**Module II**

**Programming Language Basics:**

1 The Static/Dynamic Distinction

2 Environments and States

3 Static Scope and Block Structure

4 Explicit Access Control

5 Dynamic Scope

6 Parameter Passing Mechanisms

**The Static/Dynamic Distinction:**

Among the most important issues that we face when designing a compiler for a language is

1.first issue is,what decisions can the compiler make about a program.

If a language uses a policy that allows the compiler to decide an issue, then we say that the language uses a **static policy** or that the issue can be decided **at compile time.**

On the other hand, a policy that only allows a decision to be made when we **execute** the program is said to be a **dynamic policy.**

2.Another issue is the **scope of declarations**.

The scope of a declaration of x is the region of the program in which uses of x refer to this declaration.

A language uses **static scope or lexical scope** if it is possible to determine the scope of a declaration by looking only at the program.

Otherwise, the language uses **dynamic scope**. With dynamic scope, as the program runs, the same use of x could refer to any of several different declarations of x.

**Environments and States:**

- The environment is a mapping from names to locations in the store. Since variables refer to locations, we could alternatively define an environment as a mapping from names to variables.
- The state is a mapping from locations in store to their values. That is, the state maps 1-values to their corresponding r-values, in the terminology of C.

*Environments change according to the scope rules of a language.

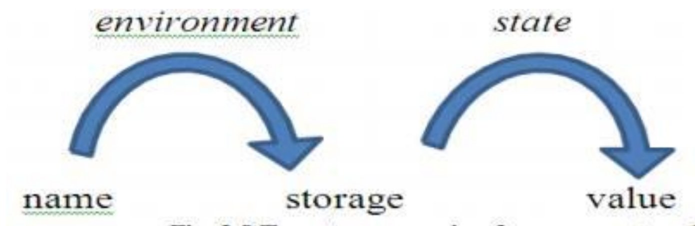


**Fig. 2.8 Two-stage mapping from names to values**

## <u>Static Scope and Block Structure</u>

Most languages, including C and its family, use static scope. we consider static-scope rules for a language with blocks, where a block is a grouping of declarations and statements. C uses braces { and } to delimit a block; the alternative use of begin and end for the same purpose dates back to Algol.

### <u>Block</u>

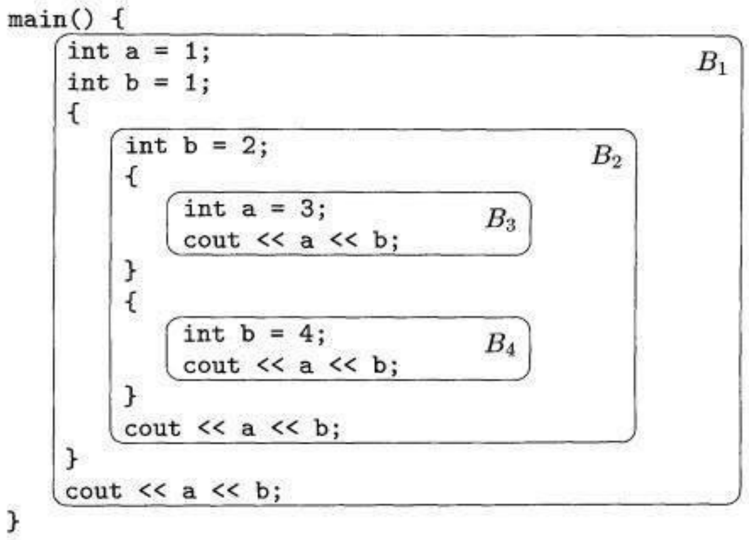
In C, the syntax of blocks is given by

1. One type of statement is a block. Blocks can appear anywhere that other types of statements, such as assignment statements, can appear.

2. A block is a sequence of declarations followed by a sequence of statements, all surrounded by braces.

Note that this syntax allows blocks to be nested inside each other. This nesting property is referred to as block structure.

