东南大学

编译原理课程设计

设计报告

成员： 71121107 张益凡

71121117 马骁宇

东南大学软件学院

二 0 24 年 5 月

1 编译对象与编译功能

1.1 编译对象

以c99.l文件为标准，可以实现C语言全集（在1999年推出的c99是C编程语言标准的过去版本）作为编译对象。将c99.l文件的格式进行了调整，如下：

{辅助定义部分}

%%

{识别规则部分}

%%

{用户子程序部分}

其中，辅助定义部分改为先以%{开始,%}结束声明用户自定义变量、常量和头文件，再声明正规表达式定义。同时，在词法定义部分，重写了一些不够严谨的正则表达式（如将L({L}|{D})*重写为{L}({L}|{D})*），其余部分没有删改。

1.2 编译功能

项目整体功能包括：

① c99.l输入文件的解析，主要在 main()函数中完成；

② 正则表达式的规范化， 主要在 strRE ()函数中完成；

③ 由正则表达式构造NFA，合并多个NFA，主要在 makeNFA()函数中完成；

④ 由NFA构造 DFA、DFA 最小化，主要在 NFA2DFA() 、minimizeDFA()函数中完成；

⑤ 词法分析器C++代码的生成， 主要在 generateCode ()函数中完成。

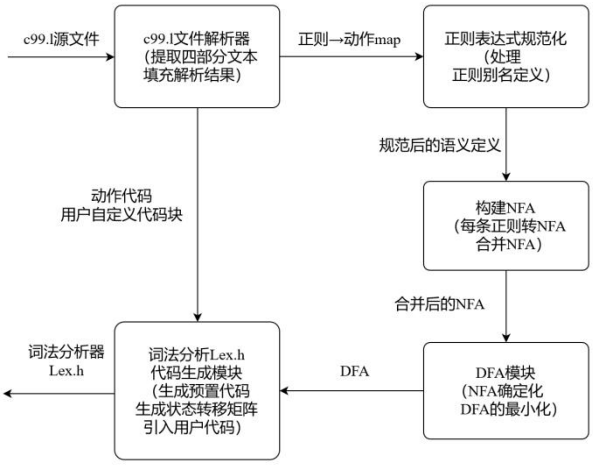
2 主要特色

1. 实现了最长匹配原则
2. 支持正则表达式四大元符号：?+*|（0 或 1 次，1 次或以上，0 次或以上，或）
3. 支持正则表达式范围与范围补（ASCII 为全集）：[A-Za-z0-9_] 、[^"]
4. 支持正则别名定义
5. 可以使用范围型转义字符\d([0-9])和\s([\t\r\n])
6. 运用C++的 regex 库以提供用于表示正则表达式和匹配结果的基本类型

3 概要设计与详细设计

3.1 概要设计

SeuLex分为c99.l文件解析器、正则表达式规范化模块、NFA 模块、 DFA 模块、词法分析Lex.cpp代码生成模块。模块之间的关系和职责可以用如下的工作流图来表示：



3.2 详细设计

3.2.1 C99.l文件解析的详细设计

Lex 文件解析器需要解析的c99.l文件主要分为四个部分：一，用户自定义变量、常量和头文件（以%{开头，以%}结尾）；二，正规表达式定义（在第一个%%之前）；三，保留字、正则表达式与相应的动作（起始于第一个%%，终止于第二个%%）；四，用户自定义子例程段（第二个%%之后）。它们可以通过对文件内容逐行扫描，对前面提到的定界符进行判别得到。注意：解析的同时要判断c99.l文件的结构是否符合要求，如果结构不正确需要报错处理。

对于第一和第四部分，处理方式简单：直接读入并存入Lex.h即可；对于第二部分，逐行读取每个正规表达式的表示符号与定义，并用结构体变量存储它们，并建立起这两者的对应关系；至于第三部分，因为每行的词法规则构成是保留字或者正则表达式 + 动作，中间用若干数量不一的制表符分隔开，所以我们先按行读取，然后按制表符将保留字或者正则表达式与动作分开，分别存入对应的变量中，此外，我们还要将第三部分使用的表示符号替换为正规式（即使用第二部分的对应关系进行替换）。

3.2.2 正则表达式规范化的详细设计，即strRE()函数

首先strRE函数出现在识别词法规则的部分，主要的作用是识别正规表达式，将正规表达式的表示符号替换为正规式。例如，在c99.l中出现的

{L}({L}|{D})*　　{ count(); return(check_type()); }

其中{L}会被替换成[a-zA-Z]、{D}会被替换成[0-9]。而c99.l文件中，语句的格式较为统一，总结可以分为“”纯关键字”” 型、“”关键字”与{RE标识符}与RE连接符混合” 型、“”符号”” 型。所以区分操作较为方便，关键在于找“{}”，也就是找到所有的RE标识符。考虑到可能有多个“{}”，函数内部的操作需要不断循环、处理遇到的所有的“{}”。这里首先就出现了第一种return情景，即函数处理语句时未遇到“{”，那么函数就可以直接返回输入的语句参数，因为原语句并不包含RE标识符；而在处理遇到“{”的情况时，又有两个分支，即“{”有对应“}”和无对应“}”的情况，接下来先处理有对应的情况，从“{”的位置出发，开始往后遍历，如果找到了下一个“}”，就跳出遍历，而如果没找到下一个“}”，那么就直到遍历完整个语句，再跳出遍历。只要结束遍历之后，j==str.size()我们就直接返回传入函数的语句参数，因为若“{}”不成对出现，说明非RE的表示符号，而是单括号本身。再回到“{”对应的情况，此时“{}”中的就是RE标识符，通过和一开始建立的“RE标识符-RE”map里去搜索，就可以将RE标识符一一转换，然后再和“{”左侧的部分与“}”右侧的部分重新拼接，并且函数return这拼接后的语句，就成功将正则表达式规范化了。

3.2.3 构建NFA的详细设计，即makeNFA()函数

NFA是允许两个节点之间有多条重边的，所以在一开始对边的结构体变量定义中，使用到看指针数组类型，具体如下

```
struct Edge {
    string symbol = "";
    Edge *next = nullptr;
    int toNode = -1;
};
```

可以看到每一个Edge的定义中使用到自身指针类型Edge*作为成员变量，表示了两点之间可以有多个重边。

makeNFA()函数部分主要靠内部不断循环使用边处理函数processEdge(int node, Edge *edge)来实现，函数部分如下：

```
void makeNFA() {
    for (Edge *i = NFA[0].edge; i != nullptr; i = i->next)
        processEdge(0, i);
}
```

