Compilation Techniques

Laboratory 3

The lexical analyzer (LA) implementation

The LA role is to group the input characters in tokens. The tokens are lexical units, indivisible from the point of view of the next compiler phases (ex: numbers, identifiers, keywords, …). In the same time all the sequences which are not relevant in the next phases (comments, spaces, new lines) are removed. A token is composed from:

- **name (type, code)** – a symbolic name (numeric constant) which allows the identification of the lexical unit. Examples: INT, ID, STR, ADD. Most of the times when we are speaking about a token, in fact we are talking about its code.
- **attributes** – associated information such as: the characters from which an identifier or string is composed, the numeric value for numeric constants, the input file line, …. If some of these attributes are mutually exclusive (the numeric value of a number will never be used simultaneously with the string necessary for an identifier), they can be grouped in a *union*, in order to occupy less memory space.

A possible structure which defines a token could be:

```
enum{ID, END, CT_INT, ASSIGN, SEMICOLON…};   // tokens codes

typedef struct _Token{
  int         code;             // code (name)
  union{
     char          *text;       // used for ID, CT_STRING (dynamically allocated)
     long int      i;           // used for CT_INT, CT_CHAR
     double        r;           // used for CT_REAL
     };
  int         line;             // the input file line
  struct _Token *next;          // link to the next token
  }Token;
```

The **next** field is needed if we use a list structure to keep the tokens. In this case we will consider the variable **tokens** as the beginning of this list and for the ease of adding new tokens we also keep another variable **lastToken** which points to the last token in list. Initially these two variables are NULL. We can implement a function which creates and add a new token in list:

```
Token      *addTk(int code)
{
  Token        *tk;
  SAFEALLOC(tk,Token)
  tk->code=code;
  tk->line=line;
  tk->next=NULL;
  if(lastToken){
     lastToken->next=tk;
     }else{
     tokens=tk;
     }
  lastToken=tk;
  return tk;
}
```

This function also needs the **line** variable which contains the current line in the input file. **SAFEALLOC** is a macro which allocates memory for a given type and exits the program if there is not enough memory:

```
#define SAFEALLOC(var,Type) if((var=(Type*)malloc(sizeof(Type)))==NULL)err("not enough memory");
```

The **err** function has a variable number of arguments and it is used in the same manner as **printf** (it allows any number of arguments and placeholders). **err** prints its arguments as an error message and exits the program:

```
void	err(const char *fmt,...)
{
  va_list	va;
  va_start(va,fmt);
  fprintf(stderr,"error: ");
  vfprintf(stderr,fmt,va);
  fputc('\n',stderr);
  va_end(va);
  exit(-1);
}
```

To print the errors which can appear at a specific position in the input file, an **err** function variant can be used, which also has as parameter a token, from which it extracts the line where the error took place:

```
void	tkerr(const Token *tk,const char *fmt,...)
{
  va_list	va;
  va_start(va,fmt);
  fprintf(stderr,"error in line %d: ",tk->line);
  vfprintf(stderr,fmt,va);
  fputc('\n',stderr);
  va_end(va);
  exit(-1);
}
```

Because we want each compiler module to be as much as possible independent from the other modules (decoupling), in the lexical analysis phase we will create the complete tokens list from the input file. Afterwards, in the syntactic analysis phase, we will use only this list.

The most important part of the LA is implemented in a function „int **getNextToken**()" which at every call adds in the tokens list the next token from the input string and returns its code. This function will be called until it will return the code of the token END, which marks the end of the file. In the end all resulted tokens will be in the **tokens** list.

The getNextToken() implementation with explicit states

It is an easy to code implementation, even if it implies more code. It starts with the transition diagram (TD) which has all the lexical definitions. Here is the algorithm:

- We consider a variable which contains the current state, initially 0 (the initial state)
- In an infinite loop, at each iteration:
  - For the current state we test the current character according with all the possible transitions from that state
  - If a transition which consumes that character is found, this is consumed and the new state becomes the current one, according to the selected transition
  - If no transition which consumes that character was found, but the current state has an "else" transition, the destination state of this transition becomes the current state, without consuming the input character
  - If the current state is a final state a new token is created, its attributes are set and it is returned

If we consider the following AtomC definitions:

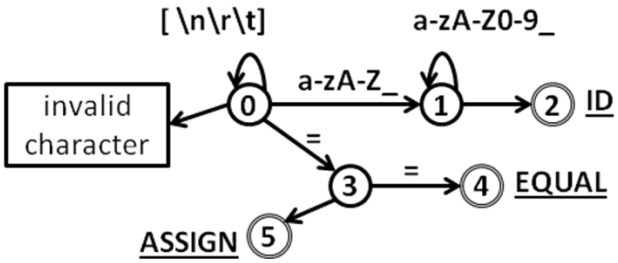
ID: [a-zA-Z_] [a-zA-Z0-9_]* ;

