Introduction a l'analyse syntaxique appliquée au 42sh

David Giron – Chef Koala

Avec l'aimable participation

du grand Chuck Norris

Sommaire

- Qui sont les Koalas et que vous veulent-ils ?
- L'eval-epxr en vrai et notions de base d'analyse syntaxique
- Application au 42sh et notions avancées d'analyse syntaxique

Pas de panique les enfants. Ca a l'air compliqué vu d'ici, mais je serais là pour vous accompagner tout au long de cette conférence.

KOALA

Qui sont les Koalas et que vous veulent-ils ?

Aucun mal, soyez tranquilles, ce sont des gens biens.

KOALA

Les Koalas

- Koala est un acronyme qui signifie : Kind Of Advanced Languages Assistants.

- Les Koalas sont un groupe de professeurs et d'assistants dirigées par le chef Koala David GIRON.

- Les Koalas encadrents le C++ (David GIRON), le KOOC (Lionel AUROUX), le Java (Dimitri DARSEYNE), l'UML (Francois CARRUBA), le C# (Neils FREIER) et la programmation fonctionnelle (David GIRON).

- Les slides de cette conference vont sont accessibles a l'adresse : www.epitech.net/~koala/ressources/conference_tech1_2011.ppt

C'est Lionel Auroux qui a inventé le nom de « Koalas ». Il a l'habitude d'appeler tout ses projets « kind of quelque chose ». Je trouve ca très amusant !

KOALA

Pourquoi cette conférence ?

Pour vous apporter des connaissances et une méthodologie intéressantes pour le 42sh.

- Pour démystifier l'eval expr et bien comprendre le but que ce projet cherchait a atteindre : L'initiation à l'analyse syntaxique.

Pour vous intoduire à un vocabulaire et des pratiques que vous retrouverez en 2ème et 3ème année et un peu partout...

Vous voyez ? cette conférence peut vraiment vous aider à réussir votre année, donc soyez bien attentifs. D'ailleurs, Je vous conseille de prendre des notes.

KOALA

L'eval-expr en vrai

Voila, les choses sérieuses commencent. Ne soyez pas inquiets, je suis avec vous. Tout se passera bien.

KOALA

Rappel de la problématique de l'eval-expr

- « Écrire un programme capable de reconnaître une expression arithmétique et de l'évaluer en fonction des priorités d'opérations. »

- Nous allons en extraire 2 sous-problèmes pour introduire les notions voulues plus simplement :

1) Reconnaître une expression arithmétique.

2) Évaluer cette expression en fonction des priorités d'opérateurs.

Diviser un problème pour le simplifier est une méthode qui est très utile dans la vie de tout les jours. Souvenez-vous en !

KOALA

Reconnaître une expression arithmétique

- Soit l'expression arithmétique suivante :

$$1 + 1$$

- Les '1' sont appelés “opérandes” de l'expression.
- Le '+' est appelé “opérateur” de l'expression.

- Appelons “num” la famille des chiffres de '1' à '9'.

- Et appelons “ope” la famille des opérateurs '+', '-', '*', '/' et '%'.

Je pense que jusque là, vous ne devriez pas avoir de problèmes à suivre. N'hésitez pas à poser des questions avant d'être perdus.
Vous devriez vraiment prendre des notes vous savez...

KOALA

Notion de « grammaire formelle »

- D'après les deux familles que nous avons décrites sur le slide précédent, on pourrait adopter la notation suivante pour les exprimer plus clairement et plus précisément :

num ⟶ '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

ope ⟶ '+' | '-' | '/' | '*' | '%'

- On pourrait donc généraliser l'expression du slide précédent de la façon suivante :

exp ⟶ num ope num

La porté d'un « ou » (le pipe) s'étend au groupe encadré par ce pipe au pipe suivant ou la fin de la règle. Le premier groupe ainsi définit s'étend naturellement du début de la règle au premier pipe.

KOALA

Précisions et vocabulaire

- Une grammaire formelle, ou simplement grammaire, est composée de règles telles que nos règles 'exp', 'num' ou 'ope'. L'ensemble de ces règles permet de décrire précisément un langage. Dans notre exemple, le langage est l'ensemble des expressions arithmétiques composees d'un operande suivi d'un operateur suivi d'un operande.

