CS 280 Programming Language Concepts

Spring 2025

Project Assignment 1

Building a Lexical Analyzer for

A Simple Ada-Like (SADAL) Language

Project Assignment 1

- Objectives
  - building a lexical analyzer for a small programming language, called Simple Ada-Like (SADAL) Language
  - Writing a C++ program to test the lexical analyzer.

- Notes:
  - Read the assignment carefully to understand it.
  - Make a list of the lexical rules of the language and the assigned tokens.
  - Understand the functionality of the testing program, the required information to be collected and printed out.

Project Assignment 1 Statement

- In this programming assignment, you will be building a lexical analyzer for small programming language, called Simple Ada-Like (SADAL), and a program to test it. This assignment will be followed by two other assignments to build a parser and an interpreter to the language. Although, we are not concerned about the syntax definitions of the language in this assignment, we intend to introduce it ahead of Programming Assignment 2 in order to determine the language terminals: reserved words, constants, identifier, and operators. The syntax definitions of the SADAL language are given below using EBNF notations. However, the details of the meanings (i.e. semantics) of the language constructs will be given later on.

Simple Ada-Like (SADAL) Language Definition

1. Prog ::= PROCEDURE ProcName IS ProcBody
2. ProcBody ::= DeclPart BEGIN StmtList END ProcName ;
3. ProcName ::= IDENT
4. DeclPart ::= DeclStmt { DeclStmt }
5. DeclStmt ::= IDENT {, IDENT } : [CONSTANT] Type [(Range)] [:= Expr] ;
6. Type ::= INTEGER | FLOAT | BOOLEAN | STRING | CHARACTER
7. StmtList ::= Stmt { Stmt }
8. Stmt ::= AssignStmt | PrintStmts | GetStmt | IfStmt
9. PrintStmts ::= (PutLine | Put) (Expr) ;
10. GetStmt := Get (Var) ;
11. IfStmt ::= IF Expr THEN StmtList { ELSIF Expr THEN StmtList } [ELSE StmtList] END IF ;
12. AssignStmt ::= Var := Expr ;

Simple Ada-Like (SADAL) Language Definition

```
13.Expr ::= Relation {(AND | OR) Relation }
14.Relation ::= SimpleExpr [  ( = | /= | < | <= | > | >= )
   SimpleExpr ]
15.SimpleExpr ::= STerm {  ( + | - | & ) STerm }
16.STerm ::= [ ( + | - ) ] Term
17.Term  ::= Factor { ( * | / | MOD ) Factor }
18.Factor ::= Primary [** Primary ] | NOT Primary
19.Primary ::= Name | ICONST | FCONST | SCONST | BCONST |
   CCONST | (Expr)
20.Name ::= IDENT [ ( Range ) ]
21.Range ::= SimpleExpr [. . SimpleExpr ]
```

Tokens of the SADAL Language

- The language identifiers are referred to by the *IDENT* terminal, which is defined as a sequence of characters that starts by a letter, and can be followed by zero or more letters, digits, or underscore '_' characters. **Note that all identifiers are case insensitive.** *IDENT* regular expression is defined as:

```
IDENT ::= ( Letter) ( _ ? ( Letter | Digit ) )*
Letter ::= [a-z A-Z]
Digit ::= [0-9]
```

- Integer constant is referred to by *ICONST* terminal, which is defined as a sequence of one or more digits followed by by an optional exponent. It is defined as:

```
ICONST ::= Digit+ Exponent?
Exponent ::= E ( +? | - ) Digit+
```

Tokens of the SADAL Language

- Floating-point constant is a real number referred to by *FCONST* terminal. It is defined as a sequence of one or more digits, forming the integer part, a decimal point, and followed by a one or more digits, forming the fractional part, and an optional exponent. The regular expression definition for FCONST is as follows:

```
FCONST ::= Digit+ \. Digit+ Exponent?
```

  - For example, the values 5.25, 1.75E-2, 0.75, and 1.0 are valid numeric values, while the values 5., .25, 5.7.4 are not. Note that "5." is recognized as an integer followed by a DOT.
- A string literal is referred to by *SCONST* terminal, which is defined as a sequence of characters delimited by double quotes. For example,
  - "Hello to CS 280." are valid string literals. While, 'Hello to CS 280." Or "Hello to CS 280.' are not.

