Projet

Un interpréteur pour la création d'images

Version du 12 mars 2021

Il existe de nombreux logiciels permettant de créer des images, par exemple, Paint, Gimp, Photoshop. Pour tous ces exemples, l'utilisateur transmet ses instructions à la souris. La souris offre une prise en main intuitive mais ne permet pas d'automatiser les taches que l'utilisateur souhaite effectuer ni de communiquer avec d'autres programmes.

Dans ce projet vous allez concevoir et implanter un interpréteur pour la création d'images. L'interpréteur va prendre ses instructions dans un fichier texte et dessiner à l'écran une image construite à partir de ces instructions. Les instructions permettent de déplacer verticalement ou horizontalement un curseur virtuel. Si le curseur est en position basse chaque déplacement laisse une trace noire, s'il est en position haute le déplacement n'écrit rien. Les instructions permettent aussi de tourner le curseur dans le sens trigonométrique (c'est-à-dire dans le sens inverse de rotation des aiguilles d'une montre), par an angle donné en degrés.

La grammaire La grammaire du langage est la suivante, les non-terminaux sont en gras et en italiques, les terminaux (jetons) en police `code` :

- ***programme*** → ***declarations instruction***
- ***declarations*** → `Var identificateur ;` ***declarations*** | ε
- ***instruction*** → `Avance` ***expression*** | `Tourne` ***expression*** | `BasPinceau` | `HautPinceau` | `identificateur =` ***expression*** | `Debut` ***blocInstruction*** `Fin`
- ***blocInstruction*** → ***instruction*** `;` ***blocInstruction*** | ε
- ***expression*** → `nombre` ***expressionSuite*** | `identificateur` ***expressionSuite*** | `(` ***expression*** `)` ***expressionSuite***
- ***expressionSuite*** → `+` ***expression*** | `-` ***expression*** | ε

avec les définitions suivantes de jetons :

- `identificateur` → [a-z][a-zA-Z0-9]*
- `nombre` → [1-9][0-9]* | 0

Exemple Le programme suivant dessine ainsi un carré dont les côtés ont longueur 10.

```
Var angle;
Debut
angle = 0 - 90;
BasPinceau;
Avance 10;
Tourne angle;
Avance 10;
Tourne angle;
Avance 10;
```

```
Tourne angle;
Avance 10;
Fin
```

1 Première Étape

Il est conseillé d'écrire le programme en plusieurs étapes de difficulté croissante. Dans un premier temps, écrivez un analyseur lexical et syntaxique afin de construire l'arbre de syntaxe abstraite d'un programme décrit par la grammaire ci-dessus. Ensuite, écrivez un interpréteur pour l'arbre construit.

Il faudra veiller à ce que toute variable utilisée soit bien déclarée au début du programme et vérifier que chaque variable est déclarée au plus une fois.

Nous choisissons aussi que le curseur est initalement placé aux coordonnées (0,0), c'est-à-dire un bas à gauche du canvas graphique, en orientation nord (c'est-à-dire vers le haut du canvas), et le pinceau en position haute.

Gestion des erreurs Votre programme doit traiter proprement toutes les erreurs qui peuvent se produire pendant l'interprétation, c'est-à-dire afficher un message explicite, détaillant l'erreur rencontrée, et puis terminer l'interpréteur. Les erreurs à capturer sont :

— division par 0 ;

— le curseur sort du canevas.

2 Deuxième Étape

Ajoutez des instructions de type

`Si` ***expression*** `Alors` ***instruction*** `Sinon` ***instruction***

et

`Tant que` ***expression*** `Faire` ***instruction***

produites par le non-terminal ***instruction***. Nous choisissons pour l'instruction `Si Alors Sinon` que l'instruction après le `Alors` est exécutée seulement si l'expression est différente de 0. Nous choisissons aussi pour l'instruction `Tant que` que la boucle s'arrête si l'expression est égale à 0 .