- Une règle est composée de la façon suivante:

  - Un membre gauche qui définie l'identifiant de la règle.
  - Une flèche qui sépare les deux membre, la pointe vers le membre droit.
  - Un membre droit qui énumère les différents éléments qui composent la règle.

La première règle à être appelée est « l'axiome » de la grammaire. Lorsqu'une règle est appliquée, on dit qu'on la « dérive ».

KOALA

Notion d'élément terminal et non-terminal

- On peut trouver 2 types d'éléments dans le membre droit d'une règle:

- Les elements terminaux, notés 'T' ou 'Σ', tels que '1', '2', '9', '+' ou '*' dans notre exemple. Cet ensemble est aussi appellé “alphabet” ou encore “vocabulaire terminal”.

- Les éléments non-terminaux, notés 'N', tels que num ou ope dans notre exemple. Une regle de grammaire peut donc etre recursive.

De plus, les regles se notent 'R' et l'axiome 'S'. C'est plus pratique pour prendre des notes.

KOALA

Règles récursive

- L'état actuel de notre grammaire ne permet pas de reconnaitre des expressions de la forme suivante :

$$1 + 23 * 456$$

- Nous devons donc étendre notre grammaire afin de pouvoir reconnaître des expressions semblables à celle du dessus. Pour cela, nous allons appeler les règles pertinentes récursivement. Notre grammaire devient donc :

S ⟶ num ope S | num

num ⟶ '1' num | '2' num | '3' num | '4' num
| '5' num | '6' num | '7' num | '8' num | '9' num
| '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

ope ⟶ '+' | '-' | '/' | '*' | '%'

Voilà, la grammaire commence à devenir précise. Malheureusement, la notation devient vite difficilement lisible. Nous allons voir dans le prochain slide comment simplifier tout cela.

KOALA

Forme de Backus-Naur 1/2

- Cette syntaxe a été conçue par John Backus et Peter Naur lors de la création de la grammaire du langage Algol 60. Initialement appelée Backus normal form, elle est devenue la « forme de Backus-Naur » à la suggestion de Donald Knuth, l'inventeur du langage TeX. On l'abrège par « BNF » pour plus de simplicité.

- La BNF (re)définit un certain nombre de symboles supplémentaires par rapport à la notation d'une grammaire formelle :

- « ::= » Remplace avantageusement pour la notation informatique la flèche servant à définir une règle.

- « [ ] » Permet de définir des sous-ensembles dans les éléments composant une règle.

- « * » L'ensemble qui précède ce symbole peut être lu 0 ou n fois.

- « + » L'ensemble qui précède ce symbole peut être lu 1 ou n fois.

- « ? » L'ensemble qui précède ce symbole peut être lu 0 ou 1 fois.

Je me souviens quand on inventait Algol avec des amis, j'ai toujours trouvé que ce langage a été une grande amélioration de ses successeurs !

KOALA

Forme de Backus-Naur 2/2

- Notre grammaire peut donc être traduite simplement en BNF, ce qui améliore grandement sa lisibilité:

```
S      ::=    num [ ope num ]*

num    ::=    [ '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ]+

ope    ::=    '+' | '-' | '/' | '*' | '%'
```

- Toutes les expressions grammaticales qui suivront seront exprimées en BNF à partir de maintenant.

Maintenant qu'on a une notation simple, visuelle et précise, on va pouvoir rajouter la priorité des opérateurs et les parenthèses. Je sais que vous êtes inquiets, mais rassurez-vous je veille sur vous.

KOALA

Priorité des opérateurs

- En arithmétique, les opérateurs '*', '/' et '%' ont une priorité superieure à celle des opérateurs '+' et '-'. Nous allons donc adapter notre BNF pour prendre en compte cette priorité:

```
S          ::=   ope_low
ope_low    ::=   ope_high [ ['+' | '-'] ope_high ]*
ope_high   ::=   num [ ['*' | '/' | '%' ] num ]*
num        ::=   [ '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ]+
```

- On peut donc remarquer que les règles ayant la priorité la plus haute sont celles qui sont les plus profondes, ici, la règle 'ope_high'.

