编译原理大作业

Deadline：

1、词法分析器（包括代码及要求的文档）2021 年 10 月 24 日 11：59PM（GMT+8）
2、语法分析器（包括代码及要求的文档）2021 年 11 月 7 日 11：59PM (GMT+8)

一、目标

本次大作业为编写一个编译器前端（包括词法分析器和语法分析器），（1）使用自动机理论编写词法分析器，（2）自上而下或者自下而上的语法分析方法编写语法分析器。

1、手工编写 C--语言的词法分析器，理解词法分析器的工作原理，熟练掌握高级语言的单词符号的正规式表示，词法分析器的工作流程并编写源代码，识别出单词的二元属性，填写符号表。

2、手工编写 C--语言的语法分析器，理解自上而下/自下而上的语法分析算法的工作原理；理解词法分析与语法分析之间的关系。语法分析器的输入为 C--语言源代码，输出为按扫描顺序进行推导或归约的正确/错误的判别结果，以及按照最左推导顺序/规范规约顺序生成语法树所用的产生式序列。

二、软件需求

1、词法分析器

（1）完成 C--语言的词法分析器，词法分析器的输入为 C--语言源代码，输出识别出单词的二元属性，填写符号表。单词符号的类型包括关键字，标识符，界符，运算符，整数，浮点数，单字符，字符串。每种单词符号的具体要求如下：

关键字包括 while for continue break if else float int char void return

运算符（OP）包括 + - * / % = > < == <= >= != ++ -- && || += -= *= /= %=

界符（SE）包括 ( ) { } ; , [ ]

标识符（IDN）为字母、数字和下划线（_）组成的不以数字开头的串

整数（INT）、浮点数（FLOAT）的定义与 C 语言相同

单字符（CHAR）定义与 C 语言相同为单引号字符，例如： 'c'

字符串（STR）定义与 C 语言相同为双引号字符串，例如：" string"

（2）实现语言：C/C++/Java/Python；

（3）操作目的：生成符号表；将待分析代码转化为语法分析器可接受的序列。

2、语法分析器

（1）完成 C--语言的语法分析器，语法分析器的输入为 C--语言代码的 token 序列，输出用最左推导或规范规约产生语法树所用的产生式序列。C--语言文法须包含以下操作（具体语法参见附件）：

①声明语句（变量声明）； ②表达式及赋值语句分支语句：if else； ③循环语句：for while；④算数表达式四则运算；⑤变量自增、自减运算

例如：采用 LL（1）语法分析方法构建语法分析器需要完成以下三部分内容：

① FIRST 集合、FOLLOW 集合；② LL(1)预测分析表；③对待编译代码解析得到语法树或产生式的规约序列。

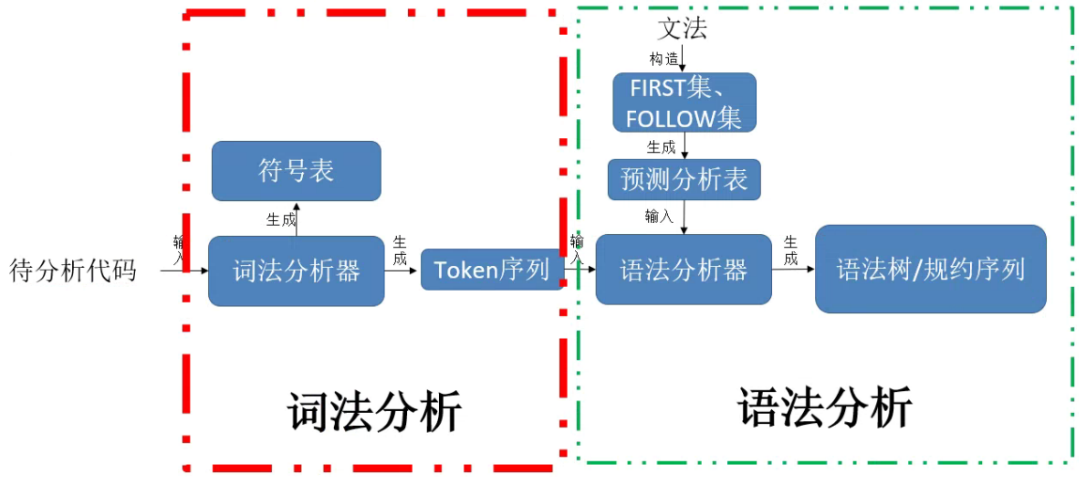

图 1 LL（1）词法语法分析器流程图

（2）实现语言：C/C++/Java/Python；

三、输出示例

1、词法分析输出示例

```
void main(){
   int i = 2;
   while(i <= 4){
       i = i+1;
}
}
```

代码 1 待测 C--代码

（1）输出词素序列：

```
[void, main, (, ), {, int, i, =, 2, ;, while, (, i, <=, 4, ), {, i, =, i, +, 1, ;, }, }]
```

代码 2 待测 C--代码词素序列

（2）输出 TOKEN 序列：

```
void  <VOID,_>
main  <IDN,main>
(  <SE,_>
)  <SE,_>
{  <SE,_>
int  <INT,_>
i  <IDN,i>

=  <OP,_>
2  <CONST,2>
;  <SE,_>
while  <WHILE,_>
(  <SE,_>
i  <IDN,i>

<  <OP,_>
4  <CONST,4>
)  <SE,_>
{  <SE,_>
i  <IDN,i>

=  <OP,_>
i  <IDN,i>

+  <OP,_>
1  <CONST,1>
;  <SE,_>
}  <SE,_>
}  <SE,_>
```

