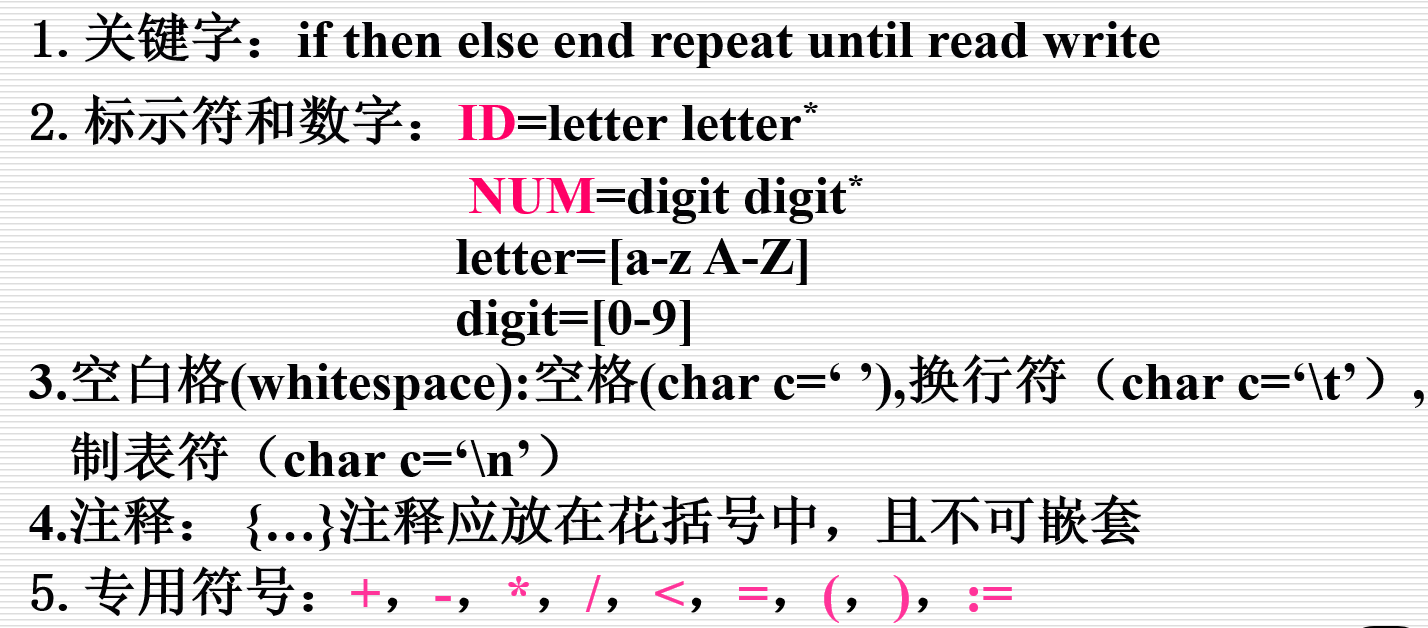
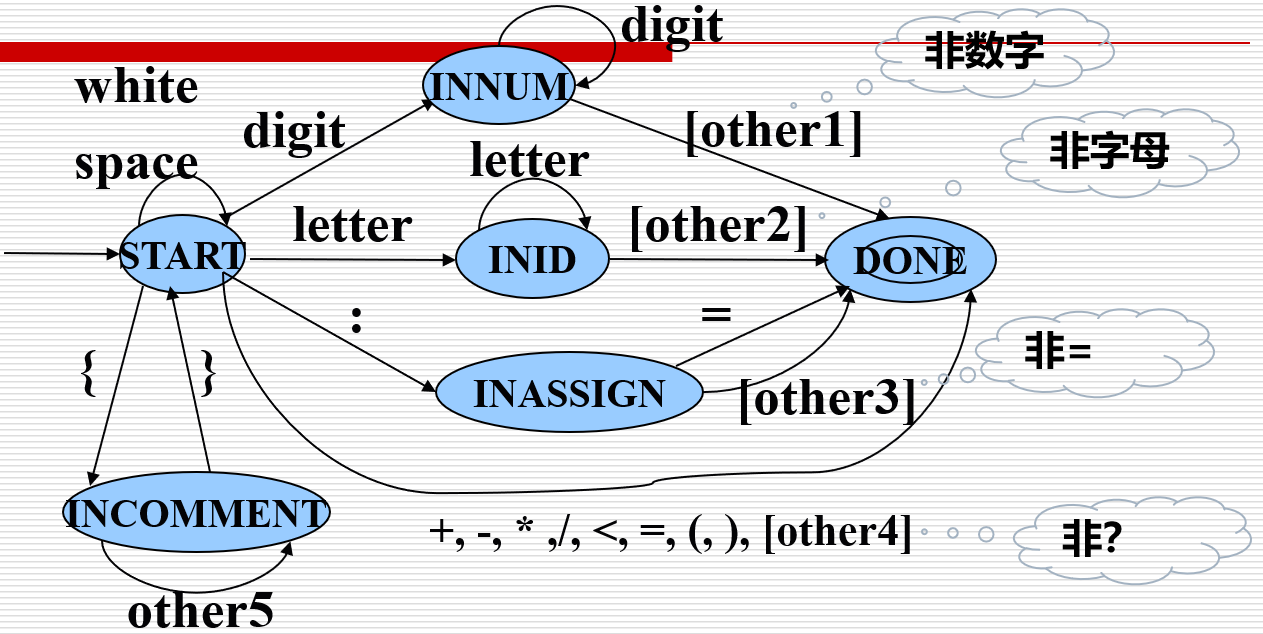
# Tiny词法分析程序撰写示例