Example



```
main() {
    int a = 1;
    int b = 1;
    {
        int b = 2;
        {
            int a = 3;
            cout << a << b;
        }
        {
            int b = 4;
            cout << a << b;
        }
        cout << a << b;
    }
    cout << a << b;
}
```

Figure 1.10: Blocks in a C++ program

Program in Fig. 1.10 has four blocks, with several definitions of variables a and b. As a memory aid, each declaration initializes its variable to the number of the block to which it belongs.

For instance, consider the declaration int a = 1 in block B1. Its scope is all of B1, except for those blocks nested (perhaps deeply) within B1 that have their own declaration of a.

B2, nested immediately within B1, does not have a declaration of a, but B3 does.

B4 does not have a declaration of a, block B3 is the only place in the entire program that is outside the scope of the declaration of the name a that belongs to B1. That is, this scope includes B4 and all of B2 except for the part of B2 that is within B3.

The scopes of all five declarations are summarized in below table

| DECLARATION | SCOPE |
|---|---|
| `int a = 1` | $B_1 - B_3$ |
| `int b = 1` | $B_1 - B_2$ |
| `int b = 2` | $B_2 - B_4$ |
| `int a = 3` | $B_3$ |
| `int b = 4` | $B_4$ |

ure 1.11 Scopes of declarations in Example 1.6

**Explicit Access Control**

Through the use of keywords like public, private, and protected, object-oriented languages such as C + + or Java provide explicit control over access to member names in a super class.

These keywords support encapsulation by restricting access. Thus, private names are purposely given a scope that includes only the method declarations and definitions associated with that class and any "friend" classes (the C + + term). Protected names are accessible to subclasses. Public names are accessible from outside the class.

**Dynamic Scope**

Any scoping policy is dynamic if it is based on factor(s) that can be known only when the program executes. The term dynamic scope, however, usually refers to the following policy: a use of a name x refers to the declaration of x in the most recently called procedure with such a declaration. Dynamic scoping of this type appears only in special situations. We shall consider two examples of dynamic policies: macro expansion in the C preprocessor and method resolution in object-oriented programming.

**Declarations and Definitions**

Declarations tell us about the types of things, while definitions tell us about their values. Thus,

int i is a declaration of i, while i = 1 is a definition of i.

**Parameter Passing Mechanisms**

For a procedure call .,the great majority of languages use either "call-by-value," or "call-by-reference," or both.

**a) Call - by – Value**

In call-by-value, the actual parameter is evaluated (if it is an expression) or copied (if it is a variable). The value is placed in the location belonging to the corresponding formal parameter of the called procedure. . Call-by-value has the effect that all computation involving the formal parameters done by the called procedure is local to that procedure, and the actual parameters themselves cannot be changed.

**b) Call - by - Reference**

In call-by-reference, the address of the actual parameter is passed to the callee as the value of the corresponding formal parameter. Uses of the formal parameter in the code of the callee are implemented by following this pointer to the location indicated by the caller. Changes to the formal parameter thus appear as changes to the actual parameter.

If the actual parameter is an expression, however, then the expression is evaluated before the call, and its value stored in a location of its own. Changes to the formal parameter change this location, but can have no effect on the data of the caller.

**c) Call - by – Name**

A third mechanism — call-by-name — was used in the early programming language Algol 60. It requires that the callee execute as if the actual parameter were substituted literally for the formal parameter in the code of the callee, as if the formal parameter were a macro standing for the actual parameter (with renaming of local names in the called procedure, to keep them distinct). When the actual parameter is an expression rather than a variable, some unintuitive behaviors occur, which is one reason this mechanism is not favored today.