图 2 TOKEN 序列示例

注：符号表中关键字填写是全部大写，例如： void <**VOID**,_>

2、语法分析输出示例（以预测分析为例）

（1）输出 FIRST 集和 FOLLOW 集：

```
FIRST:
const: : * / % + - ++ -- > >= < <= == != = += -= *= /= %= ; && and || or ) ,
type: IDN ( FLOAT CHAR STR INT
lop: ( IDN FLOAT CHAR STR INT
bool_expression: )
```

代码 3 FIRST 集、FOLLOW 集部分输出示例

（2）输出预测分析表：

预测分析表	!=	%	%=	&&	(	)
S						
fnc						
type						
args						
arg						
func_body						
block						
vars						
stmts						
stmt						
assign_stmt					assign_stmt-> expression ;	
jump_stmt						
branch_stmt						
result						
logical_expression						
bool_expression				bool_expression -> lop expression bool_expression		
lop				lop -> &&		

表格 1 预测分析表输出示例

注：预测分析表可以存储为 csv 格式、excel 格式或者在命令行中打印表格。

（3）输出规约序列：

（序号）栈顶符号-面临输入符号（执行动作）【十个空格】栈顶到栈底符号串【十个空格】选用规则

```
(1) S-void          S          S -> func
(2) func-void          func          func -> type IDN ( args ) func_body
(3) type-void          type IDN ( args ) func_body          type -> void
(4) void-void (跳过)            void IDN ( args ) func_body
(5) IDN-IDN (跳过)           IDN ( args ) func_body
(6) (-( (跳过)            ( args ) func_body
(7) args-)          args ) func_body          args -> $
(8) )-) (跳过)           ) func_body
(9) func_body-{          func_body          func_body -> block
(10) block-{          block          block -> { stmts }
(11) {-{跳过          { stmts }
(12) stmts-int          stmts }          stmts -> stmt stmts
```

代码 4 规约序列部分输出示例

注：分析栈左端为栈顶，输入串过长没有在输出实例中进行展示，输入串即词法分析器生成的 Token 序列。

四、提交要求

1、 源代码

包括词法分析器、语法分析器

2、 开发报告

对项目的开发过程进行详细的描述，包括（1）词法分析器算法描述，输出格式说明，源程序编译步骤；(2) 语法分析器的算法描述，创建的分析表（预测分析表、LR 分析表等），输出格式说明，源程序编译步骤；

3、 测试报告

在给出的测试用例上分析后词法分析器以及语法分析器的输出截图(注意需要按照要求格式输出)。

五、展示验收

每组 10 分钟时间，通过 PPT+演示的形式展示本组完成的大作业。具体时间地点待定。

附录

注：文法中出现的$代表 ε，文法中出现的 IDN（标识符)、INT（整数)、FLOAT（浮点数)、CHAR（单字符、STR（字符串）参照词法分析器中定义，所有的界符、运算符、关键字均以原本形式存在，例如：
iteration_stmt -> **while** (logical_expression) block
operation -> **++**

LL（1）形式的 C--文法：
S -> func
func -> type IDN (args) func_body
type -> int
type -> char
type -> float
type -> void
type -> $
args -> type IDN arg
args -> $
arg -> , type IDN arg
arg -> $
func_body -> ;
func_body -> block
block -> { define_stmts stmts }
define_stmts -> type IDN define_stmt define_stmts
define_stmts -> $
define_stmt -> init vars ;
init -> = const
init -> $
vars -> , IDN init vars
vars -> $
stmts -> stmt stmts
stmts -> $
stmt -> assign_stmt

stmt -> jump_stmt
stmt -> iteration_stmt
stmt -> branch_stmt
assign_stmt -> expression ;
jump_stmt -> continue ;
jump_stmt -> break ;
jump_stmt -> return isnull_expr ;
iteration_stmt -> while (logical_expression) block_stmt
iteration_stmt -> for (isnull_expr ; isnull_expr ; isnull_expr) block_stmt
branch_stmt -> if (logical_expression) block_stmt result
result -> else block_stmt
result -> $
logical_expression -> ! expression bool_expression
logical_expression -> expression bool_expression
bool_expression -> lop expression bool_expression
bool_expression -> $
lop -> &&
lop -> ||
block_stmt -> { stmts }
isnull_expr -> expression
isnull_expr -> $
expression -> value operation
operation -> compare_op value
operation -> equal_op value
operation -> ++
operation -> --
operation -> $
compare_op -> >
compare_op -> >=
compare_op -> <
compare_op -> <=
compare_op -> ==
compare_op -> !=
equal_op -> =
equal_op -> +=
equal_op -> -=
equal_op -> *=
equal_op -> /=
equal_op -> %=
value -> item value'
value' -> + item value'
value' -> - item value'
value' -> $
item -> factor item'

item' -> * factor item'
item' -> / factor item'
item' -> % factor item'
item' -> $
factor -> (value)
factor -> IDN call_func
factor -> const
call_func -> (es)
call_func -> $
es -> isnull_expr isnull_es
isnull_es -> , isnull_expr isnull_es
isnull_es -> $
const -> num_const
const -> FLOAT
const -> CHAR
const -> STR
num_const -> INT