TD/TP 10 : interprète de mini-Logo

Gilles Peskine

13 décembre 2002

1 Dessin à la tortue

Dans le langage Logo, on peut afficher des dessins à l'aide d'une *tortue*. À tout instant, la tortue a une *position* (coordonnées x et y du pixel où est la tortue) et un *cap* (direction vers laquelle pointe la tortue).

Voici un extrait du fichier `tortue.h`.

```
void avance (float d);
void tourne_droite (float a);
void tourne_gauche (float a);
void regle_position (float x, float y);
void regle_cap (float a);
void regle_crayon (int i);
```

Écrire ces fonctions à l'aide des fonctions de la librairie `graphics`. Ces fonctions seront placées dans un fichier `tortue.c`.

On utilisera les fonctions suivante de la bibliothèque `graphics` :

```
void gr_plot(int x, int y);
void gr_moveto(int x, int y);
```

2 Découpage d'une chaîne en mots

Écrire une fonction qui découpe une chaîne de caractères en mots (séparés par un ou plusieurs espaces, sauts de lignes, etc). Cette fonction devra renvoyer un pointeur vers un tableau de pointeurs vers des chaînes de caractères (les mots en question). Utiliser un pointeur nul pour marquer la fin du tableau.

```
char **decoupe(char *s);
```

C'est (en simplifié) le travail qu'effectue le shell pour passer d'une ligne de commande à `argv`.

3 Syntaxe du langage

On voudrait pouvoir écrire à la volée des commandes de dessin à la tortue. Par exemple, le programme mini-Logo suivant dessine un carré :

```
avance 100
td 90
avance 100
td 90
avance 100
td 90
avance 100
td 90
```

(`avance 100` fait avancer la tortue de 100 pas, et `td` la fait tourner à droite de 90°.)

La syntaxe de mini-Logo est la suivante : chaque ligne contient une instruction, puis ses arguments (pour l'instant, des nombres), avec des espaces comme séparation. En outre, si une ligne contient un signe `#` au début ou après un espace, ce signe débute un commentaire qui se termine à la fin de la ligne.

On définit une variable qui contient une description des instructions autorisées.

```
struct commande {
    char *nom; /* nom de l'instruction */
    int arite; /* nombre d'arguments */
};
struct commande commandes[] = {
    {"avance", 2},
    {"td", 1},
    {"tg", 1},
    ...
};
```

Écrire un vérificateur de syntaxe mini-Logo : il doit rejeter les lignes incorrectes (instruction inconnue, mauvais nombre d'arguments, arguments non numériques).

Pour lire les nombres à virgule flottante, on dispose de la fonction `strtod` (déclarée dans `stdlib.h`) :

```
double strtod(char *s, char **endp);
```

La fonction `strtod` lit un nombre réel depuis `s`, et place dans `*endp` l'adresse du premier caractère qui ne fait pas partie du nombre.

4 Organisation de l'interprète

Comme l'interprète mini-Logo est constitué de plusieurs parties qui n'ont pas grand-chose à voir entre elles, nous allons le découper en plusieurs fichiers :

- `tortue.c` pour la manipulation de la tortue.
- `tortue.h` qui déclare les fonctions de manipulation de la tortue.
- `texte.c` qui contient les fonctions d'utilité générale : `getline` (voir TD 7), `decoupe`, etc.
- `texte.h` qui déclare les fonctions d'utilité générale.
- `interprete.c` qui contient l'interprète mini-Logo proprement dit.
- `interprete.h` qui déclare les fonctions exportées par l'interprète.
- `main.c` qui contient essentiellement une fonction `main` qui ouvre la fenêtre graphique et lance l'interprète.

Écrire les fichiers d'en-têtes. (Au fur et à mesure qu'on étoffera le programme, il faudra les compléter.)

Quels en-têtes faut-il inclure dans les différents fichiers `.c` ?

Écrire un `Makefile` pour l'interprète. On appellera le programme compilé `minilogo`.

