**需求分析**

**用户需求分析**

**用户需求**

按照源语言（Pascal语言和C语言）的语法和语义，基于Linux操作系统和GCC编译器，设计并实现将Pascal源程序翻译为等价的C源程序的编译程序。

**功能需求**

1. 语法支持：基础语法：变量声明、控制结构、函数/过程、类型系统。
2. 高级语法：嵌套作用域、指针等。
3. 结构等效：begin/end到{}，procedure到对应类型函数，writeln到printf的映射。
4. 类型映射：Integer->int，Real->float，动态数组转C指针+数组长度。
5. 标准库匹配：Read/Write到stdio的匹配。
6. 错误处理：处理语法错误和语义错误。

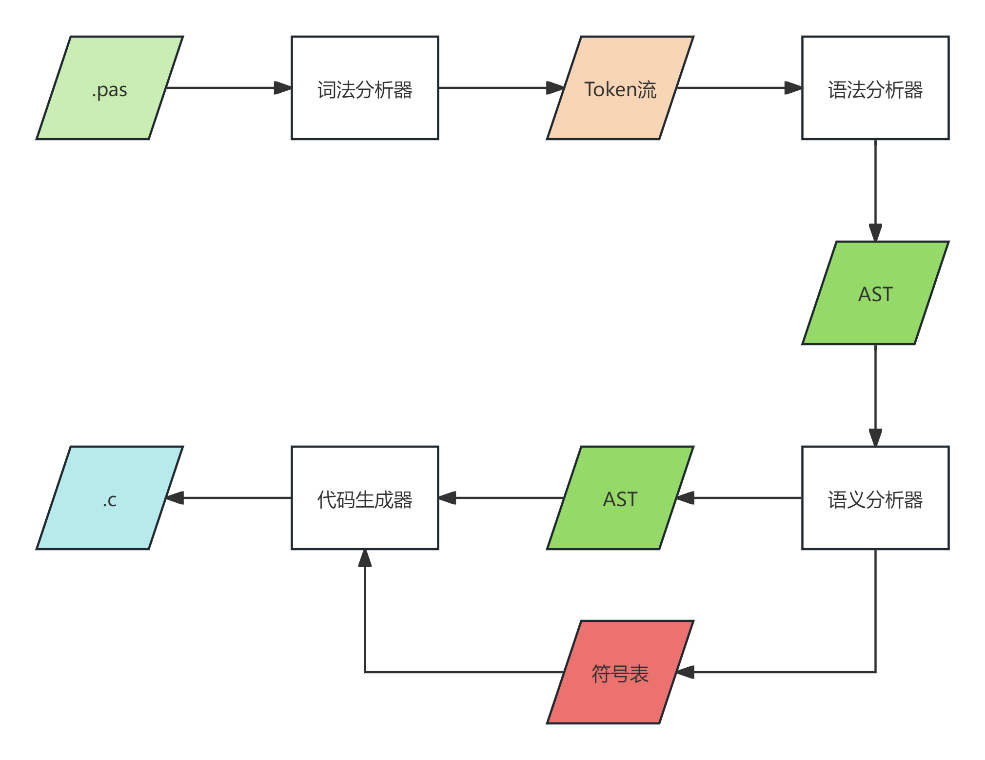
**需求分析**

用户需要的编译程序为一个基于Linux和GCC的Pascal2C的编译器，因此整个编译器需要的主要模块和功能包含以下四个：

1. 词法分析器：对pascal源程序文件的字符流进行处理分类，生成对应的Token流；
2. 语法分析器：对来自词法分析器的Token流进行语法分析，生成语法生成树；
3. 语义分析器：处理来自语法分析器的语法生成树，进行类型检查和符号表管理；
4. 代码生成器：根据语义分析器的结果按照一定的规则生成对应的C程序。

**数据流图**

根据上述的需求分析，对应的项目总体数据流图如下：



**开发环境**

* 操作系统：Linux、Windows
* 开发组件：Flex、Bison、yacc
* 编译器：gcc

**词法分析**

**功能**

输入一个指定的Pascal源程序，对程序中的所有字符进行单词分类，处理无法识别的词法错误，并跳过注释、分隔符，最后输出一个与Pascal程序完整对应的Token流。

**输入**

|  |
| --- |
| Pascal源程序 |

**输出**

|  |
| --- |
| Token流 |

**处理逻辑**

使用Flex组件实现对输入字符流的正则表达式处理，词法分析正则表达式如下：

    DIGIT       [0-9]

    LETTER      [a-zA-Z]

    ID          {LETTER}({LETTER}|{DIGIT})\*

    INTEGER     {DIGIT}+

    EXPONENT    [eE][+-]?{DIGIT}+

    REAL        ({DIGIT}+"."{DIGIT}\*{EXPONENT}?)|({DIGIT}\*"."{DIGIT}+{EXPONENT}?)|({DIGIT}+{EXPONENT})