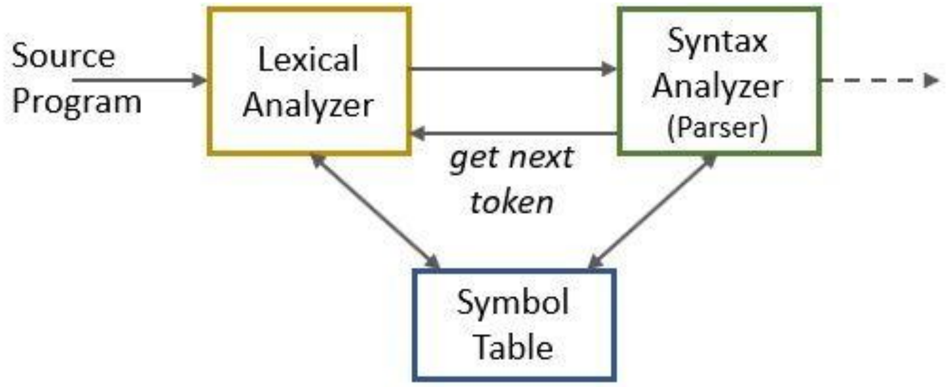
**<u>Aliasing</u>**

When parameters are effectively passed by reference ,two formal parameters can refer to the same location; such variables are said to be aliases of one another. This possibility allows a change of one variable to change another.

Eg:A is an array belonging to a procedure p,and p calls another procedure q(x,y) with call q(a,a).suppose parameters are passed by value. Now x and y have become aliases to each other. If within q there is an assignment x[10]=2 then the value of y[10]also becomes 2.

**The Role of the Lexical Analyser**

The first phase of a compiler, the main task of the lexical analyser is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyser to interact with the symbol table as well.



Lexical Analyzer's Interaction with Parser

1. The lexical analyzer phase has the scanner or lexer program implemented in it which produces tokens only when they are commanded by the parser to do so.
2. The parser generates the *getNextToken* command and sends it to the lexical analyzer as a response to this the lexical analyzer starts reading the input stream character by character until it identifies a lexeme that can be recognized as a token.
3. As soon as a token is produced the lexical analyzer sends it to the syntax analyzer for parsing.
4. Along with the syntax analyzer, the lexical analyzer also communicates with the symbol table. When a lexical analyzer identifies a lexeme as an **identifier** it enters that lexeme into the symbol table.
5. Sometimes the information of identifier in **symbol table** helps lexical analyzer in determining the token that has to be sent to the parser.

Apart from identifying the tokens in the input stream, the lexical analyzer also

- E**liminates** the blank space/white space and the comments of the program. Such other things include characters the separates tokens, tabs, blank spaces, new lines.

- Another task is correlating error messages generated by the compiler with the source program.
- For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message.

  Sometimes, lexical analyzers are divided into a cascade of two processes:

  a) Scanning consists of the simple processes that do not require tokenizationof the input, such as deletion of comments and compaction of consecutive whitespace characters into one
  b) . b) Lexical analysis proper is the more complex portion, which produces tokens from the output of the scanner.

**<u>Lexical analysis Versus Parsing</u>**

There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.

1. **Simplicity of design** is the most important consideration. The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks. For example, a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer.
2. **Compiler efficiency is improved.** A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.
3. **Compiler portability is enhanced**. Input-device-specific peculiarities can be restricted to the lexical analyzer.

**<u>Token,Pattern and Lexemes.</u>**

**1.A token** is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit.
E.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes.

2. **A pattern** is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.

**3.A lexeme** is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as aninstance of that token.

Example 3.1 : Figure 3.2 gives some typical tokens, their informally described patterns, and some sample lexemes. To see how these concepts are used in practice, in the C statement
printf("Total = %d\n", score);
both printf and score are lexemes matching the pattern for token id, and"Total = %d\n" is a lexeme matching literal.
In many programming languages, the following classes cover most or all of the tokens:

| TOKEN | INFORMAL DESCRIPTION | SAMPLE LEXEMES |
|---|---|---|
| **if** | characters `i`, `f` | `if` |
| **else** | characters `e`, `l`, `s`, `e` | `else` |
| **comparison** | `<` or `>` or `<=` or `>=` or `==` or `!=` | `<=`, `!=` |
| **id** | letter followed by letters and digits | `pi`, `score`, `D2` |
| **number** | any numeric constant | `3.14159`, `0`, `6.02e23` |
| **literal** | anything but `"`, surrounded by `"`'s | `"core dumped"` |

## Attributes for Token

- When more than one lexeme can match a pattern, the lexical analyzer must

provide the subsequent compiler phase’s additional information about the particular lexeme that matched.
❖For example, the pattern for token number matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program.