SPACE: [\n\r\t] ;

ASSIGN: '=' ;

EQUAL: '==' ;

A possible TD is:



If we consider that the input file is already read in memory, we appended a string terminator (0) after its characters and the variable **pCrtCh** is a pointer to the current character, a **getNextToken** implementation could be:

```
int     getNextToken()
{
int             state=0,nCh;
char            ch;
const char      *pStartCh;
Token           *tk;

while(1){    // infinite loop
    ch=*pCrtCh;
    switch(state){
        case 0:             // transitions test for state 0
            if(isalpha(ch)||ch=='_'){
                pStartCh=pCrtCh;        // memorizes the beginning of the ID
                pCrtCh++;               // consume the character
                state=1;                // set the new state
                }
            else if(ch=='='){
                pCrtCh++;
                state=3;
                }
            else if(ch==' '||ch=='\r'||ch=='\t'){
                pCrtCh++;       // consume the character and remains in state 0
                }
            else if(ch=='\n'){  // handled separately in order to update the current line
                line++;
                pCrtCh++;
                }
            else if(ch==0){     // the end of the input string
                addTk(END);
                return END;
                }
            else tkerr(addTk(END),"invalid character");
            break;
        case 1:
            if(isalnum(ch)||ch=='_')pCrtCh++;
                else state=2;
```

```
            break;
        case 2:
            nCh=pCrtCh-pStartCh; // the id length
            // keywords tests
            if(nCh==5&&!memcmp(pStartCh,"break",5))tk=addTk(BREAK);
            else if(nCh==4&&!memcmp(pStartCh,"char",4))tk=addTk(CHAR);
            // … all keywords …
            else{                       // if no keyword, then it is an ID
                tk=addTk(ID);
                tk->text=createString(pStartCh,pCrtCh);
                }
        return tk->code;
        case 3:
            if(ch=='='){
                pCrtCh++;
                state=4;
                }
            else state=5;
            break;
        case 4:
            addTk(EQUAL);
            return EQUAL;
        case 5:
            addTk(ASSIGN);
            return ASSIGN;
        }
    }
}
```

It can be seen that this implementation closely follows the transitions diagram and so it is quite easy to implement. For IDs it was shown how the keywords can be tested. Technically speaking the keywords have the same definition as an ID, but they have a special meaning. Of course in the more advanced implementations associative vectors can be used for these tests or the transition diagram can also contain all keywords.

The function **createString** creates a new copy, dynamically allocated, for the string which begins with **pStartCh** and ends with **pCrtCh** (excluding this character). This function is used to set the **text** attribute of the IDs. To set the numeric attributes of CT_INT and CT_REAL, functions which convert strings to numeric values can be used (**strtol**, **atof**) or these conversions can be made directly in code.

Observation: there are other possibilities to implement the function **getNextToken**, for example by using implicit states. In this implementation there is no need for TD but the algorithm tries to implement directly the regular definitions logic. In many case this implementation requires less code and it is more readable, but it is a bit more complex.

To test the LA, it will be a function which shows all the tokens (**showTokens**). This function will iterate the list **tokens** and it will print the numeric codes of all tokens. If some tokens also have attributes (ID, CT_INT, …) , these will also be printed. An example for the above enum and the program „speed=70;”: 0:speed 3 2:70 4 1.

Ideas for testing and debugging:

- For different tokens there will be tests for different correct and erroneous combinations, in order to see if the results are the expected ones. For example for CT_INT these inputs can be tested:
    - correct: 0, 25, 017, 9, 0xAFd9
    - erroneous: 08, 0xG

- If the tokens are not correctly recognized and from the TD of from the implementation the problem cannot be solved, in **getNextToken** a **printf** can be put before the states *switch.* This will print both the state and the current character at the beginning of each transition. In this way the path taken inside TD can be clearly visualized.

In the end of the compiler execution, all dynamically allocated memory will be freed. This can be done with a function **done**() which frees all the used data structures.

Application 3.1: *Write the getNextToken() code corresponding to the CT_INT and COMMENT definitions. For CT_INT its attribute (its numeric value) will also be obtained.*

Homework: implement the full lexical analyzer for the AtomC language and call it with an input file to obtain all the tokens from that file. The tokens codes will be shown and where it is the case their attributes will also be shown.