Compilateur Lua - Projet IFT3150

Philippe Caron

28 août 2017

1 Préprocesseur

Le but du préprocesseur est de standardiser tous les fichiers de code de manière à ce que le compilateur puisse éventuellement traiter des fichiers suivant un certain patron plus strict de formattage. C'est important puisque rien ne garanti que tous les programmeurs ont la même éthique d'écriture. Dans ce cas-ci particulièrement, le préprocesseur est très utile car le langage Lua a une syntaxe plus permissive que la plupart des autres langages impératifs. Alors que beaucoup exigent le fameux «;» à la fin de chaque ligne, en Lua il n'est nécessaire que pour séparter deux instructions consécutives ambiguës. Il existe également plusieurs forme équivalentes pour exprimer la même chose, ce qui pourrait devenir compliquer pour un compilateur. Finalement, les expressions ne sont pas forcément correctement parenthésées au moment de la compilation, le préprocesseur permet de compléter les parenthèse manquante.

1.1 Fonctions du préprocesseur

Afin de faciliter au maximum le travail du compilateur, le préprocesseur accompli plusieurs tâches diverses, allant du formattage à la précompilation. Les voici :

1.1.1 Détection d'erreur

Il est facile de faire des erreurs en programant, mais pas toujours de les trouver. Le simple fait de chercher une parenthèse mal fermée peut représenter une perte de temps importante. Par exemple dans le segment de code suivant, on voit qu'une parenthèse fermante manque.

```
1  local x = print(test(i)
```

Exemple 1 – Mauvais parenthèsage

Un des rôles du préprocesseur va être d'avertir l'usager de son erreur. Si on tente d'exécuter luna sur le code suivant, on obtient ceci :

```
FILE: tmp.lua
Error at line 1: Parenthesis mismatch.
local x = print(test(i)
                ^
```

Comme on peut le voir, le programme indique une erreur en pointant sur la parenthèse qui n'a pas été fermée. Ce genre d'aide est très utile à l'usager, et lui permet de sauver du temps. De plus en détectant les erreurs plus haut dans le processus de compilation et en les supportant, on réduit le risque d'erreur fatale plus tard dans le processus de compilation, où un code incorrect est très difficile à prévoir et peut facilement faire planter le compilateur.

1.1.2 Effacement des commentaires

Une fonction bête mais essentielle du préprocesseur est d'effacer les commentaire laissés par l'usager. Cette supression permettra au compilateur de lire le fichier Lua sans avoir à se préocupper de fragments de code inutiles. Un fragment de code comme celui-ci :

```
--Bonjour
local x = "tout" .. --[[le]] "monde"
--[[Il
   fait
   beau]]
local y = "dehors"
```

Exemple 2 – Commentaires multiples

en devient un beaucoup plus simple :

```

local x = ("tout" .. monde)
local y = "dehors"
```

1.1.3 Indexation

Même si le but ici n'était pas d'écrire un éditeur de texte pour le langage, l'idée de faire une fonction d'indexation semblait très raisonnable. En effet, que ce soit par presse ou par paresse, les programmeurs vont occasionnellement écrire du code mal formaté et il est alors très pratique de pouvoir l'indexer automatiquement. Le préprocesseur va également conserver les choix personnel du programmeur quand à ses «saut de ligne», de sorte qu'un code comme celui-ci :

```
if
   y==x and
            b==3 and
   c==4
then
if y==2 then
 print("y=2")
   elseif
 x==3 then
   print("x=3")
   else
  print(c)
   end
end
```

Exemple 3 – Code mal formaté

va être transformé par le préprocesseur pour produire un code bien indenté :

```
if
   (((y == x) and
   (b == 3)) and
   (c == 4))
then
   if (y == 2) then
      print ("y=2")
   else if
         (x == 3) then
         print ("x=3")
```

```
      else
         print (c)
      end
   end
end
```

Bien évidemment, une telle fonction requière une certaine quantitée de ressource en comparaison avec une autre qui n'aurait pas autant d'éléments à prendre en charge. Il faut cependant réaliser qu'elle permet de lire des nombres et des opérateurs collés et ensuite de les décoller. Un des avantages immense de cette partie du processus sera évidente ultérieurement, alors que l'écriture d'une fonction de *parsing* deviendra triviale.

