Carnet de Travaux Pratiques
Analyse syntaxique
Licence 3 Informatique

Julien Bernard

Table des matières

1 Objectif du TP d'Analyse Syntaxique **2**

2 Description du langage Turtle **2**
2.1 Syntaxe du langage . . . 2
2.1.1 Éléments de base . . . 2
2.1.2 Expressions . . . 2
2.1.3 Commandes simples . . . 3
2.1.4 Bloc de commandes . . . 4
2.1.5 Boucle . . . 4
2.1.6 Variables . . . 4
2.1.7 Procédures . . . 4
2.2 Sémantique du langage . . . 5
2.3 Langage cible . . . 5
2.4 Exemple de programme Turtle . . . 5

3 Travail attendu **6**
3.1 Déroulement des séances . . . 6
3.2 Séance #1 : Flex . . . 6
3.3 Séance #2 : Bison . . . 7
3.4 Séance #3 à #6 : Turtle . . . 8
3.4.1 Exemple . . . 9
3.4.2 Aide . . . 10
3.5 Rendu . . . 10

1 Objectif du TP d'Analyse Syntaxique

Le but du TP d'Analyse Syntaxique est de concevoir et d'implémenter un langage pour faire du dessin, un peu à la manière de la tortue LOGO. Lors des premiers TP, il sera nécessaire de prendre en main les outils qui seront utilisés par la suite. Puis dans un deuxième temps, il faudra implémenter un maximum d'éléments du langage au sein d'un interpréteur qui ciblera des primitives de base.

2 Description du langage Turtle

Le langage qui suit est décrit volontairement de manière non-formalisée. Une partie du travail consiste à formaliser les spécifications données sous la forme d'une grammaire.

2.1 Syntaxe du langage

2.1.1 Éléments de base

Commentaires Un commentaire commence à partir d'un `#` et termine à la fin de la ligne.

Nombre Un nombre est un réel représenté par le type `double` au sein de l'interpréteur. Exemples : `1`, `3.14`, `1e10`, `-2.0`.

Couleur Une couleur est représentée :
- soit par trois nombres compris dans l'intervalle $[0, 1]$, séparés par un ou plusieurs espaces, représentant respectivement les canaux rouge, vert et bleu ;
- soit par un des mots-clefs précisés dans la table 1.

Mot-clef	Équivalent numérique
`red`	`1.0 0.0 0.0`
`green`	`0.0 1.0 0.0`
`blue`	`0.0 0.0 1.0`
`black`	`0.0 0.0 0.0`
`gray`	`0.5 0.5 0.5`
`cyan`	`0.0 1.0 1.0`
`yellow`	`1.0 0.0 1.0`
`magenta`	`1.0 1.0 0.0`

TABLE 1 – Couleurs préfé finies avec un mot-clef

Angle Un angle est représenté par un nombre et est exprimé en degrés.

2.1.2 Expressions

Une expression est :
- soit un nombre ;

— soit l'opération unaire `-` (opposé) ;
— soit une opération binaire parmi `+` (addition), `-` (soustraction), `*` (multiplication), `/` (division), `^` (puissance) ;
— soit une fonction interne ;
— soit un nom de variable ;
— soit une expression entre parenthèses.

Les fonctions internes requises sont :

— `sin` qui prend un paramètre qui représente un angle et qui calcule le sinus de son paramètre ;
— `cos` qui prend un paramètre qui représente un angle et qui calcule le cosinus de son paramètre ;
— `tan` qui prend un paramètre qui représente un angle et qui calcule la tangente de son paramètre ;
— `sqrt` qui prend un paramètre réel et qui calcule la racine carrée de son paramètre ;
— `random` qui prend deux paramètres qui indiquent les bornes de l'invervalle fermé où tirer un nombre au hasard.

Les paramètres des fonctions sont séparés par des virgules et sont entourés de parenthèses. Exemple : `random(1, 3)`

Les expressions ont toutes le type nombre.

2.1.3 Commandes simples

Une commande est une action prédéfinie à laquelle l'utilisateur peut faire appel. Une commande peut avoir zéro, un ou plusieurs paramètres. Les paramètres sont séparés par des espaces. Les paramètres peuvent être des expressions (le plus souvent), des noms de variables ou des commandes. Il n'y a pas de séparateur de commandes.

Commande d'impression La commande `print` prend un paramètre et permet d'afficher le paramètre sur une ligne de la console. Cette commande a pour but de débugguer les programmes Turtle.

Commandes du crayon Les commandes `up` et `down` permettent respectivement de lever et de baisser le crayon virtuel. Ces commandes n'ont pas de paramètres.

