# 语法分析详细设计

## 语法分析概述

在语法分析阶段，将词法分析阶段得到的记号序列作为输入，结合Pascal-S语言的语法产生式，利用yacc工具，将记号序列规约成有意义的语句，并用一棵完整的语法分析树来表示。

yacc语法分析器采用LALR(1)语法分析方法，其输入是上下文无关文法，输出是基于分析表驱动的编译器。

yacc处理冲突的规则如下：

当发生移进/归约冲突时，执行移进动作，即移进先于归约；

当发生归约/归约冲突时，用yacc源程序中第一个出现的产生式进行归约。

## 接口分析

### 1.与词法分析的接口

在yacc中规定，语法分析与词法分析之间通过一个内置的结构类型YYSTYPE连接；默认的YYSTYPE中包含的信息十分有限，于是在我们的程序中，通过自定义一个记号的类Token，并将YYSTYPE宏定义为自定义类Token，就可以使Token成为词法分析与语法分析沟通的桥梁。

*//节点定义*

class **Token**

{

public:

**string** type; *//符号类型*

**string** value; *//符号具体的值*

    int lineNo; *//所在行号*

**vector**<**Token** \*> children; *//语法树子节点 对应产生式右部*

**Token**() {}

**Token**(**string** val) : **type**(val) {}

**Token**(**string** tp, **string** val, int ln) : **type**(tp), **value**(val), **lineNo**(ln) {}

};

*//将YYSTYPE宏定义为Token*

#define **YYSTYPE** **Token** \*

### 2.与语义分析的接口

完成语法分析后，语法分析器将返回一颗完整的语法分析树，用树的根结点ParseTreeHead来表示；

**Token**\* ParseTreeHead;

## 各部分功能描述

实现语法分析器的文件yacc.y结构如图：

%{

Declarations

%}

Definitions

%%

Translation rules

%%

User subroutines

其中，Declarations部分是C 代码的声明与实现，yacc会将这部分代码直接拷贝到生成的C程序yacc.tab.c中；

Definitions部分是记号声明，可以规定文法的开始符号以及终结符类型等；以便于告诉yacc在语法分析器中记号的名称。通常来说，记号总是使用大写字母，对应产生式的终结符。任何没有声明为记号的语法符号对应产生式的非终结符，所以必须出现在至少一条规则的左边。

Translation rules 部分为通过BNF定义的产生式合集，需要注意的是，此处的产生式使用冒号代替右箭头；每条产生式规则下可以定义C语言的动作代码，在该条规则被规约时执行，用花括号括起；

User subroutines 是用户自定义程序段，可执行额外的动作代码 。

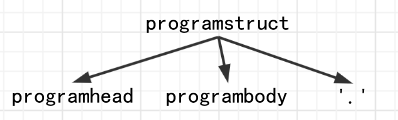
## 数据结构说明

### 1.语法分析树相关数据结构

每一个完整的Pascal-S源程序都对应一棵完整的语法分析树，每个产生式对应一类子树。树的结点用Token表示；

例如对于产生式 programstruct -> program\_head program\_body '.'

这个语句产生的子树，是整棵语法分析树的最上层的子树，其中，产生式左部programstruct是子树的根，右部的各个符号均为左部的孩子结点。



*//节点定义*

class **Token**

{

public:

**string** type; *//符号类型*

**string** value; *//符号具体的值*

    int lineNo; *//所在行号*

**vector**<**Token** \*> children; *//语法树子节点 对应产生式右部*

**Token**() {}

**Token**(**string** val) : **type**(val) {}

**Token**(**string** tp, **string** val, int ln) : **type**(tp), **value**(val), **lineNo**(ln) {}

};

*//语法分析树的头节点*

**Token**\* ParseTreeHead;

### 2.错误处理相关数据结构

yacc中有自带的错误处理函数yyerror()，基本可以满足错误信息的输出要求；

但为了更加详细地输出和存储错误信息，特定义了一个字符串数组syntaxErrorInformation，用以存储错误信息；同时对yyerror()函数进行重载，以处理不同情形下的语法错误。

bool haveSemanticError=false; *//是否有语法错误*

**vector**<**string**> syntaxErrorInformation; *//存放语法错误信息*

*//语法错误处理函数 参数为错误描述信息 以及错误所在行号*

void **yyerror**( const char \*s, **YYLTYPE** \*loc);

void **yyerror**( const char \*s, int line, int col);

void **yyerror**( const char \*s, int startLine, int startCol, int endLine, int endCol);

