Le langage EZ

Kevin Hivert, Timothée Napoli

31 mai 2016

Table des matières

1 Introduction

Ce rapport présente le travail effectué lors du projet annuel du M2 SILI de l'année 2015-2016 par Kevin Hivert et Timothée Napoli.

Le sujet du projet annuel a été la rédaction d'un compilateur du nouveau langage de programmation, le langage EZ (prononcer "easy").

Le rapport est séparé en 4 parties :

— Définition du langage EZ
— Implémentation du compilateur
— Les pistes d'amélioration du langage
— Une brève conclusion

2 Le langage EZ

Le langage EZ est un nouveau langage de programmation destiné aux débutants en programmation. L'esprit du langage est de limiter le plus possible les aspects techniques de la programmation (comme la manipulation des pointeurs) tout en offrant un langage performant, simple et expressif.

Le langage permet l'usage du paradigme de programmation impératif, et introduit des notions de programmation fonctionelle. Éssayant de s'inspirer de langages modernes tels que Rust ou encore Haskell (eux même influencés par des langages comme Erlang ou Scala), tout en étant plus simple à appréhender comme peut l'être PASCAL pour des néophytes en programmation.

Le langage supporte :

- Programmation structurée (définition de fonctions, de procédures, de types de données définis par l'utilisateur, instruction de contrôle de flux).
- Récursivité (les fonctions peuvent s'appeller elles-même).
- Typage fort (les variables, paramètres de fonctions ont un type déterminé).
- Notions de "type d'accès" (on peut interdire la modification d'un paramètre par une fonction).
- Types de donnée complexe définis dans le langage comme les vectors (tableaux dynamiques) et les optionals (variables pouvant avoir ou non une valeur).
- Importation de fonctions utilisables en EZ et écrites en C++.

2.1 Les éléments du langage EZ

2.1.1 Identifiants

Les identifieurs sont utilisés pour nommer les différents éléments du langage, comme les variables, les types de données (structures), les fonctions et les procédure. Un indentifiant ne peut pas être un mot clé réservé du langage (voir en annexe pour une telle liste).

2.1.2 Les types primitifs du langage EZ

Le langage EZ met à disposition un ensemble de types primitifs utilisables à tout moment dans le programme.

Types simples (ou scalaires)
Les types listés dans le tableau ci-dessous sont les types simples :

Nom de type	Valeurs possibles	Exemple de déclaration	Exemple de valeur
boolean	true, false	local b is boolean	b = true
integer	valeurs d'un int64_t	local i is integer	i = -42
natural	valeurs d'un uint64_t	local n is natural	n = 42
real	valeurs d'un double	local r is real	r = -0.42
char	valeurs d'un char	local c is char	c = 'x'
string	chaîne de caractère	local s is string	s = "toto"

Le type vector

Le type vector est plus complexe que les autres. Il permet de définir un tableau dynamique d'éléments d'un type donné.

Par exemple, pour définir une variable locale de type "tableau dynamique d'entiers", on écrit :

```
local v is vector of integer
```

On peut définir des vecteurs de vecteurs :

```
local v is vector of vector of integer
```

Pour accéder au nième élément d'un vector, la syntaxe est :

```
v[n] = 42   // Assigne 42 au nieme element du vector v
```

Lorsqu'une variable a le type vector, on peut utiliser les méthodes de vector définis par le langage, comme dans l'exemple suivant :

```
v.remove(n) // Retire le nième élément du vector v
v.push(42)  // Ajoute 42 à la fin du vector v
```

L'ensemble des méthodes de vector disponible est listé en annexe.

Le type optional
Le type optional permet de définir des variables pouvant avoir une valeur ou non. Cela est nécessaire pour définir des structures de données récursives comme les arbres.

