Licence 3 Informatique
Module : théorie des langages
et compilation

Projet
city

L'objectif de ce projet est de créer un interpréteur pour un langage permettant la génération automatisée d'une ville constituée de maisons et de routes. Le but de ce langage est de permettre facilement la création de graphes dont les sommets sont représentés par les maisons et les arcs par les routes afin de tester des algorithmes portant sur des graphes.
Ces maisons seront placées sur une grille hexagonale. Pour nous repérer dans cette grille nous utiliserons 3 coordonnées (x, y, z). La Figure 1 montre les coordonnées dans un repère hexagonal de rayon 3. Vous trouverez de très bonnes explications ainsi que de nombreuses illustrations à cette adresse : `https://www.redblobgames.com/grids/hexagons/`

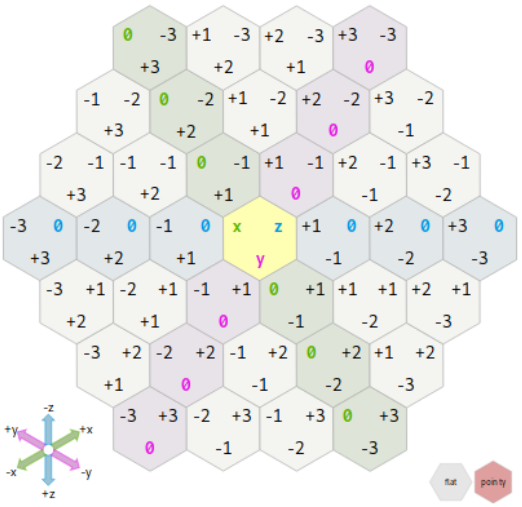

FIGURE 1 – Coordonnées dans un repère hexagonal

Les spécificités du langage sont détaillées dans la suite du sujet.
Le projet peut se faire seul ou en binôme et est à déposer au plus tard le vendredi 20 décembre 2019 sur Moodle. Vous déposerez une archive à votre nom au format zip ou tar.gz contenant le code de votre projet. En cas de binôme, vous indiquerez dans le nom de l'archive vos deux noms et ne déposerez l'archive qu'une fois.

1 Généralités

Le projet est divisé en trois parties : la reconnaissance du langage, la structure de données et la gestion de l'interface graphique.
La reconnaissance du langage sera effectuée dans un premier temps. Dans cette première partie nous souhaitons avoir un produit final qui lit un fichier en entrée contenant un programme avec la syntaxe définie ci-dessous (cette partie gérée seule ne vous donnera quasiment pas de point). Dans une seconde partie nous souhaitons gérer une structure de données adaptée à la manipulation d'un graphe

et du langage proposé, une bonne implémentation de cette partie doit permettre de générer facilement une sortie. Ce sont ces deux premières parties qui vous permettrons d'obtenir des points. Il est possible d'obtenir la note de 20/20 en gérant parfaitement ces deux premières parties. Les explications nécessaires à la gestion de l'interface graphique seront fournies le lundi 2 décembre lors d'une mise à jour du sujet.

La description du langage est divisée en 6 sections allant de 2.1 à 2.6. Pour chaque section un nombre de point est donné à titre indicatif. Les sections peuvent être gérées les unes après les autres. Pour obtenir des points il faut absolument gérer la reconnaissance du langage et la structure de données pour chaque section (par exemple pour les instructions de base vous devez être en capacité d'afficher une matrice d'adjacence une fois le parsing d'un programme en entrée effectué).

Afin de tester votre projet, vous pouvez récupérer les exemples disponibles et la base bison sur Moodle.

Nous insistons sur le fait qu'un langage fonctionnel même sans interface graphique doit avoir une structure de données adéquate. Il ne suffit pas de reconnaître une règle et de faire un affichage "règle détectée" pour obtenir des points !

Vous pouvez utiliser la bibliothèque d'expressions utilisée durant les TPs comme base pour votre structure de données.

Attention, vous serez pénalisés si :
— Le projet ne compile pas (erreur de compilation) : **4 points**
— La grammaire est ambigüe (avertissements bison) : **1 point**
Pour chaque consigne du projet votre structure de données doit être agrémentée de nouveaux éléments. En fin de projet, le parsing d'un fichier en entrée doit créer un graphe autonome qui peut être manipulée en utilisant seulement votre structure de données et permettre de générer une sortie (par exemple : une matrice d'adjacence, le résultat d'un plus court chemin, ...).

2 Langage

2.1 Instructions de base ∼4 points

La base de notre langage doit permettre la construction de maisons et de routes afin de former un graphe.