❖Tokens have at most one associated attribute, although this attribute may have a structure that combines several pieces of information. Information about an identifier - e.g., its lexeme, its type, and the location at which it is first found (in case an error message about that identifier must be issued) - is kept in the symbol table.

Example: The token names and associated attribute values for the Fortran statement
E = M * C ** 2 are written below as a sequence of pairs.
<id, pointer to symbol-table entry for E>
<assign-op>
<id, pointer to symbol-table entry for M>
<mult-op>
<id, pointer to symbol-table entry for C>
<exp-op>
<number, integer-value 2 >

**<u>Lexical Errors</u>**

A lexical error refers to an error that occurs during the lexical analysis phase of the compilation process.
A lexical error occurs when the compiler encounters an invalid or unrecognized token in the source code. This can happen due to various reasons, such as misspelled keywords, using an undefined identifier, or using an incorrect operator.
Example:

```
1.Void main()
2.{
3.int x=10, y=20;
4.char*a;
5.a= &x;
6.x= 1xab;
7.}
```

In this code, 1xab is neither a number nor an identifier. So this code will show the lexical error

Note that in certain pairs, especially operators, punctuation, and keywords, there is no need
For an attribute value.

example:

```
 Void main()
 {    int x=10, y=20;
char * a;
    a= &x;
    x= 1xab;  .
 }
```

In this code, 1xab is neither a number nor an identifier. So this code will show the lexical error.

## SPECIFICATION OF TOKENS

There are 3 specifications of tokens:
1)Strings
2) Language
3)Regular expression

## Strings and Languages

An **alphabet** or character class is a finite set of symbols.
A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet.
A **language** is any countable set of strings over some fixed alphabet.

In language theory, the terms "sentence" and "word" are often used as synonyms for "string." The length of a string s, usually written |s|, is the number of occurrences of symbols in s. For example, banana is a string of length six. The empty string, denoted ε, is the string of length zero.

## Operations on strings

The following string-related terms are commonly used:

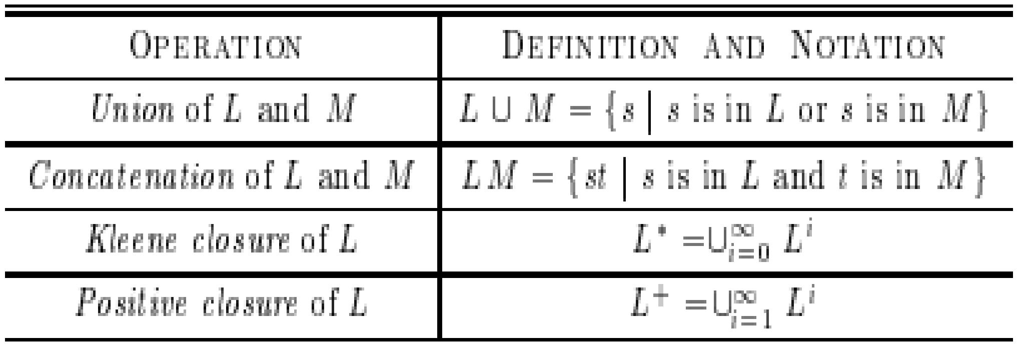
1. A **prefix** of string s is any string obtained by removing zero or more symbols from the end of string s. For example, ban is a prefix of banana.

2. A **suffix** of string s is any string obtained by removing zero or more symbols from the beginning of s. For example, nana is a suffix of banana.

3. A **substring** of s is obtained by deleting any prefix and any suffix from s. For example, nan is a substring of banana.

4. The **proper prefixes, suffixes, and substrings** of a string s are those prefixes, suffixes, and substrings, respectively of s that are not ε or not equal to s itself.
5. A **subsequence** of s is any string formed by deleting zero or more not necessarily consecutive positions of s For example, baan is a subsequence of banana.