### 3.文法的终结符以及起始符

*//%start规定文法开始符号*

%start programstruct

*//终结符的类型*

  enum **yytokentype**

  {

    PROGRAM = 258,

    CONST = 259,

    VAR = 260,

    ARRAY = 261,

    OF = 262,

    PROCEDURE = 263,

    FUNCTION = 264,

    \_BEGIN = 265,

    END = 266,

    IF = 267,

    THEN = 268,

    CASE = 269,

    FOR = 270,

    TO = 271,

    DOWNTO = 272,

    DO = 273,

    ELSE = 274,

    REPEAT = 275,

    UNTIL = 276,

    WHILE = 277,

    IDENTIFIER = 278,

    UINUM = 279,

    UFNUM = 280,

    CHAR = 281,

    LETTER = 282,

    INTEGER = 283,

    TYPE = 284,

    ASSIGNOP = 285,

    RELOP = 286,

    EQUAL = 287,

    ADDOP = 288,

    ADD = 289,

    MINUS = 290,

    MULOP = 291,

    NOT = 292,

    RECORD = 293,

    REAL = 294,

    BOOLEAN = 295,

    BOOL\_CONSTANT = 296,

    MUL = 297,

    UMINUS = 298

  };

## 算法描述

### 1.对单词进行语法分析

在主程序中，通过调用yacc的内置函数yyparse()，来进行语法分析；

在yyparse()函数中，每遇到一个符号，都会调用lex的内置函数yylex()对符号进行词法分析，生成该符号对应的Token，并返回记号的类型；

得到记号的类型后，yacc语法分析器将查找能够匹配当前记号的规则，根据LALR(1)分析方法执行移进或规约动作；每当规约一条规则，就自动执行该条规则关联的用户代码。

其中，yacc.y文件中出现的产生式均参照书本中提供的Pascal-S语言产生式合集。

### 2.构建语法分析树

yacc语法分析器没规约一条规则时，会自动执行该条规则关联的用户代码，在本程序中，即生成语法分析树。

每规约到一个产生式，按如下步骤构建语法分析树：

① 为产生式的左部构造一个结点

② 将产生式右部的记号分别加入到左边结点的子结点集合中;

以产生式compound\_statement -> \_BEGIN statement\_list END为例:

其中 $$ 指代产生式左部；$1指代产生式右部第一个符号，$2指代产生式右部第二个符号，以此类推；

compound\_statement : \_BEGIN statement\_list END {

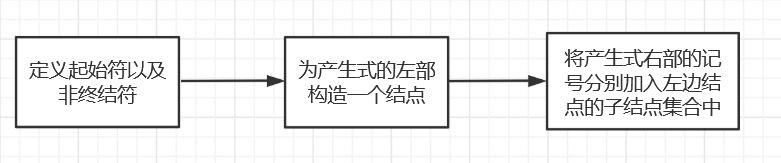
                    $$ = new Token("compound\_statement");

                    $$->children.push\_back($1);

                    $$->children.push\_back($2);

                    $$->children.push\_back($3);

                };



### 3.错误处理

本程序中，采用了出错产生式的方法来进行错误处理以及错误恢复，即通过增加产生错误结构的产生式，扩充源语言的文法，然后根据扩充后的文法构造分析程序。

如果分析程序在分析过程中使用到了这些扩充的产生式，表明输入记号序列中出现错误，且已经被识别，此时应当记录错误信息；

语法分析中涉及到的错误类型详见语法分析-测试文档。

以if语句中缺少关键字then为例：

产生式1为if语句对应的正确无误的产生式，而产生式2对应if语句中缺少”then”关键字的错误情况；当识别的Pasca-S代码中出现这种错误时，语法分析程序会自动将该语句按照下面的产生式2来规约，不妨碍程序继续执行的同时，会输出报错信息，提示该行缺少then关键字。

statement : IF expression THEN statement else\_part {

                    $$ = new Token("statement");

                    $$->children.push\_back($1);

                    $$->children.push\_back($2);

                    $$->children.push\_back($3);

                    $$->children.push\_back($4);

                    $$->children.push\_back($5);

                } | IF expression error statement else\_part {  //缺少then关键字

                    $$ = new Token("statement");

                     yyerror("missing keyword \"then\"", @2.last\_line, @2.last\_column+1);

                }