## Tiny语言的词法



## Tiny语言的词法规则对应的确定性有穷自动机模型



## 基于自动机模型的Tiny语言的词法程序如下：

### Scan.h

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* File: scan.h \*/

/\* The scanner interface for the TINY compiler \*/

/\* Compiler Construction: Principles and Practice \*/

/\* Kenneth C. Louden \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef \_SCAN\_H\_

#define \_SCAN\_H\_

/\* MAXTOKENLEN is the maximum size of a token \*/

#define MAXTOKENLEN 40

/\* tokenString array stores the lexeme of each token \*/

extern char tokenString[MAXTOKENLEN+1];

/\* function getToken returns the

\* next token in source file

\*/

TokenType getToken(void);

#endif

### Scan.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* File: scan.c \*/

/\* The scanner implementation for the TINY compiler \*/

/\* Compiler Construction: Principles and Practice \*/

/\* Kenneth C. Louden \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "globals.h"

#include "util.h"

#include "scan.h"

/\* states in scanner DFA \*/

typedef enum

{ START,INASSIGN,INCOMMENT,INNUM,INID,DONE }

StateType;

/\* lexeme of identifier or reserved word \*/

char tokenString[MAXTOKENLEN+1];

/\* BUFLEN = length of the input buffer for

source code lines \*/

#define BUFLEN 256

static char lineBuf[BUFLEN]; /\* holds the current line \*/

static int linepos = 0; /\* current position in LineBuf \*/

static int bufsize = 0; /\* current size of buffer string \*/

static int EOF\_flag = FALSE; /\* corrects ungetNextChar behavior on EOF \*/

/\* getNextChar fetches the next non-blank character

from lineBuf, reading in a new line if lineBuf is

exhausted \*/

static int getNextChar(void)

{ if (!(linepos < bufsize))

{ lineno++;

if (fgets(lineBuf,BUFLEN-1,source))

{ if (EchoSource) fprintf(listing,"%4d: %s",lineno,lineBuf);

bufsize = strlen(lineBuf);

linepos = 0;

return lineBuf[linepos++];

}

else

{ EOF\_flag = TRUE;

return EOF;

}

}

else return lineBuf[linepos++];

}

/\* ungetNextChar backtracks one character

in lineBuf \*/

static void ungetNextChar(void)

{ if (!EOF\_flag) linepos-- ;}

/\* 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;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* the primary function of the scanner \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* function getToken returns the

\* next token in source file

\*/

TokenType getToken(void)

{ /\* index for storing into tokenString \*/

int tokenStringIndex = 0;

/\* holds current token to be returned \*/

TokenType currentToken;

/\* current state - always begins at START \*/

StateType state = START;

/\* flag to indicate save to tokenString \*/

int save;

while (state != DONE)

{ int c = getNextChar();

save = TRUE;

switch (state)

{ case START:

if (isdigit(c))

state = INNUM;

else if (isalpha(c))

state = INID;

else if (c == ':')

state = INASSIGN;

else if ((c == ' ') || (c == '\t') || (c == '\n'))

save = FALSE;

else if (c == '{')

{ save = FALSE;

state = INCOMMENT;

}

else

{ state = DONE;

switch (c)

{ case EOF:

save = FALSE;

currentToken = ENDFILE;

break;

case '=':

currentToken = EQ;

break;

case '<':

currentToken = LT;

break;

case '+':

currentToken = PLUS;

break;

case '-':

currentToken = MINUS;

break;

case '\*':

currentToken = TIMES;

break;

case '/':

currentToken = OVER;

break;

case '(':

currentToken = LPAREN;

break;

case ')':

currentToken = RPAREN;

break;

case ';':

currentToken = SEMI;

break;

default:

currentToken = ERROR;

break;

}

}

break;

case INCOMMENT:

save = FALSE;

if (c == EOF)

{ state = DONE;

currentToken = ENDFILE;

