编译方法实验报告

57119101 王晨阳
2022 年 1 月 1 日

Content

Motivation

通过实验更好地掌握编译原理这门课，强化有关词法分析和语法分析的知识，并将理论应用到实践中来。

Content Description

实验包含两部分。

第一部分是词法分析的实现。 这部分需要输入一组正则表达式（或者自己设计的 lex 文件）作为基本语法，然后输入按照这个语法编写的程序。 输出是对该程序进行词法分析后得到的 token 序列。

第二部分是语法分析的实现。 这部分需要 CFG（或者自己设计的 yacc 文件）和一个程序的输入。 输出语法分析结果。

Ideas

对于第一个实验，我们考虑首先将正则表达式转换为 NFA，然后将 NFA 转换为 DFA，再最小化 DFA，最后将生成的 DFA 应用于我们的程序以获得 token 序列。

第二个实验，先生成了 LR 分析表，然后生成了 LALR 分析表。 最后，在前面的实验的基础上，通过分析表对程序进行分析。

Assumptions

囿于时间不足的问题，我选择了一个较为简单的语言：

PASCAL

这是个很古老的语言（几乎没人用了），但我之前学过，对它很熟悉。

它的 lex 文件

```
%{
#include <stdio.h>
#include "y.tab.h"

int line_number = 0;

void yyerror(char *message);

%}

%x COMMENT1 COMMENT2

white_space       [ \t]*
digit             [0-9]
alpha             [A-Za-z_]
alpha_num         ({alpha}|{digit})
hex_digit         [0-9A-F]
identifier        {alpha}{alpha_num}*
unsigned_integer  {digit}+
hex_integer       ${hex_digit}{hex_digit}*
exponent          e[+-]?{digit}+
i                 {unsigned_integer}
real              ({i}\.{i}?|{i}?\.{i}){exponent}?
string            \'([^'\n]|\'\')+\'
```

```
25  bad_string        \'([^'\n]|\'\')+
26
27  %%
28
29  "{"                 BEGIN(COMMENT1);
30  <COMMENT1>[^}\n]+
31  <COMMENT1>\n            ++line_number;
32  <COMMENT1><<EOF>>    yyerror("EOF in comment");
33  <COMMENT1>"}"        BEGIN(INITIAL);
34
35  "(*"                BEGIN(COMMENT2);
36  <COMMENT2>[^)*\n]+
37  <COMMENT2>\n            ++line_number;
38  <COMMENT2><<EOF>>    yyerror("EOF in comment");
39  <COMMENT2>"*)"       BEGIN(INITIAL);
40  <COMMENT2>[*)]
41
42  and                 return(AND);
43  array               return(ARRAY);
44  begin               return(_BEGIN);
45  case                return(CASE);
46  const               return(CONST);
47  div                 return(DIV);
48  do                  return(DO);
49  downto              return(DOWNTO);
50  else                return(ELSE);
51  end                 return(END);
52  file                return(_FILE);
53  for                 return(FOR);
54  function            return(FUNCTION);
55  goto                return(GOTO);
56  if                  return(IF);
57  in                  return(IN);
58  label               return(LABEL);
59  mod                 return(MOD);
60  nil                 return(NIL);
61  not                 return(NOT);
62  of                  return(OF);
63  packed              return(PACKED);
64  procedure           return(PROCEDURE);
65  program             return(PROGRAM);
66  record              return(RECORD);
67  repeat              return(REPEAT);
68  set                 return(SET);
69  then                return(THEN);
70  to                  return(TO);
71  type                return(TYPE);
72  until               return(UNTIL);
73  var                 return(VAR);
```

```
while                   return(WHILE);
with                    return(WITH);

"<="|"=<"               return(LEQ);
"=>"|">="               return(GEQ);
"<>"                    return(NEQ);
"="                     return(EQ);

".."                    return(DOUBLEDOT);

{unsigned_integer}      return(UNSIGNED_INTEGER);
{real}                  return(REAL);
{hex_integer}           return(HEX_INTEGER);
{string}                return{STRING};
{bad_string}            yyerror("Unterminated string");

{identifier}            return(IDENTIFIER);

[*/+\-,^.;:()\[\]]      return(yytext[0]);

{white_space}           /* do nothing */
\n                      line_number += 1;
.                       yyerror("Illegal input");

%%

void yyerror(char *message)
{
   fprintf(stderr,"Error: \"%s\" in line %d. Token = %s\n",
           message,line_number,yytext);
   exit(1);
}
```

