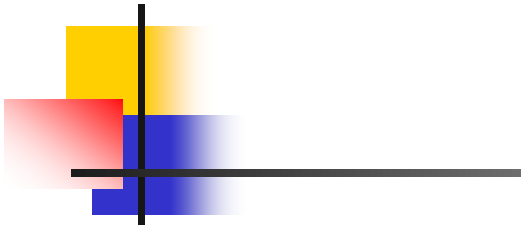
Implementation of a TINY Scanner



Tokens of the TINY Language

DFA of the TINY Scanner

The Code to Implement the DFA

The DFAs for the Tokens of TINY

Reserved Words	Special Symbols	Other
if	+	number
then	-	
else	*	
end	/	
repeat	=	
until	<	Identifier
read	(	
write	)	
	;	
	:=	

- 1. Special Symbols
  - + - * / = < ( ) ;
  - :=
- 2. Number
- 3. ID
- 4. White space, comments
  - Comments are enclosed in curly brackets {…} and cannot be nested.

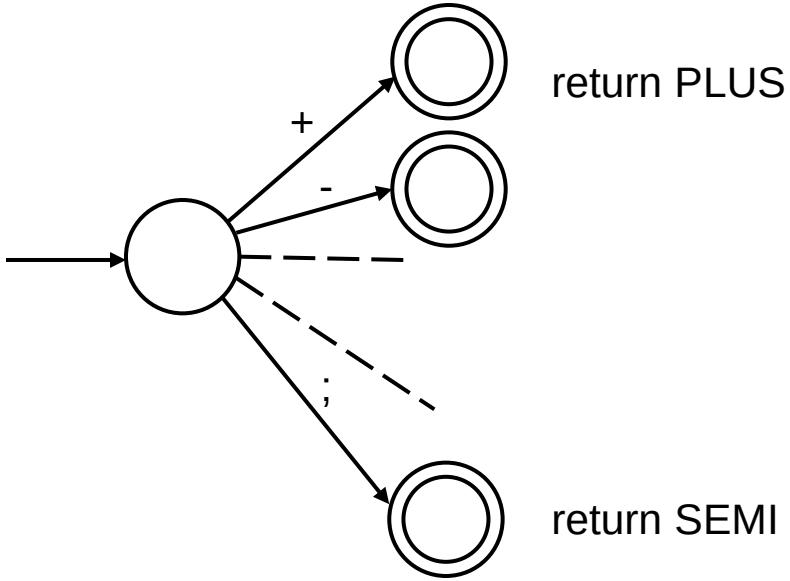
Tokens of the TINY Language

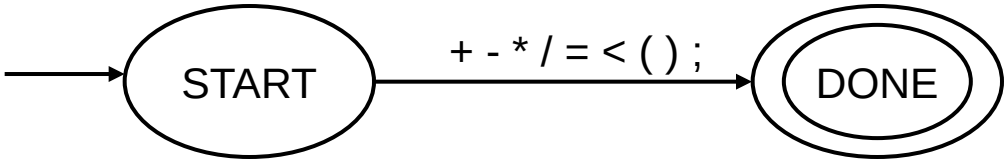
DFA of the TINY Scanner

The Code to Implement the DFA

The DFAs for the Tokens of TINY

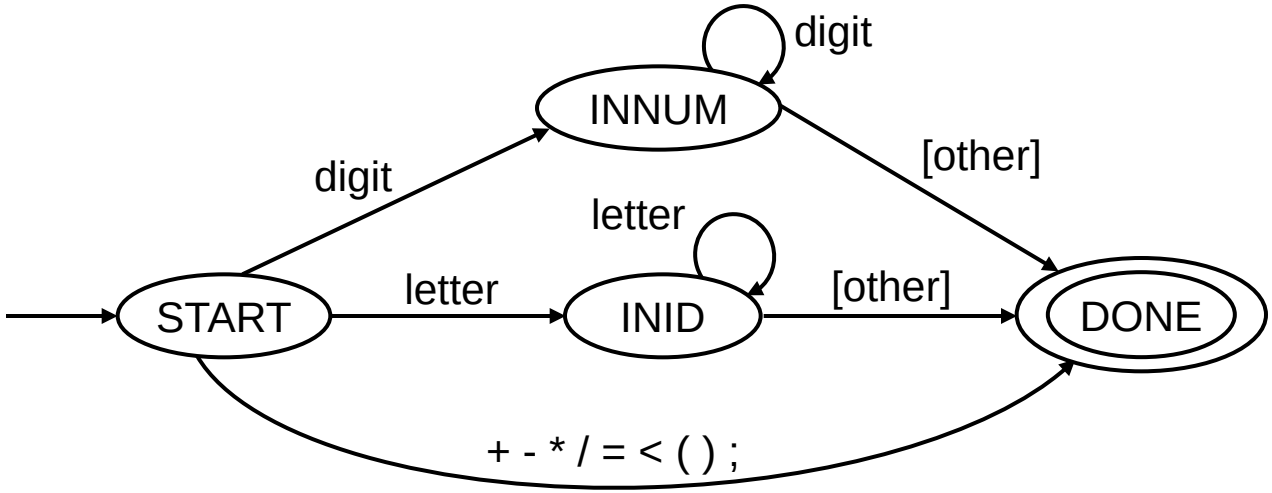
- 1.The DFA for the special symbols except assignment





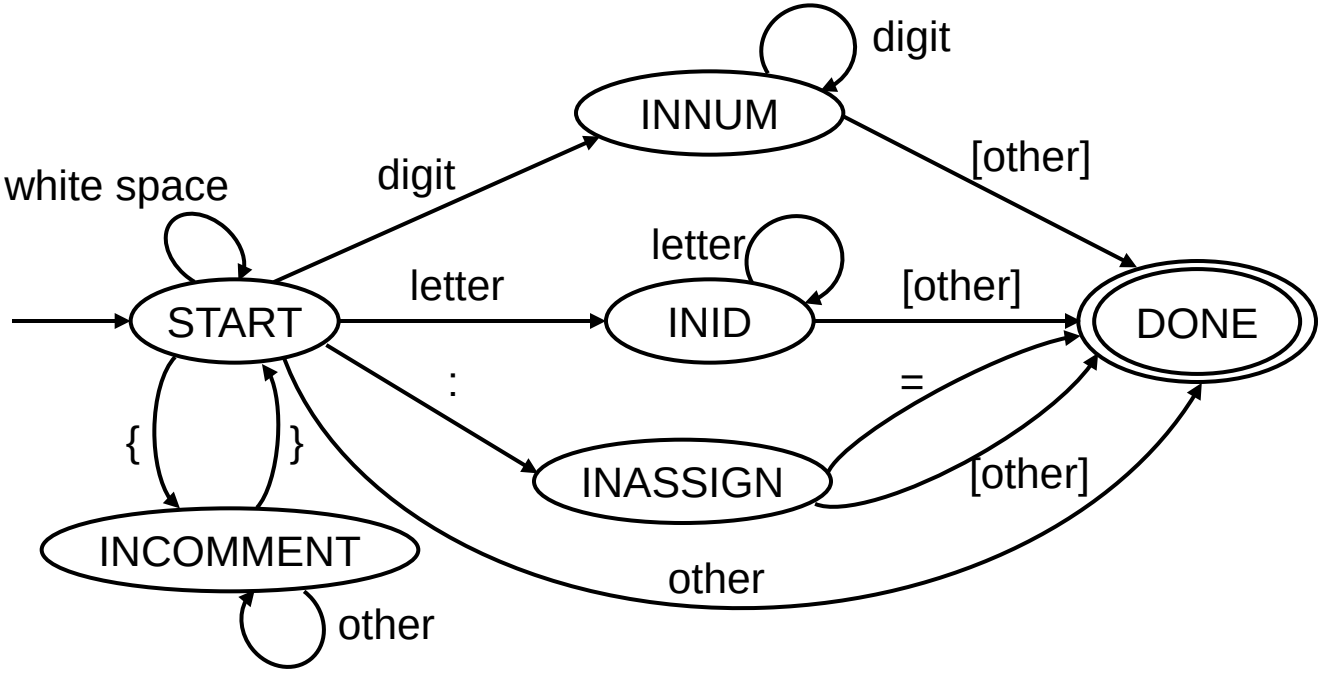
The DFAs for the Tokens of TINY

- 2. Combine the DFA with DFAs that accept numbers
- 3. Combine the DFA with DFAs that accept identifiers

