haute école
neuchâtel berne jura
arc ingénierie
saint-imier le locle delémont

VSL COMPILER

Rapport

Auteur : David Jacot, Anthony Mougin

Lieu et date : Saint-Imier, le 26/01/09

Table des matières

1 Introduction

Le compilateur VSL a été développé avec le module python PLY (Python Lex Yacc). La structure de base a été développée selon le TP4 (séparation des fichiers d'analyse lexicale, syntaxique et sémantique). Son but est de compiler les fichiers d'extension *.vsl afin de les convertir en byte code pouvant être lu par une machine virtuelle de notre conception. Tout au long de ces différentes analyses, un contrôle d'erreur sera effectué sur les parties du programme. Chaque fichier du compilateur (sauf les fichiers binaires) aura comme préfixe ***vslcomp_...***.

2 Analyse lexicale

L'analyse lexicale correspond à la partie LEX de PLY. Elle permet la détection des différents Tokens dans le texte. Elle ne regarde pas si les enchaînements sont bons, mais seulement si les éléments du programme sont bien des éléments autorisés. Cette partie se situe dans le fichier ***vslcomp_lex.py***. Ce dernier dispose d'une fonction *main* qui est là pour tester certains fichiers (l'option -v permet d'afficher chaque token trouvé).

2.1Initialisation

La fonction init() permet d'initialiser l'analyse lexicale (à appeler pour chaque compilation de fichier, réinitialise le compteur de ligne, etc...).

2.2Les types de tokens

Les tokens sont séparés en 4 listes distinctes:

- Les constantes réservées : ***reserved_const***

  Ce sont les constantes du programme (True, False, Pi, ...)

- Les fonctions réservées : ***reserved_func***

  Les fonctions déjà prédéfini dans le programme. Pour plus d'information sur ces fonctions, consulter le manuel d'utilisateur.

- Les mots réservés : ***reserved_words***

  Les mots réservés (Comme pour les boucles, la déclaration des fonctions, des importations, ...)

- Les Tokens généraux : ***tokens***

Les Tokens qui sont les composants principaux du programme (Les chaînes de caractère, les opérations, les parenthèses ...)

2.3 Tokens ignorés

Certains éléments du programme sont détectés mais volontairement ignorés. C'est le cas pour les commentaires (de deux types). Le type *commentaire* n'est pas rentré dans la liste des tokens, car il n'est pas utilisé plus tard. On défini juste la fonction permettant de les reconnaître mais cette fonction ne retourne pas le token trouvé. On ignore aussi, comme dans le TP4, les espaces et les tabulations grâce à la variable *t_ignore*.

2.4 Gestion des erreurs

La méthode t_error(t) permet de récupérer les éléments du programme qui ont été découverts mais qui ne figurent dans aucune liste de tokens. Le message affiché sera donc une erreur du compilateur de ce type:

**** Error line 4: Illegal character '$'.**

La variable global ***lexicalerror*** permet de récupérer plus haut le nombre d'erreurs lexical découvert.

3 Analyse syntaxique

L'analyse syntaxique permet de vérifier la cohérence de l'enchaînement des tokens selon la grammaire définie. La manière avec laquelle on définit la grammaire a déjà été expliquée dans les TP vu en cours. Cette analyse, aussi appelé ***parsing*** en anglais ce situe principalement dans le fichier ***vslcomp_parser.py***. Le parsing passe aussi par la construction d'un arbre appeler AST (Abstract Syntax Tree). Le fichier principal contient une méthode *main* permettant de tester le parsing du fichier en affichant l'arbre dans la console.

3.1 La grammaire

La grammaire est de type hors-contexte. ***NT -> (N ou NT)****. Elle est de type BNF. Elle est composée de 123 règles que je n'énoncerai pas ici. Elles se situent en annexes. L'élément de départ est l'élément non terminal *mainprogram*.

3.2 AST

Les différentes classes pour construire les éléments de l'arbre se situent dans le fichier ***vslcomp_ast.py***. Elles dérivent toutes d'une classe ***Node***. Toutes ces classes ont des noms d'attributs très différents, contrairement au TP vu en cours, qui était beaucoup plus réduit. Ce choix a été fait en vue de facilité la compréhension lors de la partie sur l'analyse sémantique que nous verrons plus tard. En effet, certain bloc de code étant assez conséquent, la relecture d'une multitude d'accès à des éléments d'un tableau sans savoir à quoi ces derniers correspondaient devenait fastidieuse. Chaque élément de type Node mets en place des méthodes permettant leur affichage en ASCII dans la console.