通过遍历Edge数组里的所有的边，并且对于每一个边（即边）进行处理，实现NFA的构造。

接下来主要对于processEdge函数进行分析。首先获得输入参数Edge上的symbol参数。第一步判断symbol的长度，如果symbol长度<=1，那就说明边上的字符为空字符或单字符，不存在复合字符的情况，不需要对与边进行处理。实际情况如下：

```
string str = edge->symbol;
if (str.size() <= 1)
    return;
```

然后处理其他情况，第一种情况，检测到str（边上的字符，后同）第一个字符为“[”符号，此时可能就出现了字符为[1-9]{[0-9]}*{((u|U)|(u|U)?(l|L|ll|LL)|(l|L|ll|LL)(u|U))}这样的情况当然也有另外的情况如“[1-9]”“[a-z A-Z]”，经有一段表达式...首先来处理最简单的情况，也就是仅有一段正则表达式的情况。

首先，仍然是处理“[”的另一半问题，如果没找“]”，就说明是普通“[”，直接结束总的第一种情况，进入到另一种情况下处理。

接下来，如果找到了另一半，那就是正常的正则表达式，对于正则表达式的处理，这里采用取巧的方法，由于1-9加上大小写的所有字母符号，基本已经囊括了所有的键盘字符了，就采用ASCII码的方式，从32-126遍历全部的键盘输入符号，然后每一个输入符号和这里的正则表达式进行正则匹配，如果匹配上了，那么就利用这个输入符号在processEdge输入参数Node的原Node和下一个Node两点之间新增加一条边，这样就将单边上的单个正则表达式完全解析为多条边的一个DFA。

```
bool b = true;
for (int c = 32; c < 128; c++) {
    string st = "";
    st += (char) c;
    if (regex_match( s: st, re: regex( s: str))) {
        if (b) {
            edge->symbol = st;
            b = false;
        } else
            newEdge( str: st, edge->toNode, fromNode: node, &: NFA);
    }
}
```

for循环从32到128，ASCII码转字符，然后进行regex匹配，匹配成功加一条边，在进入下一个循环。最简单的情况结束。

然后，基于最简单的情况，考虑一个更为复杂的情景，“[]”后面有剩余字符，以及单个“[”的处理。（补充，若“[]”后有剩余字符，都是以“‘[]”连接符“剩余字符””形式出现的）

针对“[]”后面有剩余字符，首先提取原有的“[]”内的字符，作为pro以及提取连接符后面的部分作为rest，连接符则有“*”“+”“?”“|”四种。

接下来对于四种连接符分别进行解释如何处理，当遇到“*”时，首先，新增一个节点用于插在原有边连接的两点之间，同时将原有边的起点与新增节点用一个新的边连接（原有边起点→新增节点），并且将边的symbol置为ε，将原有边的终点与新增节点用一个新的边连接（新增节点→原有边终点），并且将边的symbol置为rest，而新增节点与其自身再增加一个边（即指向自己的边），并且将symbol置为pro。

当遇到“+”时，与“*”的情况大致一样，唯一区别在于原edge的symbol不在置为“ε”，而是pro。

当遇到“?”时，大致流程仍与“*”一致，但是不同点在于，newNode没有指向自己的边，new1成为一条空边，由node指向newNode。

当遇到“|”时，大致流程相对来说较为简单，只需增加一条边new1，作为原edge的平行边，指向仍然是node指向nextNode，new1的symbol改为pro，而原edge的symbol改为rest。

在这里，我们额外补充遇到单个“[”的情况，这时需要将“[”与后续的部分拆分开来，大致流程为，增加一个新节点newNode，同时增加一条新边new1（newNode→nextNode），其symbol为rest；原edge修改为node→newNode，并且symbol为pro。

以上，就处理完了“[”之后可能遇到的所有情况。接下来，处理其他情况，比如说第二种情况，遇到了“(”时的处理，类似于“[”,首先判断是否成对出现，当“()”成对出现时，并且后续无其他字符时，将“(” “)”中间夹的部分提取出来，并且去重新做pro。而如果遇到了单“(”，“()”后包含连接符的情况，处理方式与“[”一致。

第三种情况，遇到了“\””的处理，本质上就是对于“””的处理，首先仍然是判断是否成对出现“””””，如果成对出现就说明，出现了关键字，例如“”else””，如果同时后续无连接符，那么处理方式如下，检测关键字的第一个字符（例如else的e），并且将edge的symbol改为该字符，同时检测关键字有无后续字符（例如else就是l，然后时s，然后是e，最后结束），如果有一个后续字符，就加一个newNode和一个newE，原edge改为（原edge起点→newNode），newE的symbol置为后续字符，指向为newNode→nextNode。

而如果遇到了“/”单符号或者后续有连接符的情况，处理同“[”。

第四种情况，遇到了“\\”，本质是遇到了转义符“\”的情况，c99.l中可能遇到转义符的情况如下，“\\” “\”” “\’” “\t” “\v” “\n” “\f” “\0” “\?” “\r” “\a” “\b”,由于转义符之后总是有别的字符跟随，所以就将转义符部分提取出来作为pro（比如遇到“\t”,就将“\t”作为pro），而转义符后面的部分用作rest，然后仍然依照处理“[”后有连接符或者单边“[”的处理方式，处理pro与rest。

第五种情况，即不属于以上四种任何一种情况时，处理方法如下，将第一个字符单独取出来作为pro，剩下部分作为rest。然后依照处理“[”后有连接符或者单边“[”的处理方式，处理pro与rest。本质上相当于将字符串拆分并且逐个加边或者作其他处理。

以上，完成对于makeNFA()函数的分析。

3.2.4 构建DFA的详细设计，即NFA2DFA() 、minimizeDFA()函数

确定有限状态自动机类构造时，接收一个 NFA ，进行确定化和可选的最小化。

NFA2DFA()函数中的详细设计

使用子集法(ε-闭包)进行 NFA的确定化由NFA M={S, Σ, f, S_0, Z}构造一个等价的DFA M’={Q, Σ, δ, I_0, F}的算法如下：

① 取I_0 = ε-Closure(S_0)，加入状态集Q，这里定义状态集合 I 的 ε-Closure(I)为 I 中的任何状态s经任意条ε边能到达的状态的集合；

