Exercise 1

Let $L \in \{0,1\}^*$ be a language formed by the entities X, Y, and Z:

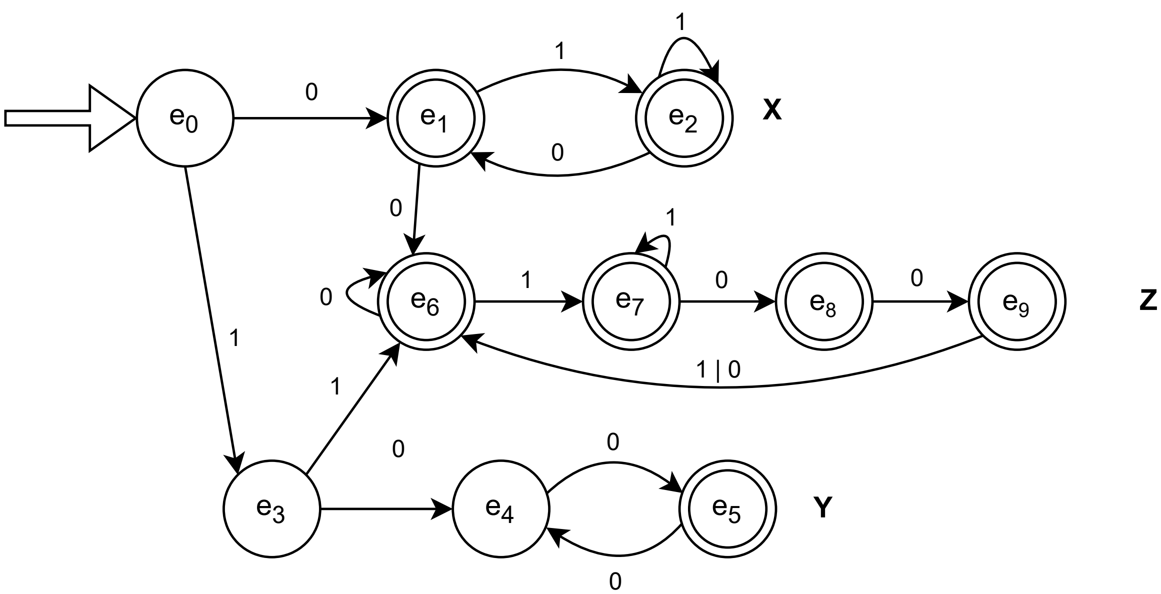
- **X** : Starts with '0' and contains no consecutive '0's.
- **Y** : Starts with '1', followed by a sequence with an even number of '0's.
- **Z** : Starts with either two '0's or two '1's, followed by a sequence of '0's and '1's, and does not contain '101'.

Construct the corresponding automaton to recognize all these entities.

Note

Never loop on the start state, as this can lead to undesired behavior. Doing so makes having independent branches impossible since the compiler has only one automaton to match all entities.

Solution :



Exercise 2

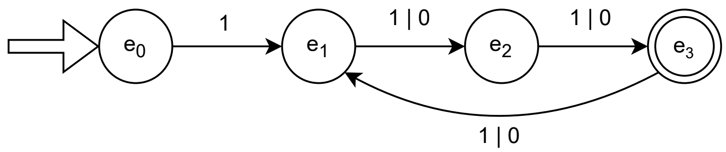
Give the grammar and automaton for each language

- $L_1 = \{w \in 1.\{0,1\}^* \ / \ |w| \equiv 0 \ [3] \}$
- $L_2 = \{w \in \{a,b\}^* \ / \ w = a^n \ b^m \ , m > n \}$
- $L_3 = \{w \in \{a,b,c\}^* \ / \ w = a^i \ c \ b^j \ , i \equiv 0[2] \ , j \equiv 1[2] \}$

Solution :

L$_1$

$S \rightarrow 1A$
$A \rightarrow 1B \mid 0B$
$B \rightarrow 1C \mid 0C$
$C \rightarrow 1A \mid 0A \mid \epsilon$



L$_2$

Solution$_1$

$S \rightarrow aAb \mid bB$
$A \rightarrow aAb \mid bB$
$B \rightarrow bB \mid \epsilon$

Solution$_2$

$S \rightarrow aSb \mid Sb \mid b$

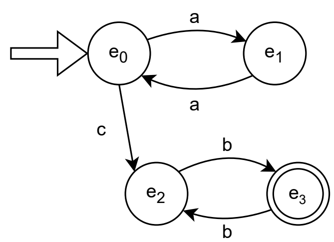
No Automaton

This grammar is neither right-regular nor left-regular, it is a Type 2 grammar. Therefore, it does not have a corresponding automaton.