Ainsi, à la fin de la construction de l'AST, Le parsing retourne un élément de type ***Node*** qui est la racine du programme. Si ce n'est pas le cas, alors il y a dû avoir des erreurs.

3.3 Gestion des erreurs

La gestion des erreurs se fait tout au long du parsing. Certaines méthodes, comme l'accès à un tableau, nécessite une gestion particulière. Ainsi, certaines méthodes peuvent avoir une gestion interne des erreurs. C'est pour éviter de parcourir l'arbre pour vérifier ces erreurs que la gestion a été intégrée lors de sa construction.

Il existe aussi une méthode ***p_error(p)*** qui récupère les tokens mal placés et affiche un message d'erreur:

*Ex: ** Error line 2: Syntax error. NUMBER is not attempt here.*

La liste peut être longue, la plupart du temps c'est dés le premier token que vient l'erreur.

A chaque erreur. C'est la variable ***parserror*** qui est incrémentée.

3.4 Construction récursive

Pour parser un document, on utilise la méthode ***parseImport()***. Cette methode parse le document en cours, puis étudie les importations encore à traiter. Pour chaque document, on réutilise la méthode ***parseImport().*** On remplace la valeur de la case actuellement traitée dans le tableau des importations par le résultat. Ainsi, à la sortie du premier appel de la méthode, on dispose de l'arbre du programme principal, et d'un tableau contenant tous les arbres des différentes importations (les importations non-valides sont supprimées du tableau).

4 Analyse du parsing et de l'AST

Cette étape, située entre l'analyse syntaxique et l'analyse sémantique, effectue un certain nombre de vérifications :

- Test des définitions de fonctions:

  Cette partie permet de vérifier si une méthode n'est pas déclarée deux fois. Le langage n'accepte pas le polymorphisme, ainsi deux fonctions avec le même nom sont considérées comme identiques.

  Pour ce faire, on construit un dictionnaire qui a pour clef le nom des méthodes et pour valeur un tableau contenant la ou les lignes où elles sont déclarées. Si la longueur d'un tableau est supérieure à deux, alors la fonction associée est déclarée plus d'une fois. Une erreur est alors affichée, en indiquant les lignes des déclarations.

- Test des appels de fonctions:

  Cette partie remplie deux tâches:

  - Test sur la fonction principale ***main***.

    On test si la fonction ***main*** existe bien, et que celle-ci a bien un seul attribut (les arguments du programme).

  - Test sur les fonctions en générale

    - Test si la fonction est bien définie (dans le tableau de définitions)
    - Test si le nombre de paramètres lors de l'appel correspond au nombre de paramètres nécessaires.
    - Test si l'appel de la fonction nécessite un retour de valeur et si la définition retourne bien une valeur.

Ces deux tests sont capables d'afficher des erreurs. Ces erreurs sont considérées comme des erreurs de parsing. C'est donc la variable global ***parserror*** qui sera incrémentée.

- Test des fonctions non-utilisées:

  Permet de détecter les fonctions qui sont définies, mais ne sont jamais appelées dans le code.

- Test des variables non-utilisées:

  Permet de détecter l'affectation d'une variable qui n'est jamais lue.

Ces deux tests sont capables d'afficher des avertissements. C'est la variable

global ***warningfound*** qui sera incrémentée.

5 Analyse sémantique

C'est la partie qui est chargée de l'interprétation de l'AST. Son but est de convertir chaque élément en élément du byte code (chaîne de caractère). Cette partie est contenue dans le fichier ***vslcomp_compiler.py***. Elle contient elle aussi une fonction main capable de compiler un programme donné en paramètre.

5.1 Compilation principale

La compilation principale est une fonction nommé ***compile*** et qui prend en paramètre l'AST principale à compiler. On compile tout d'abord tout les AST contenu dans le tableau d'importation du parser que l'on concatène à la compilation du programme principal. On obtient ainsi le byte code total a générer que l'on retourne. Toutes ces opérations sont chronométrées.

5.2 Construction du Byte Code

La construction du byte code se fait récursivement grâce aux méthodes ***compile*** ajoutées à chaque élément de l'AST qui devra être compilé (certains éléments sont utilisés seulement pour stocker des informations utilisables par d'autre ***Node***). L'ajout se fait grâce au décorateur ***addToClass()*** du TP4. Chaque méthode retourne le morceau du byte code crée.