② 接着定义状态集合 I 的 a 弧转换f(I,a)，表示可从 I 中的某一状态经过一条 a 弧到达的状态的全体。则当状态集Q中有状态$I_i = \{s_1, s_2, \ldots, s_j\}$，$s_k \in S, 0 \le k \le j$，而且 M 机中有$f(\{s_1, s_2, \ldots, s_j\}, a) = \cup_{k=0}^{k} f(s_k, a) = \{s_1, s_2, \ldots, s_t\} = I_t$，便对于每个输入符号 a，计算 U= ε-Closure(f(I_i,a))即δ(I_i,a) (DFA的状态转移)= U。并且，若U不在状态集Q中，则将 U 加入Q中；

③ 重复第②步，直至Q中不再有新的状态加入为止；

④ 取F={I|$I \in Q$，且$I \cap Z \neq \emptyset$}。

在上面这个NFA确定化的算法中，我们借助Node struct结构体中定义的set<int> nodes保存NFA状态编号，将一个DFA状态对应于NFA状态子集，从而在构建DFA节点时通过set数据结构的=相等判断，知晓是否新建过对应该ε-闭包的DFA节点，若无则通过newDFANode()函数新建节点。

同时，对于每一个新建的DFA节点（即新的ε-闭包），都需通过makeDFATable()函数构建对应的跳转表，此处跳转表采用map数据结构map<string, set<int>>symbols记录所有非终结符与对应的NFA节点编号集合，便可完成跳转表的可视化。

另外，我们知道，子集构造法的每个 DFA 状态都对应于若干个 NFA 状态，那么在一个 DFA 接收态中有没有可能包括多种动作代码不同的 NFA 接收态？我们发现答案是肯定的。这种动作代码的冲突问题需要借助动作的优先级，即借助 Lex 的正则-动作部 分“先定义的优先高”的规则进行解决。

minimizeDFA()函数中的详细设计

DFA的最小化算法[又称划分法(Partition)]的原则是将DFA中的状态划分成不相交的子集，在每个子集内部其状态等价，而在不同子集间状态均不等价（即可划分的）。最后，从每个子集中任选一状态作为代表，消去其他的等价状态。将那些原来射入其他等价状态的弧改射入相应的代表状态。

按照这个原则构造算法如下：

① 首先将状态集F划分为终态集和非终态集，即$\prod_0 = \{I_0^1, I_0^2\}$，设$I_0^1$为非终态集，$I_0^2$为终态集；

② 假定k次划分后，已含有m个子集，记作$\prod_k = \{I_k^1, I_k^2, \ldots, I_k^m\}$。这些子集到现在为止都是可区分的，然后继续考察这些子集是否还可以划分。设任取一子集$I_k^i = \{s_1, s_2, \ldots, s_t\}$，若存在一个输入字符a，使得$f(I_k^i, a)$不全包含在现行$\prod_k$的某子集中，这说明$\prod_k$中存在不等价的状态，他应该还可一分为二，做一次划分；

③ 重复步骤②，直到所含的子集数不再增加为止。到此$\prod$中的每个子集都是不可再分的了；

④ 对每个子集任取一个状态为代表，若该子集包含原有的初态，则此代表状态便为最小化后DFA的初态；若该子集包含原有的终态，则此代表状态便为最小化后DFA的终态。

以上过程可以在有限步内结束，因为状态数是有限的。

需要注意的是，对于终态，不能把它们放在同一个划分，而要在一开始就全部构建独立的划分（即一个新状态不能包含多个终态），这是因为终态上绑定着“动作代码”属性，因此它们之间不是真正等价的。

4 测试用例与结果分析

测试用例

测试代码：

```
int test(int x)
{
    if (x==0) {
          return 1;
    }
    else _x=test(x-2)+3;
      bool b=true;
    return x;
}
```

结果：

```
yytext: int TokenName:INT
yytext: test TokenName:IDENTIFIER
yytext: ( TokenName:(
yytext: int TokenName:INT
yytext: x TokenName:IDENTIFIER
yytext: ) TokenName:)
yytext: { TokenName:{
yytext: if TokenName:IF
yytext: ( TokenName:(
yytext: x TokenName:IDENTIFIER
yytext: == TokenName:EQ_OP
yytext: 0 TokenName:CONSTANT
yytext: ) TokenName:)
yytext: { TokenName:{
yytext: return TokenName:RETURN
yytext: 1 TokenName:CONSTANT
yytext: ; TokenName:;
yytext: } TokenName:}
yytext: else TokenName:ELSE
yytext: _x TokenName:IDENTIFIER
yytext: = TokenName:=
yytext: test TokenName:IDENTIFIER
yytext: ( TokenName:(
yytext: x TokenName:IDENTIFIER
yytext: - TokenName:-
yytext: 2 TokenName:CONSTANT
yytext: ) TokenName:)
yytext: + TokenName:+
yytext: 3 TokenName:CONSTANT
yytext: ; TokenName:;
yytext: bool TokenName:BOOL
yytext: b TokenName:IDENTIFIER
yytext: = TokenName:=
yytext: true TokenName:IDENTIFIER
yytext: ; TokenName:;
yytext: return TokenName:RETURN
yytext: x TokenName:IDENTIFIER
yytext: ; TokenName:;
yytext: } TokenName:}
```

Yacc部分

1 编译对象与编译功能

1.1 编译对象

我们以c99.y文件为标准，将文件的格式修改呈现如下：

- **说明部分**

%{

头文件表、宏定义、数据类型定义、全局变量定义

%}

语法开始符号定义、语义值类型定义、终结符号定义、运算符优先级及结合性定义

%%

- **语法规则部分**

%%

- **用户子程序部分 (包含主程序、错误信息报告程序、词法分析程序、其它程序段)**

原c99.y说明部分内没有%{...%}内容，为了便于分析，我们补充了这一部分，只不过内容为空而已，同时以“%start”标记的文法开始符号是translation_unit，为了省去在语法规则部分检索文法开始符号的步骤，我们与原版Yacc 的行为保持一致，如果定义段中没有“%start”的说明，Yacc自动将语法规则部分中第一条语法规则左部的非终结符作为语法开始符，从而我们删去说明部分“%start”的定义说明，将语法规则中

translation_unit: external_declaration

| translation_unit external_declaration;