Commandes de déplacement La commande `forward` (ou `fw`) et la commande `backward` (ou `bw`) permettent respectivement d'avancer et de reculer le crayon virtuel. Ces commandes prennent un paramètre qui est la distance de déplacement. La commande `position` (ou `pos`) prend deux paramètres et permet de placer le crayon aux coordonnées indiquées par les paramètres (abscisse puis ordonnée).

Commandes d'orientation La commande `right` (ou `rt`) et la commande `left` (ou `lt`) permettent respectivement de tourner à droite et à gauche. Ces commandes prennent un paramètre qui est l'angle de rotation. La commande

`heading` (ou `hd`) prend un paramètre qui indique l'angle absolu pour l'orientation. L'angle 0 pointe vers le nord, et les angles positifs vont dans le sens horaire. Ainsi, l'est est à 90°, le sud à 180° et l'ouest à 270° (ou -90°).

Commande de couleur La commande `color` permet de changer la couleur courante. Elle prend en paramètre une couleur.

2.1.4 Bloc de commandes

Il est possible de grouper des commandes dans un bloc de commandes en les entourant avec des accolades (`{` et `}`). Un bloc de commandes est considéré comme une commande.

2.1.5 Boucle

La commande `repeat` permet de répéter une commande un certain nombre de fois. La commande prend donc deux paramètres. Le premier est un nombre qui indique le nombre de répétitions à faire, en considérant la valeur entière par défaut (donnée par `floor(3)`). Le second est une commande ou un bloc de commande à répéter.

2.1.6 Variables

Il est possible de définir des variables et de leur attribuer une valeur de type nombre. Les noms de variables sont des suites de lettres majuscules et de chiffres, mais doivent nécessairement commencer par une lettre.

Les variables ont une portée globale, même si elles sont définies à l'intérieur d'un bloc de commande.

La commande `set` permet d'attribuer une valeur à une variable. Elle prend deux paramètres. Le premier paramètre est le nom d'une variable. Le second paramètre est une expression dont le résultat est affecté immédiatement à la variable.

Il existe trois variables prédéfinies :

— `PI` qui a pour valeur 3.14159265358979323846 ;
— `SQRT2` qui a pour valeur 1.41421356237309504880 ;
— `SQRT3` qui a pour valeur 1.7320508075688772935.

2.1.7 Procédures

Il est possible de définir des procédures. Une procédure est un alias pour une commande ou un bloc de commandes. Une procédure n'a pas de paramètres. Les noms de procédures suivent les mêmes règles que les noms de variables.

La commande `proc` permet de définir une procédure. Elle prend en paramètre un nom et une commande. On ne doit pas pouvoir définir de procédure à l'intérieur d'une procédure.

La commande `call` permet d'appeler une procédure. Elle prend en paramètre le nom d'une procédure précédemment définie.

2.2 Sémantique du langage

Fichier Un fichier contient une suite de commandes qui sont exécutées séquentiellement. Toute erreur arrête l'interprétation du fichier instantanément avec un message d'erreur le plus explicite possible sur l'erreur standard et un code de retour non-nul.

Conditions de départ Au départ, l'orientation est au nord et le crayon se situe aux coordonnées $(0, 0)$ et est baissé. La couleur courante est le noir.

Portée des variables Une variable a une portée globale. Elle peut être utilisée dès qu'elle a été définie avec `set`. L'utilisation d'une variable non-définie provoque une erreur.

Portée des procédures Une procédure a une portée globale. Elle peut être appelée avec `call` dès qu'elle a été définie avec `proc`. L'utilisation d'une procédure non-définie provoque une erreur. La redéfinition d'une procédure déjà définie provoque une erreur.

Non-respect des pré-conditions Le non-respect d'une pré-conditon d'une des fonctions internes ou des opérateurs provoque une erreur. Parmi le non-respect des pré-conditions, on trouve : une racine carrée d'un nombre négatif, un intervalle non-valide pour `random`, des paramètres en dehors du domaine de définition pour l'opérateur puissance.

2.3 Langage cible

L'interpréteur du langage doit produire sur sa sortie standard une suite de primitves de base. Elles sont au nombre de trois :

- `Color` change la couleur courante ;
- `MoveTo` change la position courante par la positon passée en paramètre ;
- `LineTo` trace une ligne entre la position courante et la position passée en paramètre qui devient alors la nouvelle position courante.

Au départ, la position courante est $(0, 0)$ et la couleur courante est le noir. L'écran virtuel est situé sur le domaine $[-500, 500] \times [-500, 500]$. Sortir du domaine ne provoque pas d'erreur (mais rien n'est affiché en dehors du domaine). L'axe des x est orienté de gauche à droite, l'axe des y est orienté de haut en bas.

