# Implémentation d'un générateur de byte-code et d'un interprêtateur pour le langage PseudoC

*Projet du module "Compilation"* - ENSIAS

**Komlan Akpédjé KEDJI**

Mars - Avril 2008

Sous la direction de: **Professeur Habilité Karim BAINA**

# Table des matières

**Résumé**

Dans le cadre du cours de compilation, il est implémenté une collection d'outils (PCC : PseudoC Compiler Collection) permettant d'exécuter du code écrit en PseudoC (une version allégée du langage C). Le présent document documente les diverses stratégies et compromis utilisés tout au long du processus, et discute de leur pertinence vu les résultats obtenus.

# CHAPITRE 1

# Introduction

> *"Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it."*
> – Alan J. Perlis (Epigrams in programming).[3]

S'il existe un trait caractéristique du développement d'un compilateur, il s'agit bien de la complexité [1].Le présent rapport présente les stratégies utilisées pour contrôler cette complexité, dans le cadre de l'implémentation d'une suite d'outils permettant d'exécuter du code PseudoC.

De [1] nous retenons qu'une stratégie éfficace pour contrôler la complexité d'un compilateur est d'utiliser des représentations intermédiares bien définies. Ceci permet de décrire la transformation du program source en exécutable comme une série de transformations sur un flux de données. Le présent rapport est structuré de manière à réfléter ce processus.

En effet, le passage d'un code source en PseudoC à l'exécution traverse les étapes suivantes :

*Code Source (PseudoC) -> Code Intermédiare -> Byte-Code -> Exécution*

1. Il n'est donc pas surprennant que d'importants travaux de recherche soient menés sur le sujet, voir [1].

# CHAPITRE 2

## L'analyseur lexical de code PseudoC

L'analyseur lexical a été généré par Flex, à partir du fichier d'entrée `pc_lexer.lex`.

En dehors du code classique d'analyse lexical, les fonctionnalités suivantes sont introduites :

- Sauvegarde des positions (ligne, colonne) de chaque token. Ceci permettra par la suite de donner des messages d'erreur utiles lors de la compilation.
- Vérifications de base sur les chaines de caractères : les chaines invalides (contenant des retour chariot) et celles trop longues [1] sont détectées.

1. Les chaines littérales en PseudoC ont une longueur maximale de 256 caractères.

# CHAPITRE 3

## L'analyseur syntaxique de code PseudoC

L'analyseur lexical a été généré par Bison, à partir du fichier d'entrée `pc_parser.y`.

Le fichier `pc_parser.y` implémente la grammaire du PseudoC. Le compilateur construit étant à une passe, l'AST (Abstract Syntax Tree) est implicite, et les actions sémantiques effectuent directement les appels nécessaires pour générer le code intermédiare. En effet, la construction d'un AST ne se justifie pas vu la simplicité des structures syntaxiques du PseudoC.

> *"There are two ways of constructing a software design ; one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."*
> – C. A. R. Hoare

Une règle de la grammaire a donc la forme suivante :

```
instruction: WRITESTR '(' STR ')' ';'
    { emit(OPWRITESTR, $3); }
    ...
```

Ici, le token WRITESTR est reconnu, suivi d'une parenthèse ouvrante, un argument (une chaine de caractère littérale), puis une parenthèse fermante. L'action correspondante est de générer l'instruction de code intermédiaire nommée OPWRITESTR, avec comme argument, le troisième token, ie STR.

Les actions sémantiques font des appels au module implémentant la table de symboles (pc_symtable.c). Cette dernière est implémentée comme une table de hachage [1]. Ceci permet de compiler de larges programmes PseudoC, sans pénalité de performance majeure (causée par une recherche linéaire dans une liste chainée).

1. L'implémentation de la table de hashage est contenue dans le fichier uthash.h, et provient du projet UTHASH sur http ://uthash.sourceforge.net.

# CHAPITRE 4

# Le générateur de code intermédiare

Le fichier `ic_generator.c` implémente la génération de code intermédiare. Le point central est la fonction `emit` dont le prototype est :

```
void emit(int instruction, ...);
```

`emit` prend un nombre variable d'arguments (suivant l'instruction) et génère le code approprié. Pour rendre les modifications plus simples, la correspondance entre le numéro d'une instruction du code intermédiare, et la représentation textuelle de l'opérateur, est maintenue dans le fichier `ic_generator.h` (tous les modules désirant faire cette conversion y font appel). Ceci permet de modifier la représentation textuelle des opérateurs en mettant à jour un seul fichier.