Pour définir une variable de type "entier optionel", on écrit :

```
local x is optional integer
```

Pour savoir si une variable de type optional a une valeur, on peut utiliser la méthode **is_set** :

```
if x.is_set() then
    // Ici l'on peut accéder à la valeur de x sans risque
    print x.get(), "\n"
endif
```

Pour accéder à la valeur d'une variable optional, on utilise la méthode **get** qui retourne une valeur du type de l'optional. Si la variable optional n'est pas préalablement défini, le programme s'arrêtera.

Pour donner une valeur à une variable optional, on utilise la méthode **set** :

```
x.set(42)   // Maintenant, x.get() est l'entier 42
```

Le type function
Le type function permet de définir des variables qui sont en fait des fonctions.

Pour définir une variable de type fonction, on écrit :

```
local f is function(in x is integer) return integer
```

Pour assigner une valeur à une variable de type fonction, on utilise les lambda functions (une partie de ce document sera dédié à expliquer de quoi il s'agit).

```
f = lambda (in x is integer) return integer is return x * 2
```

2.1.3 Les valeurs

Les valeurs sont l'un des éléments de base du langage EZ. On les rencontre dans les expressions (voir ci-dessous).

Voici quelques exemples de valeurs :

```
42          // Une valeur de type integer
"toto"      // Une valeur de type string
true        // Une valeur de type boolean
x           // Une valeur référençant la valeur de la variable 'x'
f(x)        // L'appel de la fonction 'f' avec pour paramètre la variable 'x'
x.y         // Référence du membre 'y' de la structure 'x'
empty Arbre // Optionelle de type Arbre (optional Arbre) vide
v[32]       // Référence du 32eme membre du vecteur v
```

2.1.4 Les expressions

Les expressions dans le langage EZ sont ce que l'on rencontre dans les paramètres d'un appel de fonction, lors de l'affectation d'une variable, ou bien lors de la définition d'une condition pour une instruction de type contrôle de flux.

Les expressions peuvent être :
— Des calculs arithmétiques ;
— Des comparaisons ;
— Des expressions booléennes ;
— Des appels de fonctions (ou procédures) ;
— La définition d'une fonction lambda ;
Voici quelques exemples d'expressions :

```
2 + 3 * 4           // Calcul arithmétique
x <= 2              // Comparaison
x <= 2 and x >= 0   // Expression booléenne
f(x)                // Appel de fonction

/* Une définition de lambda function */
lambda (in x is integer) return integer is return x * 2
```

Certaines parties d'une expression peuvent être groupés en les plaçants entre parenthèses :

```
(2 + 3) * 4
(2 + 5) * (x + 32)
```

Calculs arithmétiques

Le langage EZ donne accès à 5 opérateurs de calcul arithmétique :

```
5 + 6       // Addition
5 - 6       // Soustraction
5 * 6       // Multiplication
5.0 / 2     // Division
5 % 3       // Modulo (ou reste de la division euclidienne)
```

Pour l'opérateur '%', les valeurs de part et d'autre de l'opérateur doivent être des entiers.

Comparaisons

Le langage EZ donne accès aux opérateurs de comparaisons suivants :

```
x < y       // x est plus petit que y
x <= y      // x est plus petit ou égale à y
x > y       // x est plus grand que y
x >= y      // x est plus grand ou égale à y
x == y      // x est égale à y
x != y      // x est différent de y
```

Opérateurs booléens

Le langage EZ donne accès à 3 opérateurs booléens :

```
x and y     // x et y
x or y      // x ou y
not x       // non x
```

2.1.5 Les instructions du langage EZ

Le langage EZ donne accès à différentes instructions :
— L'instruction "print" ;
— L'instruction "read" ;
— L'instruction "return" ;
— Les affectations ;
— Les instructions de contrôle de flux ;
— Les appels de fonctions ;

L'instruction print

Cette instruction permet l'affichage sur la sortie standard des valeurs des expressions passés en paramètre. Chaque expression donnée en paramètre à l'instruction print doit être séparé par une virgule.

```
// Affiche "Hello, world" et saute la ligne
print "Hello, world\n"

// Affiche "f(x) = " suivi de la valeur de l'appel de f(x) puis saute une
// ligne
print "f(x) = ", f(x), "\n"
```

L'instruction read

Cette instruction permet la saisie par l'utilisateur de la valeur d'une variable.
Cette instruction ne prends qu'un unique paramètre qui doit être le nom d'une variable.

```
// Lit la valeur de la variable x
read x
```

La variable passé en paramètre doit avoir un type simple (entier, réel, caractère, chaîne de caractères, booléen).