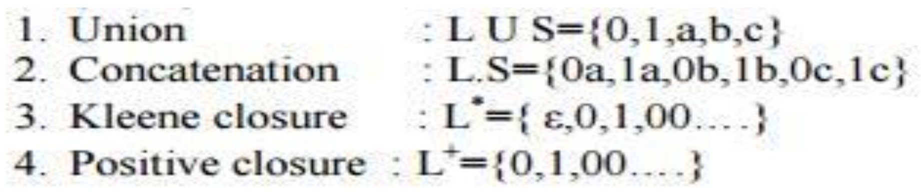
<u>**Operations on languages:**</u>
The following are the operations that can be applied to languages:
1. Union
2. Concatenation
3. Kleene closure
4. Positive closure



| OPERATION | DEFINITION AND NOTATION |
|---|---|
| *Union* of $L$ and $M$ | $L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$ |
| *Concatenation* of $L$ and $M$ | $LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$ |
| *Kleene closure* of $L$ | $L^* = \cup_{i=0}^{\infty} L^i$ |
| *Positive closure* of $L$ | $L^+ = \cup_{i=1}^{\infty} L^i$ |

Figure 3.6: Definitions of operations on languages

The following example shows the operations on strings: Let L={0,1} and S={a,b,c}



1. Union : L U S={0,1,a,b,c}
2. Concatenation : L.S={0a,1a,0b,1b,0c,1c}
3. Kleene closure : $L^*$={ ε,0,1,00….}
4. Positive closure : $L^+$={0,1,00….}

## Regular Expressions

Regular expression is an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions serve as names for a set of strings. Programming language token scan be described by regular languages.

Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expressions is known as regular grammar.The language defined by regular grammar is known as regular language.
Each regular expression r denotes a language L(r).

Example for regular expression: C language identifier can be represented in regular expression
as

**letter ( letter | digit )***

- The vertical bar above means union, the parentheses are used to group sub expressions, the star means \zero or more occurrences of," and the just a position of letter with the remainder of the expression signifies concatenation.

Here are the rules that define the regular expressions over some alphabet Σ and the languages that those expressions denote:

1.ε is a regular expression, and L(ε) is { ε }, that is, the language whose sole member is the empty string.

2. If 'a' is a symbol in Σ, then 'a' is a regular expression, and L(a) = {a}, that is, the language with one string, of length one, with 'a' in its one position.

Suppose r and s are regular expressions denoting the languages L(r) and L(s).

Then,
a) (r)|(s) is a regular expression denoting the language L(r) U L(s).
b) (r)(s) is a regular expression denoting the language L(r)L(s).
c) (r)* is a regular expression denoting (L(r))*

d) (r) is a regular expression denoting L(r).

some conventions in the case of pairs of parentheses

- The unary operator * has highest precedence and is left associative.
- Concatenation has second highest precedence and is left associative.
- **|** has lowest precedence and is left associative.

Example : Let Σ = {a,b};

1. The regular expression a|b denotes the language{a,b}.

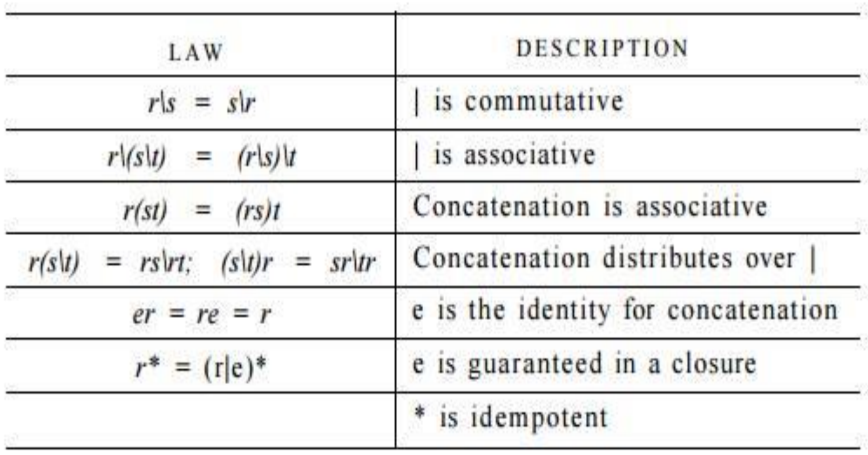
2. (a|b)(a|b) denotes {aa,ab, ba,bb} the language of all strings of length two over the alphabet .Another regular expression for the same language is aa|ab|ba|bb.