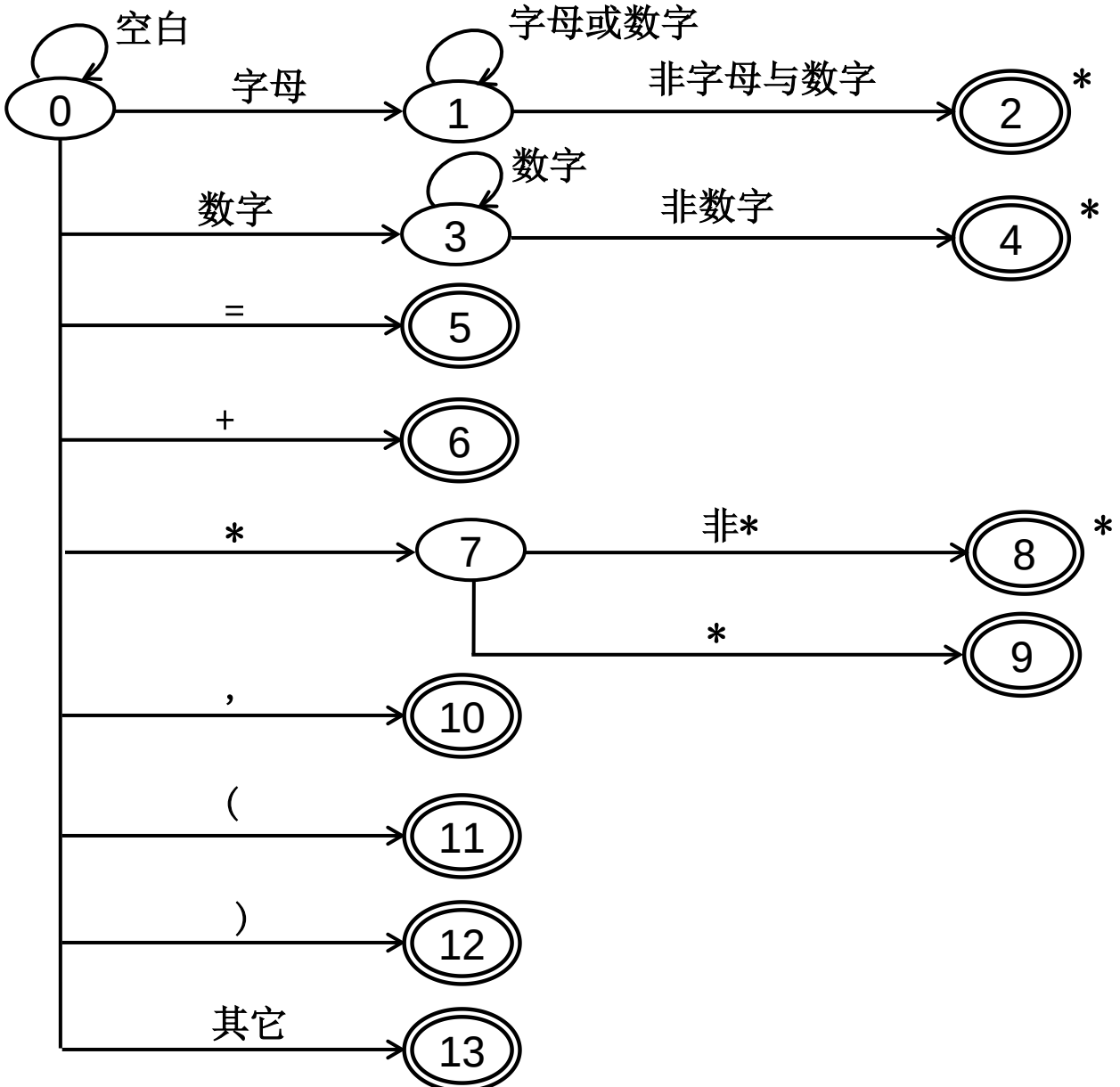


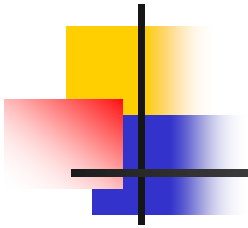
The DFAs for the Tokens of TINY

- 4. The DFA extended by adding
  - white space,
  - comments,
  - and assignment.
- The DFA considers reserved words to be the same as identifiers, and then to look up the identifiers in a table of reserved words.