1. Pour commencer la construction nous déclarons le programme principal à l'aide de l'instruction `Construire` suivie d'une accolade ouvrante { de la liste des instructions du programme et pour terminer le programme une accolade fermante }. Il est aussi possible de spécifier le rayon de la ville entre parenthèses avant l'accolade ouvrante. Si aucune taille n'est spécifiée la ville sera de 5. L'appel du programme principal initialisera la structure de graphe si elle n'existe pas encore. Si le graphe existe déjà elle ne fera que mettre à jour la taille de la ville.
2. Il est possible de construire une maison en utilisant l'instruction `Maison` de deux façons :
   — construire à un emplacement aléatoire (dans la limite de la taille de la ville) :
   `Maison`
   — construire à un emplacement précis :
   `Maison coordonnées`
   Pour chaque maison construite on ajoute un sommet à notre graphe dans notre structure de données.
   Les `coordonnées` sont des couples d'expressions arithmétiques sur des entiers notées entre parenthèses. Par exemple, les coordonnées `(-1+2, 4-2, -1)` ou `(3/2, 5/2, -1*-1)` correspondent

à `(1, 2, 1)`.

3. Il est possible de construire une route entre deux maisons avec l'instruction `Route` avec la syntaxe suivante :
   `Route coordonnées -> coordonnées`
   L'ajout d'une route correspond à l'ajout d'un arc orienté d'un sommet vers un second.
4. Enfin nous voulons pouvoir ajouter des commentaires commençant par `%%` pour les commentaires sur une ligne et commençant par `%/` et finissant par `/%` pour les commentaires sur plusieurs lignes.

Un premier exemple de programme basique (Exemple 1) :

```
Construire (5){
    %%Début du programme
    Maison
    %/
        Construit une maison à des coordonnées
        aléatoires dans la limite de la taille
        de la ville
    /%
    Maison (0,0,0)
    %%Construit une maison à l'emplacement de coordonnées (0,0,0)
    Maison (2,-1,-1)
    Route (0,0,0) -> (2,-1,-1)
    %%Construit une route entre les maisons situées aux coordonnées (0,0,0) et (2,-1,-1)
    %%fin du programme
}
```

Exemple 1: Base d'un programme.

2.2 Gestion des maisons ∼5 points

Pour faciliter la manipulation du graphe et donc de la ville, nous souhaitons pouvoir déplacer ou supprimer une maison ou une route. Pour cela on utilisera une structure de données adaptée afin que chaque maison soit accessible au choix par ses coordonnées ou par son indice avec `maison[i]`. On affecte un indice à la création de chaque maison, ainsi, la première maison créée sera `maison[1]`, la seconde `maison[2]` et ainsi de suite.

Les fonctions suivantes pourront utiliser la notation `maison[i]` ou des coordonnées pour faire référence à une maison.

1. Chaque maison possède une orientation de base de 90° (porte vers le bas sur la représentation graphique). Il est possible de faire pivoter une maison de deux manière :
   — `Tourner <coord|ind> <horaire|!horaire>` pour faire pivoter la maison aux coordonnées données dans le sens horaire ou anti-horaire.
   — `Orienter <coord|ind> nombre°` pour orienter directement la maison à une orientation précise (entre 0° et 360°)
2. On ajoute aussi la possibilité de Déplacer une maison avec l'instruction :
   `Déplacer <coord|ind> -> coord`. Lors du déplacement d'une maison il faudra penser à mettre à jour les routes vers les bonnes coordonnées si besoin.
3. On peut supprimer une maison avec l'instruction suivante `Détruire <coord|ind>`. Si on supprime une maison il faudra penser à supprimer toutes les routes reliées à cette maison.
4. On peut supprimer une route avec l'instruction suivante `Détruire <coord|ind> -> <coord|ind>`.
5. Enfin on veut pouvoir afficher les informations liées à une maison en utilisant les instructions suivantes :
   — `Position maison[i]` affiche la position de la maison d'indice i.
   — `Orientation <coord|ind>` affiche l'orientation de la maison d'indice i ou aux coordonnées données.

— `Voisinage <coord|ind>` affiche les coordonnées de l'ensemble des maisons qui sont reliées à la maison i ou aux coordonnées données. On affichera aussi la distance relative entre ces deux maisons (ce qui correspond au poids de l'arc).

Exemple de programme avec la gestion des maisons en annexe (Exemple 2).

2.3 Utilisation de variables et expressions ∼3 points

Pour mieux maîtriser la gestion des maisons nous ajoutons la gestion des variables, des expressions arithmétiques et des erreurs.