5.3 Fichier auxiliaire

Le fichier ***vslcomp_operations.py*** contient toutes méthodes permettant de convertir les opérations du programme en leur équivalent en byte code

5.4 Gestion des erreurs

La seule opération de test lors de la compilation et le test des opérations sur les strings. En effet, cet élément du programme est un cas spécial sur les opérations. Elle affecte la variable globale ***compilerror***.

A mesure que le langage se complexifierait, d'autres tests pourraient s'ajouter.

6 Optimisation

6.1 Optimisation de l'AST

Lors de la compilation des définitions de fonction de l'AST, on peut savoir si ces dernières sont utilisées ou non. On peut alors éviter d'implémenter ces définitions inutilisées dans le byte code.

Pour les variables non utilisées, c'est plus compliqué. On ne peut pas supprimer les assignations d'éléments non utilisés, car il est possible que la partie droite de l'assignation soit un appel de fonction.

6.2 Optimisation du byte code

Après que le byte code soit entièrement construit, on le passe dans une fonction d'optimisation. Cette fonction possède plusieurs optimisations différentes:

- Double labels:

  Il est possible que dans le code, plusieurs label s'enchaînent.

  Ex: lab1: lab2: PUSHS "*b*"

  Or, la machine virtuelle n'accepte pas ce format. On va alors supprimer le premier label (ici *lab1*) et remplacer tous les sauts à ce label par le deuxième label.

- Enchaînement label – jump:

  Le code peut présenter un label suivi d'un saut. On se rend compte que cette opération et inutile. Il suffit alors d'aller remplacer tous les appels de saut à ce label par un appel de saut au label du jump qui le suit. (Un exemple semble inévitable !):

  Ex:

  JMP lab1

  ...

  lab1: JMP lab2

  ...

  lab2: ...

Au lieu d'aller au label lab1, puis de repartir au label lab2, on a meilleur temps d'aller directement au lab2.

Optimisé:

JMP lab2

...

JMP lab2

...

lab2: ...

- Enchaînement Set – Get:

  Lorsque dans le byte code, il y a un SET suivi d'un GET sur la même variable, Le haut de la pile est attribuée à la variable, puis elle est enlevée, puis rajoutée. On comprend vite qu'il n'est pas nécessaire d'enlever le haut de la pile. On remplace donc l'enchaînement SET-GET par l'op-code prévu à cet effet : SETN (qui permet d'attribuer une variable sans enlever le haut de la pile).

- Enchaînement Get – Set:

  Si on trouve la suite GET puis SET sur la même variable. On réatribue à la variable sa propre valeur (*b=b;*). On peut donc enlever ces deux lignes dans le byte code.

6.3 Niveaux d'optimisation

Il existe plusieurs niveaux d'optimisation, définit par un entier:

- 0: Seule la première est effectuée (obligatoire)
- 1: La première, la deuxième
- 2: La première, la deuxième et la troisième
- 3: Toutes les optimisations sont effectuées

7 Messages

7.1 Fichier séparé

L'affichage de messages se fait à travers plusieurs méthodes situées dans le fichier ***vslcomp_message.py***.

7.2 Types de messages

- Erreur : ***error()***

  Affiche une erreur avec le message et le numéro de ligne fourni.

- Avertissement : ***warning()***

  Affiche un avertissement avec le message et le numéro de ligne fourni.

- Succès : ***success()***

  Affiche le succès d'une certaine opération, sur un fichier donné. On peut aussi définir si l'opération comporte des avertissements et la durée de l'opération.

- Échec : ***failed()***

  Affiche l'échec d'une opération sur un certain fichier, en affichant le nombre d'erreurs survenues.

7.3 Code d'erreur

Le programme dispose de plusieurs numéros de sortie de programme afin d'informer le système de la manière dont le programme a quitté:

- ***EXIT_SUCCESS (0)*** : Le programme a quitté normalement
- ***LEXICAL_ERROR_CODE (1)*** : Échec de l'analyse lexical
- ***PARSING_ERROR_CODE (2)*** : Échec de l'analyse syntaxique
- ***COMPILE_ERROR_CODE (4)*** : Échec de l'analyse sémantique
- ***NOTHING_COMPILE_ERROR_CODE (8)*** : Aucun fichier à compiler

Les codes d'erreurs ont été choisis pour être cumulés. Le code d'erreur ***3*** correspond donc à un échec de l'analyse lexical et un échec de l'analyse syntaxique.

