Documentation de travail : Création d'un compilateur pour le langage Léa

Michael Färber

19 décembre 2012

Résumé

Nous avons créé un compilateur pour le langage Léa, qui est capable de créer du code Java à partir des fichiers Léa.

1 Cahier des charges précisé

Notre compilateur est capable de créer du code Java pour un fichier Léa donné. Un fichier Léa peut contenir les définitions suivantes :

- Définition des constantes.
- Définition des types.
- Définition des fonctions.

Partout dans le code Léa, on peut écrire des commentaires de façon suivante :

```
// This is a one-line comment.

/* This is a more complex,
   tremendously long,
   not very expressive
   multi-line comment. */
```

Tout fichier Léa ne peut pas utiliser des constantes, types et fonctions définis dans les autres fichiers Léa.

1.1 Définition des constantes

On peut définir des constantes sans donner leur types, de façon suivante :

```
THIS_IS_FALSE = True;
MEANING = 42;
PI = 3.14;
FIRST_LETTER = 'A';
SHAKESPEARE = "To be or not to be ...";
FIBONACCI = [1, 2, 3, 5, 8];
TUPLE = (MEANING, PI, SHAKESPEARE, FIBONACCI);
```

1.2 Définition des types

On peut définir des nouveaux types de façon suivante :

```
date = (int, int, int);

person = struct
{
        name : string;
        birthday : date;
}

person_list = struct
{
        elem : person;
        next : int_list;
}
```

Nous supportons l'utilisation des types suivants :

- Types de base :
  - bool : booléen
  - int : entier
  - float : virgule flottante
  - string : chaîne de caractères
- list of TYPE : liste d'éléments de TYPE
- (TYPE_1, TYPE_2, ..., TYPE_N) : n-uplet
- struct { ELEM_1 : TYPE_1, ..., ELEM_N : TYPE_N } : structure

1.3 Définition des fonctions

Nous distinguons deux cas différents : fonctions qui retournent une valeur (appelés procedure), et fonctions qui ne retournent pas une valeur (appelés function). Les fonctions peuvent prendre plusieurs arguments, qui sont passés par valeur.

Nous ne permettons pas d'avoir des fonctions surchargés. Aussi, nous ne permettons pas des variables des types comme arguments pour les fonctions. Par contre, nous permettons des fonctions récurrentes.

```
procedure print_fibonacci(n1 : int, n2 : int, iter : int, count : int)
{
        if (iter >= count)
        {
                return;
        }

        writeln(n1.toString());
        n3 : int := n1 + n2;
        print_fibonacci(n2, n3, iter+1, count);
```

```
}

procedure print_int_list(l : list of int)
{
        i : int := 0;
        while (i < l.length())
        {
                writeln(l[i].toString());
                i := i+1;
        }
}

function main(args : list of string) : int
{
        writeln("Hello World!");

        writeln("Fibonacci numbers:");
        print_int_list(FIBONACCI);

        return 0;
}
```

Tous les fichiers Léa doivent contenir une fonction main, qui prend une liste de chaînes de lettres et qui retourne un entier. Cette fonction est appelée au début du programme, avec des arguments donnés.

Nous supportons les instructions suivantes :

- VARIABLE := EXPRESSION;
- return EXPRESSION;
- FUNCTION(PAR_1, ..., PAR_N);
- if (BOOL_EXPRESSION) INSTRS
- if (BOOL_EXPRESSION) INSTRS ELSE INSTRS
- while (BOOL_EXPRESSION) INSTRS
- repeat INSTRS while (BOOL_EXPRESSION);

Dans les instructions, on peut avoir des expressions suivantes :

- Toutes les constantes sont des expressions.
- EXPRESSION $ EXPRESSION, où $ peut être :
  - == : égalité
  - != : inégalité
- BOOL_EXPRESSION $ BOOL_EXPRESSION, où $ peut être :
  - && : ET
  - || : OU
  - ! : NON
- NUMBER_EXPRESSION $ NUMBER_EXPRESSION, où $ peut être :
  - $+$: addition
  - $-$: soustraction
  - $*$: multiplication

    - / : division
    - < : inférieur
    - > : supérieur
    - <= : inférieur ou égal
    - >= : supérieur ou égal
  - INT_EXPRESSION % INT_EXPRESSION : modulo
  - − NUMBER_EXPRESSION : négation
  - STRING_EXPRESSION + STRING_EXPRESSION : concaténation

Il y a des fonctions predéfinis :

  - writeln : Prend une chaîne de caractères et l'affiche sur l'écran, terminé avec une nouvelle ligne.
  - write : Prend une chaîne de caractères et l'affiche sur l'écran, pas terminé avec une nouvelle ligne.
  - read : Attend l'entrée de l'utilisateur et la retourne comme chaîne de caractères.

Pour les types de base, la fonction toString retourne une chaîne de caractères représentant la valeur. Pour les listes, la fonction length retourne le nombre des objets dans la liste.

2 Problèmes envisagés

Le plus grand problème pendant l'implémentation était la coordination du développement des parts différents. Tandis que la coordination entre syntacticien et sémanticien marchait bien grâce à leur relation proche (même origine, mêmes cours, même résidence, ...), la coordination avec le développeur était plus difficile, parce-que on ne se pouvait pas voir assez souvent pour discuter le travail. Aussi, elle pouvait seulement commencer son travail après grands parts de la partie des autres étaient finis — à cause de ça, il fallait faire beaucoup de travail à la fin du projet pour elle, engendrent assez souvent des conflits avec leurs examens et autres projets.

