CS 280
Programming Language Concepts

Spring 2025

Lexical Analysis
Chapter 4

Topics

- Introduction
- Lexical Analysis
- The Parsing Problem
- Recursive-Descent Parsing

Introduction

- Language implementation systems must analyze source code, regardless of the specific implementation approach
- Nearly all syntax analysis is based on a formal description of the syntax of the source language (BNF)

- The syntax analysis portion of a language processor nearly always consists of two parts:
  - A low-level part called a *lexical analyzer* (mathematically, a finite automaton based on a regular grammar)
  - A high-level part called a *syntax analyzer*, or parser (mathematically, a push-down automaton based on a context-free grammar, or BNF)

Lexical Analysis

- A lexical analyzer is a pattern matcher for character strings.
- A lexical analyzer is a "front-end" for the parser
- Identifies substrings of the source program that belong together – *lexemes*
  - Lexemes match a character pattern, which is associated with a lexical category called a *token*
  - `sum` is a lexeme; its token may be `IDENT`
  - Transform sequence of characters into sequence of tokens.

Lexical Analysis (continued)

- The lexical analyzer is usually a function that is called by the parser when it needs the next token
- Three approaches to building a lexical analyzer:
  - Write a formal description (using regular grammars) of the tokens and use a software tool that constructs a table-driven lexical analyzer from such a description (e.g., lex on UNIX, or flex)
  - **Design a state diagram that describes the tokens and write a program that implements the state diagra**m
  - Design a state diagram that describes the tokens and hand-construct a table-driven implementation of the state diagram

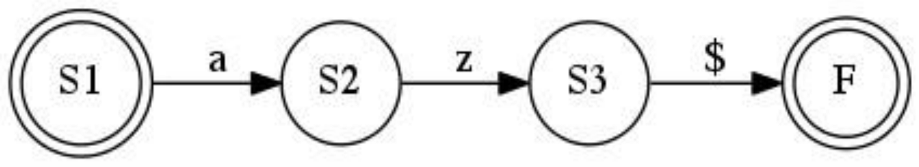
Matching Strings to Regular Expressions

RegExpr	**Meaning**
x	a character x
\x	an escaped character, e.g., \n
M | N	M or N
M N	M followed by N
M*	zero or more occurrences of M
M+	One or more occurrences of M
M?	Zero or one occurrence of M
[*characters*]	choose from the characters in []
[aeiou]	the set of vowels
[0-9]	the set of digits
. (that's a dot)	Any single character

You can parenthesize items for clarity

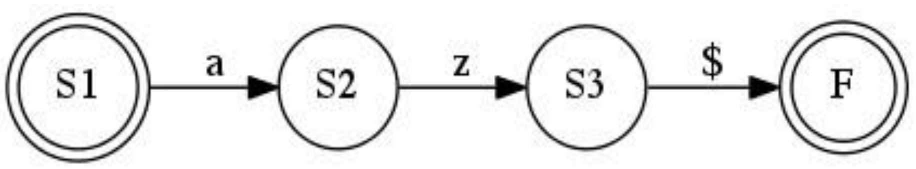
Formal mechanism: Finite State Automata (FSA)

- A finite automaton consists of a finite number of states and a finite number of transitions between states
- Each of the states represents a set of possible tokens, the start state represents the set of all possible tokens
- A subset of states is called accepting. Accepting state corresponds to a single token type.
- Each transition is labelled with a character (or character set)
  - Example: FSA to recognize the string “az”



Formal mechanism: Finite State Automata (FSA)

- The automaton starts in the start state and processes input character by character. Each character triggers a transition to a new state
- If the automaton reaches an accepting state, the token type has been determined



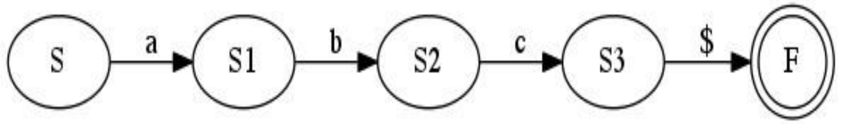
- FSA to recognize the string "az"
  - It is deterministic: there is no ambiguous transition from any one of the states
  - Error conditions are not shown; any character other than those labeled on an arc is an error