Vous pouvez remarquer au passage que les règles 'ope_low' et 'ope_high' sont construites sur le même modèle.

KOALA

Gestion des parenthèses

- Les opérateurs parenthèse '(' et ')' permettent d'ajouter un ou plusieurs niveaux de priorité supplémentaires à une expression arithmétique. Modifions notre BNF en conséquence:

```
S         ::=   ope_low
ope_low   ::=   ope_high [ ['+' | '-'] ope_high ]*
ope_high  ::=   pth [ ['*' | '/' | '%' ] pth ]*
pth       ::=   '(' S ')' | num
num       ::=   [ '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ]+
```

- Le contenu d'un paire de parenthèses est une expression à part entière, la dérivation de l'axiome leur est donc appliqué. Notre grammaire est dite récursive.

Voyons maintenant comment appliquer en code tout ce que nous venons de voir ensemble.

KOALA

De la grammaire au code

- Une fois la grammaire d'une expression (et par extension d'un langage) définie, deux outils d'analyses doivent être mis en place dans le programme:

- L'analyseur lexical, ou « lexer », chargé de découper le flux d'entré du programme en « mots » ou « tokens » qui sont les unités lexicales à analyser.

- L'analyseur syntaxique, ou « parser » qui représente l'implémentation de la BNF du langage, chargé de vérifier que l'ordre des tokens fournis par le lexer respecte les règles grammaticales établies et de construire un « arbre de syntaxe abstraite ».

Il existe des outils tels que lex/yacc, AntLR, LLGen ou encore CodeWorker pour générer ces codes, mais je vous déconseille fortement de les utiliser tant que vous ne saurez pas coder convenablement un lexer et un parser à la main.

KOALA

L'analyse lexicale 1/2

- L'analyse lexical d'un flux de caractères consiste à le découper en mots, appelés « tokens » ou « lexèmes ».
- Les lexèmes doivent être divisés en groupes selon leur type. Ce seront ces types qui seront utilisés par l'analyseur syntaxique pour déterminer si l'ordre des tokens est conforme à la grammaire.
- Les lexèmes forment un sous langage, on pourrait donc les décrire par une grammaire également. L'implémentation d'un lexer classique passe par une description des lexèmes sous forme d'expressions rationnelles (ou tout autre procédure de reconnaissance lexicale).
- Notre langage comporte 6 types de lexèmes. Les expressions rationnelles correspondantes sont les suivantes:

NUM = [123456789]+	OPE_LOW = [+-]	OPE_HIGH = [*/%]
PTH_BEG = (	PTH_END =)	

« Expression rationnelle » est la traduction de l'anglais « regular expression » souvent traduit à tort en « expression régulière »

KOALA

L'analyse lexicale 2/2

- Les lexèmes et leur type sont organisés dans une liste, dans l'ordre où ils apparaissent sans vérification de cohérence dans leur ordre. Cela signifie qu'une expression lexicalement correcte peut être syntaxiquement fausse.
- Soit l'expression suivante:

$$(1 + 23) * 456$$

- Après analyse lexicale selon les expressions rationnelles du slide précédent, on obtiendra la liste de lexèmes suivante:

lexème	(	1	+	23	)	*	456
type	PTH_BEG	NUM	OPE_LOW	NUM	PTH_END	OPE_HIGH	NUM

Il est très pratique de faire commencer sa liste par un token vide de type BEGIN et de la terminer par un autre token vide de type END. De plus, les tokens non reconnus peuvent être regroupés sous le type UNKNOWN. Ces pratiques faciliteront la vie à l'analyseur syntaxique par la suite.

KOALA

L'analyse syntaxique

- L'analyse syntaxique consiste à vérifier que le flux de lexèmes fournis par l'analyseur lexical correspond bien à la grammaire spécifiée pour le langage, de l'axiome au texte analysé, puis à construire un arbre de syntaxe abstraite.

