TP, et Projets possibles pour Lex et Yacc

1 TP

1.1 Premier contact

Récupérez les exemples sur le serveur. Compilez; si nécessaire, adaptez les fichiers makefile... Exécutez les programmes d'exemple.

1.2 Calculette: flottants, boucles, tests

Corriger le bug de la calculette. Convertir les nombres de a calculette en nombres flottants. Ajouter une boucle for au programme de la calculette. Ajouter les tests si-alors-sinon à la calculette. L'ajout des fonctions n'est pas demandé pour un TP.

1.3 Calculette: d'infixe à postfixe

Modifiez la calculette; vous construirez un arbre représentant l'expression en entrée. Elle n'est donc pas évaluée. Par exemple, à la lecture d'une somme de 2 expressions, il faut créer un noeud de type Plus dont le fils gauche et le fils droit sont les noeuds qui représentent les deux arguments de la somme. Idem pour le produit.

Ensuite vous écrirez cette expression en notation postfixe.

Vous afficherez aussi l'arbre (sur le terminal) en utilisant des indentations.

1.4 Yacc: Calculette postfixe

Récupérez la calculette en notation postfixe, dans le répertoire postfixe. Modifiez les règles yacc pour que les nombres soient des entiers, puis que pour ces entiers soient reconnus par des règles yacc (ceci a été vu en cours). Vous pouvez supprimer la soustraction ($a - b$ peut être calculée comme $a + opp(b)$), la division, et l'exponentiation, pour simplifier. La fonction yylex d'analyse lexicale retournera des lexèmes: DIGIT, SPACE, "n", "+", "*". La difficulté est la suivante: "1 2" doit être reconnu comme 2 nombres, 1 et 2, et pas comme l'entier 12. C'est l'analyse syntaxique qui doit donc gérer les espaces entre les expressions. Une solution est dans le répertoire POSTFIXE2.

2 Le projet

Envoyez moi une archive tar de votre projet, à l'adresse dmichel@u-bourgogne.fr. Mettez votre nom en commentaire dans vos sources. Joignez le makefile, quelques fichiers d'exemples, et les quelques explications nécessaires dans un fichier LISEZ-MOI (n'abusez pas des erreurs d'orthographe: elles m'exaspèrent).

N'utilisez pas de lettres accentuées dans vos sources, vos fichiers de données, votre langage!

Choisissez un des projets suivants. Eventuellement, proposez moi votre propre sujet, mais demandez mon accord.

3 Projets faciles

Ces projets sont faciles. Ils suffisent pour comprendre Lex et Yacc. Une journée de travail suffit, mais n'attendez pas le dernier moment pour commencer .

Si vous trouvez ces projets ennuyeux, vous pouvez choisir un projet dans la section suivante.

Vous pouvez travailler en binôme, ou seul. Une grosse journée de travail suffit, mais n'attendez pas le dernier moment pour commencer. N'hésitez pas à me poser des questions; mon adresse électronique est "dmichel@u-bourgogne.fr".

3.1 Grapheur et régions

Modifiez le "grapheur"[1], pour dessiner des régions définies par des inéquations, ainsi que des combinaisons booléennes d'inéquations. Il faut représenter des conditions simples : $f(x, y) = 0$, $f(x, y \leq 0$, $f(x, y) \geq 0$, et des combinaisons et-ou de conditions (utilisez un arbre binaire, avec des noeuds "et" et des noeuds "ou", éventuellement des noeuds unaires "not".

3.2 Grapheur et régions avec profondeurs

Modifiez le "grapheur" pour dessiner des régions définies par des inéquations. Chaque région a une couleur associée, et une profondeur associée (un entier). Vous afficherez les régions par profondeur décroissante.

3.3 Fonctions cos et sin

Modifiez le "grapheur" pour prendre en compte les fonctions cos et sin. Pour calculer le cosinus ou le sinus d'un intervalle, il vous faudra décomposer l'intervalle en parties monotones. Penser à afficher les courbes $y - \cos x = 0$, et $y - \sin x = 0$ pour contrôler. Enfin, ajoutez des variables pour définir des constantes comme dans "c=2.5 ;", et pour factoriser comme dans: "soit t=(x+y)(x-y) dans (t*x-1)*(t*y+1)". Choisissez librement votre syntaxe; elle doit cependant être cohérente, donc n'hésitez pas à modifier la syntaxe du grapheur.