1.1.4 Parenthèsage

Comme dans tous les langages à notation infixe, le compilateur doit s'assurer d'avoir un parenthèsage complet avant d'évaluer une expression.

```
  local x=3+4-2-3*-2^-2+4/5%8
```

Exemple 4 – Parenthèsage incomplet

Une expression du genre est plutôt difficile à évaluer, c'est pourquoi le préprocesseur va former les expressions une à une en suivant l'ordre de priorité prescrit par le langage. Le résultat est le suivant :

```
local x = ((((3 + 4) - 2) - (3 * (- (2 ^ (- 2))))) + ((4 / 5) % 8))
```

On constate immédiatement une lisibilité accrue ; il en va de même pour le compilateur. Pour les étapes suivante on sait qu'il y a 1 ou 2 opérande par opérateur, il n'y aura donc que deux cas à gérer.

1.1.5 Précompilation

La ligne obtenue dans la précédente section est en réalité obtenue en exécutant `./luna -npc -s tmp.lua`, c'est à dire en spécifiant au compilateur qu'on ne veut pas précompiler. Sans la *switch* `-npc`, le résultat est en fait le suivant :

```
local x = 6.55
```

Le principe est d'éviter au programme compilé des calculs facilement évitable. Au fur et à mesure que les parenthèses sont formées, le préprocesseur vérifie si celles-ci sont possible à évaluer (qu'elles ne contiennes pas de variable). Si l'évaluation est réussie, il remplace la parenthèse par le résultat de sont évaluation.

1.1.6 Standardisation

L'objectif final, et non le moindre, du préprocesseur est de limiter le langage à un sous emsemble sur lequel il le compilateur pourra compter. Le langage Lua offre de nombreuses contraction et macros que le préprocesseur transforme en version plus basique.

```
tmp = { allo = 9 }

function tmp.test(a)
   return a.x:sub(i, i)
end

local function loc(n)
   return n + 1
end
```

Exemple 5 – Utilisation des contractions et des macros

Comme on peut le voir dans l'exemple ci-haut, il est possible de référer aux indices textuels d'une table en utilisant «`table.index`»; cela reviens exactement au même qu'écrire «`table ["index"]`». La seconde forme sera priorisé dans le texte final puisqu'elle est standard et fonctionne avec tous les types d'index. On remarque aussi qu'il est possible de déclarer des fonction sans utiliser explicitement l'opérateur de déclaration «=». Cette déclaration en ligne est pratique car agréable à l'oeil, mais ce n'est pas un format conventionnel que le compilateur pourra traiter, il faut donc aussi les développer :

```
tmp = { allo = 9 }

tmp ["test"] = function (a)
   return a : sub (i , i)
end
local loc ; loc = function (n)
   return (n + 1)
end
```

Lors de l'expansion de la déclaration d'une fonction locale, un point-virgule s'ajoute au code [1]. Ce point-virgule est en fait essentiel car en Lua les assignations sont faites après l'évaluation de toute les valeurs. Sans le point-virgule, lorsque la fonction `loc` serait évaluée, le terme «loc» n'existerait pas encore et elle ne pourrait pas s'y référer. Par contre le deux-point dans l'expression «`a : sub (i , i)`» n'est pas transformé. Ce choix a été fait pour des raison d'optimisation. Cette optimisation est particulièrement visible dans les expressions de la forme suivante :

```
table.x.y.z:sub(1, 1) ==
  table ["x"] ["y"] ["z"] ["sub"] (table ["x"] ["y"] ["z"] , 1 , 1)
```

Exemple 6 – Développement sous-optimal

On peut voir que développer le «`:`» est en fait sous-optimal, dans le second cas, la table «`table`» devra être indexée deux fois, une fois à 4 niveau de profondeur, la seconde à 3. En conservant le «`:`», on peut implémenter une manière beaucoup plus efficace de réaliser cette opération en rajoutant des instructions au compilateur.