3. a* denotes the language consisting of all strings of zero or more a's, that is, {**ϵ**, a, aa, aaa..}

4. (a|b)* denotes the set of all strings consisting of zero or more instances of a or b, that is, all strings of a's and b's: {**ϵ,** a, b, aa, ab, ba, bb, aaa…} Another regular expression for the same language is {ab } .

5. a|a*b denotes the language {a,b, ab, aab, aaab,….}, that is, the string a and l strings consisting of zero or more a's and ending in b.

A language that can be defined by a regular expression is called a regular set. If two regular expressions r and s denote the same regular set, we say they are equivalent and write r = s. For instance, (a|b)=(b|a). There are a number of algebraic laws for regular expressions;



| LAW | DESCRIPTION |
|---|---|
| $r \mid s = s \mid r$ | $\mid$ is commutative |
| $r \mid (s \mid t) = (r \mid s) \mid t$ | $\mid$ is associative |
| $r(st) = (rs)t$ | Concatenation is associative |
| $r(s \mid t) = rs \mid rt;\ (s \mid t)r = sr \mid tr$ | Concatenation distributes over $\mid$ |
| $er = re = r$ | e is the identity for concatenation |
| $r^* = (r \mid e)^*$ | e is guaranteed in a closure |
| | * is idempotent |

Figure 3.7: Algebraic laws for regular expressions

**Regular set**

A language that can be defined by a regular expression is called a regular set. If two regular expressions r and s denote the same regular set, we say they are equivalent and write r = s.

There are a number of algebraic laws for regular expressions that can be used to manipulate into equivalent forms.

For instance, r|s = s|r is commutative; r|(s|t)=(r|s)|t is associative.

**<u>Regular Definitions</u>**

Giving names to regular expressions is referred to as a Regular definition. If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

$d_l \rightarrow r_1$
$d_2 \rightarrow r_2$

………
$d_n \rightarrow r_n$

1.Each $d_i$ is a distinct name.
2.Each $r_i$ is a regular expression over the alphabet Σ U $\{d_l, d_2, . . . , d_{i-l}\}$.

Example: Identifiers is the set of strings of letters and digits beginning with a letter. Regular
definition for this set:

letter → A | B | …. | Z | a | b | …. | z | digit → 0 | 1 | …. | 9

id → letter ( letter | digit ) *

Example 3 . 6 : Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4. The regular definition

digit 0 | 1 9

digits -> digit digit*

optionalFraction - ) • . digits | e

optionalExponent ( E ( + | - | e ) digits ) | e

number -> digits optionalFraction optionalExponent

**Shorthands**

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational short hands for them.

**1. *One or more instances (+)*:**

- The unary postfix operator + means " one or more instances of" .

- If r is a regular expression that denotes the language L(r), then ( r )$^+$ is a regular expression that denotes the language (L (r ))$^+$

- Thus the regular expression a$^+$ denotes the set of all strings of one or more a's.
- The operator $^+$ has the same precedence and associativity as the operator $^*$.

**2*. Zero or one instance ( ?)*:**

- The unary postfix operator ? means "zero or one instance of".

- The notation r? is a shorthand for r | ε.

**3*. Character Classes*:**

- The notation [abc] where a, b and c are alphabet symbols denotes the regular expression a | b | c.
- Character class such as [a – z] denotes the regular expression a | b | c | d | ….|z.

- We can describe identifiers as being strings generated by the regular expression, [A–Za–z][A– Za–z0–9]*

Using these shorthands, we can rewrite the regular definition as:

letter. -> [A-Za-z_]
digit -> [0-9]
id -> letter- ( letter 1 digit )*

The regular definition of Example 3.6 can also be simplified:

digit -> [0-9]
digits —>• digit+
number -»• digits (. digits)? ( E [+-]? digits )? •