Ceci dit, c'était un problème que je n'ai pas défini la architecture du projet assez détaillé. Ça entraînait des malentendus, qui prenaient quelque temps à résoudre et qui mettaient les nerfs à rude épreuve sans raison.

3 Architecture

Le programme est démarré avec la classe lea.Main, qui initialise le lexer et le parser. La vérification de types est fait par le parser. Si le parser termine son travail sans erreurs, il retourne une table de types, une table de constantes et une table de fonctions. A partir de ça, la classe Main appelle le générateur de code Java, qui génère le code cible Java.

Nous avons implémenté :

  - Table de types : Map<String, Type>
  - Table de constantes : Map<String, Constant>

- Table de fonctions : Map<String, FunctionInfo>
  La classe FunctionInfo sauvegarde un arbre de syntaxe abstrait, le type de sortie de la fonction, et les arguments d'entrée comme liste de ArgumentInfo. La classe ArgumentInfo sauvegarde le nom et le type d'une variable d'entrée.
- Environnement : Map<String, VariableInfo>
  La classe VariableInfo sauvegarde le type d'une variable et l'information si la variable était déjà initialisé dans le programme.
- Pile d'environnement :

3.1 Types

Les types sont implémenté dans le paquet lea.types. Dedans, il y a une classe de base Type, sur laquelle tous les autres classes dans le paquet sont basés. Ils sont les suivantes :

- BoolType : booléen
- CharType : caractère
- EnumType : énumération
- FloatType : nombre flottant
- IntType : nombre entier
- ListType : liste, sauvegarde aussi le type d'éléments dans la liste
- StringType : chaîne de caractères
- StructType : structure, sauvegarde un ensemble de variables contenus
- TupleType : uplet, sauvegarde une liste de types contenus
- UnknownType : utilisé par le parser quand il ne peut pas déterminer le type d'une expression

3.2 Constantes

Similaire de notre implémentation de types, nous avons implémenté les constantes. Ils sont contenus dans un paquet lea.constants, et la classe de base s'appelle Constant. Les classes dérivés sont :

- BoolConstant
- CharConstant
- EnumConstant
- FloatConstant
- IntConstant
- ListConstant
- StringConstant
- TupleConstant

Les classes sont utilisés à la déclaration des constantes, mais il y a aussi des classes qui sont utilisés dans les fonctions ; ils sont BoolConstant, CharConstant, FloatConstant, IntConstant et StringConstant. Les classes ListConstant et TupleConstant ne sont pas utilisés dans les fonctions — au lieu de cela, on utilise ListNode et TupleNode là.

3.3 Arbres de syntaxe abstraites

Les arbres de syntaxe abstraites sont implémentés dans le paquet lea.syntax. Là, nous trouvons la classe SyntaxTree, qui est la classe de base pour toutes les autres classes dans le paquet. Cette classe SyntaxTree contient l'environnement courant et des pointeurs à un fils gauche et un fils droite, qui permettent de traverser l'arbre de syntaxe d'une façon très simple.

Nous différencions deux types de nœuds dans l'AST ; des expressions et des instructions. Les expressions sont implémenté dans la classe Expression, qui permet de déterminer le type de l'expression. Il y a plusieurs classes hérités de Expression :

- BoolExp : expressions qui sont de type booléen, comme a != b ou True || False
- ConstantLeaf : constante
- EnumExp : ? ? ?
- FunctionCall : appel de fonction
- FunctionRef : ? ? ?
- ListAccessor : expression de genre liste [i]
- ListNode : expression de genre [1, 2, 3, 4, 5]
- NumberExp : expressions qui sont de type numérique, entier ou flottant, comme 2.0∗3.14 ou 5%3
- StringExp : concaténation des chaînes de caractères, comme "Hello" + " world!"
- StructAccessor : expression de genre structure.element
- TupleNode : expression de genre (1, 'b', C)
- TypeExp : ? ? ?
- VariableLeaf : variable

Les instruction sont implémenté dans la classe Instruction. Les classes hérités sont :

- Assignment : affectation
- Case
- Condition
- ElseCondition
- Loop :
- Repeat
- ReturnNode
- Succ : succession de deux instructions
- While

3.4 Traduction en code cible Java

A faire !

4 Utilisation

Appeler « ant » exécute les tâches suivantes :

- Créer des fichiers Java à partir des fichiers CUP/JFlex (src/lea.cup et src/lea.jflex).
- Compiler tous les fichiers Java (dans src/).
- Exécuter le compilateur sur les fichiers Léa mentionnés dans build.xml (target « default »), écrivant les messages du compilateur dans data/output.txt.

A partir de tout fichier Léa, le compilateur crée un nouveau répertoire (portant le même nom comme le fichier Léa), où il place :

- Fichiers Java contenant le code cible pour le fichier Léa.
- Arbres de syntaxe abstraites pour toute fonction dans le fichier Léa, dans le format DOT.

Appeler « ant clean » efface tous fichiers qui sont créés par « ant », sauf les fichiers dans data/.