Deterministic Finite State Automata

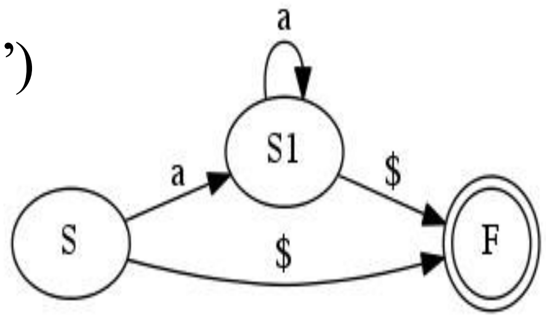
- Set of states
  - A useful representation is a graph: nodes representing states, and edges are labeled with the character that causes the transition
- Input alphabet + unique end symbol
- State transition function
  - Labelled (using alphabet) arcs in graph
- Unique start state
- A final state or an “accepting” state
- A finite state automaton is *deterministic* if for each state and each input symbol, there is at most one outgoing arc from that state for each input symbol.
  - A Deterministic FSA is also called a DFA

Examples

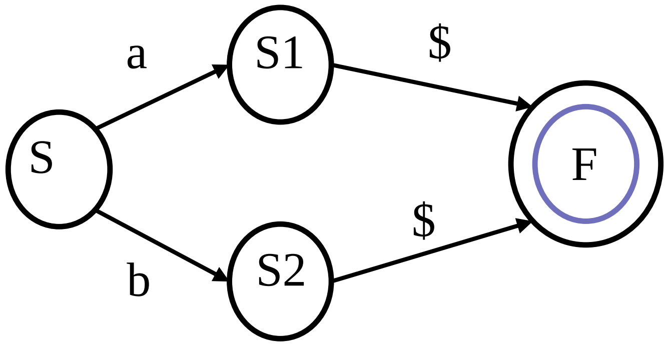
- Catenation: Sequence ("abc")
  - DFA for "a" followed by DFA for "b" followed by DFA for "c"



- Repetition (Kleene Star): Zero or more ("a*")

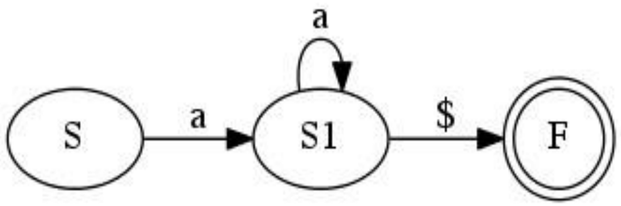


- Alternation: ("a" | "b")

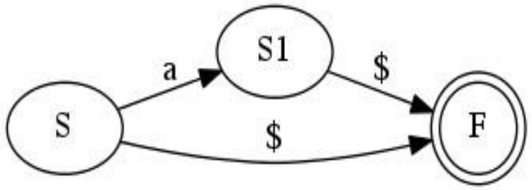


Examples

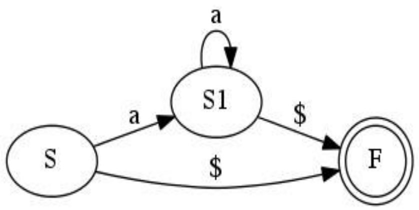
- One or more ("a+")

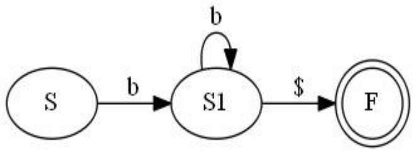


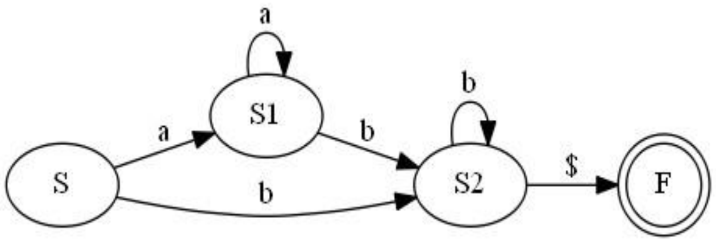
- Zero or one ("a?")



- Combinations: Zero or more a's followed by 1 or more b's (a*b+)







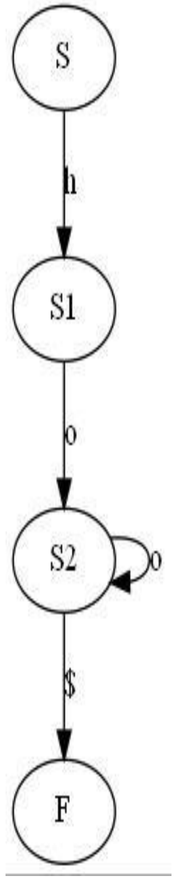
Deterministic FSA

- **Definitions**
  - A *configuration* on an FSA consists of a state and the remaining input.
  - A *move* consists of traversing the arc exiting the state that corresponds to the leftmost input symbol, thereby consuming it. If no such arc, then:
    - If no input and state is final, then accept.
    - Otherwise, error.
  - An input is *accepted* if, starting with the start state, the automaton consumes all the input and halts in a final state.