状态转换图（3.2.3，P43）





Tokens of the TINY Language

DFA of the TINY Scanner

The Code to Implement the DFA

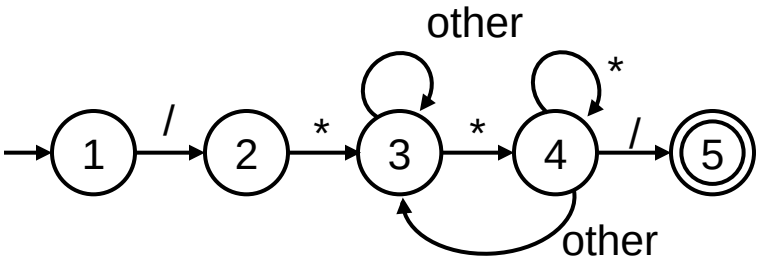
Ways to Translate a DFA into Code - II

- A better method is obtained by
  - using a variable to maintain the current state and
  - writing the transitions as a doubly nested case statement inside a loop, where
    - the first case statement tests the current state and
    - the nested second level tests the input character, given the state.

Ways to Translate a DFA into Code

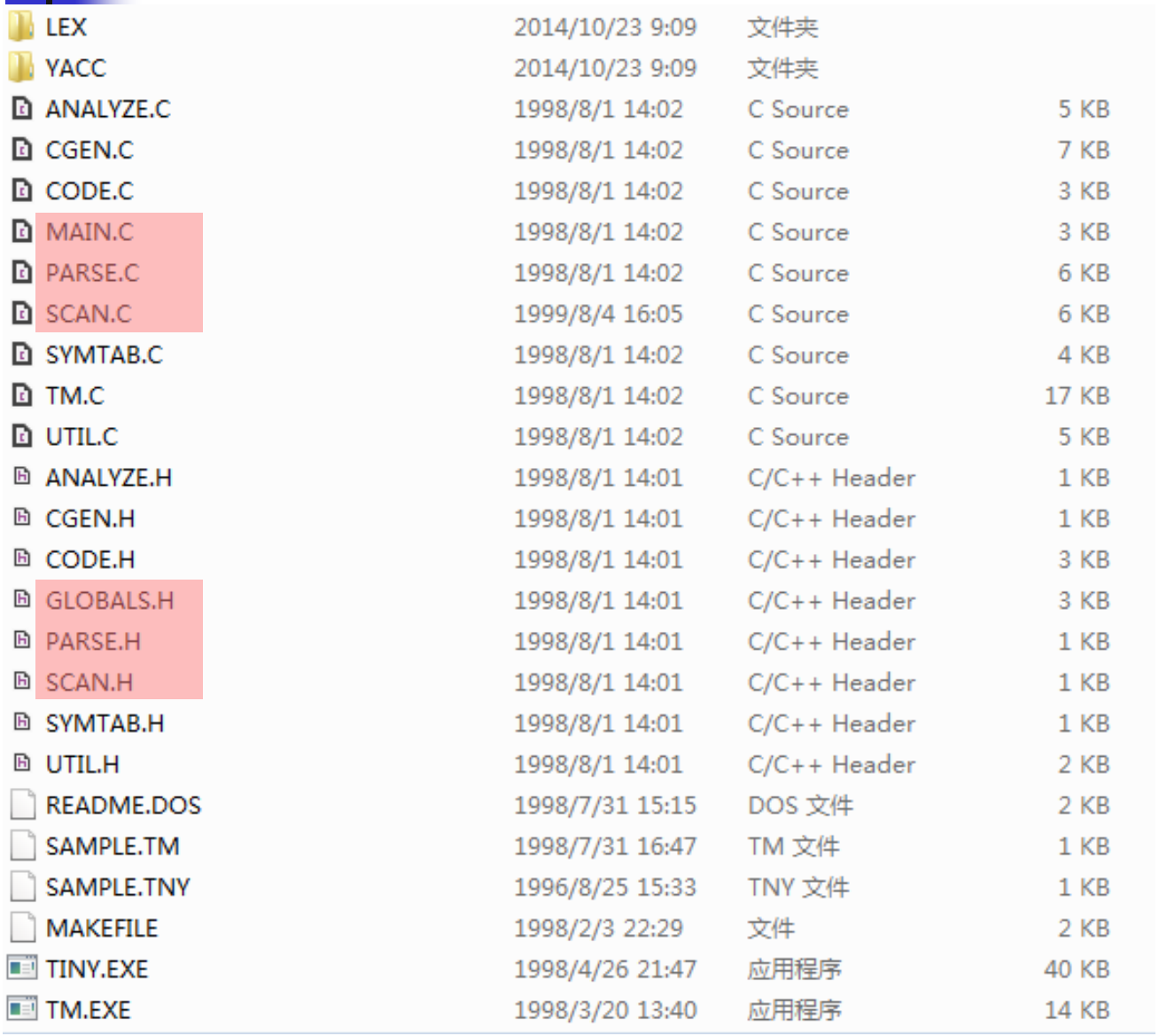
- The Code of the DFA that accepts the C comments

```
state := 1;  { start }
while state = 1, 2, 3 or 4 do
  case state of
  1: case input character of
       "/" : advance the input;
             state := 2;
       else state :=...{ error or other };
       end case;
  2: case input character of
       "*": advance the input;
            state := 3;
       else state :=...{ error or other };
       end case;
  3: case input character of
       "*": advance the input;
            state := 4;
       else advance the input; {and stay in state 3};
       end case;
```




```
  4: case input character of
       "/": advance the input;
            state := 5;
       "*": advance the input;
            { and stay in state 4 }
       else advance the input;
            state := 3;
       end case;
  end case;
end while;
if state = 5 then accept else error;
```

