Semaine 1 (Préprocesseur) - Projet IFT3150

Philippe Caron

29 août 2017

1 Avant-propos

Suite à l'extension accordée, j'ai décidé de repartir à 0 pour le projet qui avait été monté en hâte et ne constituait pas une base solide pour continuer. Cependant en raison de l'échéancier très serré, je n'ai pas pris le temps de transcrire mes notes sous format PDF, ayant peur de manquer de temps pour l'objectif principal. Les rapports de progressions suivants seront tous rapportés d'après les notes que j'ai prises sur papier.

2 Objectif

Le but de la première semaine était de terminer le préprocesseur afin d'avoir une base solide pour commencer le projet rapidement. Les principaux objectifs à remplir par le préprocesseur sont

- La détection d'erreur
- Le *parsing* adéquat des expression
- Le formatage adéquat du résultat de sortie
- Le retrait des macros «.» et « :» et parenthèsage

3 Le préprocesseur

Le préprocesseur repose sur la fonction `preprocess` qui va en fait appeler la fonction `readline` en boucle, elle même basée sur la fonction `nexttoken`. Le tout à pour but de forcer le document à avoir une certaine forme.

3.0.1 nexttoken

Cette fonction a pour principale utilité de détecter les différents «jetons» du langage même si ceux si se ramassent collés ensemble. Elle doit donc tous les connaître et comparer chaque fragment de text de au plus 3 caractère au fur et à mesure qu'elle le parcours. Un fois qu'elle a identifier un jeton, elle retourne celui-ci et la nouvelle position du curseur. Afin de préserver le formatage de l'usager, le caractère '\n' est considéré comme un jeton. Il pourra ainsi être incorporer au code sortant.

En lisant des jettons comme des string, la fonction les formatte immédiatement de la bonne manière, de même si elle lit un commentaire, elle le saute, comme si il n'existait pas. Le formattage ainsi que le retrait de commentaire reposent tous deux sur la fonction `strpar` qui permet de lire les string en prenant en compte des *nested brackets* («[[», «[=[», «[==[», etc.).

3.0.2 readexpr

Cette fonction lit des jettons jusqu'à ce que le jeton retourné soir le caractère de nouvelle ligne, ou un caractère délimitateur d'environnement (if, else, while, etc.). Si c'est le cas, la fonction regarde le jeton précé-

dent : si c'est un opérateur, alors la ligne est considérée comme n'étant pas finie, autrement la ligne termine et la fonction retourne la ligne lue, et la position actuelle du curseur.

Au fur et à mesure de la lecture les jetons sont ajoutés à une table. En même temps, les parenthèses sont comptées. Si la fin du fichier ou un jeton d'environnement est rencontré alors que le nombre de parenthèses n'est pas équilibré, cela déclenche une erreur.

La table de jetons est ensuites envoyée à la fonction scan qui associe les différents jetons selon les opérateurs présent pour retourné une chaîne de caractère bien parenthèsée. Le fonctionnement de scan sera discuté plus loin.

3.0.3 readline

En suite la fonction `readline` s'assure que la fonction précédent ne retourne pas juste des jetons de nouvelle ligne. Une fois que `readexpr` retourne une valeur autre, elle retourne la ligne et la position du curseur dans le texte.

3.0.4 Fonctions d'environnement

Les fonctions d'environnement appellent `readline` ou `readexpr` en fonction du patron sur lequel est basé leur environnement. Parexemple, l'environnement 'if' [très] simplifié ressemble à ceci

```
<if> := if <expr> then
  <bloc>
else
  <bloc>
end
<bloc> := <bloc> <expr>
<bloc> := <expr>
<expr> := (<expr> op <expr>)
<expr> := X
```

le patron de la fonction d'environnement pour *parser* if risque donc de ressembler à ceci :

```
1  function parseif(...)
2     nexttoken(...) -- if
3     nextexpr(...) -- <expr>
4     nexttoken(...) -- then
5     while peektoken(...) ~= "else" do
6        nextexpr(...)
7     end
8     nexttoken(...) -- else
9     while peektoken(...) ~= "else" do
10       nextexpr(...)
11    end
12    nexttoken(...) --end
13 end
```

3.0.5 preprocess

Tout en haut de la chaîne d'exécution il y a la fonction `preprocess` qui sert en quelque sorte de fonction d'aiguillage. Elle récupère les informations de d'environnement et dirige l'exécution vers la bonne fonction.

