArrayDeclaration	在数组定义时缺少某个右端]
	ExpectRBrackAtArrayUseInFactor	因子中在数组引用时缺少某个右端]
	ExpectRBrackAtArrayUseInAssignLeft	在赋值语句左端部分引用数组时缺少]
N	ArrayInitMismatchWithTemplate	数组定义及初始化时初始化内容任一维度的元素个数不匹配或缺少某一维的元素
O	ConstantTypeMismatchInVarDeclarationAndInit	变量定义及初始化中初始化的类型与赋值右端不匹配
	ConstantTypeMismatchInSwitchCase	switch选择的类型与case中常量类型不一致
P	ExpectDefaultStatement	缺少缺省语句

跳读设计表

跳读分为两个层次，分别为**词法分析时产生错误**和**语法分析阶段产生错误**，两者的跳读分别为字符级别的和Token级别的。

下面用None表示不需要跳读的情况。

词法分析跳读分析表

ErrorType	SkipUntil (excusive of common stop chars)
IllegalLetterChar	'
IllegalLetterString	"
EmptyCharOrString	None
CharLengthError	'

语法分析跳读分析表

ErrorType	SkipUntil (exclusive of common stop words)
DuplicatedName	None
UndefinedName	None
FunctionParamCountMismatch(more)	COMMA, RPARENT
FunctionParamCountMismatch(less)	None
FunctionParamTypeMismatch	COMMA, RPARENT
IllegalTypeInCondition	None
VoidFunctionWithParents	None
VoidFunctionWithValue	None
ValuedFunctionWithoutReturn	None
ValuedFunctionWithVoid	None
ValuedFunctionWithParents	None
ValuedFunctionReturnTypeMismatch	None
ArraySubIndexTypeNotInt	RBRACK
ModifyConstWithAssign	None
ModifyConstWithScanf	None
ExpectSemicnInStatementEnd	None
ExpectSemicnInFor	None
ExpectSemicnAtConstVarDeclarationEnd	None
ExpectRParentAtFunctionCall	None
ExpectRParentAtFunctionDeclaration	None
ExpectRParentAtMain	None
ExpectRParentAtExpression	None
ExpectRParentAtExpression	None
ExpectRParentAtWhile	None
ExpectRParentAtFor	None
ExpectRParentAtSwitch	None
ExpectRParentAtScanf	None
ExpectRParentAtPrintf	None
ExpectRParentAtReturn	None
ExpectRBrackAtArrayDeclaration	None

ErrorType	SkipUntil (exclusive of common stop words)
ExpectRBrackAtArrayUseInFactor	None
ExpectRBrackAtArrayUseInAssignLeft	None
ArrayInitMismatchWithTemplate	SEMICN
ConstantTypeMismatchInVarDeclarationAndInit	None
ConstantTypeMismatchInSwitchCase	None
ExpectDefaultStatement	None

具体实现

将错误处理模块进行封装单独管理，由于词法分析和语法分析分为两遍，首先进行词法分析再进行语法分析，错误输出的顺序不能得到保证，因此实现上采用优先队列管理每一个错误ErrorEntry，并对自定义的ErrorEntry进行以行为关键字的有序维护。

以下是实现的ErrorEntry和ErrorManager的接口：

```
class ErrorEntry
    {
    private:
        int row;
        int column;
        config::ErrorType type;
        std::string description;

    public:
        ErrorEntry(const int & _row, const int & _column, const
config::ErrorType & _type, const std::string & _description);
        bool operator < (const ErrorEntry & other) const;
        bool operator > (const ErrorEntry & other) const;
        std::string to_string(const bool & detailed = false);
    };

class ErrorManager
    {
    private:
        const bool enable_print_tuple = config::PRINT_ERROR_TUPLE;
        const bool enable_detailed_info = config::PRINT_DETAILED_ERROR;
        std::priority_queue<ErrorEntry, std::vector<ErrorEntry>,
std::greater<ErrorEntry> > errors;
        bool useCout;
        std::ofstream fsOut;
        bool watch;

    public:
        ErrorManager();
        ErrorManager(const std::string & fOut);
        ~ErrorManager();

    public:
```

```
        void insertError(const int & row, const int & column, const
config::ErrorType & type, const std::string & description);
        void printAllErrors();
        void watchErrors();
        bool queryWatch() const;
    };
```