Vérifiez tout d'abord que la grammaire est bien LR(1) et modifiez-la le cas échéant. Modifiez votre code afin d'intégrer la nouvelle grammaire à votre interpréteur.

3 Troisième Étape

Ajouter à la définition des expressions les opérateurs de multiplication $*$ et de division $/$, ainsi que le moins unaire. La multiplication et la division doivent prendre la priorité sur l'addition et la soustraction, c'est-à-dire une expression comme $1+3*-5$ doit être lue comme $1+(3*(-5))$.

4 Quatrième Étape

Nous proposons quelques pistes pour des extensions de votre programme. À vous de choisir les extensions que vous souhaitez réaliser :

- Ajouter une instruction permettant de modifier l'épaisseur des traits et leur couleur.
- Modifier la grammaire afin d'autoriser en plus les conditions

  `Si` ***expression*** `Alors` ***instruction***

  sans `Sinon` derrière. Attention, assurez-vous qu'elle soit LR(1).
- Ajouter un mode interprété où l'utilisateur peut entrer des commandes et voir en même temps les résultats de ces commandes dans une fenêtre graphique.

5 Rendu

Le langage de programmation utilisé doit être Ocaml, l'analyse lexicale doit être realisée avec *ocamllex*, et l'analyse grammaticale avec *menhir*.

Concernant l'affichage, nous recommandons sans l'imposer l'utilisation du module Graphics (en cas d'usage d'une autre bibliothèque, adapter votre documentation (cf section suivante) en fonction). Il est conseillé de faire le projet en binôme (mais pas de trinômes !) Le travail doit être réalisé tout au long de la suite du semestre via un dépôt git pour chaque groupe. L'usage de ce gestionnaire de version permet d'éviter toute perte de donnée, et d'accéder si besoin à vos anciennes versions.

- Votre dépôt git doit être hébergé par le serveur GitLab de l'UFR d'informatique : `https://gaufre.informatique.univ-paris-diderot.fr`
- Votre dépôt doit être rendu privé dès sa création, avec accès uniquement aux membres du groupe et aux enseignants de ce cours. Tout code laissé en accès libre sur Gaufre ou ailleurs sera considéré comme une incitation à la fraude, et sanctionné.
- Il va de soi que votre travail doit être strictement personnel : aucune communication de code ou d'"idées" entre les groupes, aucune "aide" externe ou entre groupes. Nous vous rappelons que la fraude à un projet est aussi une fraude à un examen, passible de sanctions disciplinaires pouvant aller jusqu'à l'exclusion définitive de toute université.
- Des fichiers pouvant servir de tests seront mis le 16 mars sur le dépôt git.

6 Rapport et Documentation

Il est nécessaire que vous indiquiez dans fichiez `LisezMoi.txt` les commandes qu'il faut lancer pour compiler et exécuter votre programme. De plus, il vous est demandé de brièvement expliquer (dans le fichier `LisezMoi.txt` ou dans un fichier séparé) quelles sont exactement les parties du sujet que vous avez réalisées, et dans le cas où vous travaillez en binôme comment vous avez organisé le travail dans le binôme.

Annexe : Usage de GitLab

La création d'un depôt git sur le GitLab de l'UFR permettra à votre groupe de disposer d'un dépôt commun de fichiers sur ce serveur. Chaque membre du groupe pourra ensuite disposer d'une copie locale de ces fichiers sur sa machine, les faire évoluer, puis sauvegarder les changements jugées intéressants et les synchroniser sur le serveur. Si vous ne vous êtes pas déjà servi de GitLab dans d'autres matières, cette section décrit son usage le plus élémentaire. Pour plus d'informations sur git et GitLab, il existe de multiples tutoriels en ligne. Nous contacter en cas de problèmes.