这一条语法规则移至第一条位置，以此表示translation_unit为文法开始符号。

另外，原c99.y 说明部分没有左右结合关系声明，所以我们经网上查阅，整理了C运算符优先级与左右结合关系表，添加到其中，以便解决结合性和优先级冲突。最后，直接将c99.y 作为编译对象，即使程序经过一定优化，也需耗时很久，但是最终测试只需要用输出Yacc.cpp即可。

具体请参考 .../SEULex/c99_1.y。

1.2 编译功能

项目整体功能包括：

① c99.y输入文件的解析，主要在 analysisInput()函数中完成，其中调用了ExplanationProcess()函数完成说明部分%{...%}解析，SymbolProcess()函数完成说明部分%开头标记的终结符与非终结符的解析，RuleProcess()函数解析语法规则部分，ProducerProcess()函数处理规则部分的产生式，AdditionalProgramProcess()函数处理附加程序段；

② 灵活使用set，map，vector等数据结构，存储符号、产生式等；

③ 构造LR(1)项目集，包含LR1Item类设计，closure()函数求闭包、findPredictiveSym()函数计算预测符、getFirstSet()函数计算First集；

④ 构造LR(1)项目集规范族的详细设计，主要在 ProcessItemSet() 函数中完成；

⑤ 语法分析器C++代码的生成， 主要在GenerateCode()函数中完成。

项目还有一些特色功能， 如LR(1)分析过程的可视化（包含所有产生式、终结符、非终结符、LR(1)项目集、分析表打印等）、 语法树的可视化等。

2 主要特色

① .y 文件头部声明支持%token 、%type、%left 、%right

② 动作代码部分可以使用$$ 、$1 、$2 、$3 等获取栈内元素

③ 广泛使用接口与面向对象编程，代码各部分独立，从而易于调用

④ 代码具备一定的错误处理能力

⑤ 在进行 LR(1)分析时，对符号（终结符、非终结符）、产生式、 LR(1)项目集等全部进行编号，利用编号索引，节省空间，效率较高

⑥ 对 LR(1)项目集进行了简化算法

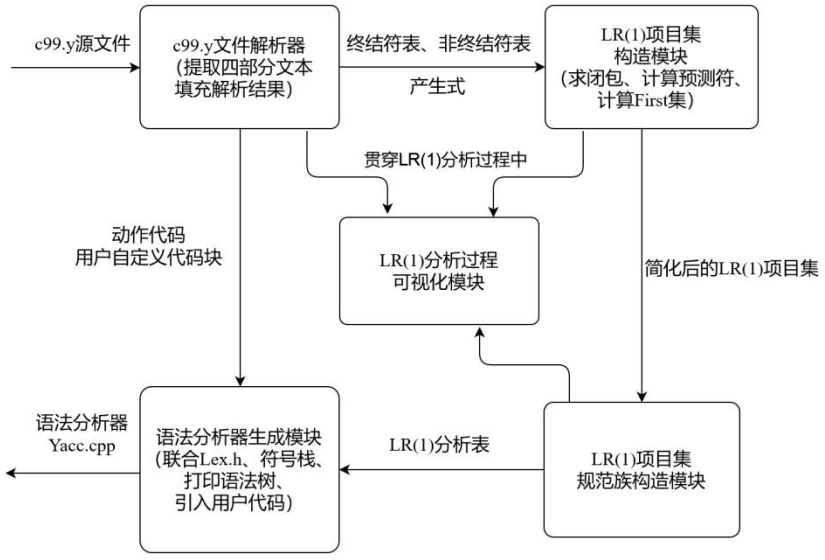
⑦ 对构造ACTION-GOTO分析表时，会处理可能出现的归约-归约冲突与移进-归约冲突，以保证语法分析不被打断

⑧ 提供了所有产生式可视化、终结符表与非终结符的打印功能，语法树可视化功能

3 概要设计与详细设计

3.1 概要设计

SeuYacc 分为 .y文件解析器、LR(1)项目集构造模块、LR(1)项目集规范族构造模块、语法分析器生成模块、LR(1)分析过程可视化模块。 模块之间的关系和职责可以用如下的工作流图来表示：



3.2 详细设计

3.2.1 C99.y文件解析的详细设计

Yacc输入文件解析器需要解析的c99.y文件主要分为四个部分：一，包含头文件表、宏定义、数据类型定义、全局变量定义的说明部分（以%{开头，以%}结尾）；二，包含语法开始符号定义、语义值类型定义、终结符号定义、运算符优先级及结合性定义的部分（在第一个%%之前）；三，语法规则部分，即产生式-动作部分（起始于第一个%%，终止于第二个%%）。对于第二部分需要先读到第一个空格判断此定义是有关终结符的还是非终结符的亦或语义值类型还是运算符优先级或结合性的，之后再读取剩下整行内容，通过特别字符做分割进行定义解析，其余部分（第一、三部分）可以通过对文件内容逐行扫描。注意：解析的同时要判断c99.y文件的结构是否符合要求，如果结构不正确需要报错处理。

对于第一部分，处理方式简单：直接读入并存入Yacc.cpp即可。

对于第二部分，先读到第一个空格判断此定义是有关哪一内容，“%token”代表定义的是终结符+语义值类型，“%type”代表定义的是非终结符+语义值类型，“%left”与“%right”是有关运算符结合性与优先级的。接下来，再逐行读取剩下整行内容，对于终结符/非终结符，输入文档中用“<>”标记语义值类型声明，如果没有规定语义值类型则置为空，之后逐空格截取字符读取每个终结符/非终结符的定义，给其编号并用相应的map存储符号与编号一一对应。在解析运算符结合性声明部分（%left 、%right）时，遵循同一行的运算符具有相同的优先级， 越后声明的运算符优先级越高的策略确定优先级关系，同时操作符用‘’标记从而能从字符串中截取出来，也作为终结符编号存入终结符map中，根据操作符结合性也相应地存入左右结合表（set数据结构）中。

对于第三部分，因为每行的语法规则构成是产生式 + 动作，每一个语法规则的左部后面会有若干数量不一的候选式，换行由“|”分隔开，我们用Producer结构体存储，先读取规则左部，理应存在非终结符表中，然后逐字符读取右部 的候选式，将对应字符串序号存入产生式右部，每一个候选式识别完毕后，会检测是否存在语法动作（以{}标记），完毕后存入producerVector, producerActionVector, producerPriorVector这三个vector数据结构中对应产生式、产生式动作与产生式优先级的存储。注意：在读取文件所有产生式定义前，要预先加入产生式S'→S，以便产生拓广文法。但由于未实现代码生成，故动作并未实现。