- L'analyse descendante retrace la dérivation de l'axiome vers les éléments terminaux. Les analyseurs syntaxique “LL“ utilisent cette méthode.

- l'analyse ascendante essaie d'associer des lexème entre eux pour former des « syntagmes » de plus en plus grands jusqu'à retrouver l'axiome. Les analyseurs syntaxique de type “LR” utilisent cette méthode.

- Quelque soit la méthode d'analyse utilisée, le but est toujours d'organiser les lexèmes sous la forme d'un arbre pour en représenter « l'information » exprimée par le langage. Par exemple, si on analysait un code C, un noeud correspondant à une définition de fonction aurait des sous noeuds contenant chacun une des expressions contenues dans la fonction, etc.

Il n'existe pas de forme d'arbre « officielle » ou « meilleure » que les autres, il faut juste que cette forme soit à même d'exprimer de façon pratique le langage.

KOALA

L'analyse ascendante

- Méthode d'analyse utilisée par yacc et AntLR, l'analyse ascendante est une méthode à la fois efficace et très complexe pour des non-initiés. C'est pourquoi cette méthode ne sera pas abordé aujourd'hui.
- Il est toutefois intéressant de connaître quelques propriétés remarquables de ces analyseurs:

- Machine à états finis contenant: une pile pour stocker les états, une table des branchements qui indique l'état vers lequel aller et une table des actions qui donne la règle de grammaire à utiliser en fonction de l'état courant et du lexème courant de l'entrée.

- Détectent plus rapidement les erreurs dans le flux de lexème

- Peuvent être prouvés formellement

Ne vous en faites pas si vous ne comprenez pas complètement de quoi il retourne, nous verrons cela plus en détail une prochaine fois, vous avez déjà beaucoup à comprendre aujourd'hui.

KOALA

L'analyse descendante 1/2

- L'analyse descendante retrace la dérivation de l'axiome vers les éléments terminaux. Cela signifie qu'en théorie, chaque règle de la grammaire est traduite en une fonction homonyme chargée de consommer des lexèmes fournis par le lexer.
- Chacune de ces fonctions renvoie un booléen renseignant la fonction appelante sur la réussite ou l'échec de la dérivation de la règle représenté.
- Si une fonction a consommé des lexèmes avant d'échouer, il est important de restaurer les lexèmes consommés dans la liste pour essayer une règle alternative.
- Les paramètres à passer à chacune de ces fonctions sont laissés à la discrétion du développeur, dans notre exemple, nous n'en mettrons pas, mais cela ne signifie en aucun cas qu'il ne faut pas en mettre.
- Nous allons voir dans le slide suivant un exemple d'implémentation de la règle ope_low.

Il existe en réalité un millier de petites choses supplémentaires à savoir sur l'implémentation d'un analyseur syntaxique descendant, mais il ne tient qu'à vous d'expérimenter et de les découvrir par vous même.

KOALA

L'analyse descendante 2/2

```
/*    ope_low ::=       ope_high [ ['+' | '-'] ope_high ]*    */

t_boolope_low( void )
{
t_bool    stop;

stop = FALSE;
if ( ope_high() )
    {
        while( !stop )
        {
                if ( !(read_OPE_LOW_token() && ope_high()) )
                        stop = TRUE;
        }
        return TRUE;
    }
return FALSE;
}
```

Comme vous pouvez le constater, un analyseur syntaxique descendant est particulièrement simple à coder. Notez au passage que la mise en arbre des informations validées n'apparaît pas dans cet exemple.

KOALA

Application au 42sh

Maintenant que vous connaissez toutes les bases, nous pouvons aborder le problème de votre projet de fin d'année.

KOALA

Généralités sur les shells et problématique du projet

- Le 42sh est le projet de fin de première année à Epitech. Ce projet consiste à programmer un interprete de commandes stable et utilisable, capable d'interpréter des commandes de façon similaire à d'autre shells comme sh, bash, zsh ou encore tcsh.
- L'ensemble des commandes qu'il est possible d'envoyer à un shell ainsi que le script shell forment un langage qui est et doit être définit par une grammaire.
- Chaque shell implémente un analyseur lexical et un analyseur syntaxique pour consommer son flux d'entrée, qu'il provienne de la sortie standard, d'un fichier ou d'une quelconque autre source. Il doit en être de même pour les votres. Les « strtowordtab » que vous avez utilisés jusqu'à aujourd'hui n'amènent qu'à un résultat: Segfault et 0 en soutenance finale, comme chaque année. Vous voilà prévenus.