Tokens of the SADAL Language

- A character constant is referred to by *CCONST* terminal. It is a character delimited by single quotes.
- A Boolean constant is referred to by *BCONST* terminal. It consists of the two Boolean values **`true`** and **`false`** reserved words.
- A comment is defined by all the characters on a line following the sequence of two dash characters, “`--`” till the end of line.
- White spaces are skipped. However, white spaces between tokens are used to improve readability and can be used as a one way to delimit tokens.

Tokens of the SADAL Language

- The operators of the language and their corresponding tokens are:

Operator Symbol	Token	Description
+	PLUS	Arithmetic addition or concatenation
-	MINUS	Arithmetic subtraction
*	MULT	Multiplication
/	DIV	Division
:=	ASSOP	Assignment operator
=	EQ	Equality
/=	NEQ	Not Equality
<	LTHAN	Less than operator
>	GTHAN	Greater then operator
<=	LTE	Less than or equal
>=	GTE	Greater than or equal
mod	MOD	Modulus
&	CONCAT	Concatenation
and	AND	Logic Anding
or	OR	Logic Oring
not	NOT	Complement
**	EXP	Exponentiation

Tokens of the SADAL Language

- The reserved words of the language and their corresponding tokens are:
  - **Note: Reserved words are not case sensitive.**

Reserved Words	Tokens
procedure	PROCEDURE
String	STRING
else	ELSE
if	IF
Integer	INT
Float	FLOAT
Character	CHAR
Put	Put
PutLine	PUTLN
Get	GET
Boolean	BOOL
true	TRUE
false	FALSE
elsif	ELSIF
is	IS
end	END
begin	BEGIN
then	THEN
constant	CONST
mod	MOD
and	AND
Or	OR
not	NOT

Tokens of the SADAL Language

- The delimiters of the language are:

Character	Token	Description
,	COMMA	Comma
;	SEMICOL	Semi-colon
(	LPAREN	Left Parenthesis
)	RPAREN	Right parenthesis
:	COLON	Colon
.	DOT	Dot

- An error will be denoted by the ERR token.
- End of file will be denoted by the DONE token.
- White spaces are skipped.

Lexical Analyzer Implementation

- You will write a lexical analyzer function, called `getNextToken` having the following signature:

```
LexItem getNextToken (istream& in, int& linenumber);
```

  - First argument is a reference to an istream object that the function should read from (input file).
  - Second reference is an integer that contains the current line number of the line read from the input file.
  - `getNextToken` returns a `LexItem` object. A `LexItem` is a class that contains a token, a string for the lexeme, and the line number as data members.

Implementation Issues

- Questions to be considered:
  - What patterns need to be recognized?
  - Is there a different approach for multi-character tokens and single character tokens?
  - What are the states?
  - How do you need to represent states?
- The lexical rules represent the patterns your lexical analyzer must recognize
  - You should understand the patterns and build a DFA representing all of the patterns
    - This will tell you what states you need in your implementation
  - Assignment requires writing code to implement the DFAs
    - Write pseudocode for the function.
    - Implement one state at a time.

Implementation Issues

- Token Types:
  - Single character tokens, such as +, -, *, /, =, <, >, . (dot), (,):
    - These are easy to recognize.
  - Two or more characters tokens such as "**", ">=" and "/="
    - First character in "**" is an MULT token, the second character '*' appended to the first forms the EXP token.
  - For multi-character tokens, such as IDENT, ICONST, FCONST, and SCONST, create a state for each type of token, for example INID for recognizing identifiers (IDENT), ININT for recognizing integer constants (ICONST), and INSTRING for recognizing string literals (SCONST) .
    - Transition from one state to another state is possible. For example from the state of recognizing ICONST to the state of recognizing FCONST.