8 Machine Virtuelle

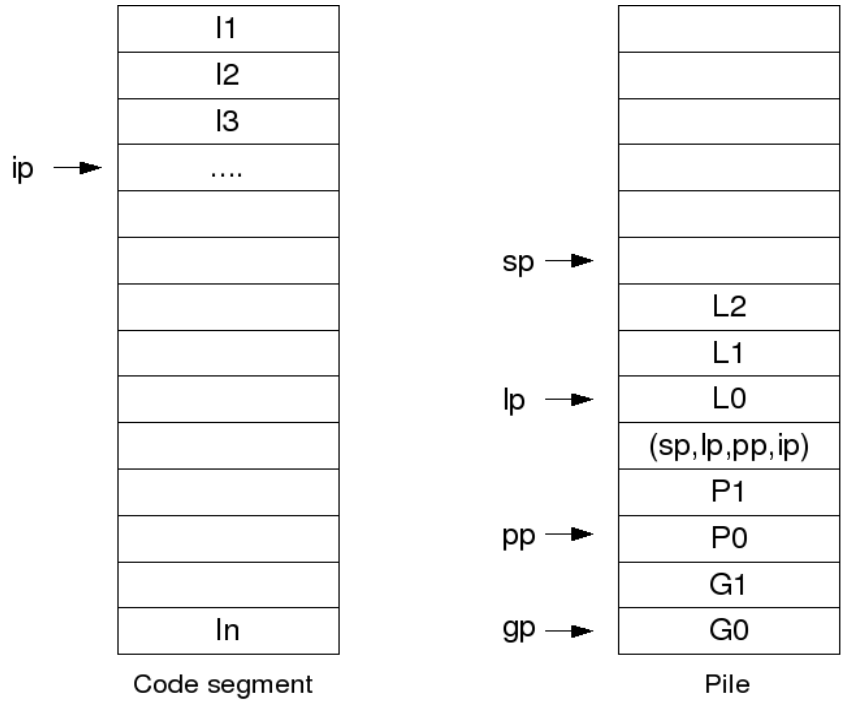
Le présent chapitre décrit les grandes lignes de la conception de la machine virtuelle. Tout d'abord, nous avons décidé de développer une machine virtuelle à pile. Soit une machine où toutes les opérations s'effectuent sur une pile contrairement à une machine à registre.

L'objectif était de concevoir une machine permettant de gérer les opérations standards (arithmétiques, logiques, ..), les sauts conditionnels, les nombres (entier et float), les chaînes ce caractères, les tableaux, les fonctions et les entrés – sorties standards.

Dans les faites, nous sommes allés légèrement plus loin en lui ajoutant les fonctions trigonométriques, logarithmiques, exponentielles, la gestion des paramètres passé en ligne de commande, la gestion des fichiers, ... Tous les opcodes sont décrits dans le fichier source de la machine virtuelle.

8.1 Organisation de la machine

La machine est composée d'un segment de code qui contient les opcodes à exécuter et d'une pile qui contient les données. Dans les faites, elle possède également un tas mais, dans notre cas, nous laissons cette tâche à python.



Elle contient également quatre registres :

- Instruction Pointer : Prochaine instruction à exécuter.
- Stack Pointer : Prochaine case vide dans la pile.
- Local Pointer : Adresse de la première variable locale dans la pile.
- Parameter Pointer : Adresse du premier paramètre dans la pile.

8.2 Opcodes

Avant de se lancer dans la conception de la machine virtuelle, il a fallut établir la structure du byte code. Afin de rendre la machine virtuelle la plus simple possible, nous avons choisi de faire un byte code très simple et de laisser la logique dans le compilateur.

Chaque ligne ce code a la forme suivante :

```
<label>: <opcode> <paramètres>
```

Le label et les paramètres sont optionnels.