[1] http://ufrsciencestech.u-bourgogne.fr/master1/mi1-tc5/SOURCES/GRAPHER_INTERVAL_3/

3.4 Animation

Modifiez le "grapheur", en ajoutant une variable z. Vous créerez des animations en faisant varier la valeur de z, et en visualisant les points x, y, z tels que $f(x, y, z) = 0$. Prévoir des boucles pour faire varier z : par exemple avec la syntaxe "for z = 0 to 10 step 0.01 do". Ce qui est visualisé est donc une coupe d'une surface par un plan.

3.5 Courbes

Modifiez le "grapheur", en ajoutant une variable z. Des courbes en 3D seront décrites par l'intersection de 2 surfaces $f(x, y, z) = 0$ et $g(x, y, z) = 0$. Vous dessinerez la projection de ces courbes sur le plan Oxy, autrement dit tout point (x, y, z) de la courbe sera affiché en (x, y). Ce grapheur peut afficher les courbes paramétrées $x = f(t), y = g(t)$; il suffit de renommer t en z, et de considérer: $x - f(z) = y - g(z) = 0$. Le principe est de subdiviser l'espace 3D.

3.6 Scènes 2D (à 1 ou 2)

Décrivez des scènes 2D, ou 3D, dans un format libre, et visualisez les en openGl. Par exemple, les scènes 2D sont constituées de carrés, de cercles, de polygones, de segments. Les scènes 3D sont constituées de cubes, tétraèdres, sphères, etc. Le langage de description n'a pas à fournir les variables, les boucles, ni les fonctions. Vous pouvez réutiliser ou vous inspirer du fichier fig.cpp.

4 Projets un peu plus difficiles

Attention, les projets suivants sont un peu plus difficiles, et nécessitent plus de temps. Vous pouvez vous mettre à plus de 2 pour les faire. Faire ces projets vous apprendra beaucoup.

4.1 Petit interprête (1 à 2 jours de travail)

Attention, ce projet est plus difficile ! Modifiez la calculatrice, pour écrire un petit interprète. La syntaxe de votre langage est libre. Seul le type entier est demandé. Votre langage doit permettre de définir la fonction factorielle et la fonction fibonacci avec la récursion naïve. Pour gérer l'évaluation, vous pouvez utiliser une pile de couples : (nom de variable, valeur entière). La valeur d'une variable est la valeur entière stockée la plus haut dans la pile. Pour évaluer un appel de fonction, il faut évaluer (par un appel récursif à votre fonction d'évaluation) les arguments de la fonction, puis empiler les couples pour les paramètres de la fonction et leurs valeurs. Variante : vous pouvez gérer une pile de valeurs par nom de variable.

4.2 Grapheur 3D (2 ou 3 jours de travail, à 4)

Une surface en 3D est décrite par une équation $f(x, y, z) = 0$, par exemple la sphère est décrite par: $f(x, y, z) = x^2 + y^2 + z^2 - 1 = 0$. Subdivisez une boite englobante par la même méthode que le grapheur 2D; au niveau terminal de la récursion, partitionnez le voxel (un petit cube) en 5 (ou 6) tétrèdres; pour chaque tétrèdre, évaluez la fonction en chacun de ses sommets ; par interpolation linéaire le long des arêtes du tétraèdre, en déduire un polygone plan : un triangle ou un quadrilatère; il donne une approximation de l'intersection de la surface avec le tétraèdre. Affichez tous ces polygones. Sur internet, vous trouverez les équations définissant des surfaces classiques (quadriques, tores, cyclides, surfaces de Cayley, ombrelles de Whitney (Whitney umbrella), "supershape" [2], etc).

4.3 Mini forth (2 jours de travail, 3 étudiants)

Ré-écrivez le mini interprète forth sur ce site (miniforth.ml). Il est actuellement écrit en ocaml (moins de 100 lignes); lisez ses commentaires; 200 ou 300 lignes de C doivent suffire. Le programme ocaml n'effectue pas d'analyse syntaxique, ni lexicale. Vous devez l'effectuer. Une syntaxe possible est: ":fib dup 2 le if dup -1 add fib swap -1 add fib add else fi ." Une autre syntaxe possible est: "def fib=[dup 2 le if [dup -1 add fib swap -1 add fib add] []]". Ceci décrit la fonction fibonacci. La commande dup duplique le sommet de la pile, la commande swap échange les 2 éléments en sommet de pile, lt veut dire "lower than". Ce language est sans variable. Les entiers négatifs seront reconnus par lex (ou flex). Eventuellement, lisez un tutoriel sur Forth ou sur Joy pour plus de détail. Seul le type entier est demandé, et la possibilité de définir des fonctions récursivement, comme factorielle et Fibonacci.