$\mathbf{L_3}$

$S \to aaS \mid cA$

$A \to bbA \mid b$



	a	b	c
e_0	e_1	/	e_2
e_1	e_0	/	/
e_2	/	e_3	/
e_3	/	e_2	/

Example with 'aacbb#'

E_c	t_c	Action
e_0	'a'	E_c = M[e_0,'a'] = e_1 t_c = t_s = 'a'
e_1	'a'	E_c = M[e_1,'a'] = e_0 t_c = t_s = 'c'
e_0	'c'	E_c = M[e_0,'c'] = e_2 t_c = t_s = 'b'
e_2	'b'	E_c = M[e_2,'b'] = e_3 t_c = t_s = 'b'
e_3	'b'	E_c = M[e_3,'b'] = e_2 t_c = t_s = '#' Since $E_c \neq \emptyset$ and $E_c \notin F$: token isn't regonized because the automaton didn't reach a final state

Exercise 3

For each instruction, provide the corresponding grammar:

- Affectation, where the right-hand side is an arithmetic expression with optional parenthesis.
- An 'if' statement with an optional 'else'.
- A 'while' loop.
- A 'switch-case' statement.

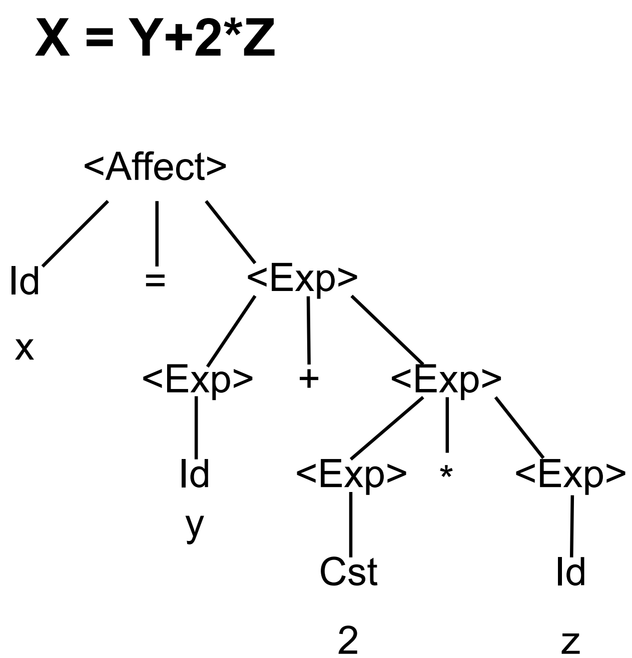
Solution :

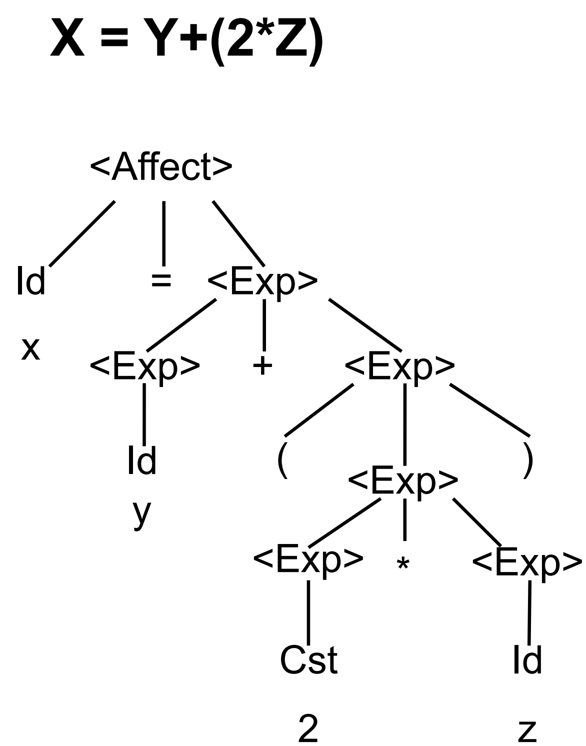
Affectation

< Affect > $\rightarrow$ Idf = < Exp >

< Exp >$\rightarrow$ Cst | Idf | < Exp > + < Exp > | < Exp > - < Exp > | < Exp > * < Exp > | < Exp > / < Exp > | < Exp > mod < Exp > | (< Exp >)

Tree Examples





Exercise 4

let the source code be :

```
INT adr1 , adr2 ;
REAL x_1 ,y ;
adr1=vale ; y=valer ;
```

- ‘adr_1‘, ‘adr_2‘, ‘x‘, and ‘y‘ are identifiers composed of alphanumeric characters. They must start with a letter and have a maximum length of 7 characters.
- ‘vale‘ is an integer constant with a maximum of 5 digits and a maximum value of 32768.
- ‘valr‘ is a floating-point constant with a maximum length of 9 characters.