最后对于所有解析出来的产生式再做一次整合，填充入名为producerMap的map<int, vector<int>> 结构中，以产生式的左部符号编号为key，收集对应候选式的所有编号装入vector中， 供下层使用。

使用的变量

```
set<int> leftTable; // 左结合表
set<int> rightTable; // 右结合表
map<string, int> nonterminalTable; // 非终结符表 符号映射至数字编号
map<string, int> terminatorTable; // 终结符表 符号映射至数字编号
map<int, int> PriorityTable; // 优先级表，前一项编码后一项优先级
map<int, string> tokenTypeTable; // 存储语义值类型的表，前一项编码后一项语义值类型

struct Producer//产生式数据结构
{
  int left; // 语法规则的左部符号编号
  vector<int> right; // 右部候选式中符号编号排列
};
vector<Producer> producerVector; // 存储所有的产生式
vector<int> producerPriorVector; // 存储所有的产生式优先级
vector<string> producerActionVector; // 存储所有产生式的动作

// 终结符<数字编号,在分析表中的列号>,第二项为了构造LR(1)分析表
map<int, int> terminSet;
// 非终结符<数字编号,在分析表中的列号>,并同时存储一个到action表头的对应关系
map<int, int> nonterminSet;
// 以产生式的左部为关键字，以对应产生式的编号为内容的vector
map<int, vector<int>> producerMap;
```

3.2.2 构造LR(1)项目集的详细设计，包含LR1Item类设计，closure()函数、findPredictiveSym()函数、getFirstSet()函数

1. closure()函数

(1) 构造closure()函数的算法，我们采用的算法依据的是上学期编译原理课上用的中文教材《编译原理及编译程序构造》中第6.4.1章“构造LR(1)项目集规范族的算法”中的closure部分，具体思路如下：

构造 LR(1)项目集规范族的算法

构造 LR(1)项目集族的算法本质上和构造 LR(0)项目集族的算法是一样的，先介绍两个函数：CLOSURE 和 GO。

(1) 函数 CLOSURE(I)—I 的项目集。

①I 的任何项目都属于 CLOSURE(I)；

②若项目(A→α·Bβ,a)属于 CLOSURE(I)，B→γ是一个产生式，那么对于 FIRST(βa)中每个终结符 b，如果(B→·γ,b)原来不在 CLOSURE(I)中，则把它加进去；

③重复步骤②，直至 CLOSURE(I)不再扩大为止。

因为(A→α·Bβ,a)属于 CLOSURE(I)，那么(B→·γ,b)当然也属于 CLOSURE(I)，其中 b 必定是跟在 B 后面的终结符，即 b∈FIRST(βa)，若 β=ε，则 b=a。

(2) 具体代码设计

① 代码的传入参数是一个LR1Item的set，也就是一个集合。而且传引用，也就是说最终在函数里对于set的修改会落实到set的实际变化中。

② 代码首先获得一个项目集里的所有的已有的项目，利用一个LR1Item类型的queue来存该项目集里的所有的项目，然后利用循环遍历itemSet的LR1Item，首先取得所有的已有项目放入queue中。

③ 利用findPredictiveSym()函数计算当前处理项目的搜索符（预测符）。

④ 针对项目还未进栈的句柄部分（即圆点右侧的部分），逐个进行判断是否还有终结符以及产生式，如果有这样的一个产生式，为其添加圆点形成项目、利用checkEqual()函数判断该项目在不在项目集中，如果不在则加入项目集，并且为其计算并添加搜索符，这里需要将最终结果落实到closure()的传入参数LR1Item的set，所以需要对set进行insert。

⑤ 同时，加入新项目的项目集，在这里意味着一开始的queue尾部push了一个新的LR1Item。

2. findPredictiveSym()函数

(1) 本函数的意义在于“A→a · Bb”，B为currentSym，b为nextSymbol，算的是产生式左边为B的推出式的搜索符（预测符）。

(2) 具体代码设计

① 代码的传入参数是一个LR1Item项目item，和该项目的搜索符集合，一个int的set，这里用predictiveSymbol表示。

② 首先，如果当前“ · ”已经到了项目的尾部，即句柄完全进栈，那么这里predictiveSymbol就是item的自己的predictiveSymbol，无需做更多的变化。

③ 进入“ · ”未到项目的尾部的情况，还可以分为两种情况，代码中我们利用getNextSymbol()函数首先获得nextSymbol。然后第一种是例如“L→ · i”这种情景，对于这种情况可以通过将 · 后的nextSymbol与终结符表中的所有符号一一比对，只要发现这个nextSymbol是终结符，就将其加入搜索符。由于开始的终结符表存为了map形式，也就是说我们直接在map中find这个nextSymbol，匹配了即加入搜索符。

④ 第二种情景则是最为普遍的情况，即“A→a · Bb”这样的情形，这时需要计算b和已有搜索符 α 组成的bα的first集，代码中采取的方法，是首先利用producerMap（一个全局的map变量，记录了所有的产生式）取出了nextSymbol对应的所有产生式记为v1，然后将v1和已有的predictive Symbol放入getFirstSet()函数中去计算处理。

3. getFirstSet()函数

(1) 本函数完成了对于传入的符号的first集的计算以及同时完成了计算出first集之后，对于项目的predictiveSymbol也就是搜索符的修改。

(2) 具体代码设计：

① 函数传入的参数有int类型的producerID（因为代码中，每一个符号我们用一个不重复的数字进行对应），一个int类型的set的引用，记为firstSet。

② 首先，if判断，若利用producerVector查询producerID对应符号的产生式，发现产生式右侧为ε，那么直接结束函数，并返回false（函数的返回值为bool类型）。

③ 进入else，根据first的定义，if该符号的右部为一个终结符，则直接将该终结符加入到first集（即firstSet）中。

④ 进入else，当为非终结符时，设置一个bool类型的epsilonFlag，置为false，接着取出所有的该非终结符的右侧表达式，譬如$\alpha=X_1X_2...X_n$，则需要各自取出，然后对每个X继续getFirstSet()，只要出现了一个子getFirstSet为空的情况，就将epsilonFlag置为true，说明有空串。