2.4 Exemple de programme Turtle

Voici un exemple de programme Turtle très simple qui trace un triangle avec une couleur pour chaque côté.

```
color red
fw 100
right 120
color green
fw 100
right 120
```

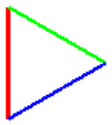

FIGURE 1 – Résultat obtenu en exécutant l'exemple

```
color blue
fw 100
right 120
```

Voici un résultat possible de l'interprétation du fichier précédent.

```
Color 1.000000 0.000000 0.000000
LineTo 0.000000 -100.000000
Color 0.000000 1.000000 0.000000
LineTo 86.602540 -50.000000
Color 0.000000 0.000000 1.000000
LineTo 0.000000 0.000000
```

Le dessin produit par ce programme est donnée dans la figure 1.

3 Travail attendu

3.1 Déroulement des séances

Ce projet se déroule sur six séances. Les deux premières séances sont consacrés aux outils, `flex` et `bison`, qui seront utilisés par la suite. Les quatre séances restantes sont dédiées à la réalisation de l'interpréteur proprement dit. Le planning qui suit est purement indicatif et vous permet de situer votre avancement globalement. Les questions des TP vous guident dans votre apprentissage et dans votre compréhension des outils.

Des archives vous sont fournies pour vous aider à démarrer. Ces archives contiennent des codes sources et un fichier `CMakeLists.txt` pour construire le programme. Pour cela, une fois dans le répertoire des sources, vous devez créer un répertoire `build`, puis y entrer et exécuter `cmake` qui va créer un `Makefile`, et enfin construire le programme via la commande `make`.

```
mkdir build
cd build
cmake ..
make
```

3.2 Séance #1 : Flex

Flex est un analyseur lexical. Il permet de créer une suite de *tokens* qui seront analysés par l'analyseur syntaxique. Il prend en entrée une description

des *tokens* via des expressions régulières et renvoie un fichier C contenant une fonction `yylex()` qui réalise l'analyse.

Le manuel de Flex se trouve à l'URL :

`https://westes.github.io/flex/manual/`

Il est nécessaire de lire ce manuel jusqu'à la section «8 Actions» incluse. Votre attention sera en particulier attirée par la section «5 Format of the Input File».

Puis, après avoir téléchargé l'archive `tp1-flex.tar.gz`, vous compilerez les fichiers comme indiqué précédemment. L'exécutable `flex-test` analyse son entrée standard et affiche le résultat sur la sortie standard.

```
echo "0123 toto42" | ./flex-test
```

Question 1 Définir une règle pour reconnaître le mot-clef `true`.

Question 2 Définir une règle pour reconnaître les nombres entiers (attention, un entier ne peut pas commencer par 0, sauf le nombre 0). Exemples : `0`, `123`, `10000`

Question 3 Définir une règle pour reconnaître les nombres hexadécimaux. Exemples : `0x0`, `0x29A`, `0xDeadBeef`.

Question 4 Définir une règle pour reconnaître les flottants. Exemples : `500.`, `500.0`, `5e2`, `5E2`, `5e+2`, `0.05`, `.05`, `5e-2`, `5.0e-2`.

Question 5 Définir une règle pour reconnaître les identifiants des variables et des procédures de Turtle. Exemples : `I`, `PI`, `CIRCLE`, `POLY5`.

Question 6 Définir une règle pour reconnaître les identifiants du langage C. Exemples : `foo`, `foo2`, `_foo`, `foo_bar`, `Foo42Bar43`.

Question 7 L'ordre des règles a-t-il une importance ?

Question 8 À quoi sert la règle `[\n\t ]*` ?

Question 9 À quoi sert la règle `.` placée en dernière position ?

3.3 Séance #2 : Bison

Bison est un analyseur syntaxique. Plus précisément, c'est un générateur d'analyseur syntaxique, appelé aussi compilateur de compilateur. Il prend en entrées une suite de *tokens* fournis par Flex et réalise une analyse. Le développeur choisit le résultat de cette analyse. Dans ce TP d'introduction, vous évaluerez directement le résultat. Dans la suite du projet, vous construirez un arbre de syntaxe abstrait.

Le manuel de Bison se trouve à l'URL :

`https://www.gnu.org/software/bison/manual/html_node/`

Il est nécessaire de lire ce manuel jusqu'à la section «5.3 Operator Precedence» incluse.

Puis, après avoir téléchargé l'archive `tp2-bison.tar.gz`, vous compilerez les fichiers comme indiqué précédemment. L'exécutable `bison-test` analyse son entrée standard et affiche le résultat sur la sortie standard.

```
echo "3 * PI" | ./bison-test
```

La compilation génère un fichier `parser.output` qui contient l'automate des items LR(0) (vu au CM #3). En cas d'erreur dans la grammaire, un message d'erreur indique succinctement le nombre de conflits de type *shift/reduce* et de type *reduce/reduce* et le détail des conflits est alors détaillé dans ce fichier `parser.output`.

Question 10 Où est défini `NAME` utilisé dans `lexer.l` ?