Vous êtes libres pour ce projet de faire supporter à votre shell une grammaire d'un shell existant ou bien d'implémenter la votre. Toutefois je vous le déconseille vivement à moins que ne vous sentiez aussi à l'aise dans le domaine que les personnes qui animent cette conférence... Vous devriez vous contenter d'implémenter la grammaire de sh.

KOALA

Exemple de traitement d'une commande 1/6

- Soit la commande à interpréter suivante:

```
$>cat koala.txt && ls | wc -l
```

- En lisant cette commande, on remarque qu'elle est composée de deux parties distinctes: afficher le contenus du fichier koala.txt, puis en cas de réussite, lister les fichiers du répertoire courant et rediriger la sortie standard pour compter les lignes.

- Nous allons maintenant analyser cette commande et grace à un analyseur lexical et à un analyseur syntaxique qui va la valider, puis la « récrire » sous forme d'un arbre de syntaxe abstraite.

Avec les Asteks, on rigole encore du dernier étudiant qui s'est cru assez malin pour coder son 42sh sans utiliser les méthodes qui vous sont présentées aujourd'hui. Le pauvre, il y a cru jusqu'au bout...

KOALA

Exemple de traitement d'une commande 2/6

- Definissons une grammaire simplifiée pour notre shell:

OPE_AND = &&
OPE_PIPE = |
WORD = [\d\w_-./]+

S ::= [exp]+

exp ::= and_exp

and_exp ::= pipe_exp [OPE_AND exp]?

pipe_exp ::= command [OPE_PIPE pipe_exp]?

command ::= com [com_arg]*

com ::= WORD

com_arg ::= WORD

Cette grammaire est evidement tres restrainte, elle ne vous est presente qu'a titre indicatif. Une veritable grammaire de shell est bien plus complexe.

KOALA

Exemple de traitement d'une commande 3/6

- La première chose à faire est donc d'analyser lexicalement cette commande.
- Après analyse lexicale, nous avons donc la liste de lexèmes suivante:

lexème	*cat*	*koala.txt*	*&&*	*ls*	*|*	*wc*	*-l*
type	*COM*	*COM_ARG*	*OPE_AND*	*COM*	*OPE_PIPE*	*COM*	*COM_ARG*

- Le découpage que nous avons utilisé ainsi que les types de lexèmes sont arbitraires, rien ne vous empêche de voir les choses autrement.

Un bon découpage du flux permet de se faciliter les choses pour l'analyse syntaxique. Assurez-vous que la sortie de votre lexer couvre bien tout les cas complexes que les Asteks ne manqueront pas de tester en soutenance.

KOALA

Exemple de traitement d'une commande 4/6

- Viens maintenant la phase d'analyse syntaxique. Cette phase est cruciale puisque c'est elle qui va créer l'arbre de syntaxe abstraite (ou « AST ») que vous parcourerez pour exécuter votre commande. Il est donc vital de réfléchir à un design intelligent et pratique de votre arbre pour prendre en compte des choses comme l'ordre de certains tokens, la priorité entre eux, etc.
- Nous vous conseillons vivement d'utiliser une analyse syntaxique descendante pour votre 42sh. Ces parsers sont très simple à mettre en place car ils sont le reflet en code d'une grammaire formelle ou d'une BNF.
- Par exemple, une fonction « read_command() » renverrait « true » si le lexème courant est de type COM et est suivit de 0 ou n lexèmes de type COM_ARG.
- Autre exemple, (peut-être au dessus de « read_command() »...), la fonction « read_and() » renverrait « false » si elle lisait un token de type COM suivit par un token de type OPE_AND suivit par le token de fin de flux...
- Chaque fonction qui valide une séquence de tokens en renvoyant « true » doit ajouter ce token dans l'arbre à l'endroit voulu. Il pourrait donc être malin de passer en paramètre de la fonction un pointeur vers l'endroit voulu...