Implementation Issues

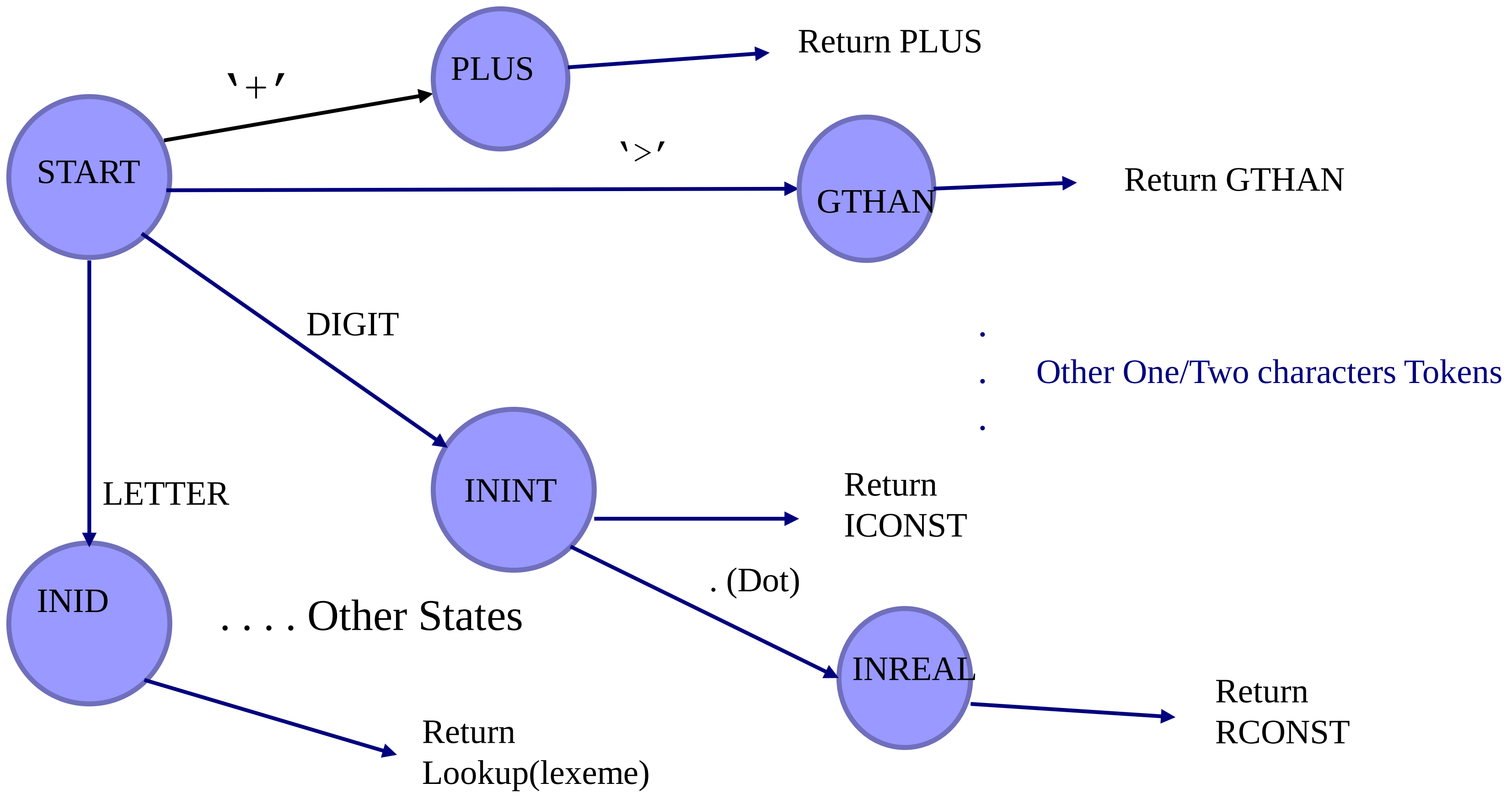
- Keywords' tokens such as *put*, *if*, *elsif*, and *else* are treated as identifiers
  - Each identifier is checked against a directory of keywords mapped to tokens. If the keyword identifier is found, then its reserved word token is returned, otherwise an identifier token (IDENT) is returned.