当然，这还是太复杂了。鉴于写 pascal 的人很少取用高级功能，因此语法糖给去掉了大部分程序还是可以跑起来。所以我**删除了一些不常见的内容**（比如面向对象）。

我对 lex 进行了一些更改，以使我更容易使用程序来处理它们。

```
(~()+(~))+(~+)+(~*)+(:)+(;)+(,)+(-)+(=)+(:.=)+(<.>)+(<.=)+(>.=)+(>)+(<)+(o.r)+
(a.n.d)+(m.o.d)+(d.i.v)
([0-9]).([0-9])*
([a-z]+[A-Z]+_).([a-z]+[A-Z]+[0-9]+_)*
(m.a.i.n)+(i.f)+(t.h.e.n)+(e.l.s.e)+(w.h.i.l.e)+(f.o.r)+(d.o)+(b.e.g.i.n)+
(e.n.d)+(v.a.r)+(i.n.t.e.g.e.r)+(r.e.a.l)+(f.u.n.c.t.i.o.n)+(a.r.r.a.y)+
(p.r.o.c.e.d.u.r.e)+(r.e.s.u.l.t)+(p.r.o.g.r.a.m)
```

在错误处理方面，由于时间限制，我只考虑了几个错误情况。 其他情况直接报错，不提示具体错误原因。 但是程序的鲁棒性并不是特别强，仍然存在少量错误会导致直接崩溃。

Related FA

无法在此展示 FA。 这是因为语法规则很多，**导致FA很大**（它已经获得了数千行格式化数据），**不能直接在报告中写出来**。 如果想看可以自己跑一下。

Important Data Structures

为了简化程序的实现，我使用了很多STL中的数据结构。

在词法分析程序中，我分别定义了rules、NFA、NFA state 和 DFA 的结构。 在结构中，我使用 map 等数据结构来存储边和节点的内容。

同样，在语法分析程序中，我定义了 LR 分析表的结构。 该表按行分为 LR 状态，每个状态有若干个 LR 项。

以上就是程序存储源数据、中间过程、分析结果最重要的数据结构。

下面是一点例子：

```
class State
{
public:
    set<State*> NFAStates;

    typedef vector<State*> Table;
    typedef set<State*>::iterator StateIterator;
    multimap<char, State*> Transition;
    int StateID;
    bool Accept;
    bool Marked;
    int  GroupID;
    /*
    Some other things
    */
}
```

Core Algorithms

核心算法有以下几个部分：

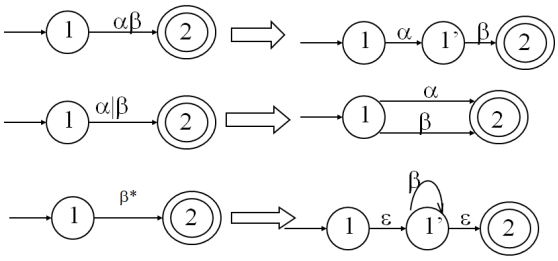
RE to suffix mode

正则表达式的识别和处理与带括号的四种算术运算类似。 我使用类似于**波兰语表达**的方法来处理它。

通过将 RE 转换为后缀表达式，我们可以使用堆栈来处理它。 每次取一个字符，如果是变量（即RE中的标识符），则压入栈中； 如果是运算符，则从堆栈中取出相应数量的操作数进行运算，并将结果压回到堆栈中。

RE to NFA

- 将正则表达式 r 拆分为子表达式
- 对于 r 中的每个 symbol
  - ε:
  - a:
- 将它们组合起来



程序直接使用我上面提到的栈模拟这个过程，也就是 Thompson NFA 方法。

```
for (int i = 0; i < (int)PostfixRE.size(); i++)
{
    char Ch = PostfixRE[i];

    if (!IsOperator(Ch))
    {
        if (Ch == '~')
        {
            i++;
            Ch = PostfixRE[i];
        }
        CreateOneCharNFA(Ch);
    }
    else if (IsOperator(Ch))
    {
        if (Ch == '.')
            Concat();
        else if (Ch == '*')
            Closure();
        else if (Ch == '+')
            Or();
    }
```

```
}
```

NFA to DFA

- I_0 =ε-closure(S_0),$I_0 \in Q$
- 对所有的 $I_i \in Q$，I_t =ε-closure($move(I_i, a)$)，即 I_i 经过一步（ε 不计入步数）可以到达的所有 state，一个 I_i 可能会有多个 I_t。若 $I_t \notin Q$，则把 I_t 放入 Q
- 重复上一步，直到没有新 state 可以被放入 Q
- $F = \{I \mid I \in Q, I \cap Z \neq \varnothing\}$

