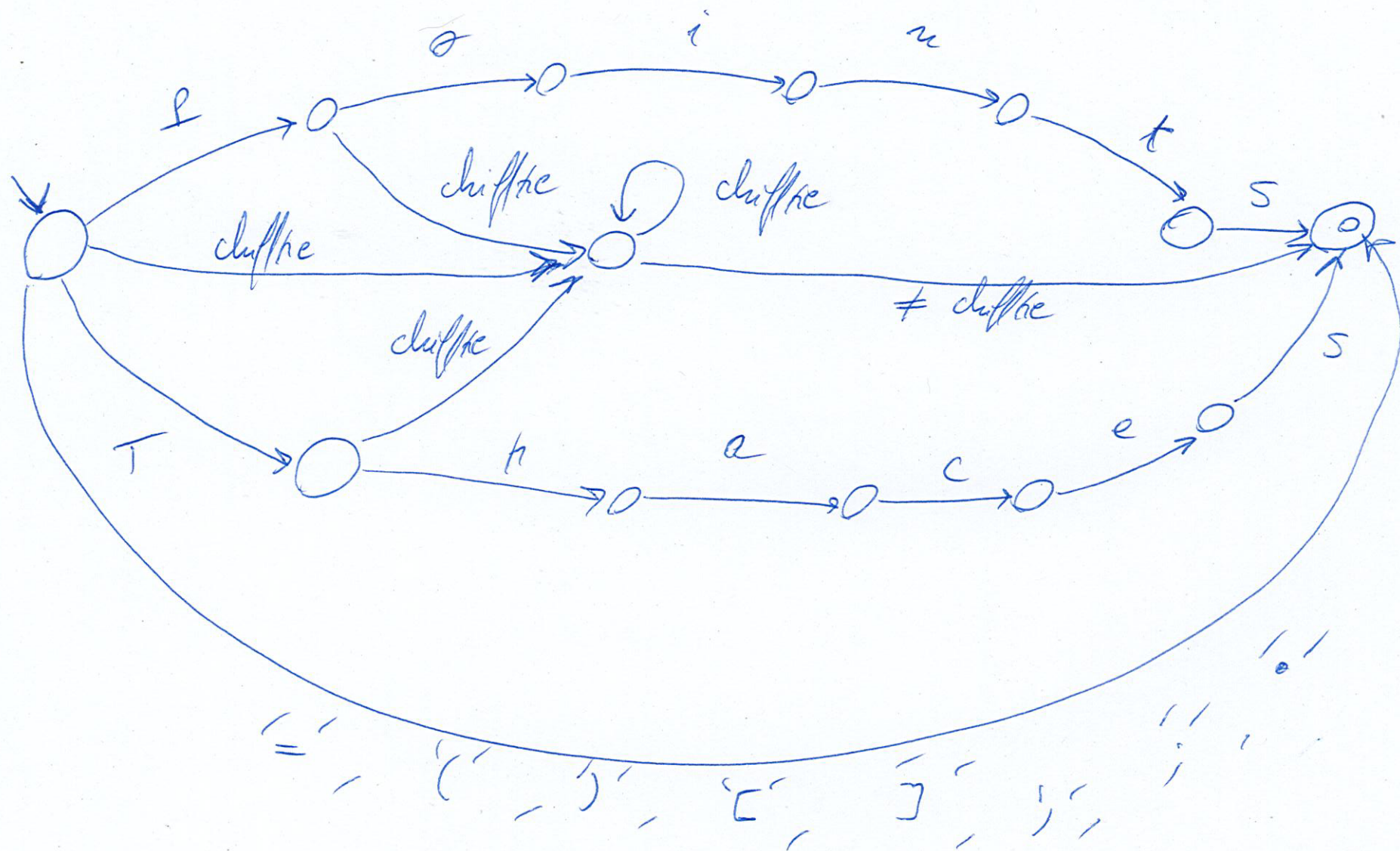


Automate d'analyse lexicale



Q1.

1. Lexeme incorrect : #
2. Automate d'analyse lexicale : voir fichier

=====

Q2.

1. Le langage proposé est régulier. En effet, il ne contient pas d'imbrications non bornées de structures :

- pas de parenthèses ni crochets imbriqués
- pas de répétitions ou concaténations imbriquées

Il peut donc être reconnu par un automate fini.

2. Erreur syntaxique : P1 == 12

=====

Q3.

1. contenu du fichier table_points.h

// définition des types

```
typedef struct {
    char nom[20] ;
    int x ;
    int y ;
} Point ;

typedef struct {
    Point TPoints[200] ;
    unsigned int nbPts ; // nbre de points présents
} TabPoints ;
```

// variable globale

TabPoints TP ; // table des points

// primitives

```
void initTP() ;
// initialise TP à vide
```

```
void insererTP (char *n, int x, int y) ;
// ajoute (n,x,y) dans TP
```

```
int estPresentTP (char *n) ;
// vrai ssi n est present dans TP
```

```
void coord (char *n, int *x, int *y) ;
// renvoie dans x,y les coord du point n present dans TP
```