语义分析与中间代码生成部分

中间代码设计

采用如下的四元式中间代码：

OP	OUT	INL	INR	Meanings	Example
ADD_IR	c	a	b	加法运算	c=a+b
MINUS_IR	c	a	b	减法运算	c=a-b
MULT_IR	c	a	b	乘法运算	c=a*b
DIV_IR	c	a	b	除法运算	c=a/b
LOAD_IR	c	arr	idx	取数组元素值	c=arr[idx]
STORE_IR	c	arr	idx	向数组元素赋值包括单变量初始化	arr[idx]=c
MOVE_IR	b	a		单变量赋值操作	b=a
BEQ_IR	label	exp1	exp2	相等跳转	if exp1==exp2 goto label
BNE_IR	label	exp1	exp2	不等跳转	if exp1!=exp2 goto label
BLE_IR	label	exp1	exp2	小于等于跳转	if exp1<=exp2 goto lable
BLT_IR	label	exp1	exp2	小于跳转	if exp1<exp2 goto label
JUMP_IR	label			无条件跳转	jump label
PARA_IR	names			定义函数参数列表（#分隔参数）	para #name1#name2
PUSH_IR	names			传入实参列表	push #name1#name2
DEPUSH_IR	paramcount			弹栈参数个数占用空间	depush 3
CALL_IR	funcname			调用函数	call calc
RET_IR	x			函数返回语句，空为无返回值	ret x | ret
MOVRET_IR	b			将函数返回值放入指定变量	b = RET
READ_IR	name	type		读入指定类型的变量	read x int
WRITE_IR	name	type		输出指定类型的值	write x char
STRING_IR	newname			输出字符串常量	STRING_IR __String__0
SETLABEL_IR	label			设置标签label	label_0:
EXIT_IR	exitLabel			主函数中退出整个程序	exit exitLabel

OP	OUT	INL	INR	Meanings	Example

中间代码数据结构

为了便于中间代码层面的优化，以及后续目标代码生成工作，此处采用 **程序 - 函数 - 基本块 - 中间代码** 的层次结构进行中间代码储存结构设计，其中**程序**记录了全局量和程序中的所有字符串以及包括main的若干个函数，**函数**则记录了参数表、函数名和基本块信息，**基本块**则由若干四元组中间代码组成。

程序-MIR

顶层中间代码数据结构包含全局的常量、变量、待输出的字符串，以及若干函数单元。

中间代码最外层程序数据结构的设计包括**存储和添加信息与查询内容**等接口，具体接口如下：

```
class MIR
    {
    private:
        bool sealed;
        MapDeclareString globalStrings; // strName -> strValue
        MapDeclareChar globalChars; // name -> initList
        MapDeclareInt globalInts; // name -> initList
        std::vector<inter::Proc> procedures; // no global proc used

    public:
        MIR(const bool & _sealed = false);

    public:
        const MapDeclareString &queryGlobalStrings() const;
        const MapDeclareChar &queryGlobalChars() const;
        const MapDeclareInt &queryGlobalInts() const;
        std::vector<inter::Proc> queryFunctions() const;
        const inter::Proc &queryMainProc() const;

    public:
        void declareGlobalString(const std::string & _strName, const std::string
& _strContent);
        void declareGlobalChar(const std::string & _name, const int & _count,
const std::vector<char> & _initValues = {});
        void declareGlobalInt(const std::string & _name, const int & _count,
const std::vector<int> & _initValues = {});
        void declareLocalChar(const std::string & name, const InitChar &
initChar);
        void declareLocalInt(const std::string & name, const InitInt & initInt);
        void newProc(const std::string & _procName);
        void addParam(const std::string &_name, const std::string &_type);
        void doneGenerationToBlocks();
        void addQuad(const Quad & _quad);
        void assertNotSeal() const;
        void assertIsSeal() const;
        void print(const std::string &_fOut) const;
    };
```

函数-Proc

函数作为**程序**的子模块，对包括Main函数在内的函数进行管理，其中包括参数具体信息的记录、声明的局部变量的记录、以及根据记录完成的四元组序列生成基本块的功能。

在实现上，采用了在语义分析中直接把四元组插入到当前函数的四元组序列中，并在添加完毕后设置不可变标记，并生成相应的不可变基本块数据结构；同时包含输出整个函数的中间代码信息的接口，相应的接口如下所示：

```
class Proc
    {
    private:
        bool isBlockForm;
        std::string procName;
        ParasList parasList;
        MapDeclareChar localChars;
        MapDeclareInt localInts;
        std::vector<inter::Quad> lines;
        std::vector<inter::Block> blocks;

    public:
        Proc(const std::string & _procName, const bool & _isBlockForm = false);

    public:
        const string &queryProcName() const;
        const ParasList &queryParasList() const;
        const MapDeclareChar &queryLocalChars() const;
        const MapDeclareInt &queryLocalInts() const;
        const std::vector<inter::Quad> &queryLines() const;
        const std::vector<inter::Block> &queryBlocks() const;

    public:
        void assertBlockForm() const;
        void addParam(const std::string &_name, const std::string &_type);
        void addLocalChar(const std::string & name, const InitChar & initChar);
        void addLocalInt(const std::string & name, const InitInt & initInt);
        void addQuad(const Quad & _quad);
        void buildBlocks();
        void print(std::ofstream &_fsOut) const;
    };
```