其中，ε-closure(T) 为 T 中的 NFA state s 只经过 ε-transition 就可以得到的 NFA states 的集合

同样，我使用了一个简单的模拟并不断搜索，直到没有新的状态。

下面是核心代码

```
while (!UnVisitedStates.empty())
{

    State *CurDFAState = UnVisitedStates[UnVisitedStates.size() - 1];
    UnVisitedStates.pop_back();

    std::set<char>::iterator iter;
    for (iter = InputSet.begin(); iter != InputSet.end(); ++iter)
    {
        std::set<State *> MoveRes, EpsilonClosureRes;

        Move(*iter, CurDFAState->GetNFAState(), MoveRes);
        EpsilonClosure(MoveRes, EpsilonClosureRes);

        StateIterator MoveResItr;
        StateIterator EpsilonClosureResItr;

        bool bFound = false;
        State *s = NULL;
        for (int i = 0; i < (int)DFATable.size(); ++i)
        {
            s = DFATable[i];
            if (s->GetNFAState() == EpsilonClosureRes)
            {
                bFound = true;
                break;
            }
        }
        if (!bFound)
        {
            State *U = new State(EpsilonClosureRes, NextStateID++);
            UnVisitedStates.push_back(U);
```

```
33                DFATable.push_back(U);
34                CurDFAState->AddTransition(*iter, U);
35            }
36            else
37            {
38                CurDFAState->AddTransition(*iter, s);
39            }
40        }
41    }
```

Minimizing the number of States of a DFA

- 构建初始分割集合 Π_0，该集合包含两组 state：accepting states 和 non-accepting states。

  $\Pi_0 = \{I_0^1, I_0^2\}$
- 对于 Π_{i-1} 中的每组 state，对其进行划分。该组中的任意两个 state 会被划分到新集合的同一组中，当且仅当对于任意输入，这两个 state 的转换结果到达同一组。将所得的划分结果记录为 Π_i。重复执行直到 $\Pi_i = \Pi_{i-1}$
- 在 Π_i 的每组中选取一个代表，它们构成了新的 state 集合
- 若存在 dead state（即非 accepting 且对于所有输入，只能到达它自己）则删了它

  删除所有由 start state 不能到达的 state

Calculation of FIRST and FOLLOW

- $\mathrm{FIRST}(\alpha)$ 计算

  X 为任意 symbol 或 string
  - 若 X 为 terminal symbol，则 $\mathrm{FIRST}(X) = \{X\}$
  - 若有 $X \to \varepsilon$，则将 ε 加入 $\mathrm{FIRST}(X)$
  - 若 X 为 non-terminal 或 string，且 $X \to Y_1 Y_2 \cdots Y_k, Y_j \in (V_N \cup V_T)$

    若 $Y_1, \cdots, Y_{i-1}, i < k$ 中均有 ε symbol，则向 $\mathrm{FIRST}(X)$ 中加入 $\mathrm{FIRST}(Y_i) - \{\varepsilon\}$

    若 $Y_1, \cdots, Y_k$ 中均有 ε symbol，则向 $\mathrm{FIRST}(X)$ 中加入 ε
- $\mathrm{FOLLOW}(A)$ 计算
  - S 为 start symbol，则将 \$ 放入 $\mathrm{FOLLOW}(S)$
  - 若 G 中有 $B \to \alpha A \beta$，则将 $(\mathrm{FIRST}(\beta) - \varepsilon)$ 放入 $\mathrm{FOLLOW}(A)$
  - 若有 $B \to \alpha A$ 或 $B \to \alpha A \beta$ 且 $\mathrm{FIRST}(\beta)$ 包含 ε，则将 $\mathrm{FOLLOW}(B)$ 放入 $\mathrm{FOLLOW}(A)$

Constructing LR and LALR tables

- $\mathrm{closure}(I)$
  - I 中所有 items 都属于 $\mathrm{closure}(I)$
  - 若 $(A \to \alpha \cdot B\beta, a)$ 属于 $\mathrm{closure}(I)$，对于每个 production $B \to \gamma$ 和 $\mathrm{FIRST}(\beta a)$ 中的每个 terminal symbol b，若 $(B \to \cdot\gamma, b)$ 不在 $\mathrm{closure}(I)$ 中，则加入进去。重复这一步骤
- $\mathrm{goto}(I, X)$

$\text{goto}(I, X) = \text{closure}(J)$，其中 J 为 I 中所有形如 $(A \to \alpha X \cdot \beta, a)$ 或 $(A \to \alpha \cdot X\beta, a)$ 的 items