Voyons a present un exemple de code pour illustrer tout cela.

KOALA

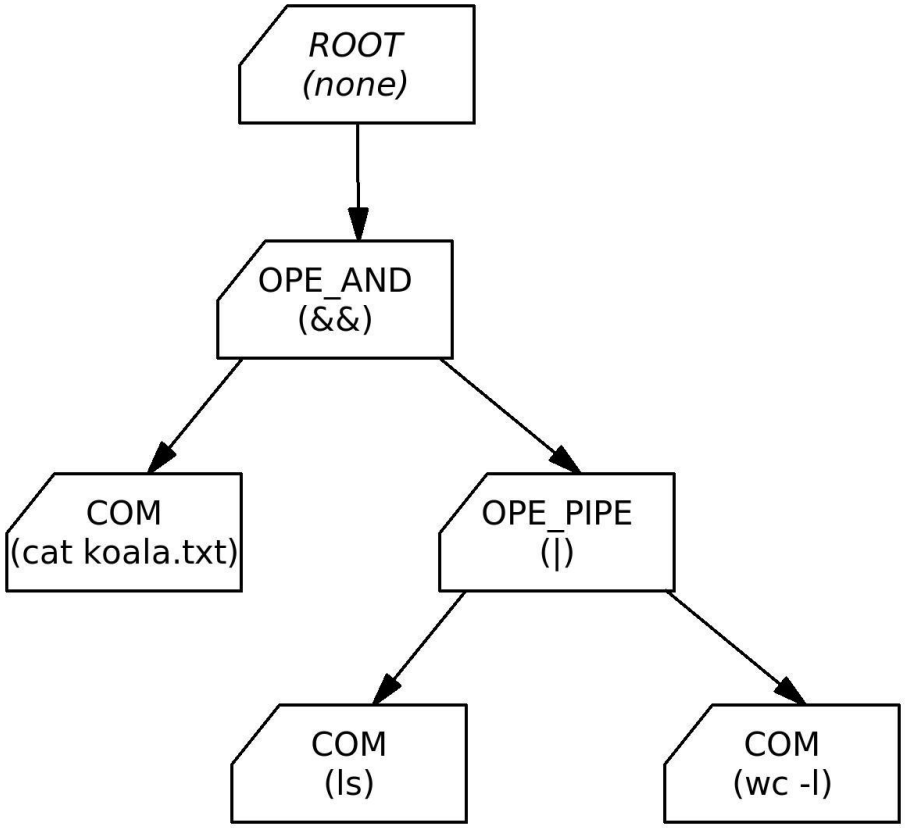
Exemple de traitement d'une commande 5/6

```
/*    pipe_exp        ::=      command [OPE_PIPE pipe_exp]?    */

t_boolread_pipe_exp( t_ast_node *tree )
{
        t_ast_node    *pipe_node;

        pipe_node = xmalloc(1 * sizeof(*pipe_node));
        pipe_node->type = PIPE_EXP;
        pipe_node->left = NULL;
        pipe_node->right = NULL;
        if ( read_command(pipe_node->left) )
           {
                read_OPE_PIPE_token() && read_pipe_exp(pipe_node->right);
                add_node_to_tree(tree, pipe_node);
                return TRUE;
           }
        free(pipe_node);
        return FALSE;
}
```

Nous allons maintenant voir ce que vous attendez depuis longtemps: Un arbre de syntaxe abstraite de la commande de notre exemple.

KOALA

Exemple de traitement d'une commande 6/6



Chaque noeud est bien evidement une structure contenant un type, une donnée, une liste de pointeurs vers les fils du noeud ainsi que tout ce que vous jugerez utile ou pratique d'ajouter. Souvenez-vous qu'il n'y a pas d'arbre idéal, vous aurez donc chacun le votre.

KOALA

Questions !!!

Ne soyez pas timides, je sais qu'il y en a. N'hésitez pas !

KOALA