The Code to Implement the DFA



LEX	2014/10/23 9:09	文件夹	
YACC	2014/10/23 9:09	文件夹	
ANALYZE.C	1998/8/1 14:02	C Source	5 KB
CGEN.C	1998/8/1 14:02	C Source	7 KB
CODE.C	1998/8/1 14:02	C Source	3 KB
MAIN.C	1998/8/1 14:02	C Source	3 KB
PARSE.C	1998/8/1 14:02	C Source	6 KB
SCAN.C	1999/8/4 16:05	C Source	6 KB
SYMTAB.C	1998/8/1 14:02	C Source	4 KB
TM.C	1998/8/1 14:02	C Source	17 KB
UTIL.C	1998/8/1 14:02	C Source	5 KB
ANALYZE.H	1998/8/1 14:01	C/C++ Header	1 KB
CGEN.H	1998/8/1 14:01	C/C++ Header	1 KB
CODE.H	1998/8/1 14:01	C/C++ Header	3 KB
GLOBALS.H	1998/8/1 14:01	C/C++ Header	3 KB
PARSE.H	1998/8/1 14:01	C/C++ Header	1 KB
SCAN.H	1998/8/1 14:01	C/C++ Header	1 KB
SYMTAB.H	1998/8/1 14:01	C/C++ Header	1 KB
UTIL.H	1998/8/1 14:01	C/C++ Header	2 KB
README.DOS	1998/7/31 15:15	DOS 文件	2 KB
SAMPLE.TM	1998/7/31 16:47	TM 文件	1 KB
SAMPLE.TNY	1996/8/25 15:33	TNY 文件	1 KB
MAKEFILE	1998/2/3 22:29	文件	2 KB
TINY.EXE	1998/4/26 21:47	应用程序	40 KB
TM.EXE	1998/3/20 13:40	应用程序	14 KB

main.c
scan.c
parse.c

globals.h
scan.h
parse.h

main( )

```
main( int argc, char * argv[] )
{
  ...
  source = fopen(pgm,"r");
  ...
  listing = stdout; /* send listing to screen */
  ...
#if NO_PARSE
  while (getToken()!=ENDFILE);
#else
  syntaxTree = parse();
   ...
  }
```

getToken()

Scan.h and Scan.c

- The principal procedure: getToken
  - consumes input characters and returns the next token recognized according to the DFA,
  - uses the doubly nested case analysis,
  - a large case list based on the state, within which are individual case lists based on the current input character.