Accès au serveur et configuration personnelle. Se connecter via l'interface web : `https://gaufre.informatique.univ-paris-diderot.fr`. Utilisez pour cela les mêmes nom et mots de passe que sur les machines de l'UFR, et pas vos compte "ENT" de paris diderot ou u-paris. Cliquer ensuite sur l'icone en haut à droite, puis sur "Settings". A droite, aller ensuite dans la section "SSH Keys", et ajouter ici la partie public de votre clé ssh (ou de vos clés si vous en avez plusieurs). Cela permettra facilitera grandement l'accès ultérieur à votre dépôt git, et vous évitera de taper votre mot de passe à chaque action.

Si vous n'avez pas encore de clé ssh, s'en générer une sur sa machine. L'usage de ssh n'est pas spécifique à git et GitLab, et permet des connections "shell" à des machines distantes. Si vous n'utilisez pas encore ssh et les clés publiques/privées ssh, il est temps de s'y mettre ! Pour plus d'information sur ssh, consulter :

`http://www.informatique.univ-paris-diderot.fr/wiki/doku.php/wiki/howto_connect`

Création du dépôt. Pour ce projet, nous vous fournissons quelques fichiers de tests. Votre dépôt git sera donc un dérivé (ou "fork") du dépôt publique du cours. En pratique :

- L'un des membres de votre groupe se rend sur la page du cours à partir du 16 Mars : `https://gaufre.informatique.univ-paris-diderot.fr/bauer/gasp/` s'identifie si ce n'est pas déjà fait, et appuie sur le bouton "fork" (vers le haut, entre "Star" et "Clone"). Attention, un seul "fork" par groupe suffit.
- Ensuite, aller dans la section "Settings" en bas à gauche, défiler un peu et cliquer sur "Visibility", et selectionner "Private" comme "projet visibility", puis "Save changes" un peu plus bas. Vérifier qu'un cadenas apparaît maintenant à côté de `gasp` quand vous cliquez sur "Projet" en haut à gauche.
- Toujours dans "Settings" en bas à gauche, mais sous-section "Members" maintenant. "Invitez" votre collègue de projet, ainsi que les logins `habermeh`, `bauer`, `habermeh`, `klimann`, `picatin` en choissisant "Maintainer" comme rôle.
- Voilà, votre dépôt sur le GitLab est prêt !

Création et synchronisation de vos copies locales de travail. Chaque membre du projet "clone" le dépôt du projet sur sa propre machine, c'est-à-dire en télécharge une copie locale : `git clone` suivi de l'addresse du projet tel qu'il apparaît dans l'onglet "Clone" sur la page du projet, champ en "SSH". Pour cela, il faut avoir installé `git` et `ssh` et configuré au moins une clé ssh dans GitLab.

Une fois le dépôt créé et cloné et en se plaçant dans le répertoire du dépôt, chaque membre pourra à tout moment :

- télécharger en local la version la plus récente du dépôt distant sur Gitlab : `git pull`.

— téléverser sa copie locale modifiée sur GitLab :
`git push`.

Avant toute synchronisation, il est demandé d'avoir une copie locale "propre" (où toutes les modifications sont enregistrées dans des "commits").

Modifications du dépôt : les commits. Un dépôt Git est un répertoire dont on peut sauvegarder l'historique des modifications. Chaque action de sauvegarde est appelée une *révision* ou "commit". L'*index* du dépôt est l'ensemble des modifications qui seront sauvegardées à la prochaine révision. La commande

`git add` suivi du nom d'un ou plusieurs fichiers

permet d'ajouter à l'index toutes les modifications faites sur ces fichiers. Si l'un d'eux vient d'être crée, on ajoute dans ce cas à l'index l'opération d'ajout de ce fichier au dépôt. La même commande suivie d'un nom de répertoire ajoute à l'index l'opération d'ajout du répertoire et de son contenu au dépôt. La révision effective du dépôt se fait par la commande

`git commit -m` suivi d'un message entre guillemets doubles.