Le code intermédiare utilisé est à zero adresse. Le code généré étant destiné à être interprêté, il est tout à fait inutile d'utiliser un code à plus d'une adresse. En effet, d'une part les opérandes ne seront jamais rangés dans les régistres réels de la machine [1], i.e. il n'y a aucun gain en performance. D'autre part, on évite ainsi la complexité liée à l'allocation des registres, et on bénéficie de la densité de code caractéristique des codes à une adresse.

1. Suggérer au compilateur C de mettre une variable dans un registre n'est pas suffisant, le standard C précise qu'il s'agit seulement d'une suggestion, pas un ordre.

> *“Any fool can make the simple complex, only a smart person can make the complex simple.”*
> – [Anonymous]

Il faut noter que le code intermédiare choisi est optimisé pour la lisibilité. Il est possible de programmer directement dans le code intermédiare. le fichier généré à l'extension `.s`, ce qui permet de l'analyser avec la coloration syntaxique sous un éditeur comme `vim`. Ceci est d'une aide précieuse lors du debuggage.

# CHAPITRE 5

# L'analyseur syntaxique de code intermédiare

Le code intermédiare généré étant du texte, il nous faut un analyseur synthaxique pour l'exploiter. Cette fonction est implémenté par le fichier `ic_parser.c`

Il s'agit de la première phase de l'interprétation. C'est une étape simple qui n'est isolée que pour des raisons de modularité. `ic_parser.c` se résume à la fonction `next_ic_inst` qui retourne, à chaque appel, une représentation structurée de la prochaine instruction du fichier contenant le code intermédiaire. Il s'agit de récupérer une instruction du langage intermédiaire, et de le transformer en une structure de données directement exploitable par le générateur de byte-code.

# CHAPITRE 6

## Le générateur de byte-code

Le fichier `bc_generator.c` implémente le générateur de byte-code.

Le byte-code généré est rangé dans un tableau `code` qui émule la RAM de l'ordinateur. Une copie est stockée dans un fichier, pour éviter de regénérer le byte-code lors des éxécutions ultérieures [1].

Le byte-code présente l'avantage d'être très compact. Les instructions sans opérande (comme les opérateurs arithmétiques) n'occupent q'un octect. Celles qui opérent sur des emplacements mémoire (comme l'assignation de variable) sont suivis d'une adresse qui est un entier stocké sur quatre octects. La taille d'une instruction varie donc entre 1 et 5 octets.

1. PYTHON utilise une tactique similaire. Avant d'interpréter un fichier `foo.py`, Python génère un fichier de byte-code `foo.pyc`. Le byte-code est utilisé lors des appels ultérieurs.

# CHAPITRE 7

# L'interpréteur

Le fichier `interpreter.c` implémente l'interprétation du byte-code, et a contient essentiellement la fonction `run`. Il faut noter que la convention d'appel (ou *stack-layout*) implémentée ici pour le PseudoC est la même que celle du langage C.

Au moment de l'éxécution, `run` utilise les structures suivantes :

- Une pile d'appels (`call_stack`). Elle contient les adresses de retour des fonctions et les copies des adresses des cadres d'activation (voir plus bas).
- Un pointeur de pile d'appels (`call_stackp`). Il pointe sur le prochain emplacement libre sur la pile d'appels.
- Une pile d'expressions (`expr_stack`). Elle contient les valeurs sur lequelles les opérateurs calculent (le code est à une adresse : tous les calculs sont faits sur la pile).
- Un pointeur de pile d'expressions (`expr_stackp`). Il pointe sur le prochain emplacement libre sur la pile d'expressions.
- Un pointer de frame (`framep`). Il contient l'adresse du début du cadre d'activation courant (i.e. celui de la fonction en cours d'exécution).
- Un compteur de programme (`instp`). Il contient l'adresse de l'instruction suivante.

# CHAPITRE 8

## Utilitaires et support de debuggage

Une série d'utilitaires a été mise au point pour faciliter l'utilisation de PseudoC Compiler Collection, et surtout faciliter le debuggage.