Data Type

- The states of the scanner are defined as an enumerated type in scan.c .

```
typedef enum
{
    START,INASSIGN,INCOMMENT,INNUM,INID,DONE
}  StateType;
```

- The tokens are defined as an enumerated type in globals.h .

```
typedef  enum
{
    ENDFILE,ERROR,
    IF,THEN,ELSE,END,REPEAT,UNTIL,READ,WRITE,
    ID,NUM,
    ASSIGN,EQ,LT,PLUS,MINUS,TIMES,OVER,LPAREN,RPAREN,SEMI
}  TokenType;
```

String value of the token

- The string value of the token is placed in the variable tokenString,
  - which is declared with a fixed length of 41, so that identifiers cannot be more than 40 characters (plus the ending null character).

```
#define MAXTOKENLEN  40
char  tokenString[MAXTOKENLEN+1];
```

Global variables

- The scanner makes use of three global variables:
  - the file variables source and listing,
  - and the integer variable lineno,
  - which are declared in **globals.h**, and allocated and initialized in **main.c**.

Reserved Word

```
#define MAXRESERVED 8
/* lookup table of reserved words */
static struct
{ char* str;
   TokenType tok;
} reservedWords[MAXRESERVED]
= {
      {"if",IF},
      {"then",THEN},
      {"else",ELSE},
      {"end",END},
      {"repeat",REPEAT},
      {"until",UNTIL},
      {"read",READ},
      {"write",WRITE}
   };
```

```
/* lookup an identifier to see if it is a
reserved word */
/* uses linear search */
static  TokenType   reservedLookup(char * s)
{ int i;
  for (i=0;i<MAXRESERVED;i++)
      if (!strcmp(s,reservedWords[i].str))
          return reservedWords[i].tok;
  return ID;
}
```

getNextChar( )

- Character input to the scanner is provided by the getNextChar function.

```
#define BUFLEN 256
static char lineBuf[BUFLEN]; /* holds the current line */

static int  getNextChar(void)
{   if (!(linepos < bufsize))
    {   lineno++;
        if (fgets(lineBuf,BUFLEN-1,source))
        {   …
            return lineBuf[linepos++];
        }
       else    {   …   }
    }
    else return lineBuf[linepos++];
}
```

ungetNextChar( )

- ungetNextChar procedure backs up one character in the input buffer.

```
static void ungetNextChar(void)
{
   if (!EOF_flag)  linepos-- ;
}
```

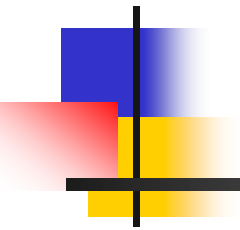
Sample program in the TINY language

```
{   Sample program
    in TINY language  -
    computes  factorial
}
read x; { input an integer }
if  0 < x then  { don't compute if x <= 0 }
    fact := 1 ;
    repeat
        fact := fact * x;
        x := x – 1
    until x = 0;
    write fact { output factorial of x }
end
```

Output of scanner given the TINY program

```
TINY COMPILATION: sample.tny
1: { Sample program
2:   in TINY language –
3:   computes factorial
4: }
5: read x; { input an integer }
5: reserved word: read
5: ID, name= x
5: ;
6: if 0<x then { don't compute if x<=0}
6: reserved word: if
6: NUM, val= 0
6: <
6: ID, name= x
6: reserved word: then
7: fact := 1;
7: ID, name= fact
7: :=
7: NUM, val= 1
7: ;
8: repeat
8: reserved word: repeat
9: fact := fact * x;
9: ID, name= fact
9: :=
9: ID, name= fact
9: *
9: ID, name= x
9: ;
10:x := x - 1
10: ID, name= x
10: :=
10: ID, name=x
10: -
10: NUM, val = 1
11:until x = 0;
11: reserved word: until
11: ID, name= x
11: =
11: NUM, val= 0
11: ;
12: write fact { output factorial of x }
12: reserved words: write
12: ID, name= fact
13:end
13: reserved word: end
14: EOF
```

When **TraceScan** and **EchoSource** are set.