1. Lors de la construction d'une maison il est possible de lui donner un nom avec la syntaxe suivante :
   `Maison M1 coordonnées`
   Le nom d'une maison commence toujours par une majuscule
2. On peut aussi accéder à une maison par son nom quelque soit l'instruction utilisée. par exemple `Position M1`
3. Les nombres peuvent être exprimés par une constante entière ou une expression arithmétique utilisant les opérateurs suivant $+$, $-$, $*$, $/$ et (). Si le résultat n'est pas un entier on arrondira à l'entier le plus proche.
4. Il sera possible d'utiliser des variables entières (notamment pour faire des compteurs) en utilisant une simple affectation $i = 1$, $j = 0$. Le nom d'une variable entière commencera toujours pas une minuscule.
5. On ajoutera maintenant la gestion d'erreurs notamment lorsque l'on essaye de construire une maison sur un emplacement déjà occupé ou lorsque l'on construit une route avec des coordonnées vides. Pensez à gérer un maximum de cas d'erreurs.

Exemple d'utilisation des variables et expressions arithmétiques en annexe (Exemple 3) :

2.4 Un peu de paillettes dans ma vie Kévin ∼2 points

Pour égayer un peu le quartier nous ajoutons la possibilité de changer la couleur des maisons. Et pour faciliter la construction automatique des maisons nous ajoutons une instruction Voisin qui ajoute une maison à une distance donnée.

1. Pour changer la couleur on peut utiliser l'expression suivante :
   `Coloriser <coord|ind|var> #rrggbb`
   Avec rr, gg et bb une valeur hexadécimale représentant le vert le rouge et le bleu. ou alors
   `Coloriser <coord|ind|var> (expression,expression,expression)`
2. Pour ajouter un voisin l'instruction :
   `Voisin <coord|ind|var> expression`
   La maison sera ajoutée automatiquement à un emplacement aléatoire autour de la maison indiquée avec une distance correspondant à l'expression arithmétique donnée et une route liera les deux maisons entre elles.

Exemple de programme avec la gestion des couleurs et voisinage (Exemple 4).

2.5 Conditionnelles et boucles ∼4 points

Il est possible de spécifier une série d'instructions sous certaines conditions. Les deux conditions possibles sont des fonctions permettant de déterminer si une case contient déjà une maison ou une route. Elles s'écrivent respectivement `occupé <coordonées>` et `vide <coordonnées>`.

La négation peut être utilisée devant une condition, avec le mot clé `non`. Par exemple, la condition `non occupée <coordonnées>` utilisée dans une conditionnelle permet de tester si la case est occupées ou non.

Une conditionnelle s'écrit de la manière suivante :

```
Si <condition>{
    ...
}
```

Il est possible d'ajouter un bloc `Sinon {}` pour spécifier les instructions à effectuer si la condition testée est fausse.

Deux types de boucles sont utilisables. La première permet d'effectuer une série d'instructions tant que la condition testée est vraie et s'utilise de la manière suivante :

```
Tant que <condition> {
    ...
}
```

La seconde permet de répéter n fois une série d'instructions. Comme précédemment, n est un entier positif et peut correspondre à une expression arithmétique. Elle s'utilise de la manière suivante :

```
Répéter <expression> fois {
    ...
}
```

Exemple de programme avec les conditions et boucles (Exemple 5).

2.6 Fonctions ∼3 points

La définition de fonctions en dehors du programme principal est possible (Exemple 4) . Une fonction se déclare comme suit :

```
void <nom>{
    ...
}
```

Si la fonction prend des arguments en paramètre, l'accès au premier argument dans la fonction se fait avec `$1` et de manière générale, l'argument n se fait avec `$n`. Il n'existe pas de limite au nombre d'arguments d'une fonction.

Il n'est pas possible de créer deux fonctions avec le même nom.

Une fonction est appelée avec son nom et d'éventuels arguments. Par exemple, `test 1 2 3` permet d'appeler une fonction test avec les arguments 1, 2 et 3.

3 Interface graphique

À partir du 2 décembre des sources permettant de gérer l'interface graphique vous seront fournies sur moodle ainsi qu'une mise à jour du sujet avec une notice permettant la manipulation de l'interface graphique. Nous souhaitons dans un premier temps que la structure de données soit gérée proprement avant d'utiliser l'interface graphique. L'utilisation de l'interface graphique ne comptera qu'en points bonus (jusqu'à 4 points pour un projet parfait), elle n'est pas nécessaire pour obtenir le graal (une note de 20/20) mais celle-ci permettra d'avoir un projet complet avec une visualisation de la ville, orientation et couleur des maisons, affichage des routes... Elle permettra aussi aux correcteurs de corriger plus

efficacement vos projets. Nous utiliserons un jeu de tests pour vérifier le bon fonctionnement de votre projet puis nous regarderons le code de votre projet en détail. Nous vous fournirons une partie du jeu de test le 2 décembre pour que vous puissiez avoir une idée des cas qui doivent fonctionner. Le but étant de gérer le langage de façon complète. N'hésitez pas à fournir des fichiers de tests que vous avez créé pour que l'on puisse voir quels sont les exemples qui fonctionnent. Les tests qui ne vous seront pas divulgués devront fonctionner pour obtenir les points de chaque partie du projet. Ce n'est pas parce que les exemples que vous avez en votre possession fonctionnent que vous aurez tous les points.