1.2 Mécanisme

Le préprocesseur a de nombreuses fonctions à remplir. Ces fonctions sont pour la plupart cependant simplement destinée à faciliter la tâche des compilateurs ultérieurement et à favoriser les chances de succès, c'est-à-dire qu'aucune d'entre elles ne fait réellement partie du processus de compilation. Le tout semblant un peu superflu, on souhaite réduire au maximum le temps que le programme prend pour effectuer cette tâche. La première version du préprocesseur était très simple mais demandait le lire le texte à plusieurs reprise en modifiant une chose à chaque fois. La version présentement utilisée ne fait qu'un seul passage, et exécute chacune de ces fonctions simultanément.

1.2.1 Fonction de *parsing*

La reconnaissance de texte en Lua n'est pas la chose la plus évidente. Il faut donc un fonction robuste capable de différencier tous les termes réservés du langage de ceux utilisable pour les variables, de différencier une variable d'un opérateur même s'il n'y a pas d'espace entre les deux, d'un chiffre et ainsi de suite... Pour Luna, cette fonction s'appelle `nexttoken`. Pour un endroit quelconque dans le texte, lorsqu'appelée cette fonction retournera le prochain élément du langage, déjà formaté dans le cas d'un string, ainsi que la nouvelle position du lecteur.

```
tmp = °[[Paul profitait du soleil.
```

1. https ://www.lua.org/manual/5.3/manual.html, section 3.4.11

```
  - "Il n'y avait pas beaucoup de monde sur la plage aujourd'hui", pensait-
    il.]]
```

Exemple 7 – Lecture d'un string

Dans l'exemple ci-haut position du lecteur est donné par «°». Suite à l'appel de la fonction `nexttoken` le programe se retrouve dans cet état :

```
tmp = "Paul profitait du soleil.\n\t- \"Il n'y avait pas beaucoup de monde
    sur la plage aujourd'hui\", pensait-il." °
```

Le nouveau *string* produit utilise la notation classique avec les guillements anglais, prenant soin de bien protéger les caractère spéciaux, comme le *newline* et le *tab* ainsi que les guillements présents avec la barre oblique inversée.

1.2.2 Lecture de ligne

À chaque ligne la fonction `nexttoken` est appelée jusqu'à ce que lecteur rencontre le caractère *newline*, atteigne un délimitateur d'environnement, ou atteigne la fin du fichier. Au fur et à mesure que les mots sont lus, ils sont ajouter à une table d'expressions. C'est cette table qui sera utilisée pour le parenthèsage.

1.2.2.1 Parenthèsage

Le système de parenthèsage est plutôt simple, il est réalisé par les fonction `comb1` et `comb2`, depuis la fonction `scan`. Essentiellement, `scan` les appelles successivement avec une liste d'opérateur à associer (dans l'ordre de priorité du langage). La fonction de combinaison parcours la table, et lorsqu'elle trouve un opérateur de ses paramètres, elle va tout d'abord tenter de résoudre le calcul posé par cet opérateur grâce à `trysolve`. Dans le cas d'un échec elle va retourner un string qui correspond au deux opérandes et l'opérateur entre parenthèses.

```
expr = { "a", "+", "b", "*", "c", "(", "3", "+", "(", "-", "2", ")", ")"}
scan(expr):
> scan { "3", "+", "(", "-", "2", ")" }
   > scan { "-", "2" }
      -- niveau unaire
      > comb1 { {"-" , "2"}, "not", "-", "~", "#"}
         > trysolve ("-", "2", nil)
         << "-2"
      << "-2"
   << "(-2)"
      -- niveau additif
      > comb2 { {"+", "(-2)"}, "3", "+", "-")
         > trysolve ("+", "(-2)", "3")
         << "1"
      << "1"
<< "(1)"
   -- niveau multiplicatif
   > comb2 { {"a", "+", "b", "*", "c (1)"}, "*", "/", "\\", "%" }
   -- niveau additif
   > comb2 { {"a", "+", "(b * c (1))"}, "+", "-")
<< "(a + (b * c (1)))"
```

Exemple 8 – Trace partielle d'exécution de la fonction scan