Implementation Issues (cont'd)

- Possible States for the types of tokens:
  - START, INID, INSTRING, ININT . . .
  - This means, starting from the START state, I have seen a character to make a transition into a state where you will be collecting characters for an identifier, a string constant, an integer constant, etc.
- Each state has different rules *inside* that state (relevant to the DFA representing that token definition). For example,
  - INID state has rules for collecting zero or more letters, digits, or underscores..
  - INSTRING state has a rule for collecting all characters following a single or double quote for a string (note, newline in string is an error, and also note there are no escape characters inside the string) till a terminating quote is found. A string must be terminated by the same type of quote that is used for opening that string literal.

Implementation Issues (cont'd)

- Combined DFAs Diagrams



Implementation Issues (cont'd)

- Use an enumerated type to define the states such as:

```
enum TokState {START, INID, INSTRING, ININT, . . .}
lexstate = START;
```

- `getNextToken` outline
  - Set initial state to START.
  - Loop, reading a character at a time from the input file till EOF is reached or an ERR is found. For each read character,
    - Select a block of code to execute based on the current state
      - Might need to change state based on a character (ex: a letter in the START state indicates the beginning of an identifier, so change the current state to INID).
    - Return when a token is recognized.
  - If the end of file is found, return a DONE token; else, return an ERR token with a meaningful message.

Implementation Issues (cont'd)

Pseudocode

```
if( lexstate == START ) {
// START code
  . . .
}
else if( lexstate == INID ){
 // INID code
. . .
}
else if (. . .)
 //other states follow
 . . .
```

```
switch( lexstate ) {
case START: // START code
      . . .
      break;
case INID: // INIT code
      . . .
      break;
//other states follow
. . .
}
```

Implementation Issues (cont'd)

- A header file, "lex.h", is provided for you. You MUST use the provided header file. You may NOT change it. "lex.h" includes the following
  - Definitions of all the possible token types
  - Class definition of `LexItem`
  - Some function prototypes

```
extern ostream& operator<<(ostream& out, const LexItem& tok);
extern LexItem id_or_kw(const string& lexeme, int linenum);
extern LexItem getNextToken(istream& in, int& linenum);
```

- You should implement the lexical analyzer function "lex.cpp" in one source file.
- You should implement a test main program in another source file.

```
class LexItem { //Class definition of LexItem
        Token   token;
        string  lexeme;
        int     lnum;
public:
        LexItem() {
                token = ERR;
                lnum = -1;
        }
        LexItem(Token token, string lexeme, int line) {
                this->token = token;
                this->lexeme = lexeme;
                this->lnum = line;
        }

bool operator==(const Token token) const { return this->token ==
token; }
bool operator!=(const Token token) const { return this->token !=
token; }

Token   GetToken() const { return token; }
string  GetLexeme() const { return lexeme; }
int     GetLinenum() const { return lnum; }

};
```

constructors

overloaded operators

getter methods

```
//a segment of the getNextToken code
LexItem getNextToken(istream& in, int& linenum){
  enum TokState { START, INID, ININT, . . .} lexstate = START;
  string lexeme;
  char ch;
  while(in.get(ch)) {
         switch( lexstate ) {
         case START:
               . . .
             break;
         Case ININT:
               . . .
               Break;
         //Other states will follow
         . . .
         . . .
  }
```

Implementation Issues (cont'd)

- Points to be considered
  - Your `getNextToken` function might need to look at the next character from input to decide if the token is finished or not.
    - Method 1: use the `peek()` method, to examine the next character, and only read it if it belongs to the token.
    - Method 2: if you read a character that does not belong to the token, use the `putback()` method to put it back, so that you get() it next time
  - Any error detected by the lexical analyzer should result in a `LexItem` object to be returned with the ERR token, and the lexeme value equal to the string recognized when the error was detected.
  - Note also that both ERR and DONE are unrecoverable. Once the `getNextToken` function returns a `LexItem` object for either of these tokens, you shouldn't call `getNextToken` again.