=====

2. Construction de la table des points

```
void constTabPoints(char *nomfichier) {
    demarrer(nomfichier) ;
    initTP() ;
}
```

```

    Rec_Pgm() ;
}

void Rec_Pgm() {
    if (LC.nature == POINTS) avancer() else erreur() ;
    Rec_SeqPts()
    if (LC.nature == TRACES) avancer() else erreur() ;
    // on suspend ici la lecture du fichier
}

void Rec_SeqPts() {
    Rec_Point() ; Rec_SuiteSeqPts() ;
}

void Rec_SuiteSeqPts() {
    if (LC.nature == PVIRG)
        Rec_SeqPts() ;
}

void Rec_Points() {
    char *n ;
    int x, y ;
    if (LC.nature == NPOINT) {
        n = LC.chaine ; // nom du point courant
        avancer()
    } else erreur() ;
    if (estPresentTP(n)) erreur() ; // point déjà présent
    if (LC.nature == EGAL) avancer() else erreur() ;
    if (LC.nature == PARO) avancer() else erreur() ;
    if (LC.nature == ENTIER) {
        x = LC.valeur ; // coord x du point courant
        avancer()
    } else erreur() ;
    if (LC.nature == VIRG) avancer() else erreur() ;
    if (LC.nature == ENTIER) {
        y = LC.valeur ; // coord y du point courant
        avancer()
    } else erreur() ;
    if (LC.nature == PARF) avancer() else erreur() ;

    insererTP (n, x, y) ; // on insere le point dans TP
}

```

=====
Q4

1. contenu du fichier table_traces.h

// définition des types

```

typedef struct {
    char nom[20] ;
    int l ;
} Trace ;

```

```

typedef struct {
    Trace TTrace[200] ;
    unsigned int nbTraces ; // nbre de traces présentes
} TabTraces ;

```

// variable globale

TabTraces TT ; // table des traces

```

// primitives

void initTT() ;
// initialise TT à vide

void insererTT (char *n, int l) ;
// ajoute (n,l) dans TT

int estPresentTT (char *n) ;
// vrai ssi n est present dans TT

void lg (char *n, int *l) ;
// renvoie dans l la longueur de la trace n présente dans TT

```

2.

```

void constabTraces(nonfichier) {
    // on reprend la suite de la lecture du fichier ...
    if (LC.nature == VIRG) avancer() else erreur() ;
    initTT() ;
    Rec_SeqTraces()
    if (LC.nature == FSEQ)
        arreter(nomfichier)
    else
        erreur() ;
}

void Rec_SeqTraces() {
    Rec_Traces() ;
    Rec_SuiteSeqTraces() ;
}

void Rec_SuiteSeqTraces() {
    if (LC.nature == PVIRG) {
        avancer() ;
        Rec_SeqTraces() ;
    }
}

void Rec_Traces() {
    char *n ; // nom de la trace courante
    int lg ; // longueur de la trace courante
    if (LC.nature == NTRACE) {
        n = LC.chaine ;
        avancer()
    } else erreur() ;
    if (estPresentTT(n)) erreur() ; // trace déjà présente
    if (LC.nature == EGAL) avancer() else erreur() ;
    Rec_SeqSegment(&lg) ;
    insererTP(n, lg) ; // on insere la trace et sa longueur dans la table
}

void Rec_SeqSegment(int *l) {
    int l1, l2 ;
    Rec_Segment(&l1) ;
    Rec_SuiteSegment(&l2) ;
    *l = l1 + l2 ; // concaténation des 2 segments
}

void Rec_SuiteSeqSegment(int *l) {
    int l1, l2 ;
    if (LC.nature == DOT) {

```

```

        avancer() ;
        Rec_Segment(&l1) ;
        Rec_SuiteSegment(&l2) ;
        *l = l1 + l2 ; // concaténation des 2 segments
    } else
        *lg = 0 ; // cas epsilon, segment vide
}

void Rec_Segment(int *l) {
    int l1, n ;
    Rec_SegElementaire(&l1) ;
    Rec_Repetition(&n) ;
    *l = l1 * n ; // n repetition du segment de longueur l1
}

void Rec_Repetition(int *n) {
    if (LC.nature == ENTIER) {
        *n = LC.valeur ;
        avancer() ;
    } else
        *n = 1; // cas epsilon, pas de répétition
}

void Rec_SegElementaire(int *l) {
    switch (LC.nature) {
        case NTRACE :
            if (estPresentTT(LC.chaine)) {
                lg (n, l) ; // on recupere la longueur de la trace
                avancer() ;
            } else erreur() ; // trace non définie
            break ;
        case CROCHO :
            avancer() ;
            if (LC.nature == NPOINT) {
                if (estPresentTP(LC.chaine)) {
                    *l = 1
                    avancer() ;
                } else erreur() ; // point non défini
            } else erreur() ;
            Rec_SuiteSeqChaine(l) ;
            break ;
        default : erreur() ;
    }
}

void Rec_SuiteSeqChaine(int *l) {
    switch (LC.nature) {
        case CROCHF :
            avancer() ;
            break ; // fin de la sequence de points ...
        case VIRG :
            avancer() ;
            if (LC.nature == NPOINT) {
                if (estPresentTP(LC.chaine)) {
                    *l = *l + 1 // on ajoute un point
                    avancer() ;
                } else erreur() ; // point non défini
            } else erreur() ;
            Rec_SuiteSeqChaine(l) ;
            break ;
        default : erreur() ;
    }
}

```

=====
Q5

1. Programme principal

```
int main(int argc, char *argv[]) {  
    if (argc == 2) {  
        constTabPoints(argv[1]) ;  
        constTabTraces(argv[1]) ;  
    } else  
        erreur() ;  
    return 0 ;  
}
```

2. Longueur euclidienne des traces

les principaux changements sont :

- modifier la formule de calcul de la longueur d'un segment
élémentaire constitué d'une séquence de points
- prendre en compte la distance entre le dernier point d'un segment
et le premier du segment suivant :
 - dans le cas d'une concaténation
 - dans le cas d'une répétition

une solution "simple" consiste à ce que Rec_Segment produise explicitement
une séquence de points (et non sa longueur) :

```
void Rec_Segment(SeqPt *s) ;  
// reconnaît une description de segment et renvoie la séquence de points  
correspondante.
```

On peut alors calculer la longueur de s en additionnant les distances entre tous
les points consécutifs qui la compose ...