Examples

- RE: ho+
  - String: hooo

State	Input	Next input character	Next state	Remaining Input
S	hooo$	h	S1	ooo$
S1	ooo$	o	S2	oo$
S2	oo$	o	S2	o$
S2	o$	o	S2	$
S2	$	$	F	


$$\begin{tikzcd}
S \arrow[d, "h"] \\
S1 \arrow[d, "o"] \\
S2 \arrow[d, "\$"] \arrow[loop right, "o"] \\
F
\end{tikzcd}$$

State Diagram Design

- A naïve state diagram would have a transition from every state on every character in the source language - such a diagram would be very large!
- In many cases, transitions can be combined to simplify the state diagram
  - When recognizing an identifier, all uppercase and lowercase letters are equivalent
    - Use a character class that includes all letters
  - When recognizing an integer literal, all digits are equivalent - use a digit class
- Reserved words and identifiers can be recognized together (rather than having a part of the diagram for each reserved word)
  - Use a table lookup to determine whether a possible identifier is in fact a reserved word.

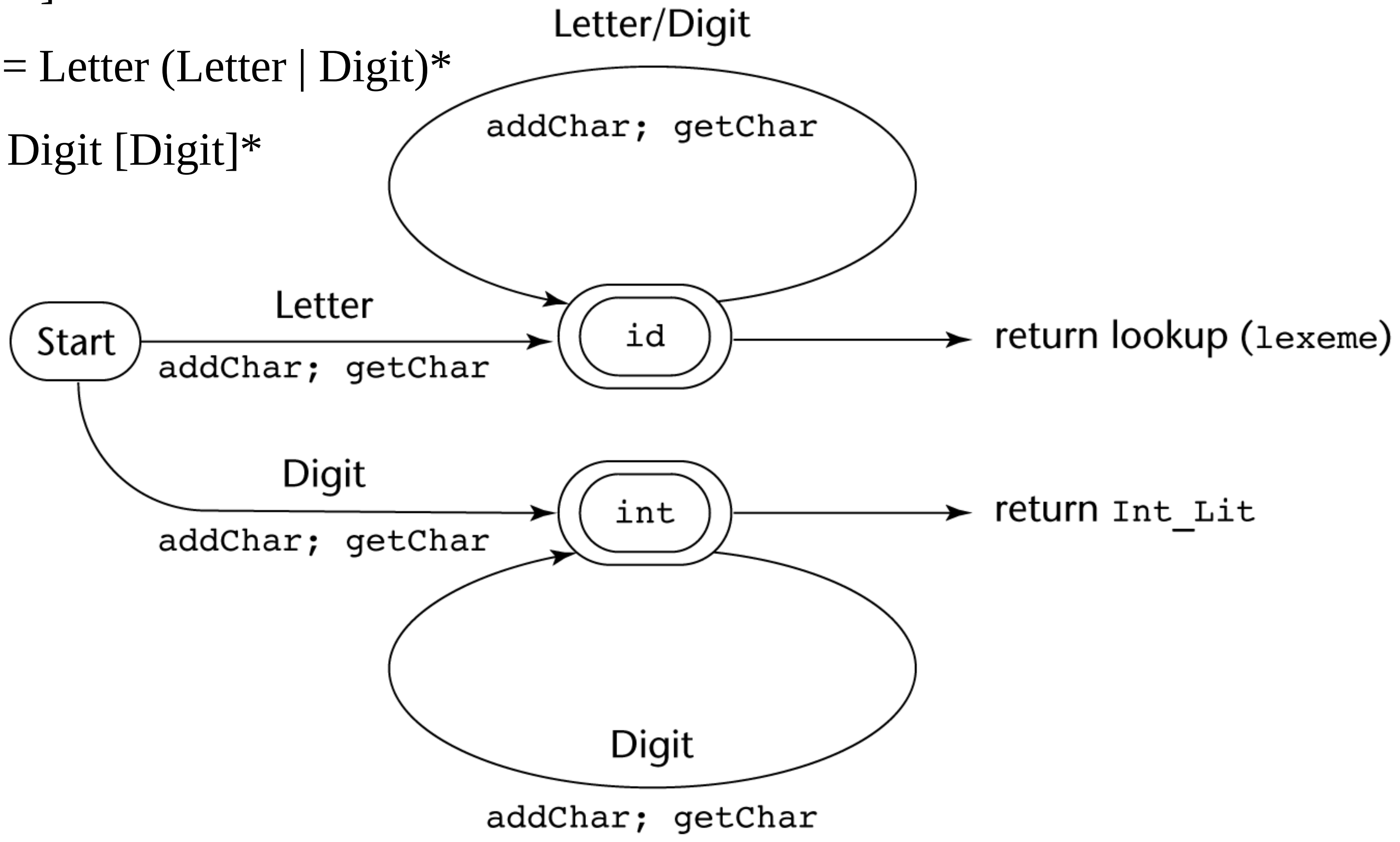
State Diagram

Letter::= [a-z_A-Z]

Digit::= [0-9]

Identifier::= Letter (Letter | Digit)*

Int_Lit ::= Digit [Digit]*



DFAs in Lexers