- 构造 the sets of LR(1) items
  - 初始化 $C = \text{closure}(\{(S' \to \cdot S, \$)\})$
  - 对于 C 中的每个 items 集合 I 和每个 symbol X，如果 goto(I,X) 非空且不属于 C，则加入到 C 中。不断重复直到 C 不再扩大
- $C = \{I_0, I_1, \cdots, I_n\}$
- 若 $(A \to \alpha \cdot a\beta, b) \in I_k$ 且 $\text{goto}(I_k, a) = I_j$，则 $\text{action}[k, a] = S_j$
- 若 $(A \to a\cdot, a) \in I_k$，则对所有 $a \in \text{FOLLOW}(A)$，$\text{action}[k, a] = r_j$，$j$ 为 production $A \to \alpha$ 的编号。其中 A 不为 S'
- 若 $\text{goto}(I_k, A) = I_j, A \in V_N$，则 $\text{goto}[k, A] = j$
- 若 $(S' \to S\cdot, \$) \in I_k$，则 $\text{action}[k, \$] = \text{accept}$

这部分内容还是模拟的，但是很麻烦。

然后合并得到 LALR。

然而，这里我无法清晰截取一段程序来展示这个算法，因为在后面的输出过程中（后面会写到），我把很多功能混在一起了。

Cases

先面试我使用的输入样例

```
program example(input, output);
var x, y: integer;
function gcd(a, b: integer): result integer;
begin
    if b = 0 then gcd:= a
    else gcd:= gcd(b, a mod b)
end
begin
    READ(x, y);
    WRITE(gcd(x, y))
end
```

词法分析的输出结果为

```
program KEYWORD
example IDENTIFIER
(   SPECIALSYMBOL
input   IDENTIFIER
,   SPECIALSYMBOL
```