基本块-Block

程序由若干个基本块组成，基本块中由若干四元组序列组成，其中基本块中间不包含跳转到其他基本块的中间代码。

在程序实现上，包含输出中间代码的接口，由以下接口构成：

```
class Block
    {
    private:
        std::vector<inter::Quad> lines;

    public:
        Block(std::vector<inter::Quad>::iterator _begin,
std::vector<inter::Quad>::iterator _end);

    public:
        const std::vector<inter::Quad> & queryLines() const;
        void print(std::ofstream &_fsOut) const;
    };
```

四元组-Quad

四元组是最基本的中间代码单元，并且设计了输出中间代码的接口：

```
class Quad
    {
    public:
        config::IRCode op;
        std::string out;
        std::string inl;
        std::string inr;

    public:
        Quad(const config::IRCode & _op, const std::string & _out, const
std::string & _inl, const std::string & _inr);

    public:
        void print(std::ofstream &_fsOut) const;
    };
```

中间代码生成

在语义分析单元中插入属性翻译模块，所有的中间代码生成通过操作该属性翻译单元的接口进行实现。

全局序号管理功能

由于在中间代码生成的过程中，需要在全局层面上引入标签计数、临时变量计数、全局字符串计数等支持，因此在属性翻译模块中设计标签、临时变量和字符串的标识符生成功能。例如，程序中的第二个标签为__Label__2，第五个临时变量为__Temp__5，在需要时进行扩张后的标识符生成和管理。

在程序实现上，通过如下几个函数对全局序号进行记录和生成：

```
std::string semantic::Semantic::genTemp()
{
    tempIdx++;
    return config::tempHead + toString(tempIdx);
}

std::string semantic::Semantic::genLabel()
{
    labelIdx++;
    return config::labelHead + toString(labelIdx);
```

```
}

std::string semantic::Semantic::genGlobalString(const std::string & _strContent)
{
    stringIdx++;
    string name = config::stringHead + toString(stringIdx);
    mir.declareGlobalString(name, _strContent);
    return name;
}
```

递归录制功能

在生成如**for**、**while**、**switch**等结构的中间代码时，涉及到将一部分分支的中间代码进行录制，并需要支持一整块中间代码的插入功能，例如，**switch**的生成中，通过对每个case和default分支的录制操作，能够对整个switch语句块进行全局的管理，合理调配生成的顺序以及标签的插入；同时，这种录制的策略便于后续优化的实现，包括改变while等语句块的翻译方法，减少1个跳转指令等优化。

需要注意的点在于递归录制的支持，例如在switch语句块内部嵌套了另一个switch语句块，直接通过**是否在录制**的记录并不能有效还原录制的中间代码，因此采用多层的记录方式，每次开始录制时都在中间代码块的记录栈中压入一个空的中间代码块，即开始一层新的中间代码录制，结束当前层次的录制时，弹出顶层的中间代码块；相应在插入一整块中间代码时则判断栈是否为空，只有在栈为空时才插入到实际生成的中间代码MIR中，否则插入的中间代码块加入到栈顶的中间代码块之中，即返还到上一层的录制中。

程序实现上，通过以下属性记录递归录制的状态：

```
std::vector<std::vector<inter::Quad> > _recorded;
int _isRecording; // 记录当前录制的层数，与_recorded中代码块的个数对应，为0时表示当前不在录制状态下，可以直接对生成的中间代码进行操作
```

通过如下两个函数进行递归录制的新建一层和结束当前层的操作

```
void semantic::Semantic::startRecording()
{
    if (this->errored)
        return;
    this->_isRecording += 1;
    this->_recorded.emplace_back();
}

std::vector<inter::Quad> semantic::Semantic::endRecording()
{
    if (this->errored)
        return std::vector<inter::Quad>();
    if (this->_isRecording <= 0)
        std::cerr << "Not recording when terminating records !" << std::endl;
    this->_isRecording -= 1;
    std::vector<inter::Quad> ret = this->_recorded.back();
    this->_recorded.pop_back();
    return ret;
}
```