- Lexers are machines recognizing multiple patterns at once.
- Instead of "end of input" in a lot of places, they recognize "not part of this pattern" as being the end of the pattern. They then accept the pattern, and go back to the start state
- In an implementation, the character recognized as "not part of this pattern" might be (probably is!) the first character in the next token. The implementation would "push back" or save that character so that it's the first character processed in the next pattern

Implementations

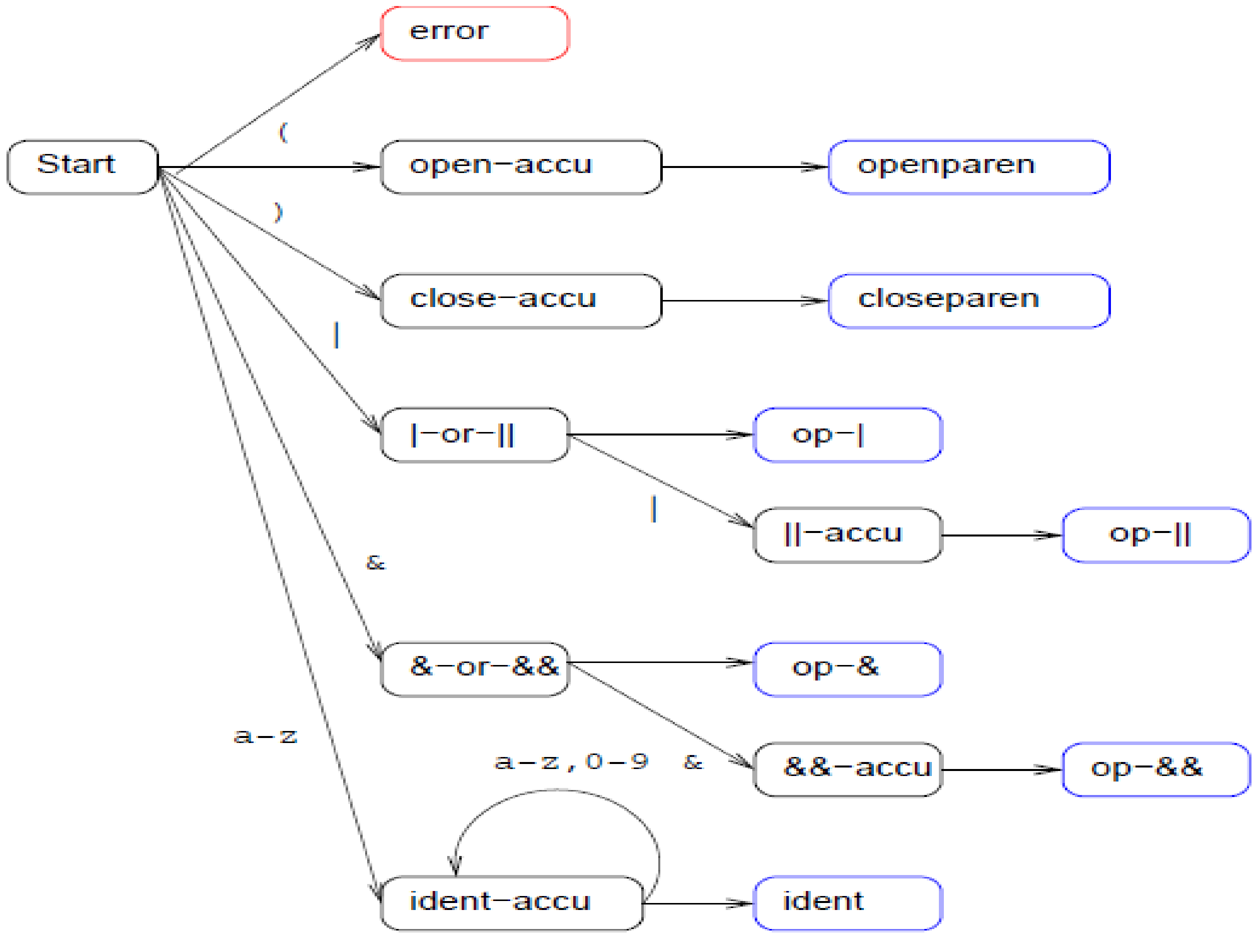
- Two implementation problems
  - Sometimes we cannot determine if a token has ended without looking at further characters
    - This is called look-ahead
    - Most existing languages can be tokenized with a one-character look-ahead
  - Some tokens may be a proper substring of another token
    - | is a substring of || in our next example
    - Examples from C++ or Java: Many reserved words like while, if, for, can also start a normal identifier.
    - > is a substring of >=
    - Solution: These conflicts are normally resolved by going to the longest match, i.e. the longest string compatible with any token determines the token type