L'instruction return

Cette instruction permet de définir la valeur de retour d'une fonction.
Cette instruction ne prends qu'un unique paramètre qui doit être une expression.

```
return 32
```

Les affectations

Les affectations permettent d'affecter à une variable une valeur.

```
x = 42      // Affecte la valeur 42 à la variable x
y = f(x)    // Affecte le résultat de f(x) à la variable y
```

Les instructions de contrôle de flux

Les instructions de contrôle de flux permettent d'exécuter un ensemble d'instructions si la condition de l'instruction est remplie.
Ces instructions sont ou bien la traditionelle construction "if/elsif/else" ou bien les boucles.
Voici les exemples de ces différentes instructions :

```
// Traditionnel if/elsif/else
if x < 10 then
    ndigits = 1
elsif x < 100 then
    ndigits = 2
else
    print "invalid number of digits"
    return 5
endif

// Sucre syntaxique qui équivaut à un "if" seul avec une unique instruction.
on x < 10 do print "x < 10 "

// Boucle while, affiche 10 fois "Je boucle".
while x < 10 do
    print "Je boucle\n"
    x = x + 1
endwhile

// Exécute les instructions entre "loop" et "until" jusqu'à ce que la condition
// soit vrai.
loop
    print "Je boucle\n"
    x = x + 1
until x > 10

// Exécute les instructions entre "do" et "endfor" pour chaque valeur de
// l'interval spécifié. Attention, la borne inférieur de l'interval est
// comprise, mais pas la borne supérieur (ici i prendra les valeurs de
// 0 à 9).
// La variable i doit avoir été préalablement définie.
for i in 0 .. 10 do
    print "Je boucle\n"
endfor
```

2.1.6 Les fonctions et procédures

Un programme EZ est composé d'un ensemble de fonctions et procédures.
Une procédure est une fonction qui n'a pas de type de retour.
Une fonction a :

— un nom (ou identifiant) ;
— un ensemble de paramètres ;
— un type de retour (pas pour les procédures) ;
— un ensemble de variables locales ;
— un ensemble d'instructions à exécuter (corps) ;

Voici quelques exemples de fonctions et de procédures :

```
function f(in x is integer) return integer
begin
    return x * 2
end

procedure lit_entier(out n is integer)
begin
    read n
end
```

```
function is_prime(in x is integer, in v is vector of integer) return boolean
    local i is integer
begin
    for i in 0 .. v.size() do
        on x % v[i] == 0 do return false
        on sqrt(x) < v[i] do return true
    endfor

    return true
end
```

Les paramètres d'une fonction

Les paramètres d'une fonction sont donnés entre les parenthèses suivants le nom de la fonction. Un paramètre peut être utilisé ensuite dans le corps de la fonction.

Un paramètre est composé de :

— un type d'accès ;
— un nom ;
— un type ;

```
<type d'accès> <nom> is <type>
```

Le type d'accès détermine les possibilités d'accès du paramètre dans le corps de la fonction. Ces types sont :

— in : le paramètre peut uniquement être utilisé dans les expressions, sa modification est impossible ;
— out : la pramètre peut uniquement être modifié, l'accès à sa valeur est impossible ;
— inout : le paramètre peut à la fois être modifié et lu.

Définition de variables locales

Après la ligne définissant la signature de la fonction, on peut rajouter un ensemble de variables locales (une par ligne). Ces variables sont ensuite utilisables dans le corps de la fonction.