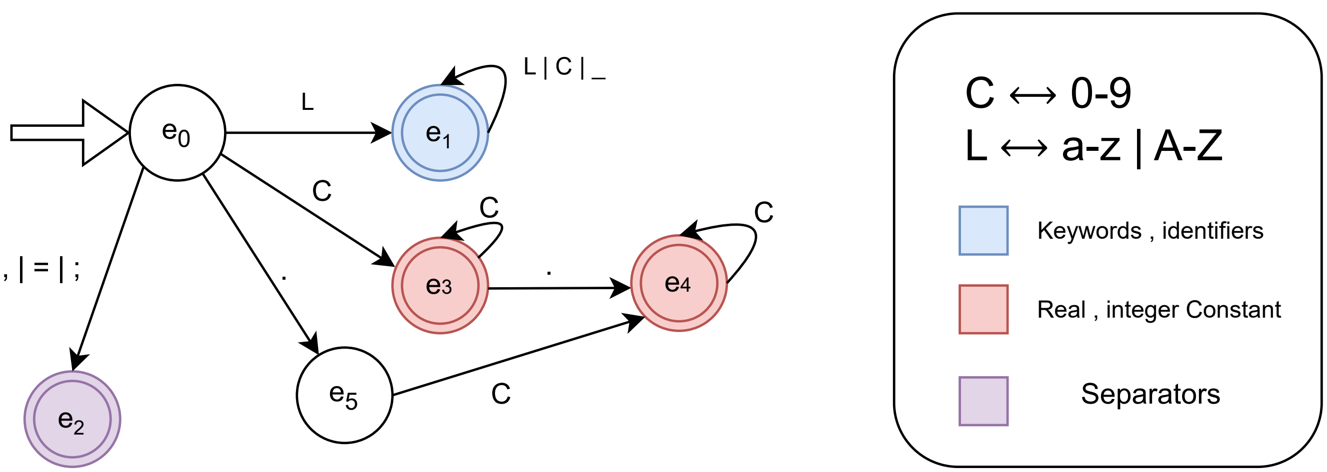
1. Define the lexical entities.
2. Construct the automaton for the source code.
3. Write the corresponding recognition algorithm.

Lexical Entities

Entities

- Identifier : 'adr_1' , 'adr_2' , 'x' , 'y' .
- Constant : 'vale' , 'valr' .
- Keyword : 'REAL' , 'INT' .
- Separator : '=' , ',' , ';' .

Automaton



Algorithm 1 Algorithm Recognition

```
Begin
Lire(Entity);                    -- Reads token
t_c ← Entity[0];                 -- Initialize with 1st character
E_c ← e_0;                       -- Intialize with initial state
F ← {e_1,e_2,e_3,e_4};           -- List Of Final States
cmpt ← 0;                        -- Initialize counter of nb character
keyword ← { 'INT' , 'REAL' };    -- List of keywords
while ( E_c != ∅ AND t_c != '#' ) do    -- Loop as long as there is no blockage and not all character of the token have been analyzed
    E_c ← M[E_c,t_c];            -- Gets next state
    t_c ← t_s;                   -- Gets next character of the token
    cmpt ← cmpt + 1;             -- Increment Counter
end while
if ( E_c != ∅ ) then             -- Check if there is a blockage
    print( 'Error the lexical entity isn't regonised by the automaton because blockage happened' );
else if ( E_c ∉ F ) then         -- Check if automaton reaches a final state
    print( 'Error the lexical entity isn't regonised by the automaton because it didn't reach a final state' );
else
    if ( E_c = e_1 ) then        -- Check if identifier or keyword
        if ( Entity ∈ keyword ) then    -- Check if keyword
            Codify the keyword and put it in the symbole table.
        else                     -- Token is identifier
            if ( cmpt > 7 ) then -- Check length of identifier
                print( 'Error the identifier exceeded the maximum character length' );
            else
                Codify the identifier and put it in the symbole table.
            end if
        end if
    else if ( E_c = e_3 ) then   -- Check if integer constant
        if (cmpt > 5) then       -- Check length of integer constant
            print( 'Error the integer constant exceeded the maximum character lenght' );
        else if (Eval(Entity) > 32768) then    -- Check value of integer constant
            print( 'Error the integer constant exceeded the maximum possible value' );
        else
            Codify the integer constant and put it in the symbole table.
        end if
    else if ( E_c = e_4 ) then   -- Check if float constant
        if (cmpt > 9) then       -- Check length of float constant
            print( 'Error the float constant exceeded the maximum character lenght' );
        else
            Codify the float constant and put it in the symbole table.
        end if
    else                         -- token is a separator
        Codify the seperator and put it in the symbole table.
    end if
end if
End
```

Note

- Identifiers and keywords share the same branch in the automaton to simplify its structure. In the recognition algorithm, we will store the keywords in an array and compare them with the token.
- In most compiler notations, formats like ‘.25‘ (meaning ‘0.25‘) and ‘45.‘ (meaning ‘45.0‘) are accepted.
- Eval convert string into integer.