Syntax of the TINY Language

Grammar of the TINY language in BNF

program → *stmt-sequence*

stmt-sequence → *stmt-sequence* **;** *statement* | *statement*

statement → *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*

if-stmt → **if** *exp* **then** *stmt-sequence* **end**

| **if** *exp* **then** *stmt-sequence* **else** *stmt-sequence* **end**

repeat-stmt → **repeat** *stmt-sequence* **until** *exp*

assign-stmt → ***identifier*** **:=** *exp*

read-stmt → **read** ***identifier***

write-stmt → **write** *exp*

exp → *simple-exp comparison-op simple-exp* | *simple-exp*

comparison-op → **<** | **=**

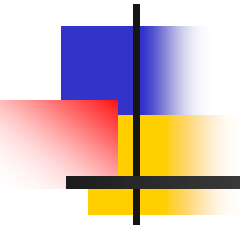
simple-exp → *simple-exp addop term* | *term*

addop → **+ | -**

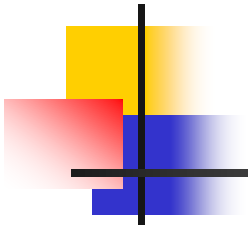
term → *term mulop factor* | *factor*

mulop → ***** | */***

factor → **(***exp***)** |***number*** |***identifier***



Introduction to C-



Lexical Conventions of C-

Syntax of C-

Sample Programs of C-

1. 关键字

   else if int return void while

2. 专用符号

   + - * / < <= > >= == != = ; , ( ) [ ] { } /* */

3. 标识符ID和整数NUM，通过下列正则表达式定义：

   ID=letter letter*

   NUM=digit digit*

   letter = a|…|z|A|…|Z

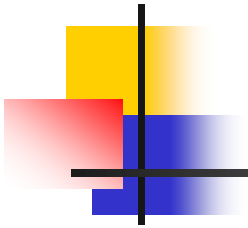
   digit = 0|…|9

4. 注释用/*…*/表示，可以超过一行。注释不能嵌套。

Lexical Conventions of C-

Syntax of C-

Sample Programs of C-



1. program → declaration-list
2. declaration-list → declaration-list declaration | declaration
3. declaration → var-declaration | fun-declaration
4. var-declaration → type-specifier **ID ;** | type-specifier **ID [NUM];**
5. type-specifier → **int** | **void**
6. fun-declaration → type-specifier **ID (**params**)** compound-stmt
7. params → param-list | **void**
8. param-list→ param-list , param | param
9. param → type-specifier **ID** | type-specifier **ID [ ]**
10. compound-stmt → **{** local-declarations statement-list **}**
11. local-declarations → local-declarations var-declaration | empty
12. statement-list → statement-list statement | empty
13. statement → expression-stmt | compound-stmt| selection-stmt
    | iteration-stmt | return-stmt
14. expression-stmt → expression **;** | **;**

15. selection-stmt → **if (** expression **)** statement
    | **if (** expression **)** statement else statement
16. iteration-stmt → **while (** expression **)** statement
17. return-stmt → **return ;** | **return** expression **;**
18. expression → var = expression | simple-expression
19. var → **ID** | **ID [** expression **]**
20. simple-expression → additive-expression relop additive- expression | additive-expression
21. relop → **<=** | **<** | **>** | **>=** | **==** | **!=**
22. additive-expression → additive-expression addop term | term
23. addop → **+** | **-**
24. term → term mulop factor | factor
25. mulop → ***** | */*
26. factor → **(** expression **)** | var | call | **NUM**
27. call → **ID (** args **)**
28. args → arg-list | empty
29. arg-list → arg-list **,** expression | expression

Lexical Conventions of C-

Syntax of C-

Sample Programs of C-

```
int gcd (int u, int v)      /* calculate the gcd of u and v */
{
      if (v == 0) return u;
      else return gcd(v,u-u/v*v);      /* v,u-u/v*v is equals to u mod v*/
}
void main()
{
      int x;   int y;   int temp;
      x = input();
      y = input();
      if  (x < y)
      {
            temp = x;
            x = y;
            y = temp;
      }
      output(gcd(x,y));
}
```