FSA Example

- Assume C style expressions with parentheses, |, ||, &, && and identifers as token types:
  - openparen (”(”)
  - closeparen (”)”)
  - op-| (”|”)
  - op-|| (”||”)
  - op-& (”&”)
  - op-&& (”&&”)
  - ident (a--z{a–z0–9})
- Convention: Transitions without explicit label are assumed to be labelled with all characters not mentioned in another transition from the same state

FSA Example

Automaton:



Implementations

- When lexing, we can “look ahead” one character to see if it belongs in the current lexeme we are building. With C++ streams, we can use “peek()” at the next character and only “read()” if we need it.

- We can also read a character, decide that the character belongs to the *next* token and not this token, and “putback()” the character that we read.

Implementations

- **Use of a Lexer by a Parser**
  - Remember, syntactic analysis or parsing needs a stream of tokens
  - How to parse?
    - Read a token at a time and match as you read the tokens (this is what’s usually done; it’s easier)
  - The Lexer is generally a getToken() function of some kind
  - Automatic lexer generators make a function called yylex()

Implementations

- Convenient utility subprograms:
  - **`getChar`** - gets the next character of input, puts it in **`nextChar`**, determines its class and puts the class in **`charClass`**
  - **`addChar`** - puts the character from **`nextChar`** into the place the lexeme is being accumulated, **`lexeme (string or an array).`**
  - **`lookup`** - determines whether the string in **`lexeme`** is a reserved word (returns a code)

Implementations

```
/* front.c - a lexical analyzer system for simple
arithmetic expressions */
#include <stdio.h>
#include <ctype.h>
/* Global declarations */
/* Variables */
int charClass;
char lexeme [100];
char nextChar;
int lexLen;
int token;
int nextToken;
FILE *in_fp, *fopen();
```

```
/* Function declarations */
void addChar();
void getChar();
void getNonBlank();
int lex();
/* Character classes */
#define LETTER 0
#define DIGIT 1
#define UNKNOWN 99
/* Token codes */
#define INT_LIT 10
#define IDENT 11
#define ASSIGN_OP 20
#define ADD_OP 21
#define SUB_OP 22
#define MULT_OP 23
#define DIV_OP 24
#define LEFT_PAREN 25
#define RIGHT_PAREN 26
```

Implementations

```
/* main driver */
main() {
/*Open the input data file and process its contents*/
  if ((in_fp = fopen("front.in", "r")) == NULL)
      printf("ERROR - cannot open front.in \n");
  else {
      getChar();
      do {
        lex();
      } while (nextToken != EOF);
  }
}
```

Lexical Analyzer

Implementation:

→ Review `front.c` (pp. 166-170)

- Following is the output of the lexical analyzer of `front.c` when used on `(sum + 47) / total`

```
Next token is: 25 Next lexeme is (
Next token is: 11 Next lexeme is sum
Next token is: 21 Next lexeme is +
Next token is: 10 Next lexeme is 47
Next token is: 26 Next lexeme is )
Next token is: 24 Next lexeme is /
Next token is: 11 Next lexeme is total
Next token is: -1 Next lexeme is EOF
```

Syntax Analysis

Phase	*Input*	*Output*
Lexer	Sequence of characters	Sequence of tokens
Parser	Sequence of tokens	Parse tree

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 | Expr)]
21. Name ::= IDENT [(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 Boolean constant is referred to by *BCONST*, which is includes the two possible values “true” and “false”.
- 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.