En vous y mettant à 4 ou 5, vous pouvez écrire votre propre version postfixe du langage LEGO http://fr.wikipedia.org/wiki/Logo_(langage) Ce langage a été introduit par S. Pappert, et permettait d'initier des enfants à la programmation; l'idée était de faire des dessins au trait en pilotant une tortue, qui avance, tourne à gauche ou à droite.

PS: la commande dup duplique le sommet de la pile. Pour vous permettre d'écrire plus facilement des fonctions (comme factorielle, ou fibonacci), vous pouvez ajouter une autre commande dupk, qui prend un paramètre: par exemple la commande 3 dupk empile le 3, puis le dépile, et copie le 3ème élément sous le sommet de la pile en haut de la pile. Ceci permet de simuler les paramètres des fonctions: les valeurs des paramètres de la fonction sont empilées lors de l'appel de cette fonction, et dupk permet d'empiler ces valeurs quand les variables sont utilisées dans le corps de la fonction.

4.4 Langage LEGO (1 semaine, 5 ou 6 étudiants)

http://fr.wikipedia.org/wiki/Logo_(langage)

[2] http://local.wasp.uwa.edu.au/~pbourke/geometry/supershape3d

5 Ajouté en Mars 2011: Mini langage

Pour aider les étudiants, j'ai ajouté les répertoires NANO_LANGAGE et MINI en 2011. Le répertoire NANO_LANGAGE contient l'interprête d'un mini-langage de programmation fonctionnelle, sans aucune syntaxe: les programmes sont représentés par des arbres (des DAG en fait), et une fonction est fournie pour évaluer ces arbres dans un environnement donné (un environnement est un ensemble de "liaisons" entre une variable (un nom) et une valeur; les seules valeurs traitées sont les entiers, pour la simplicité). Les fonctions récursives de la factorielle et de Fibonacci sont fournies à titre d'exemple. L'archive MINI.tgz contient cet interprête et les programmes d'analyse lexicale et syntaxique.

Tous les programmes sont en ocaml, ou ocamllex (l'équivalent de Lex en Ocaml), ou ocamlyacc (l'équivalent de Yacc en Ocaml). Vous pouvez traduire en C, Lex, Yacc ces programmes pour réaliser votre projet.

Vous constaterez que la syntaxe de mini (voir le fichier exemple.mini) n'est pas très cohérente...

```
function fact x = if x <= 1 then 1 else x * fact(x-1) ;;
function fib x = if x <= 1 then 1 else fib(x-1) + fib(x-2);;
fact(5);;
fib(5);;
```

Voici quelques projets possibles, à réaliser en Ocaml ou en C ou en C++:

- ajouter la possibilité de commentaires
- rendre la syntaxe cohérente
- ajouter l'affectation de variables (avec par exemple la syntaxe a := a+1); il est probablement plus commode d'utiliser un tableau pour les environnements.
- ajouter des boucles (remarques: les boucles n'ont pas grand sens sans affectation), et des séquences d'instruction.
- ajouter les fonctions locales et les variables locales; elles sont quand même bien commodes pour le programmeur... Il y a une solution (en Ocaml) dans:
  `http://ufrsciencestech.u-bourgogne.fr/master1/mi1-tc5/COURS_ALGO_HTML/cours_algo013.html`
- ajouter les valeurs flottantes
- gérer des listes, ou plutôt des arbres binaires; il faut déclarer deux nouveaux types pour les expressions, le sous type: Paire, constituée de 2 expressions, et le sous type Empty (ou Nil). Les paires seront allouées par new, malloc, etc, dans le tas, ou bien elles seront allouées dans un tableau, et c'est leur adresse (ou leur indice) qui sera empilée. Il faut les routines: cons(e1, e2), first(paire), second(paire), isnil(p), et la constante nil. Vous écrirez une fonction de tri, ou des fonctions sur les listes, comme map, ou fold_left, fold_right du langage ocaml.
- ajouter l'accès à des fonctions graphiques, 2D (ou 3D pour les courageux) et programmer votre langage Logo.
- compiler les arbres dans un langage à pile, tel que le mini-forth. Utiliser le kdup pour accéder à la valeur des variables. Principe: reprendre la fonction eval, mais évaluer "symboliquement": la pile contient des expressions, et au

moment de compiler l'accès à une variable x, vous cherchez la position de x dans la pile symbolique.

- traduire les programmes en C. Par exemple le programme:

```
function fact x = if x <= 1 then 1 else x * fact(x-1) ;;
```

sera traduit en:

```
function fact( int x) {
if (x <= 1) then { return (1); } else { return (x*fact(x-1)); }
}
```

Vous compilerez ces programmes avec gcc, pour vérifier qu'ils fonctionnent.