⑤ 然后进入while循环，while判断条件为epsilonFlag==true同时没有计算到右侧表达式末尾则继续循环，在循环内，首先将epsilonFlag置false，接下来if处理的符号为非终结符，则继续获得当前的符号的右侧符号，同样的继续上述的步骤，即譬如$\alpha=X_1X_2...X_n$，则需要各自取出，然后对每个X继续getFirstSet()，只要出现了一个子getFirstSet为空的情况，就将epsilonFlag置为true，说明有空串。而如果处理的字符串为终结符，则在插入该终结符进入first集（即firstSet）中。

⑥ 最后跳出各种循环与if...else语句，return true，同时由于firstSet作为传引用的方式引入，也完成了对于first集的处理

4. minimizeItemset()函数

(1) 本函数的意义在于对一个LR1项目集的简化，由于生成完一个LR1项目集的closure之后，往往会出现，一个项目，不同的预测符同时出现在项目集中，要对于这样的情况进行简化。

(2) 具体代码设计：

① 函数传入参数为一个未简化的LR1项目集s1，类型为一个LR1Item的multiset，返回值为一个简化后的LR1项目集miniset，类型为multiset；

② 函数首先定义miniset并且清空；

③ 然后开始针对s1进行循环，循环内定义一个搜索符的set为predictiveSymSum然后定义bool变量findSimilar，初始为false，继续在miniset内创建一个循环（到此函数已有二层循环），如果外层循环子与内层循环子相似，这里使用自定义的checkLike()函数，则置findSimilar为true，同时将外层循环子的搜索符赋值给predictiveSymSum，这里就完成了搜索符合并。记录下此时的内层循环子的位置pos，跳出内层循环，接着判断findSimilar是否为true，如果为false，那么就直接将外层循环子插入到miniset，因为没有相似项目。如果为true，那么令一个LRItem itm=pos，并且根据pos在miniset中找到对应位置，擦除，再根据predictiveSymSum，将该合并后的搜索符加入到，miniset中去即可。最后return miniset；

④ checkLike()函数自定义的一个功能函数，作用是判断两个项目是否相似，遵守规则为产生式一样，预测符不一样，返回true。主要完成了两个项目对于产生式的比较与搜索符的比较，首先比较产生式，若产生式不一致，则直接返回false，其次再去比较搜索符，若搜索符不一致则返回true，若一致，则返回false，因为二者就是同一项目，这并不会出现再实际情况中。

3.2.3 构造LR(1)项目集规范族的详细设计，包含ProcessItemSet()函数

该部分构建了LR(1)自动机并生成了语法分析表，而这两件事是同步进行的。我们采用的算法依据的是上学期编译原理课上用的中文教材《编译原理及编译程序构造》中第6.4.1章“构造LR(1)项目集规范族的算法”，具体思路如下：

构造 LR(1)项目集规范族的算法

构造 LR(1)项目集族的算法本质上和构造 LR(0)项目集族的算法是一样的，先介绍两个函数：CLOSURE 和 GO。

(1) 函数 CLOSURE(I)—I 的项目集。

①I 的任何项目都属于 CLOSURE(I)；

②若项目(A→α·Bβ,a)属于 CLOSURE(I)，B→γ是一个产生式，那么对于 FIRST(βa)中每个终结符 b，如果(B→·γ,b)原来不在 CLOSURE(I)中，则把它加进去；

③重复步骤②，直至 CLOSURE(I)不再扩大为止。

因为(A→α·Bβ,a)属于 CLOSURE(I)，那么(B→·γ,b)当然也属于 CLOSURE(I)，其中 b 必定是跟在 B 后面的终结符，即 b∈FIRST(βa)，若 β=ε，则 b=a。

(2) GO 函数。

令 I 是一个项目集，X 是一个文法符号，函数 GO(I,X)定义为：

GO(I,X)=CLOSURE(J)

其中，J={任何形如(A→αX·β,a)的项目|(A→α·Xβ,a)∈I}。

可见在执行转换函数 GO 时，搜索符并不改变。

(3) 构造拓广文法 G′的 LR(1)项目集族 C 的算法如下：

```
PROCEDURE ITEMSETIR(1)
    BEGIN
        C:={CLOSURE({(S′→·S,#)})};
        REPEAT
            FOR C 中的每个项目集 I 和 G′的每个文法符号 X  DO
                IF GO(I,X)非空且不属于 C THEN 把 GO(I,X)加入 C 中
        UNTIL C 不再扩大为止
    END;
```

接下来是具体代码设计并实现的部分。

我先介绍一下我们在这部分里使用到的重要的自定义的数据结构、变量和函数：

① 结构体LR1State，表示LR(1)项目集，即自动机中的一个状态，它包含一个用于标识该状态的状态编号stateID（整数类型，从0开始），一个包含该状态下所有项目的集合itemSet（它是LR1Item类型的multiset），一张标明当前状态通过特定符号可以到达什么状态的表格moveItemSet（它是unordered_map类型）。

② LR(1)分析表、项目集转换动作表actionTable。

③ 项目集队列Q，存储了当前待处理的项目（它是LR1Item类型的队列）。

④ 函数closure(multiset<LR1Item>& itemSet)，求当前项目集的闭包，它的算法与先前“构造LR(1)项目集规范族的算法”的第一部分一致。

最后，算法实现步骤如下：

① 求初态项目集I_0：从$(S'\rightarrow \bullet S,\#)$项目开始求闭包，再简化（该操作如下），然后将它存入项目集队列Q中。

例如，仍以 G(Ex)文法为例，构造其 LR(1)项目集族。初态项目集 I_0，从(S′→ · S，#)项目开始求闭包可得图 6.9 所示的初态项目集：

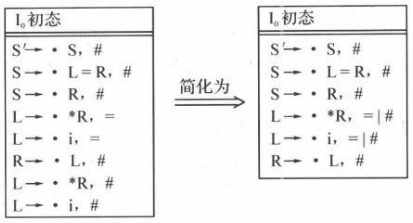


图 6.9 初态项目集

② 判断Q是否为空，为空则结束并返回1，不为空则转③。

③ 弹出队列Q顶的一个项目集itemset；提前定义好actionTable的一行f，长度为maxnum，是终结符和非终结符个数之和。

④ 对于当前项目集中的各项目求后继项目集（其算法与先前“构造LR(1)项目集规范族的算法”的第二部分类似），同时填写分析表actionTable。注意：(1) 对于新产生的项目集需要求闭包并最小化后再放入；(2) 对于每个“新”状态，要判断是否与之前的某个状态相同。

