Projet

Un interpréteur pour un langage de programmation simple

Version du 22 mars 2024
Date limite : 20 mai 2024 à midi

1 Sujet

Le but du projet est d'implémenter un interpréteur pour le langage LATSI (Langage très simple d'instructions). Une partie de la grammaire de LATSI en forme EBNF (voir Cours 3) est donnée dans la figure 1 (EOF signifie fin de fichier et CR retour à la ligne). On rappelle que {...} signifie la répétition et [...] un groupe optionnel. Dans la grammaire, le symbole `"("` désigne juste `(`. Pareil pour `")"` qui désigne `)`. Les autres parenthèses ne désignent pas des terminaux. Par contre dans la définition de `<string>`, le symbole " est utilisé comme terminal. Les caractères autorisés entre deux guillemets sont précisés entre les crochets. (le blanc est inclus).

```
<programme>    : :=  {<ligne>} EOF
<ligne>        : :=  <nombre> <instr> CR
<instr>        : :=  IMPRIME <expr-list>
                     | SI <expression> <relop> <expression> ALORS <instr>
                     | VAVERS <expression>
                     | ENTREE <var-list>
                     | <var> = <expression>
                     | FIN
                     | REM <string>
<expr-list>    : :=  (<string>| <expression>) {, (<string>| <expression>) }
<var-list>     : :=  <var> {, <var>}
<expression>   : :=  [+|-] <term> {(+|-) <term>}
<term>         : :=  <facteur> {(*|/) <facteur>}
<facteur>      : :=  <var> | <nombre> | "("<expression>")"
```

`<var>`, `<nombre>`, `<chiffre>`, `<relop>` et `<string>` sont donnés par les expressions rationnelles suivantes :

```
<var>       : :=  A | B | C ... | Y | Z
<nombre>    : :=  <chiffre> <chiffre>*
<chiffre>   : :=  0 | 1 | 2 | 3 | ... | 8 | 9
<relop>     : :=  <(>|=| ϵ) | >(<|=| ϵ) | =
<string>    : :=  "[  ,'_ ; :().a-zA-Z]*"
```

FIGURE 1 – La syntaxe de LATSI

Un programme simple est donné ci-dessous :

```
5  REM "Ce programme est formidable."
10 IMPRIME "Bonjour Paris"
20 I = 0
30 SI I > 10 ALORS VAVERS 40
35 IMPRIME I, " "
37 I = I + 1
39 VAVERS 30
40 FIN
50 IMPRIME "ne s'imprime pas"
```

2 Sémantique de LATSI

Un programme peut être vide mais déclenche une erreur à l'exécution. Quand un programme LATSI contient plusieurs lignes avec le même numéro, la dernière ligne est utilisée. Les lignes d'un programme ne sont pas forcément ordonnées par leur numéro. L'exécution d'un programme LATSI commence par la ligne avec le plus petit numéro. La ligne est exécutée (voir ci-dessous une description de chaque instruction) et on passe soit à la ligne non vide suivante, soit à la ligne indiquée par l'instruction VAVERS. Si, à la fin, il n'y a plus de ligne non vide, le programme s'arrête.

2.1 Variables

LATSI a 26 variables prédéfinies. Ces variables sont de type entier. Chaque valeur est initialisée avec 0.

2.2 Instructions

- IMPRIME $e_1, e_2, \ldots, e_k$ affiche sur l'écran la valeur des expressions e_1 à e_k. Par exemple, IMPRIME $3 + 5$, $19 + 1$,`"Hello World"` imprime 820Hello World.
- SI e_1 *op* e_2 ALORS i compare la valeur de e_1 avec la valeur de e_2 avec *op* et en fonction du résultat de la comparaison, soit exécute l'instruction i, soit passe à la ligne suivante. Si i n'est pas une instruction VAVERS, on passe également après à la ligne non vide suivante. Les comparaisons <> et >< signifient toutes les deux différent.
- VAVERS e saute à la ligne donnée par la valeur de l'expression e. Si cette ligne n'existe pas une erreur est affichée à l'exécution.
- ENTREE $x_1, \ldots, x_k$ demande à l'utilisateur d'entrer k valeurs entières sur le clavier et les affecte aux variables correspondantes. Si l'utilisateur n'entre pas des entiers, on lui en redemande.
- $x = e$ affecte la valeur de l'expression e à la variable x
- FIN arrête l'exécution d'un programme.
- REM permet de faire un commentaire qui n'a aucun effet sur l'exécution.

2.3 Travail demandé

2.3.1 Étape 1 et 2

Déterminer des lexèmes adéquats et implémenter un lexeur correspondant. En même temps, implémenter avec Menhir un analyseur syntaxique pour un programme LATSI. Cela

permet de définir avec Menhir les jetons (tokens) utilisés par le lexeur. Ici, on peut comme vu en cours ne pas construire un arbre de syntaxe abstrait.

2.3.2 Étape 3

Implémenter un interpréteur pour LATSI sans l'instruction VAVERS. Pour cela, il faut
- définir des types d'OCaml pour l'arbre de syntaxe abstraite qui représente un programme composé de lignes avec une instruction ;
- étendre l'analyseur de l'étape précédente pour qu'il produise un arbre de syntaxe abstraite ;
- implémenter un interpréteur pour LATSI sans VAVERS. Pour cela, on doit pouvoir évaluer les expressions. Il faut donc pouvoir mémoriser la valeur des 26 variables.

2.3.3 Étape 4

Étendre l'interpréteur pour LATSI avec l'instruction VAVERS.

3 Extensions

- Ajouter la possibilité d'avoir plusieurs affectations qui se suivent sur la même ligne : `X=1,Y=2+4`. Ici les affectations sont évaluées comme si elles étaient sur deux lignes qui se suivent.
- Ajouter la possibilité d'avoir des instructions de la forme `X,Y,Z = 1,2+5,4` avec un nombre quelconque de variables à gauche du `=` et le même nombre d'expressions à droite. Ici, les expressions à droite sont évaluées simultanément. Donc si `X` vaut 2 et on fait `X,Y = 5,X+1`, `Y` vaudra 3 après. `X,Y = 2` déclenche une erreur.
- Ajouter une instruction `SOUSROUTINE <expression>` (qui saute vers la ligne donnée par l'expression) et une instruction `RETOURNE` (qui revient à la première ligne non vide après la dernière instruction `SOUSROUTINE` exécuté).

4 Rendu

Le langage de programmation utilisé doit être OCaml, l'analyse lexicale doit être réalisée avec *ocamllex*, et l'analyse grammaticale avec *menhir*. Nous vous conseillons de vous servir des raccourcis de la bibliothèque standard de menhir.

Le projet est à faire en binôme, les deux membres d'un binôme peuvent être dans des groupes de TP différents. Les monômes sont autorisés mais seront notés comme des binômes.

Le travail doit être réalisé tout au long de la suite du semestre via un dépôt git pour chaque groupe (binôme ou monôme). 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 à **tous** les 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.

5 Rapport et Documentation

Il est nécessaire d'indiquer dans le fichier `LisezMoi.txt` les commandes qu'il faut lancer pour compiler et exécuter votre programme. Vous êtes libres de vous servir de `dune`, `ocamlbuild`, ou d'écrire des Makefile à la main. 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.

La date des soutenances sera annoncée plus tard.