Commençons par noter que l'analyseur syntaxique est par défaut très verbeux. Il donne des détails précis sur les erreurs rencontrées pour permettre au programmeur des les répérer et les corriger le plus vite possible. Un exemple de sortie est le suivant :

```
Duplicated identifier: `i': line 68, line 70
Type mismatch: variable `tableau':
    decl[ARRAY]: line 67, use[INT]: line 80
Function `longueur' used as a variable:
    decl: line 4, use: line 119
Redefined function: `afficher': line 8, line 186
Undeclared variable: `total': line 98
Function prototype mismatch: `size': line 1, line 2
    First: Function `size':       INT -> INT
    Second: Function `size':      [Procedure] -> INT
Undeclared function `foobar' (line 5)
`bar' is not a function (line 6)
```

## 8.1 L'utilitaire `pseudocl`

`pseudocl` affiche une version *human-readable* d'une source PseudoC. Il est ainsi possible de visualiser le résultat obtenu après la phase d'analyse lexicale. Exemple d'utilisation :

```
$ pseudocl <<EOF
> int foo; static int bar;
> int main(void){ write_int(2*5/6); return 0;}
> EOF
INT         IDENTIFIER  SEMI_COLON
STATIC      INT         IDENTIFIER  SEMI_COLON
INT         IDENTIFIER  PAREN_OPEN  VOID
PAREN_CLOSE BRACE_OPEN  WRITE_INT   PAREN_OPEN
NUMBER      MULTIPLY    NUMBER      DIVIDE
NUMBER      PAREN_CLOSE SEMI_COLON  RETURN
NUMBER      SEMI_COLON  BRACE_CLOSE
$
```

## 8.2 L'utilitaire `opname`

Quand on ne dispose que du byte-code (on peut en avoir une représentation 'lisible' avec l'utilitaire `UNIX xxd`), il est utile de pouvoir obtenir le nom d'un opérateur à partir de son numéro donné sous forme décimale ou hexadécimale. L'utilitaire `opname` a été conçu à cet effet. Exemple d'utilisation :

```
$ opname
 Utility to get an operator name from its code
  -> code in decimal: no prefix (ex: `45')
  -> code in hexadecimal: prefix `x' (ex: `x1c')

        Usage:   opname <dec_code> | opname x<hex_code>

$ opname 30
 dec=30  hex=1e  CALL
$ opname x10
 dec=16  hex=10  POP
$ opname x41
```

```
 Unknown opcode: dec=65  hex=41
$
```

## 8.3 Mode pas à pas dans l'interpréteur

L'interprétateur fonctionne en mode pas à pas, lorque la macro `NDEBUG` n'est pas définie au moment de la compilation.

Dans ce mode, la confirmation de l'utilisateur est demandée avant l'exécution de chaque instruction du byte-code, et de très utiles informations (état de la pile d'appel, état de la pile d'instruction, code et nom de l'instruction suivante, ...) sont affichées. Exemple d'éxécution :

```
-> call stack dump:
[7]: 2          (hex:2)
[6]: 0          (hex:0)
[5]: 0          (hex:0)
[4]: 42         (hex:2a)
[3]: 42         (hex:2a)
[2]: 42         (hex:2a)
[1]: 6          (hex:6)
[0]: 0          (hex:0)
-> expr stack dump:
[0]: 2          (hex:2)
next instruction: 38 (PUSHC)   at 212  (hex:d4)
press enter to continue
```

CHAPITRE 9

# Le jeu de tests

## 9.1 Tests de syntaxe

Lors du développement, il est utile de pouvoir tester la validité de l'analyseur syntaxique. Il est cependant difficile d'écrire un programme de test contenant toutes les combinaisons syntaxiques possible.

Un programe SCHEME, `generator.scm`, qui génère aléatoirement des programmes PSEUDOC syntaxiquement valides, a donc été implémenté. Une approche similaire a discutée dans [5].

Le programme `generator.scm` est lui-même une démonstration de la puissance méta-syntaxique des macros en SCHEME. En effet, le code ressemble à s'y méprendre à la spécification BNF du PSEUDOC. Extrait :

```
(define-production (int)
  (number->string (random 1000000)))

(define-production (binary-op d)
  ">" "<" "==" "<=" ">=" "!=" "+"
  "-" "*" "/" "%" "||" "&&")
```

```
(define-production (function-body)
  (string-append "{\n"
      (list-of declaration "\n\t")
      "\n"
      (list-of instruction "\n\t")
      "\n}\n"))
```

Le script shell `syntaxtest.sh` permet de lancer un nombre paramétrable de tests sur des programmes générés avec `generator.scm`.