3.2.4 LR(1)分析过程可视化的详细设计，包含PrintSymbolTable()、getSymbol()、PrintItem()、PrintItemSet()、PrintActionTable()函数

本部分负责 LR(1)分析过程可视化的详细设计，主要是分为四个方面：①对.y文件解析出来的终结符与非终结符进行可视化；②打印所有产生式；③打印LR(1)项目集；④LR(1)分析表的可视化。

① 对.y文件解析出来的终结符与非终结符进行可视化

PrintSymbolTable()函数通过map数据结构的迭代器打印出终结符与非终结符与编号，如果存在语义值类型也会随后显示出来。

② 所有产生式的可视化

通过getSymbol()函数将Producer结构体中以编号存储的产生式转化为字符串形式，同时若存在语法规则动作会随后显示出来，否则输出“空”。

③ LR(1)项目集的可视化

PrintItemSet()函数，首先输出“输出项目集”字样，通过遍历函数输入的参数（一个LR1Item的multiset i0），在遍历的每一次循环中：首先输出当前循环子（LR1Item类型）的项目号，利用getProducerID来实现，接着输出对应的产生式的左侧，利用getSymbol实现，接着进入针对产生式右部的循环，循环条件为循环子超过右部的字符数量时，在循环中，如果发现循环子的大小等于当前外部的循环子（LR1Item类型）的getCurrentPosition大小，则输出·符号，因为getCurrentPosition函数就记录一个项目集中每一个项目的·的位置。结束内部的循环，进入if-else语句进行一些情况的判断：如发现getCurrentPosition值为-1，

那么说明项目规约完毕，输出无待规约符号；对应的else意味着仍然有待规约的符号，输出待规约符号，并在后面跟随输出当前·处理到的位置处的符号利用getSymbol（当前项目.getCurrentSymbol（））来实现。在函数最后，再进行一次遍历，通过遍历当前项目的getPredictiveSym，也就是说遍历当前项目的预测符集合，将预测符集合输出到到项目的尾部。函数在输出过程使用“ ”、“\n”来控制输出精度。

④ LR(1)分析表的可视化

通过PrintActionTable()函数打印actionTable中存储的内容，对于每个项目集（状态）经过终结符达到的状态和动作（ACTION部分），以及经过非终结符达到的状态（GOTO部分）。

3.2.5 语法分析代码生成的详细设计，即GenerateCode() 函数

本部分负责从LR(1)分析表生成语法分析器的C代码。

首先，语法分析的对象是 Lex 送来的 Token 流，因此 Yacc 应与 Lex 联合使用。语法分析代码生成器会利用C语言的extern关键字，从词法分析器处引入Lex的词法分析函数string analysis(char*,int)，从而我们编写的Yacc通过readToken()函数从文件流中反复读入字符串yytext并调用 analysis()函数来分析，这一过程中我们实则来测试我们之前的Lex词法分析器代码。

然后，流式的yytext及其对应的TokenName，借助语法分析表就可以完成基本的语法分析工作。主要用到三个数据结构：一个存储符号语义值的符号栈、一个存储状态转移路径的状态栈、一个存储ACTION- GOTO表的结构actionTable。由于符号栈与状态栈操作在语法分析过程中一致，我们使用了Sym结构体：

```
struct Sym
{
    int symbol;
    int state;
};
```

定义stack<Sym> symStack;来模拟符号栈与状态栈，每次查ACTION- GOTO表时，里面记录动作为“移进”时，新建一个将栈结点（Sym）存储状态与语义信息，然后压入symStack；当记录动作为“归约”时，表内存储用于归约的相应表达式编号，执行有关动作代码，symStack弹出等同于表达式中归约符号个数的状态（我们创建了一个名为producerN的一维数组来维护每个表达式的归约符号个数），并立刻递归地解析归约而成的非终结符 号以便执行GOTO动作；当进行到ACCEPT时，语法分析成功，然后退出程序；当进行到ERROR时，报错处理。其中，语法树的打印见3.2.6。

最后，整合上述各部分，以及.y 源文件中的直接复制部分和用户代码部分，即完成了语法分析器代码的生成。将它与词法分析器代码进行联合编译，即可完成语法分析。

3.2.6 代码的语法树生成的详细设计，即printMTree()函数

1. 相关变量

首先创建相关的结构体，Tree、node。node结构体中包含，节点名称（String类型），子节点数量（int类型），子节点集合（node*类型的vector变量），Tree结构体中包含一个node节点，作为树的根节点。定义一个节点栈syntaxTreeStack，用于控制结点的记录顺序（node*的一个stack），以及一个语法树syntaxTree（Tree型变量）

2. 具体代码设计

当代码分析结果为ACCEPT时开始打印树，最初，syntaxTree的root要记录为syntaxTreeStack的top。而代码仍在分析的时候，对与syntaxTreeStack进行操作，这一部分为树能够建立起来的关键部分。

如果是在规约项目，首先创建一个空的父节点parentNode，该节点的name填上当前正在使用的规约式的产生式左部，然后进入循环，循环的跳出条件条件为循环子大于产生式右部符号数量。循环内，不断的将syntaxTreeStack的顶端push出来存入parentNode的child。并且父节点的子节点数目也同时增加。完成后，将该parentNode压入syntaxTreeStack。

剩下的情况下，就创建一个新节点newNode，newNode的name

来到实际的printMTree函数设计，函数的参数列表（string（当前节点的垂直间隔），TreeNode（当前节点），bool（判断当前节点是否为第一个子节点用isFirst命名））首先当树中的节点不为null时，就一直往下推进，输出一个垂直间隔（用于区分不同的子节点分支，因为是横过来打印的）。利用一个isFirst判断此时为新的子节点还是同一个父节点产生的子节点的兄弟节点，再进入循环，循环的跳出条件为循环子大于当前节点的子节点的数量，然后在循环内部进入函数，形成递归调用，而递归调用中的prefx值有prefix + (isFirst ? "|\t\t\t" : " \t\t\t")来决定；node则设置为当前node的子结点列表中对应下标为循环子的那一个；isFirst的bool值通过判断循环子是否等于0来决定。