Pour définir une variable locale, on utilise la syntaxe :

```
local <nom> is <type>
```

Quelques exemples :

```
local x is integer  // Défini une variable locale x de type integer
local v is vector of integer // Défini une variable locale x de type vecteur d'entiers.
```

Le corps de la fonction

Les instructions définissant le corps de la fonction sont placés entre les deux mots clés "begin" et "end". "begin" est placé après la définition des variables locales.

2.1.7 Les structures de données

Le langage EZ permet de définir ses propres types de données à l'aide du mot clé "structure".

La syntaxe est :

```
structure <nom> is
    <liste de membres>
end
```

Un membre de structure est défini de cette manière :

```
<nom> is <type>
```

Voici un exemple d'utilisation :

```
structure Personne is
    nom is string
    prenom is string
end

procedure saisie_personne(out p is Personne)
begin
    print "Nom: "
    read p.nom
    print "Prénom: "
    read p.prenom
end

procedure afficher_personne(in p is Personne)
begin
    print p.prenom, " ", p.nom
end
```

2.1.8 Les constantes

Les constantes sont des valeurs immuables définis globalement qui peuvent être accédés dans toutes les epxressions du programme.

La syntaxe de définition d'une constante est :

```
constant <nom> is <type> = <expression>
```

Exemple d'utilisation :

```
constant PI is real = 3.14

function aire_du_cercle(in rayon is real) return real
begin
    return r * PI * 2
end
```

2.1.9 Les variables globales

Les variables globales sont des variables définis globalement qui peuvent être accédés et affectés dans toute fonction du programme.

La syntaxe de définition d'une variable globale est :

```
constant STATUS_SUCCES is integer = 1
constant STATUS_ERREUR is integer = 2

global status is integer

procedure oups()
begin
    status = STATUS_ERREUR
end

procedure cool()
begin
    status = STATUS_SUCCES
```

```
end

procedure afficher_status()
begin
    if status == STATUS_ERREUR then
        print "oups"
    elsif status == STATUS_SUCCESS then
        print "cool"
    endif
end
```

2.1.10 Un programme en langage EZ

On appellera les fonctions, procédures, structures, constantes et gloables "entités" du programme.

Un programme doit commencer par le mot clé "program" suivi du nom du programme. Ce nom détermine le nom de la fonction principale du programme.

On peut donner après cet en-tête un ensemble d'entités.

La fonction principale retourne un entier, et prends en paramètre un vector de strings, correspondants aux paramètres donnés au programme lors de son invocation.

Voici un exemple de programme complet :

```
program prime_numbers

function is_prime(in x is integer, in v is vector of integer) return boolean
    local i is integer
begin
    for i in 0 .. v.size() do
        on x % v[i] == 0 do return false
        on sqrt(x) < v[i] do return true
    endfor

    return true
end

function average(in primes is vector of integer) return real
    local i is integer
    local sum is real
begin
    sum = primes.reduce(lambda (in x is integer, in y is integer) return integer is return x + y,
    return sum / primes.size()
end

function prime_numbers(in args is vector of string) return integer
    local primes is vector of integer
    local current_prime is integer
    local number_of_primes is integer
begin
    primes.push(2)
    current_prime = 3

    // Get the number of primes to fetch.
    if args.size() > 1 then
        number_of_primes = integer_from_string(args[1])
    else
        print "user defined"
        number_of_primes = 20
    endif
```

```
    while primes.size() < number_of_primes do
        on is_prime(current_prime, primes) do primes.push(current_prime)
        current_prime = current_prime + 2
    endwhile

    print "The first ", number_of_primes, " prime numbers are ", primes, "\n"
    print "Their average is ", average(primes), "\n"

    return 0
end
```

2.1.11 Ajouter des fonctions et types au langage EZ

Le langage EZ peut être améliorer en spécifiant des fonctions et types "builtins".

Ces fonctions sont listés dans le fichier "ez-builtins.ez". On liste dans ce fichier la signature des fonctions et procédures builtins, et le nom des types de données.