4 Exemples

```
Construire (6){
    Maison (0,0,0)
    %%Orienter la maison créée vers la gauche
    Tourner maison[1] horaire
    %%Orienter la même maison vers la droite
    Orienter (0,0,0) 0°
    %% Déplacer la maison 1 en (2,-1,-1)
    Deplacer maison[1] -> (2,-1,-1)
    %%Créer une seconde maison en (-1,-1,2) avec une route et supprimer la première
    Maison (-1,-1,2)
    Route maison[1] -> (-1,-1,2)
    Detruire  maison[1]
    %% La maison en position (-1,-1,2) de vient la maison 1 et la route est supprimée
    %%Construction d'une nouvelle maison avec une route et suppression de la route
    Maison (-1,2,-1)
    Route maison[1] -> maison[2]
    Detruire maison[1] -> maison[2]
    %%L'instruction inverse ne fonctionne pas car les routes sont orientées
    Position maison[2]
    %%Affiche La maison 2 est en position (-1,2,-1)
    Orientation (-1,2,-1)
    %%La maison 1 est orientée à 90°
    Voisinage maison[1]
    %%Affiche la maison 1 n'a pas de voisin
    Route maison[1] -> maison[2]
    Voisinage (-1,-1,2)
    %%Affiche la maison 1 a 1 voisin :
    %%maison 2 (4,4,4) distance relative 1
}
```

Exemple 2: Exemple de programme avec gestion des maisons.

```
Construire (6){
    Maison M1 (1+2,3/4,-2)
    Position M1
    %% Affiche La maison 1 est en position (3,1,-2)
    i=2
    i=i-1
    Maison M2 (2+(1+1)/2,i,-(1+1)*i)
    %%Affiche erreur il y a déjà une maison en (3,1,-2)
    Route M1 -> (0,0,0)
    %%Affiche erreur il n'y a pas de maison en (0,0,0)
}
```

Exemple 3: Exemple de programme avec des répètes.

```
Construire (5){
    Maison M1 (1,-1,0)
    Coloriser M1 #Ff0000
    Couleur M1
    %%Affiche la couleur de la maison 1 est (255,0,0).
    Maison M2 (2,-1,-1)
    Coloriser M2 (0,255,0)
    Couleur (2,-1,-1)
    %%Affiche la couleur de la maison 2 est (0,255,0)
    Voisin M1 1
    %%Ajoute un voisin à M1 sur une des positions suivantes (0,0,0),(1,0,-1),(2,-2,0),(1,-2,1)
        ↪,(0,-1,1). Il n'est pas possible de créer une maison en (2,-2,0).
}
```

Exemple 4: Exemple de programme avec des fonctions.

```
Construire{
    Maison M1 (1,-1,0)
    Si (occupe (2,-2,0)){
        Route M1 -> (2,-2,0)
    }
    Sinon {
        Maison (2,-2,0)
        Route M1 -> (2,-2,0)
    }
    %%Ajoute une route entre (1,-1,0) et  (2,-2,0) si une maison est présente en (2,-2,0)
        ↪sinon construit une maison en (2,-2,0) et construit une route entre (1,-1,0) et
        ↪(2,-2,0)
    i=0
    Tant que (i<5) {
        Voisin M1 1
        i=i+1
    }
    %%Construit 5 maisons autour de M1
    j=1
    Repeter i fois {
        Coloriser maison[j] (j*50,0,0)
    }
    %%Colorie les maisons de 1 à 5 en nuance de rouge
}
```

Exemple 5: Exemple de programme avec des expressions arithmétiques.

```
Construire [(rayon)]{
    Maison [nom]
    Maison [nom] (expression,expression)
    Deplacer <nom|(expression,expression)> -> (expression,expression)
    Position <nom|(expression,expression)>
    Tourner <nom|(expression,expression)> <à droite|à gauche>
    Orienter <nom|(expression,expression)> expression$\circ$
    Orientation <nom|(expression,expression)>
    Coloriser <nom|(expression,expression)> <#rrggbb|expression>
    Couleur <nom|(expression,expression)>
    Route <nom|(expression,expression)> -> <nom|(expression,expression)>
    Voisin <nom|(expression,expression)> -> expression
    Voisinage <nom|expression>

    %%Commentaires inline
    %/
        Commentaires sur plusieurs lignes
    /%


    Si (condition) {

    }
    Sinon {

    }

    Tant que (condition) {

    }

}

void nomFonction (arg1,...,argn){
}


%%    ---Bonus---

Pour (initialisation;condition;incrémentation) {
}

type nomFonction (arg1,...,argn){
        Retourne resultat
}
```

Exemple 6: Syntaxe des différents éléments du langage.