Exemple, la factorielle :

```
function main(args) {
   print(fact(500));
}

function fact(n) {
   if(n==1) ret = n;
   else ret = n*fact(n-1);
   return ret;
}
```

```
GETPROGARGS
CALL main 1
main: PUSHI 500
CALL fact 1
WRITE
PUSHI 0
EXIT
fact: ALLOC 1
GETP 0
PUSHI 1
EQ
JZ ifsep0_0
GETP 0
SETL 0
JMP endif0
ifsep0_0: GETP 0
GETP 0
PUSHI 1
SUB
CALL fact 1
MUL
SETL 0
endif0: GETL 0
RETURN 1
```

Comme vous pouvez le voir, le byte code n'est pas compliqué. La seul chose qui peut choquer est la gestion des fonctions. Une fonction est un simple label mais qui est appelé avec « call » au lieu d'un « jump ». Ainsi, il est possible de créer un nouveau contexte lors de l'appel. De plus, la fonction doit toujours se terminer par un « return » afin de restaurer le contexte précédent.

Pour plus d'information sur la byte code, veuillez vous référez à la documentation se trouvant dans le code source de vsl.

8.3 Types

Le langage VSL est faiblement typé mais la machine virtuelle ne l'est pas complètement. En effet, elle gère les types suivants :

- Integer
- Float
- String
- Array
- File

Les types doivent être respectés lors de l'exécution. Par exemple, il n'est pas possible d'additionner une chaîne de caractères et un entier. Il est nécessaire de convertir un des deux préalablement. Si cela n'est pas fait, la machine virtuelle va lancé une erreur d'exécution et s'arrêter.

8.4 Machine 0.1

La première version de la machine virtuelle a été conçu pour être totalement dynamique en s'inspirant de PLY. L'idée est que chaque opcode soit représenté par une méthode.

Exemple :

```
@opcode_args(label, int)
def op_call(self, funcName, n):
    ....
```

Grâce à cela, le parseur peut vérifier dynamiquement l'existence d'un opcode donné. De plus, cela permet de vérifier le type des paramètres. Le parseur prépare le segment de code en stockant la référence sur la méthode correspondant à l'opcode et ses paramètres.

Une fois le parsing terminé, le segment de code peut être exécuté. Pour ce faire, il suffit d'appeler dynamiquement la méthode en lui passant les paramètres.

Cette implémentation est présente dans le dossier VRE sous le nom « oldvsl ». Cette technique a l'avantage d'être totalement dynamique. Cependant, après divers tests, nous avons constaté que les performance n'était pas au rendez-vous. En effet, lorsque beaucoup d'opcodes doivent être exécutés la machine n'est pas rapide. Nous avons essayé d'utilité Psyco pour améliorer les performance mais malheureusement, Psyco n'est pas efficace sur des codes utilisant des appels dynamiques.

NB : Cette version n'implémente pas tous les opcodes. Elle n'est pas utilisable avec le code généré par VSLC.

8.5 Machine 0.2

Après les tests avec la version 0.1, nous avons décidé d'essayer d'implémenter une machine plus performante. Pour ce faire, nous avons aboli les appels dynamiques et toutes les choses que Psyco ne supporte pas.

Le parseur a du être totalement refait. Afin de préserver un peu d'évolutivité, les opcodes sont déclarés dans un tableau avec leur signature. Ceci permet au parseur de faire le même travail que dans la version 0.1.

Toutes les méthodes ont été remplacées par un « if ... elif ... elif ... ». Cette technique nous a permis de calculer le nombre de Fibonacci 40 en environ 2.5s. Avec la version précédente, le calcul s'effectuait en 40s.

Après quelques recherches supplémentaires, nous avons réussi à améliorer les performance en abaissant le temps d'exécution au environ de 1.6s pour le même calcul. Pour ce faire, nous avons changé les systèmes de « if .. elif .. » en un arbre afin de réduire le nombre de tests pour atteindre un opcode.

Cette implémentation est présente dans le dossier VRE sous le nom « vsl ».

9 Fichiers binaires

9.1 VSLC

C'est le fichier principal qui sera appelé pour compiler un ou plusieurs fichiers passés en paramètre.

9.2 VSL

VSL est la machine virtuelle qui permet d'exécuter les fichiers compilés « .cvsl ».

9.3 VMAKE

Permet de créer un makefile dans le dossier courant. Pour plus d'informations, voir la documentation utilisateur.

10 Améliorations

Voici une liste des améliorations non exhaustive:

- Ajout d'une macro *module* situé en haut d'un fichier permettant de désactiver les avertissements lors de l'importation du fichier.
- Amélioration de certains algorithmes (comme pour les optimisations) afin d'accélérer la compilation.
- Une meilleure gestion des erreurs (ajout d'un fichier avec les méthodes).
- Ajouter un peu plus d'optimisation.
- ... (et sûrement pleins d'autres !)