## 9.2 Tests de sémantique

Une élégant framework de test a été mis en place pour le compilateur PseudoC. Le framework s'appuie sur le constat qu'un programme PseudoC est à quelques modifications près, un programme C valide. On peut donc compiler un même programme avec le compilateur PseudoC (`pseudocc`) et le compilateur C (`gcc`), et comparer les sorties des deux exécutables. Une approche similaire a été utilisée par [4] pour détecter plusieurs bugs dans GCC.

Le framework de test contient les fichiers suivants :

- `libpc.c` : implémentation en C des appels système du PseudoC (`write_int`, `write_string`, ...).
- `libpc.h` : définitions de macro permettant de générer du code C ou du code PseudoC à partir d'une représentation unifiée.
- `test.h` : fonctions générales utiles pour les cas de test (valides en C et en PseudoC).
- `runtest.sh` : shell script permettant d'exécuter un test.
- `runall.sh` : shell script permettant d'exécuter tous les cas de test disponibles.
- `identical.c` : source de l'utilitaire permettant de détecter si deux fichiers sont identiques. Même si cet utilitaire donne l'impression de réinventer `diff`, il a l'avantage de terminer à la détection de la première différence entre les deux fichiers, ce qui est plus efficace dans notre cas.

> *"The good thing about reinventing the wheel is that you get a round one."*
> – Douglas Crockford (Author of JSON and JsLint)

Les cas de test implémentés sont les suivants :

**io** :
Opérations d'entrée sortie (write_int, write_string)

**math** :
Calculs arithmétiques (+, -, /, *, %)

**array** :
Utilisation des tableaux

**boolean** :
Test des opérateurs booléen (&&, ||, !, short-circuit)

**flow** :
Opérateurs de controle (if, while)

**classics** :
Fonctions classiques (factoriel, fibonnaci, ackermann, primalité, plus grand commun diviseur)

Bien entendu, le compilateur passe tous les tests avec succès.

# CHAPITRE 10

# Conclusion

> *"[…] Hence my urgent advice to all of you to reject the morals of the bestseller society and to find, to start with, your reward in your own fun. This is quite feasible, for the challenge of simplification is so fascinating that, if we do our job properly, we shall have the greatest fun in the world."*
> – E. W. Dijkstra, On the nature of computing science.[2]

Le développement de PseudoC Compiler Collection a été une immense partie de plaisir.

Le développement d'un compilateur est une occasion unique pour entrer en contact avec toutes les belles idées qui font le charme de notre métier, l'informatique. En effet, c'est l'occasion d'explorer tous les concepts, depuis les langages de haut-niveau (ici le PseudoC) jusqu'au byte-code. C'est aussi une démontration de l'efficacité qui peut résulter de la combinaison des outils en ligne de commande sous Unix (le `shell`, `xxd`, `make`, `vim`, …)

Je suis globalement satisfait du travail accompli, même s'il reste du travail à faire pour que le PseudoC Compiler Collection approche de GCC en termes de convivialité d'utilisation. Je pense en particulier à :

– Intégrer du code pour un traitement décent des arguments de la ligne de

commande avec GETOPTS.
- Inclure des scripts pour une installation dans le style `./configure ; make ; make install` (GNU AUTOTOOLS).
- Implémenter l'optimisation du code intermédiaire généré (notament le *peep-hole optimization*).
- Intégrer proprement CPP dans le processus de compilation pour que les programmes écrits en PseudoC puissent utiliser des instructions du préprocesseur C.
- Documenter proprement le code.

# Bibliographie


[1] Atze Dijkstra, Jeroen Fokker and S. Doaitse Swierstra : *The Structure of the Essential Haskell Compiler or Coping with Compiler Complexity*, http://www.cs.uu.nl/wiki/Ehc/WebHome/

[2] E. W. Dijkstra, *On the nature of computing science*, http://www.cs.utexas.edu/users/EWD/

[3] Alan J. Perlis, *Epigrams in programming*, SIGPLAN Notices Vol. 17, No. 9, September 1982, pages 7 - 13 http://www.cs.yale.edu/quotes.html

[4] Flash Sheridan, *Practical Testing of a C99 Compiler Using Output Comparison*.

[5] D. L. Bird, C. U. Munoz, *Automatic generation of random self-checking test cases*.

[6] Torben Ægidius Mogensen, *Basics of Compiler Design*.

[7] Tom Niemann, *A compact Guide to LEX & YACC*.