    NUMBER      {INTEGER}|{REAL}

    COMMENT     \{([^\}])\*\}

    WHITESPACE  [ \t\n]+

%%

    "program"       {}

    "begin"         {}

    "end"           {}

    "if"            {}

    "then"          {}

    "else"          {}

    "while"         {}

    "do"            {}

    "var"           {}

    "integer"       {}

    "real"          {}

    "boolean"       {}

    "function"      {}

    "procedure"     {}

    "div"           {}

    "mod"           {}

    "and"           {}

    "or"            {}

    "not"           {}

    {COMMENT}       {}

    ":="            {}

    "<>"            {}

    "<="            {}

    ">="            {}

    "+"|"-"|"\*"|"/" {}

    "="|"<"|">"     {}

    ";"             {}

    ","             {}

    "("             {}

    ")"             {}

    "["             {}

    "]"             {}

    ":"             {}

    "."             {}

    {NUMBER}        {}

    {ID}            {}

    {WHITESPACE}    {}

    .               {}

<<EOF>>         {}

    %%

**Token设计**

Token的数据结构设计如下：

struct Token {

    TokenType type;

    string lexstr;

    int NumSeq;

    Token\* next;

}Token;

其中TokenType表示该Token所存储的字符串对应的类型而lexstr表示Token所存储的字符串，NumSeq代表Token的序列号。

**单词分类**

Token的分类如下

|  |  |
| --- | --- |
| **类型** | **备注** |
| TOKEN\_EOF | TOKEN流结尾 |
| TOKEN\_KEYWORD | 关键字 |
| TOKEN\_SPECIAL\_SYMBOL | 特殊符号 |
| TOKEN\_OPERATOR | 运算符 |
| TOKEN\_IDENTIFIER | 标识符 |
| TOKEN\_NUMBER | 数字 |
| TOKEN\_UNKNOWN | 未知类型（错误处理） |

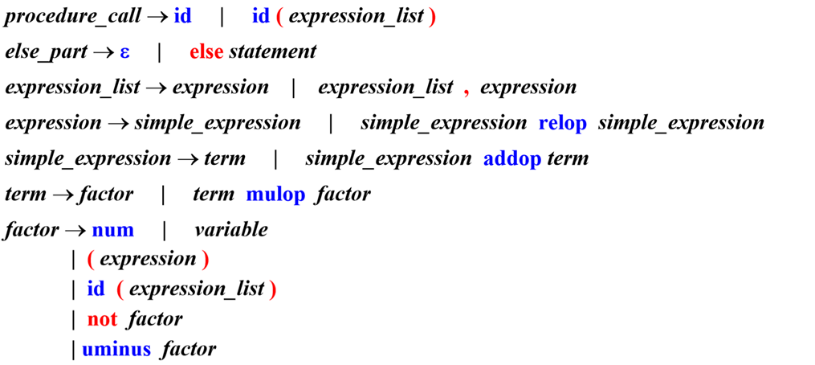
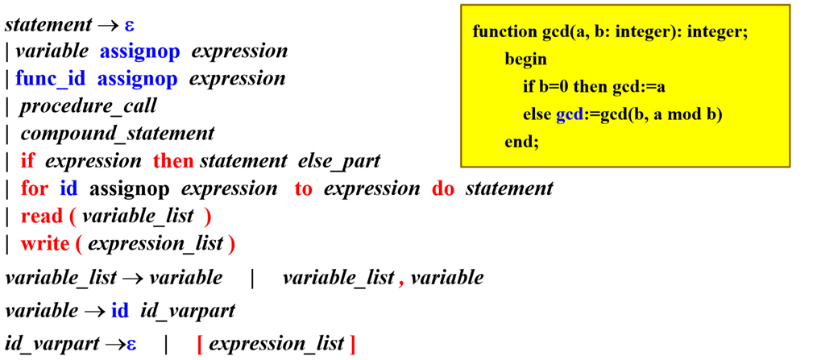
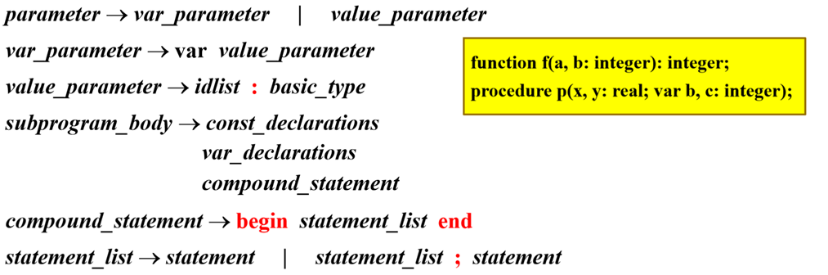
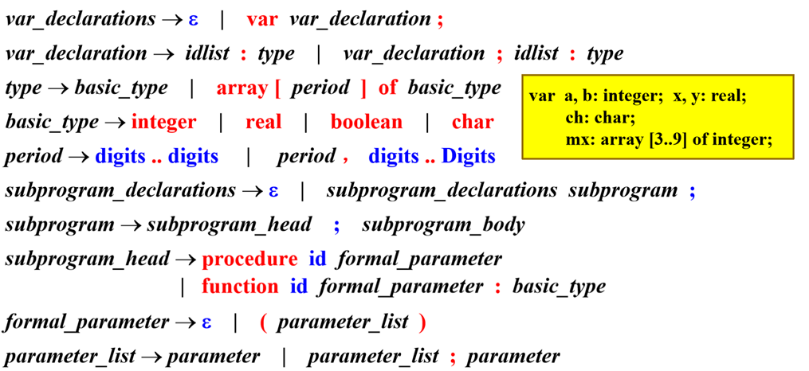
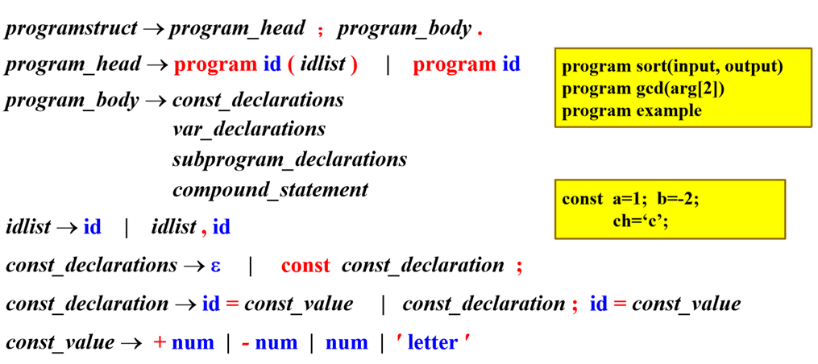
其中，关键字如下：