4 Détection d'erreur

Le processus de détection d'erreur est assez difficile à implémenter. En effet, dépendamment du type d'erreur, la cause peut être la où le curseur se trouve, ou bien il y a de ça 3 lignes. L'idée original du detecteur d'erreur était de «traîner» les valeurs `linum` (numéro de ligne) et `chnum` (numéro de caractère), et de les incrémenter ou décrémenter à mesure que le programme lisait le texte. Bien que théoriquement faisable, cette méthode s'est avérée très inefficace ; un seul oubli (dans un texte d'un peu plus de 1000 lignes) et tout le système tombe à l'eau.

La méthode actuelle est moins précise, mais est beaucoup plus fiable, elle ne se fie qu'au curseur, ainsi, lorsqu'un erreur survient, elle retourne simplement la position du curseur.

5 *Parsing*

La récupération des jetons fut assez ardue. En effet, étant donné que tous les opérateurs n'on pas forcément la même taille, il faut à chaque étape vérifier le prochain *substring* le longueur 1, 2 puis 3. La structure impérative du langage rend également difficle la prévisibilité des éléments à venir. Dans la plupart des fonction, il doit y avoir 3 ou 4 clauses d'exceptions. Par exemple, dans l'exemple suivant, `x.y` vaut toujours la même chose, mais il y a beaucoup de manière de le représenter.

```
x.y = function () ... end
function x.y () end
x["y"] = function () ... end
function x["y"] () ... end
a, x.y, c = 1, function () ... end, 2
```

Il en va de même pour beacoup d'autres éléments du langage qui ont plusieurs représentations valables (tables, string, etc.). Au final cependant, une fois toutes les exceptions couvertes, on obtient une fonction robuste (`nexttoken`) surlaque on peut se baser pour prélever du texte. En plus de rendre le code légèrement plus intuitif, le fait que tous les accès au texte partent d'une fonction réduit grandement le risque d'erreur.

6 Formatage de la sortie

En rétrospective le choix d'intégrer les caractères de changement de ligne spécifiés par l'utilisateur était vraiment une mauvaise idée. Bien que le code produit est maintenant très beau et fidèle à l'original, le traitement de tous les cas particulier engendrés par le support de cette fonctionnalité rend non seulement le code lourd et moins lisible, mais il augmente les chances de fausse erreur. Ceci dit, hormis ce détail, le formatage est très bien réussi et la fonction d'écriture fonctionne comme prévu.

7 Parenthèsage et retrait de macros

Au départ, l'objectif était de retirer les macros en premier lieu, afin de pouvoir traiter un code plus universel. Dans le cas du «.» c'est facile, et d'ailleurs c'est ce que le préprocesseur fait avant même de scanner. Cependant si on développe aussi « :» il faut le faire avant, car le résultat du développement de « :» engendre des «.». Le cas du « :» posait vraiment problème. Même si on sait que `x:y() == x.y(x)`, trouver exactement la valeur de `x` n'est pas toujours évident :

```
x.y():sub(2,3):find("a")
```

Et même si on était capable de le développer, on obtiendrait :

```
x.y().sub(x.y(),2,3).find(x.y().sub(x.y(),2,3), "a")
```

Ce on parle ici d'une séquence quasiement 3 fois plus longue à exécuter. Laisser les « :» et les traiter comme un opérateur était décidément plus intelligent. En tant qu'opérateur, on lui donne la priorité ultime; aucun opérateur n'a préséance.

Le fonctionnement du parenthèsage est décrit en détail dans le manuel. Somme toute, même si il faut parcourir une table une dizaine de fois pour obtenir le résultat final, il n'y a pas vraiment de manière plus efficace de parenthèser un séquence. Si toutes les associations étaient dans le même sens, peut-être qu'on aurait pu envisager de *pipe-liner* l'association, mais ce n'est pas le cas.

8 Test de fonctionnement

Pour s'assurer du bon fonctionnement du préprocesseur, il suffisait de l'exécuter sur lui-même, puis sur l'ancienne version, puis sur lui-même, et ainsi de suite jusqu'à ce que les deux versions soient les même, si il y a un équilibre, et que le compilateur Lua standard est en mesure d'exécuter cet équilibre, alors c'est que le fichier produit est stable et conforme à la norme. Le préprocesseur a passer ce test.

9 Conclusion

Même s'il ne s'agit pas du code le plus élégants, le préprocesseur est efficace à convertir les fichier pour les rendre prêts pour le compilateur, et même si son système de détection d'erreur est loin d'être parfait, il attrape tout de même une bonne partie des erreur de formatage de l'utilisateur.