Invocable à tout instant, la commande

```
git status
```

permet d'afficher l'état courant du dépôt depuis sa dernière révision : quels fichiers ont été modifiés, renommés, effacés, crées, etc., et lesquelles de ces modifications sont dans l'index. Elle indique également comment rétablir l'état d'un fichier à celui de la dernière révision, ce qui est utile en cas de fausse manœuvre.

Les commandes `git mv` et `git rm` se comportent comme `mv` et `rm`, mais ajoutent immédiatement les modifications associées du répertoire à l'index.

Il est conseillé d'installer et d'utiliser les interfaces graphiques `gitk` (visualisation de l'arbre des commits) et `git gui` (aide à la création de commits).

Une dernière chose : git est là pour vous aider à organiser et archiver vos divers fichiers sources. Par contre il vaut mieux ne *pas* y enregistrer les fichiers issues de compilations (binaires, répertoire temporaire tels que `_build` pour `dune`, fichiers objets OCaml `*.cm{o,x,a}`, etc).

Les fusions (merge) et les conflits. Si vous êtes plusieurs à modifier vos dépôts locaux chacun de votre côté, celui qui se synchronisera en second avec votre dépôt GitLab commun aura une manoeuvre nommé "merge" à effectuer. Tant que vos modifications respectives concernent des fichiers ou des zones de code différentes, ce "merge" est aisé, il suffit d'accepter ce que git propose, en personnalisant éventuellement le message de merge. Si par contre les modifications se chevauchent et sont incompatibles, il y a alors un conflit, et git vous demande d'aller décider quelle version est à garder. Divers outils peuvent aider lors de cette opération, mais au plus basique il s'agit d'aller éditer les zones entre `<<<<<` et `>>>>>` puis faire `git add` et `git commit` de nouveau.

Intégrer les modifications venant du dépôt du cours. Si le dépôt du cours reçoit ultérieurement des correctifs ou des évolution des fichiers fournis pour le projet, ces modifications peuvent être intégrés à vos dépôts.

— La première fois, aller dans votre répertoire de travail sur votre machine, et taper :
`git remote add prof \`
`git@gaufre.informatique.univ-paris-diderot.fr:bauer/gasp.git`
— Ensuite, à chaque fois que vous souhaiterez récupérer des commits du dépôt du cours :
`git pull prof master`
— Selon les modifications récupérées et les votres entre-temps, cela peut occasionner une opération de "merge" comme décrite auparavant.
— Enfin, ces modifications sont maintenant intégrés à votre copie locale de travail, il ne reste plus qu'à les transmettre également à votre dépôt sur GitLab :
`git push`

Les branches. Il est parfois pratique de pouvoir essayer différentes choses, même incompatibles. Pour cela, Git permet de travailler sur plusieurs exemplaires d'un même dépôt, des *branches*. Un dépôt contient toujours une branche principale, la branche "master", dont le rôle est en principe de contenir sa dernière version stable. Les autres branches peuvent servir à développer des variantes de la branche master, par exemple pour tenter de corriger un bug sans altérer cette version de référence. La création d'une nouvelle branche, copie conforme de la branche courante – initialement, master – dans son état courant, se fait par :

`git branch` suivi du nom choisi pour la branche.

Sans arguments, cette commande indique la liste des branches existantes, ainsi que celle dans laquelle se trouve l'utilisateur. Le passage à une branche se fait par

`git checkout` suivi du nom de la branche.

Pour ajouter au dépôt distant une branche qui n'est pas encore sur celui-ci, après s'être placé dans la branche :

`git push -set-upstream origin` suivi du nom de la branche

Un `push` depuis une branche déjà sur le serveur se fait de la manière habituelle. Enfin, on peut "réunifier" deux branches avec `git merge`, voir la documentation pour plus de détails.

Noter que GitLab propose également un mécanisme de "Merge Request" : il permet de proposer des modifications, soit à son propre projet, soit au projet qui a été "forké" à l'origine, les membres du projet en question pouvant alors accepter ou non ces suggestions après discussion.