短路机制

为了增强鲁棒性，在属性翻译模块中与错误处理模块连接，记录当前是否存在错误，若有错误，则属性翻译模块中所有函数都将进行短路操作，即不实际生成代码，直接返回上层逻辑。这种短路机制的引入主要是为了避免错误处理测试中，由于输入程序的错误导致编译器本身发生错误，最终导致运行时错误或死循环等。

以函数$endRecording$为例，短路机制的实现如下：

```
std::vector<inter::Quad> semantic::Semantic::endRecording()
{
    // 此处通过errored记录目前是否存在错误，若存在错误则直接以安全的状态返回
    if (this->errored)
        return std::vector<inter::Quad>();
    if (this->_isRecording <= 0)
        std::cerr << "Not recording when terminating records !" << std::endl;
    this->_isRecording -= 1;
    std::vector<inter::Quad> ret = this->_recorded.back();
    this->_recorded.pop_back();
    return ret;
}
```

中间代码输出示例

```
# Global Strings :
    ___String__1 = "-";

# Global Chars :

# Global Ints :
    ___Global__bigintarr = ;
    ___Global__bigintlength = ;
    ___Global__result = ;

___Proc__min:
    Params:
        Para int ___Local__a
        Para int ___Local__b
    LocalChars:
    LocalInts:
    Blocks:
        ************************************* New Block *************************************
        Param                   #___Local__a#___Local__b
        ?<=                     ___Label__1             ___Local__a             ___Local__b

        ************************************* New Block *************************************
        Return                  ___Local__b

        ************************************* New Block *************************************
        JP                      ___Label__2

        ************************************* New Block *************************************
        Label                   ___Label__1
        Return                  ___Local__a

        ************************************* New Block *************************************
        Label                   ___Label__2
```

```
___Proc__subtract:
    Params:
    LocalChars:
    LocalInts:
        ___Local__a = ;
        ___Local__b = ;
        ___Local__c = ;
        ___Local__flag = ;
        ___Local__i = ;
        ___Local__j = ;
        ___Local__k = ;
        ___Local__l1 = ;
        ___Local__l2 = ;
    Blocks:
        ************************************ New Block ************************************
        Param
        LD                      ___Temp__21             ___Global__bigintlength  0
        MV                      ___Local__l1            ___Temp__21
        LD                      ___Temp__22             ___Global__bigintlength  1
        MV                      ___Local__l2            ___Temp__22
        Push                    #___Local__l1#___Local__l2
        Call                    ___Proc__min

        ************************************ New Block ************************************
        DePush                  2
        MoveRet                 ___Temp__23
        MV                      ___Local__k             ___Temp__23
        Push                    #___Local__l1#___Local__l2
        Call                    ___Proc__max

        ************************************ New Block ************************************
        DePush                  2
        MoveRet                 ___Temp__24
        MV                      ___Local__j             ___Temp__24
        Push
        Call                    ___Proc__compare

        ************************************ New Block ************************************
        DePush                  0
        MoveRet                 ___Temp__25
        ?<                      ___Label__15            ___Temp__25             0
```

```
___Proc__main:
    Params:
    LocalChars:
    LocalInts:
        ___Local__i = 0 ;
        ___Local__n = ;
        ___Local__tmp = ;
    Blocks:
        ************************************ New Block ************************************
        SD                      0                       ___Local__i             0
        Read                    ___Local__n             int
        SD                      ___Local__n             ___Global__bigintlength  0
        ?<=                     ___Label__41            ___Local__n             ___Local__i

        ************************************ New Block ************************************
        Label                   ___Label__40
        Read                    ___Local__tmp           int
        *                       ___Temp__81             0                       100
        +                       ___Temp__82             ___Temp__81             ___Local__i
        SD                      ___Local__tmp           ___Global__bigintarr    ___Temp__82
        +                       ___Temp__83             ___Local__i             1
        MV                      ___Local__i             ___Temp__83
        ?<                      ___Label__40            ___Local__i             ___Local__n

        ************************************ New Block ************************************
        Label                   ___Label__41
        Read                    ___Local__n             int
        SD                      ___Local__n             ___Global__bigintlength  1
        MV                      ___Local__i             0
        ?<=                     ___Label__43            ___Local__n             ___Local__i

        ************************************ New Block ************************************
        Label                   ___Label__42
        Read                    ___Local__tmp           int
        *                       ___Temp__84             1                       100
        +                       ___Temp__85             ___Temp__84             ___Local__i
        SD                      ___Local__tmp           ___Global__bigintarr    ___Temp__85
        +                       ___Temp__86             ___Local__i             1
        MV                      ___Local__i             ___Temp__86
        ?<                      ___Label__42            ___Local__i             ___Local__n
```