4 使用说明

测试最终支持C99标准语法分析的 Lex+Yacc 使用说明

1. 在终端运行编译好的yacc.cpp文件
2. 按照提示输入各种文件名称
3. 得到结果

```
(base) matthew@MatthewdeMacBook-Air output % ./"Yacc"
Please input the file name:
../test_1.c
yytext: int TokenName:INT
yytext: main TokenName:IDENTIFIER
yytext: ( TokenName:(
yytext: ) TokenName:)
yytext: { TokenName:{
yytext: int TokenName:INT
yytext: a TokenName:IDENTIFIER
yytext: = TokenName:=
yytext: 0 TokenName:CONSTANT
yytext: ; TokenName:;
yytext: int TokenName:INT
yytext: c TokenName:IDENTIFIER
yytext: = TokenName:=
yytext: 0 TokenName:CONSTANT
yytext: ; TokenName:;
yytext: if TokenName:IF
yytext: ( TokenName:(
yytext: a TokenName:IDENTIFIER
yytext: == TokenName:EQ_OP
yytext: 2 TokenName:CONSTANT
yytext: ) TokenName:)
yytext: { TokenName:{
yytext: return TokenName:RETURN
yytext: a TokenName:IDENTIFIER
yytext: ; TokenName:;
yytext: } TokenName:}
yytext: return TokenName:RETURN
yytext: c TokenName:IDENTIFIER
yytext: ; TokenName:;
yytext: } TokenName:}
Compile sucessfully!
```

5 测试用例与结果分析

测试用例 1: test_1.c

Yacc.y 的动作代码是在规约时输出规约用到的产生式,故从上到下为规约过程，从下到上为推导过程。经验证，没有错误。不过由于其语法树较为复杂,这里没有予以打印生成

```
int main()
{
   int a = 0;
   int c = 0;
   if (a == 2)
   {
        return a;
   }
   return c;
}
```

```
yytext: int TokenName:INT
yytext: main TokenName:IDENTIFIER
yytext: ( TokenName:(
yytext: ) TokenName:)
yytext: { TokenName:{
yytext: int TokenName:INT
yytext: a TokenName:IDENTIFIER
yytext: = TokenName:=
yytext: 0 TokenName:CONSTANT
yytext: ; TokenName:;
yytext: int TokenName:INT
yytext: c TokenName:IDENTIFIER
yytext: = TokenName:=
yytext: 0 TokenName:CONSTANT
yytext: ; TokenName:;
yytext: if TokenName:IF
yytext: ( TokenName:(
yytext: a TokenName:IDENTIFIER
yytext: == TokenName:EQ_OP
yytext: 2 TokenName:CONSTANT
yytext: ) TokenName:)
yytext: { TokenName:{
yytext: return TokenName:RETURN
yytext: a TokenName:IDENTIFIER
yytext: ; TokenName:;
yytext: } TokenName:}
yytext: return TokenName:RETURN
yytext: c TokenName:IDENTIFIER
yytext: ; TokenName:;
yytext: } TokenName:}
Compile sucessfully!
```

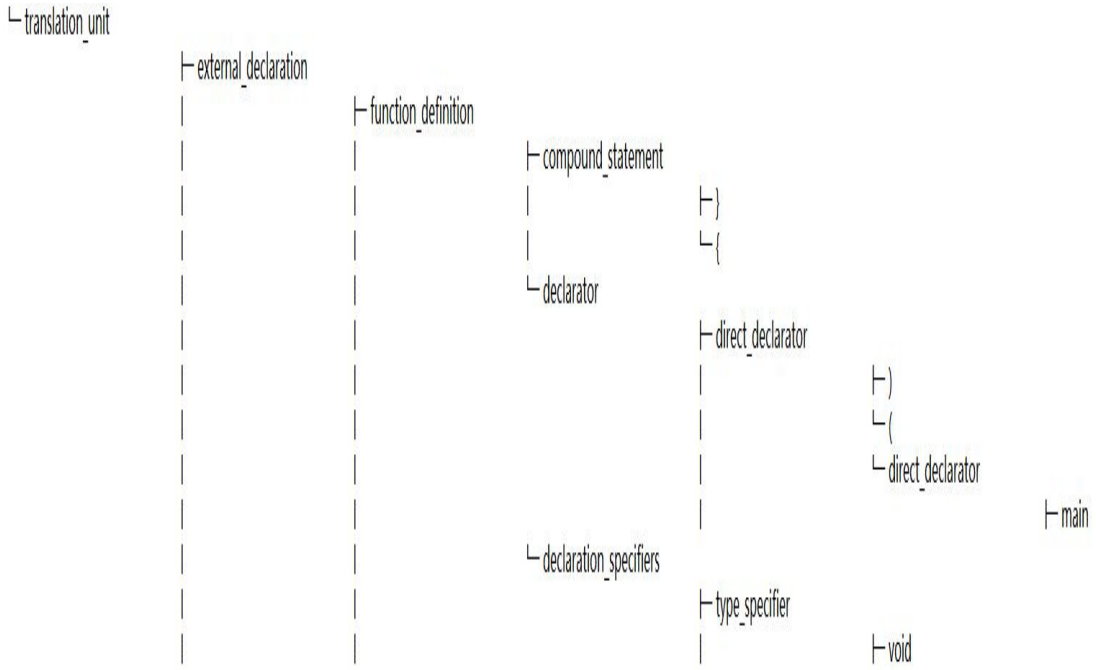
测试用例 2: test_2.c

Yacc.y 的动作代码是在规约时输出规约用到的产生式,故从上到下为规约过程，从下到上为推导过程。经验证，没有错误。由于其语法树较简单，这里进行予以打印生成。

```
void main()
{
}
```

```
yytext: void TokenName:VOID
yytext: main TokenName:IDENTIFIER
yytext: ( TokenName:(
yytext: ) TokenName:)
yytext: { TokenName:{
yytext: } TokenName:}
Compile sucessfully!
```

语法树输出结果



测试用例 3: test_3.c

Yacc.y 的动作代码是在规约时输出规约用到的产生式,故从上到下为规约过程，从下到上为推导过程。经验证，发现测试代码中有错误推导失败。这是正确的，因为测试代码给了一个不完整的的int符号。

```
void main()
{
   int a;
   in b;
   return a;
}
```

```
yytext: void TokenName:VOID
yytext: main TokenName:IDENTIFIER
yytext: ( TokenName:(
yytext: ) TokenName:)
yytext: { TokenName:{
yytext: int TokenName:INT
yytext: a TokenName:IDENTIFIER
yytext: ; TokenName:;
yytext: in TokenName:IDENTIFIER
yytext: b TokenName:IDENTIFIER
Compile Error!
```