Question 11 Que signifie `%left '+' '-'` ?

Question 12 L'ordre entre les deux directives `%left` a-t-il une importance ?

Question 13 Quel est l'arbre obtenu avec l'expression $1 + 2 + 3 + 4 * 5 * 6$? Modifier `parser.y` pour le vérifier en ajoutant des traces.

Question 14 Ajouter la gestion des parenthèses. Pour cela, il faut ajouter des *tokens* dans l'analyseur lexical puis ajouter une règle dans la grammaire.

Question 15 Ajouter un mot-clef `QUIT` qui permet de quitter l'application.

Question 16 Ajouter la gestion du moins unaire. Attention à sa précédence !

Question 17 Ajouter la gestion des commentaires au niveau de l'analyseur lexical. Les commentaires ne produisent pas de *tokens*.

Question 18 La grammaire est ambiguë. Pourquoi cela ne pose-t-il pas de problème ? Réécrire une grammaire non-ambiguë en prenant exemple sur la grammaire vue en cours à de nombreuses reprises. Quelle associativité est préférable ? Quelle solution vous paraît être la plus facile à mettre en œuvre ?

3.4 Séance #3 à #6 : Turtle

À partir de là, vous devez concevoir et coder un interpréteur pour Turtle. L'archive `turtle.tar.gz` peut vous servir de base de travail. Attention, elle contient des erreurs que vous pourrez corriger facilement si vous avez fait correctement les deux premiers TP.

Il est conseillé de procéder comme suit :

1. concevoir la grammaire, en particulier identifier les *tokens* nécessaires, mais ne pas créer d'actions ;

2. concevoir l'analyseur lexical pour tous les *tokens* ;
3. générer un arbre de syntaxe abstrait correct, ce qui implique de coder toutes les fonctions relatives à cette structure ;
4. faire une fonction qui affiche l'arbre de syntaxe abstrait comme un programme Turtle ;
5. faire une fonction qui évalue l'arbre de syntaxe abstrait et renvoie une suite de primitives de base.

De plus, dans un premier temps, vous pouvez mettre de côté les procédures et les variables. Puis, dans un second temps, vous ajouterez les procédures, puis les variables.

3.4.1 Exemple

À partir des fichiers fournis, voici comment procéder pour ajouter la commande `forward` à la grammaire et générer le nœud correspondant dans l'arbre de syntaxe abstrait.

Premièrement, il faut reconnaître le mot-clef `forward` et ajouter le token correspondant. Pour cela, on ajoute une ligne dans `turtle-parser.y` après la définition du token `NAME` :

```
%token             KW_FORWARD  "forward"
```

Puis, on reconnaît ce token dans l'analyseur lexical. On ajoute la ligne suivante dans `turtle-lexer.l`, juste après le premier `%%` :

```
"forward"              { return KW_FORWARD; }
```

Deuxièmement, il faut ajouter une règle dans la grammaire pour reconnaître la commande. Dans `turtle-parser.y`, on ajoute la commande dans la règle `cmd` :

```
cmd:
    KW_FORWARD expr   { /* ... */ }
;
```

Troisièmement, il faut remplir l'action associée à cette règle, c'est-à-dire générer le nœud adéquat. On ajoute un constructeur dans le fichier `turtle-ast.h` :

```
struct ast_node *make_cmd_forward(struct ast_node *expr);
```

Et on ajoute son implémentation dans `turtle-ast.c` :

```
struct ast_node *make_cmd_forward(struct ast_node *expr) {
  struct ast_node *node = calloc(1, sizeof(struct ast_node));
  node->kind = KIND_CMD_SIMPLE;
  node->u.cmd = CMD_FORWARD;
  node->children_count = 1;
  node->children[0] = expr;
  return node;
}
```

Enfin, on complète l'action de la règle qu'on a précédemment créée dans `turtle-parser.y` ;

```
cmd:
    KW_FORWARD expr   { $$ = make_cmd_forward($2); }
;
```

3.4.2 Aide

Si vous avez des questions générales sur le sujet, vous utiliserez le forum prévu à cet effet. Si vous êtes bloqué, une bonne idée est de trouver de l'inspiration dans la grammaire du C :

— Flex : `http://www.quut.com/c/ANSI-C-grammar-l-2011.html`
— Bison : `http://www.quut.com/c/ANSI-C-grammar-y-2011.html`

3.5 Rendu

Vous devrez réaliser un **rapport** d'une dizaine de pages expliquant ce qui a été fait, quels choix vous avez fait pour votre grammaire, comment vous avez implémenté les procédures et les variables, etc. De plus, vous aurez à rendre votre **code source** qui devra être capable de produire un exécutable `turtle`, ainsi que des **fichiers Turtle** pour montrer les capacités de votre programme. La date de rendu sera précisé ultérieurement.