Par exemple :

```
builtin structure File
builtin procedure open_file(in path is string, in accessor is string, out file is File)
builtin function is_file_open(in file is File) return boolean
builtin procedure close_file(out file is File)
builtin function read_line_from_file(inout file is File) return string
builtin function is_file_over(in file is File) return boolean
```

Pour définir ces fonction et types, on doit les écrires en C++ dans le fichier "ez/functions.hpp" :

```
#ifndef _ez_functions_hpp_
#define _ez_functions_hpp_

#include <cstdlib>
#include <cstdio>
#include <ctime>
#include <cmath>

namespace ez {

typedef std::FILE* File;

void open_file(const std::string& path,
               const std::string& accessor,
               File& file)
{
    file = fopen(path.c_str(), accessor.c_str());
}

bool is_file_open(File file) {
    return file != NULL;
}

void close_file(File& file) {
    if (is_file_open(file)) {
        fclose(file);
        file = NULL;
    }
}
```

```
std::string read_line_from_file(File file) {
    static char buf[4096];
    fgets(buf, sizeof(buf) - 1, file);
    return std::string(buf);
}

bool is_file_over(File file) {
    return feof(file);
}

}

#endif
```

On peut ensuite utiliser ces fonctions et types dans un programme EZ :

```
program cat

procedure display_file(in path is string)
    local file is File
    local line is string
begin
    open_file(path, "r", file)
    if (not is_file_open(file)) then
        print "couldn't open file ", path
    else
        while not is_file_over(file) do
            line = read_line_from_file(file)
            print line
        endwhile
        close_file(file)
    endif
end

function cat(in args is vector of string) return integer
    local i is integer
begin
    for i in 1 .. args.size() do
        display_file(args[i])
    endfor
    return 0
end
```

3 Implémentation du compilateur

Classiquement durant la phase de vérification syntaxique, un compilateur créer un arbre de syntaxe abstraite, qui est ensuite utiliser afin de réaliser la vérification sémantique.

La vérification sémantique ce fait ensuite via l'aide des tables suivantes :
- la table des lexicographique ou table des symboles
- la table des déclarations
- la table des représentations
- la table des régions

Plutôt que d'utiliser 4 tables de la sorte nous avons opté pour une structure de contexte plus proche de la signification sémantique directe du langage. Il en résulte donc un code à la fois plus clair et plus simple à comprendre pour un oeil extérieur.

Ainsi le contexte contient deux membres : le programme et la fonction courrante, ce qui nous permet de gérer les portés des variables.

Le programme contient toutes les déclarations globales comme les structures, les variables et constantes globales, ainsi que les les functions et procédures.

Les fonctions et procédures contiennent les parametres, les variables locales, ainsi que la liste d'instruction de la fonction et le type de retour pour les fonctions.

On peut toutefois noter que le fait de se passer de la table des symboles peut entrainer une perte de performance du compilateur, l'interet de cette table étant en effet d'associer les différents identifieurs à un label (typiquement grâce à une table de hachage) permettant de les retrouver en un temps constant.

Dans notre cas, nous éxecutons une simple recherche dans un tableau, la tâche est donc alourdie. En revanche, là où classiquement après avoir trouver l'identifieur, il faut examiner la table des déclarations pour connaitre son type et sa portée, nous connaissont déjà ces informations grâce aux structures utilisées.

Le compilateur ezc ne fait donc qu'une passe pour effectuer les vérifications lexicale, syntaxique et sémantique d'un programme.

Par soucis de clareté dans le code, la phase de réécriture en langage intermédiaire (dans notre cas C++) est elle réalisée lors d'un parcours du contexte généré lors de la première passe. On aurait cependant, très bien pu imaginer n'effectuer qu'une seule et unique passe pour l'ensemble des étapes de compilations.

4 Les pistes d'amélioration du langage

Enums

Tuples

Inférence de types ? ? (Exemple en Rust : création d'un vector sans type, puis au premier push d'un élément inférence du type de vector)

Modules

Threads ? ?

Gestion des erreurs