}

else if (c == '}') state = START;

break;

case INASSIGN:

state = DONE;

if (c == '=')

currentToken = ASSIGN;

else

{ /\* backup in the input \*/

ungetNextChar();

save = FALSE;

currentToken = ERROR;

}

break;

case INNUM:

if (!isdigit(c))

{ /\* backup in the input \*/

ungetNextChar();

save = FALSE;

state = DONE;

currentToken = NUM;

}

break;

case INID:

if (!isalpha(c))

{ /\* backup in the input \*/

ungetNextChar();

save = FALSE;

state = DONE;

currentToken = ID;

}

break;

case DONE:

default: /\* should never happen \*/

fprintf(listing,"Scanner Bug: state= %d\n",state);

state = DONE;

currentToken = ERROR;

break;

}

if ((save) && (tokenStringIndex <= MAXTOKENLEN))

tokenString[tokenStringIndex++] = (char) c;

if (state == DONE)

{ tokenString[tokenStringIndex] = '\0';

if (currentToken == ID)

currentToken = reservedLookup(tokenString);

}

}

if (TraceScan) {

fprintf(listing,"\t%d: ",lineno);

printToken(currentToken,tokenString);

}

return currentToken;

} /\* end getToken \*/

### Global.h

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* File: globals.h \*/

/\* Global types and vars for TINY compiler \*/

/\* must come before other include files \*/

/\* Compiler Construction: Principles and Practice \*/

/\* Kenneth C. Louden \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef \_GLOBALS\_H\_

#define \_GLOBALS\_H\_

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#ifndef FALSE

#define FALSE 0

#endif

#ifndef TRUE

#define TRUE 1

#endif

/\* MAXRESERVED = the number of reserved words \*/

#define MAXRESERVED 8

typedef enum

/\* book-keeping tokens \*/

{ENDFILE,ERROR,

/\* reserved words \*/

IF,THEN,ELSE,END,REPEAT,UNTIL,READ,WRITE,

/\* multicharacter tokens \*/

ID,NUM,

/\* special symbols \*/

ASSIGN,EQ,LT,PLUS,MINUS,TIMES,OVER,LPAREN,RPAREN,SEMI

} TokenType;

extern FILE\* source; /\* source code text file \*/

extern FILE\* listing; /\* listing output text file \*/

extern FILE\* code; /\* code text file for TM simulator \*/

extern int lineno; /\* source line number for listing \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\* Syntax tree for parsing \*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typedef enum {StmtK,ExpK} NodeKind;

typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK} StmtKind;

typedef enum {OpK,ConstK,IdK} ExpKind;

/\* ExpType is used for type checking \*/

typedef enum {Void,Integer,Boolean} ExpType;

#define MAXCHILDREN 3

typedef struct treeNode

{ struct treeNode \* child[MAXCHILDREN];

struct treeNode \* sibling;

int lineno;

NodeKind nodekind;

union { StmtKind stmt; ExpKind exp;} kind;

union { TokenType op;

int val;

char \* name; } attr;

ExpType type; /\* for type checking of exps \*/

} TreeNode;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\* Flags for tracing \*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* EchoSource = TRUE causes the source program to

\* be echoed to the listing file with line numbers

\* during parsing

\*/

extern int EchoSource;

/\* TraceScan = TRUE causes token information to be

\* printed to the listing file as each token is

\* recognized by the scanner

\*/

extern int TraceScan;

/\* TraceParse = TRUE causes the syntax tree to be

\* printed to the listing file in linearized form

\* (using indents for children)

\*/

extern int TraceParse;

/\* TraceAnalyze = TRUE causes symbol table inserts

\* and lookups to be reported to the listing file

\*/

extern int TraceAnalyze;

/\* TraceCode = TRUE causes comments to be written

\* to the TM code file as code is generated

\*/

extern int TraceCode;

/\* Error = TRUE prevents further passes if an error occurs \*/

extern int Error;

#endif

### Util.h

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* File: util.h \*/

/\* Utility functions for the TINY compiler \*/

/\* Compiler Construction: Principles and Practice \*/

/\* Kenneth C. Louden \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef \_UTIL\_H\_

#define \_UTIL\_H\_

/\* Procedure printToken prints a token

\* and its lexeme to the listing file

\*/

void printToken( TokenType, const char\* );

/\* Function newStmtNode creates a new statement

\* node for syntax tree construction

\*/

TreeNode \* newStmtNode(StmtKind);

/\* Function newExpNode creates a new expression

\* node for syntax tree construction

\*/

TreeNode \* newExpNode(ExpKind);

/\* Function copyString allocates and makes a new

\* copy of an existing string

\*/

char \* copyString( char \* );

/\* procedure printTree prints a syntax tree to the

\* listing file using indentation to indicate subtrees

\*/

void printTree( TreeNode \* );

#endif