|  |  |  |  |
| --- | --- | --- | --- |
| program | const | var | array |
| of | integer | real | boolean |
| char | procedure | function | begin |
| end | if | then | for |
| to | do | read | write |
| else | not | [while] |  |

**语法分析**

**语法结构**

语法分析器所需要处理的Pascal-S语法子集如下：



**文法及分析方法**

Pascal语言的文法为上下无关文法，而在语法分析器中可以使用yacc组件进行开发，因此可以使用yacc的LR分析技术对Pascal程序的语法进行分析与检查，生成对应的语法生成树。

**输入**

|  |
| --- |
| Token流 |

**输出**

|  |
| --- |
| 语法生成树 |

**语义分析**

**符号表**

typedef enum {

    SYM\_VARIABLE,     // 变量

    SYM\_FUNCTION,     // 函数（有返回值）

    SYM\_PROCEDURE,    // 过程（无返回值）

    SYM\_PARAMETER,    // 函数/过程的参数

    SYM\_TYPE,         // 类型定义（如 record、array）

    SYM\_CONSTANT,     // 常量

    SYM\_LABEL,        // 标签（goto 目标）

    SYM\_ENUM\_MEMBER   // 枚举成员

    // ...

} SymbolType;

typedef struct TypeInfo {

    SymbolType type;          // 类型类别（TYPE\_BASIC, TYPE\_POINTER 等）

    union {

        // 基础类型（如 integer、char）

        struct {

            varType name;       // ... 枚举内容如上

        } basic;

        // 指针类型（指向另一个 TypeInfo）

        struct {

            struct TypeInfo\* target\_type;  // 指向的类型

        } pointer;

        // 过程/函数类型

        struct {

            struct TypeInfo\* return\_type;   // 函数返回值类型（对过程为 NULL）

            struct ParamList\* params;       // 参数列表

        } proc\_func;

        // 其他类型（如数组、结构体）

    } detail;

} TypeInfo;

typedef struct ParamList {

    char\* name;              // 参数名

    struct TypeInfo\* type;   // 参数类型

    struct ParamList\* next;  // 下一个参数

} ParamList;

typedef struct SymbolEntry {

    char\* name;

    TypeInfo\* data\_type;        // 详细类型信息（新增字段）

    int scope\_level;            // 作用域

    // ... 其他字段

} SymbolEntry,\* SymbolEntryPtr;

**代码生成**

**映射关系**

Pascal的基本类型与C类型的映射(F)如下：

|  |  |
| --- | --- |
| Pascal基本类型 | C类型 |
| Integer | Int |
| Real | Float |
| char | char |
| Boolean | Bool |

对应的复合数据类型处理如下：

|  |  |
| --- | --- |
| Pascal复合类型 | C类型 |
| Array[a..b] of Type | F(Type) [b-a+1] |

对应的其他映射如下：

|  |  |
| --- | --- |
| Pascal类型 | C类型 |
| read | scanf |
| writeln | printf |