目标代码生成部分

设计目标代码生成器，输入为中间代码数据结构，输出为目标代码序列。

数据段

在全局数据段的设计中，整体上的设计方案为：

- 常量与变量一视同仁，仅在语法分析和错误处理阶段通过符号表进行检查，在生成阶段无需区分
- 单一变量和数组一视同仁
    - char型：
        - 有初始化：使用**.byte**指令进行初始化
        - 无初始化：使用**.space**开辟内存空间
    - int型：
        - 有初始化：使用**.word**指令进行初始化
        - 无初始化：使用**.space**开辟内存空间
- 具体实现上，需要每种类型初始化前进行相应的**内存对齐操作**

代码段

环境初始化与函数布局

初始化需要对全局指针**$gp**进行设置，便于后续通过全局数据段的指针与全局变量的相对偏移进行访问。

将代码段中各函数生成的代码依次排列，main函数放在最后，通过开始时的一个无条件跳转转向main，能避免main执行完毕后多余代码对结果造成影响；即**数据段——.text——环境的初始化——jmain——各函数——main函数**的布局结构。

具体生成效果如下：

```
.text
li   $gp 0x10010000
j    ___Proc__main
```

函数内代码生成

在函数开始处，通过对参数寄存器$$ai$和上一个栈帧中传入参数的获取进行参数传递，并统计函数中声明的局部变量和临时变量，对其分配栈空间。

值得注意的是，在这种设计中，对每个临时变量都分配了内存空间对其进行储存，虽然大部分都能在临时寄存器分配过程中进行处理，这种设计能够保证需要溢出时快速找到相应地址进行保存与恢复。

寄存器分配策略

临时寄存器分配

采用寄存器池对临时寄存器进行分配，核心为对临时寄存器与标识符变量之间对应关系的记录，以及是否修改过的记录；在生成过程中，需要及时主动标记有修改过的并且需要写回的寄存器；在每个基本块末尾，需要对修改过的非临时变量在内存中的值进行更新。

具体实现上，通过专门的临时寄存器池模块进行管理，在目标代码生成模块中向临时寄存器池发出寄存器申请与更新请求，具体的模块接口如下：

```
class BlockRegPool
    {
    private:
        std::queue <std::string> freePool;
        std::queue <std::string> allocPool;
        std::set <std::string> writebackRegs;
        std::unordered_map<std::string, std::string> reg2mark;
        std::unordered_map<std::string, std::string> mark2reg;

    public:
        BlockRegPool();

    private:
        mips::ObjCodes _writeBack(const std::string & _reg, const
std::map<std::string, mips::SymbolInfo> & mipsTable);
        void _untieLinks(const std::string & _reg);

    public:
        void reset();
        bool hasFree() const;
        void insertFree(const std::string & _reg);
        std::string queryReg2Mark(const std::string & _reg);
```

```
        std::string queryMark2Reg(const std::string & _mark);
        bool hasMark(const std::string & _mark) const;
        mips::ObjCodes allocBlockReg(std::string & _reg, const
std::map<std::string, mips::SymbolInfo> & mipsTable,
                                     const std::set<std::string> & _excludeRegs
= {});
        void markWriteBack(const std::string &_reg);
        mips::ObjCodes saveWriteBackRegs(const std::map<std::string,
mips::SymbolInfo> & mipsTable);
        void updateInfo(const std::string & _reg, const std::string & _mark);
        mips::ObjCodes syncLink(const std::string &_reg, const std::string
&_mark, const bool &_link,
                                const std::map<std::string, mips::SymbolInfo> &
mipsTable);
    };
```

全局寄存器分配

在临时寄存器池的基础上，对变量分配寄存器时增加对是否已经分配全局寄存器的判断，若有则只需要直接使用全局寄存器；具体的分配策略采用图着色法进行分配，启发式选取溢出变量时选择所在循环层数最少的节点进行溢出，这样能尽可能保证溢出节点的代价较小。

函数调用栈设计

以从上到下内存地址递减，栈增加的方向表示，函数调用时的内存布局如下所示：

栈帧	内容	当前指针
…	…	
前一个栈帧	… 局部变量区（含参数） 临时变量溢出区 传入参数（超过4个部分）	$fp
当前栈帧	$fp（前栈帧） $ra 局部变量区（含参数） 临时变量溢出区	$sp

其中运行时执行过程为：

压入值参数——在运行栈中保存调用方的$fp与$ra——function-call跳转到被调用方——分配局部变量区和临时变量区——更新$fp与$sp——从运行栈中取出参数

优化方案

Optim1

pass

Optim2

pass