5 L'interprète

Modifier `struct commande` (cf. exercice 3) pour qu'elle contienne suffisamment d'informations pour écrire l'interprète.

Écrire une fonction qui interprète une ligne de mini-Logo, puis une fonction qui interprète un programme mini-Logo qu'elle lit au fur et à mesure.

Écrire `main.c`.

```
interprete_instruction(char *instruction);
interprete_fichier(FILE *f);
```

6 Constantes

On souhaite utiliser des constantes nommées en mini-Logo. Par exemple, si on a une constante prédéfinie nommée `cent` dont la valeur est le nombre 100, on veut pouvoir écrire :

```
avance cent
```

Modifier l'interprète pour qu'il accepte qu'une instruction ait pour argument un nom de constante. Ajouter une table des constantes : `cent`, `pi`... Si un programme mini-Logo fait référence à un nom de constante qui n'est pas dans la table, afficher un message d'erreur. Pour l'instant, les constantes doivent être définies à l'intérieur de l'interprète.

7 Variables

On voudrait pouvoir définir des constantes directement en mini-Logo. De fait, il n'est pas plus difficile d'en faire des variables.

Ajouter à l'interprète une instruction `donne` telle que, par exemple,

```
donne cent 100
avance cent
```

fasse avancer la tortue de 100 pas.

En quoi l'instruction `donne` est-elle spéciale ?

8 Calculs

Disposant de variables numériques, aussitôt l'envie nous prend de pouvoir faire des calculs dessus.

Pour ne pas trop compliquer l'analyse syntaxique, on peut avoir des instructions `ajoute`, `soustrait`, `multiplie`, `divise` qui prennent deux arguments, effectuent l'opération arithmétique, et placent le résultat dans une variable spéciale appelée `resultat`. Ainsi pour avancer de 100 pas on pourra faire :

```
donne dix 10
multiplie dix dix
donne cent resultat
avance cent
```

Une autre syntaxe possible, plus agréable pour l'utilisateur de mini-Logo, mais plus difficile à coder dans l'interprète, est de permettre de faire des calculs dans l'argument de l'instruction `donne`. Par exemple :

```
donne dix 10
donne cent dix * dix
avance cent
```

Il peut être plus facile d'avoir deux instructions séparées, une pour une affectation simple, l'autre pour une affectation du résultat d'un calcul. Par exemple :

```
donne dix 10
calcule cent 10 * 10
avance cent
```

9 Instructions conditionnelles

Ajouter au mini-Logo un moyen d'exécuter ou non une instruction selon si une condition est vérifiée ou non. Par exemple :

```
si_egal x 3 avance 100
```

fait avancer la tortue de 100 pas, mais seulement si la variable x vaut 3.

10 Pour aller plus loin

Note : ces exercices sont plus difficiles...

10.1 Boucles simples

Ajouter au mini-Logo une instruction `repeter` qui permet d'exécuter une séquence d'instructions plusieurs fois. Par exemple, pour dessiner un carré :

```
repeter 4
  avance 100
  td 90
fin
```

On pourrait même avoir des boucles imbriquées, par exemple pour dessiner 3 carrés :

```
repeter 3
  repeter 4
    avance 100
    td 90
  fin
  avance 200
fin
```

Dans un premier temps, on pourra supposer que les boucles ne sont pas imbriquées.

10.2 Procédures

Une procédure est un sous-programme qui peut être appelé par son nom. Par exemple, voici un programme qui dessine 3 carrés en utilisant un sous-programme pour dessiner un carré :

```
pour carre
  repeter 4
    avance 100
    td 90
  fin
fin
carre
avance 200
carre
avance 200
carre
avance 200
```

10.3 Procédures avec arguments

Ajouter à l'interpréteur une possibilité pour que les procédures prennent des arguments. Par exemple, la procédure carré pourrait prendre un argument qui est la longueur du côté :

```
pour carre cote
  repeter 4
    avance cote
    td 90
  fin
fin
```