Implementation Issues (cont'd)

- Testing Program Issues
  - `main()` function that takes several command line arguments, called flags:
    - -all, -num, -str , -id, and -kw
    - filename argument must be passed to main function. Your program should open the file and read from that filename. Only one file name is allowed.
    - Read the rules for the flags and understand what does each one of them mean.
  - You need to keep a record of all the required information by creating directories for each one of them:
    - All identifiers
    - All integer, float constants
    - All character and string literals
    - All keywords

Implementation Issues (cont'd)

- Outline of the testing program
  - Process arguments: input file and flags
  - Call `getNextToken` until it returns DONE or ERR
    - Keep a record of all lexemes returned for each token type to be printed out.
  - Print out the required information according to the provided flags. See the specified format in PA 1 statement.

- **Note: Do not include the "lex.cpp" in the testing program.**

Implementation Issues (cont’d)

- Required information
  - How many lines in the input?
  - How many tokens in the input?
  - What strings and character constants are in the input?
  - What numeric constants are in the input?
  - What identifiers are in the input?
  - What keywords are in the input

- You are provided by a set of 19 test cases associated with Programming Assignment 1. Vocareum automatic grading will be based on these testing files. These are available in compressed archive “PA 1 Test Cases.zip” on Canvas assignment.

Examples

- Example 1: Real constant error
  - Input file: “realerr”

```
23.5   5.0E3
15.7E
0.758.3
0.8
```

End of
File

  - Output with –all –num flags:

```
FCONST: (23.5)
FCONST: (5.0E3)
FCONST: (15.7)
IDENT: <e>
ERR: In line 3, Error Message {0.758.}
```

Examples

- Example 2: No set flags
  - Input file: “noflags”

```
--All constants
235 "Please type the coordinates of three points"

15.25 -125 true 12e-3

4321  -0.725e+1        'Z' false
625+2.75 7.0E2
```

  - Output with no specified flags:

```
Lines: 8
Total Tokens: 16
Numerals: 9
Characters and Strings : 2
Identifiers: 0
Keywords: 0
```

Example 3: String constant error

- Input file: “idents”
- Output with –all –id flags:

```
x1, y1, r_ 2x__3, Y3, a, _b,
str1, str_2 ________ STR1 IF sTr1
x.z,y,z
//Read data
r = 5
```

```
IDENT: <x1>
COMMA
IDENT: <y1>
COMMA
IDENT: <r_>
ICONST: (2)
IDENT: <x_>
ERR: In line 1, Error Message {_}
```

Examples

- Example 4: Input file with no errors
  - Input file: “prog1”

```
procedure prog1 is
   -- { Testing all flags }

   integer x, y_1, z-, w234;
   string str := "Welcome";
   float z ;
   boolean flag := true, bing := false;

begin
    get(x);
    if x >= 1 then z:= 0.0 ;
    elsif (x = 0) then z:= -1;
    else z:=1;
    end if;
    put("Value of x= "); put(x);
    put("Value of z= ");
    putln(z);
END prog1;
```

Examples: Example 4 Cont’d

- Output with -str -id -kw -num flags:

```
Lines: 18
Total Tokens: 88
Numerals: 2
Characters and Strings : 3
Identifiers: 9
Keywords: 14
NUMERIC CONSTANTS:
0, 1
CHARACTERS AND STRINGS:
"Value of x= ", "Value of z= ", "Welcome"
IDENTIFIERS:
bing, flag, prog1, putln, str, w234, x, y_1, z
KEYWORDS:
if, else, elsif, put, get, integer, float, string, boolean,
procedure, end, is, begin, then
```