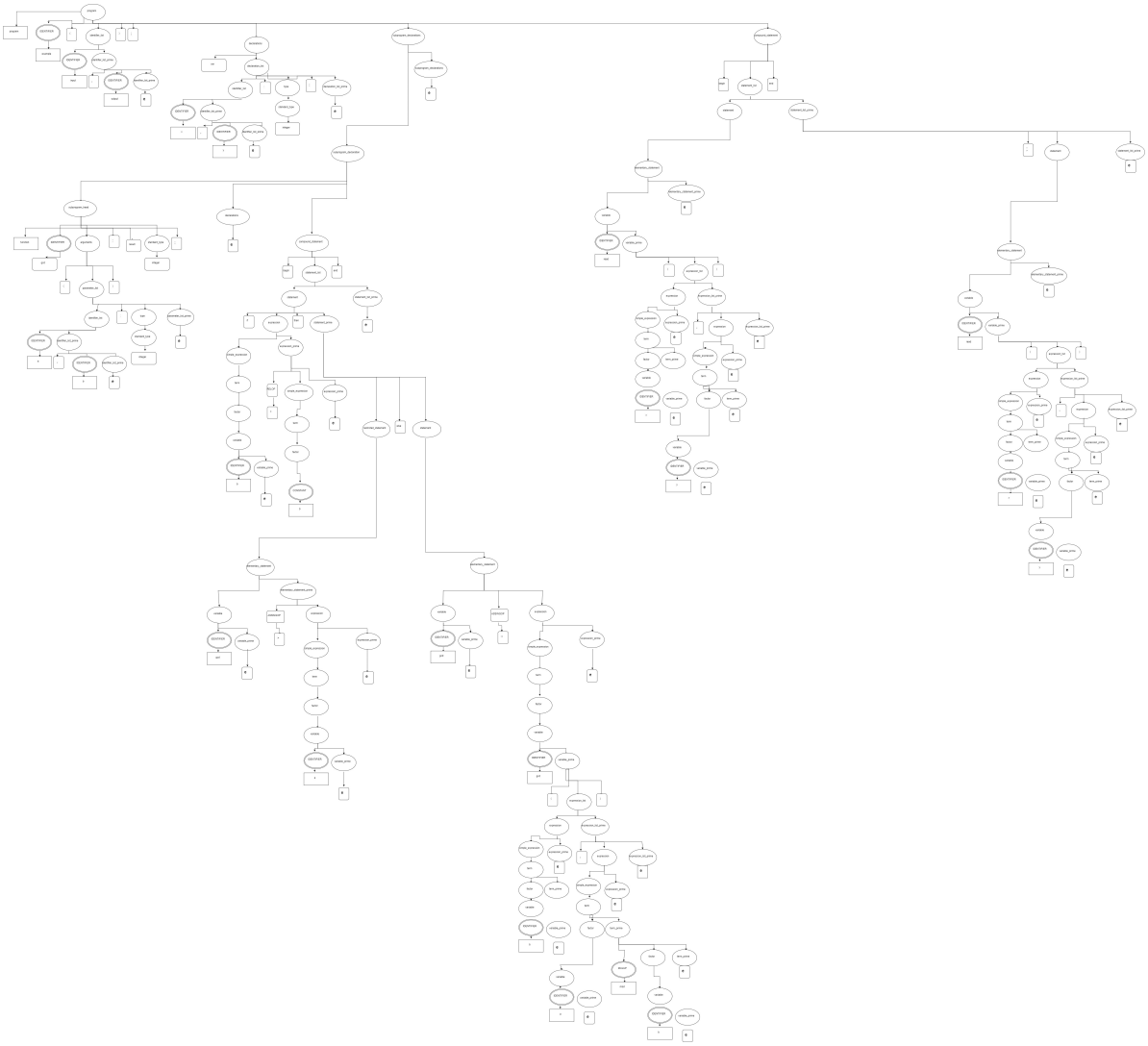
```
output  IDENTIFIER
)   SPECIALSYMBOL
;   SPECIALSYMBOL
var KEYWORD
x   IDENTIFIER
,   SPECIALSYMBOL
y   IDENTIFIER
:   SPECIALSYMBOL
integer KEYWORD
;   SPECIALSYMBOL
function    KEYWORD
gcd IDENTIFIER
(   SPECIALSYMBOL
a   IDENTIFIER
,   SPECIALSYMBOL
b   IDENTIFIER
:   SPECIALSYMBOL
integer KEYWORD
)   SPECIALSYMBOL
:   SPECIALSYMBOL
result  KEYWORD
integer KEYWORD
;   SPECIALSYMBOL
begin   KEYWORD
if  KEYWORD
b   IDENTIFIER
=   RELOP
0   CONSTANT
then    KEYWORD
gcd IDENTIFIER
:=  ASSIGNOP
a   IDENTIFIER
else    KEYWORD
gcd IDENTIFIER
:=  ASSIGNOP
gcd IDENTIFIER
(   SPECIALSYMBOL
b   IDENTIFIER
,   SPECIALSYMBOL
a   IDENTIFIER
mod MULOP
b   IDENTIFIER
)   SPECIALSYMBOL
end KEYWORD
begin   KEYWORD
READ    IDENTIFIER
(   SPECIALSYMBOL
x   IDENTIFIER
,   SPECIALSYMBOL
```

```
y   IDENTIFIER
)   SPECIALSYMBOL
;   SPECIALSYMBOL
WRITE   IDENTIFIER
(   SPECIALSYMBOL
gcd IDENTIFIER
(   SPECIALSYMBOL
x   IDENTIFIER
,   SPECIALSYMBOL
y   IDENTIFIER
)   SPECIALSYMBOL
)   SPECIALSYMBOL
end KEYWORD
```

它没有涵盖所有情况（当然那是不可能的），但足以证明程序可以正常正确运行。

语法分析程序的输出是（你可以在附录中找到大图）：

[这里，我参考了网上的资料，把语法分析的结果画成一张图。 这样子清楚多了]



经核实，结果正确。

Problems Occurred and Related Solutions

我一开始尝试编写 pascal 语言的 lex 文件，它的复杂性超乎想象。 但是后来突然发现官方提供了标准文档！ 我只需要复制粘贴，然后稍微修改一下，方便我自己的程序获取和处理。

Feelings and Comments

这个程序达到几千行，我当然不能在这么短的时间内完成，期间参考了不少资料，学到了很多新东西（比如绘制 xml 图像）。 **我参考的资料列在实验报告的最后。**

通过这个实验，我一方面进一步巩固了本学期所学的知识，牢牢掌握了生成 RE 的方法，以最小化 DFA 和 LALR 分析表。 我相信这对期末考试很有帮助。

另一方面，这个实验也提高了我的编程技巧。 通过写一个完整的项目，熟悉了面向对象编程和 C++ 的相关知识。 受益良多。

可惜这个实验时间不够多，一个人要想做好的话，工作量还是太大了。

Referrences

[1] Plex and Pyacc - Free Pascal wiki
[2] kdakan/Building a Pascal compiler with C, YACC & Lex
[3] mapron/Simple Pascal interpreter and parser
[4] luice/BUAA-Compiler-Pascal-to-x86
[5] ceciliazhou/A compiler parsing code in a language which is a subset of Turbo Pascal