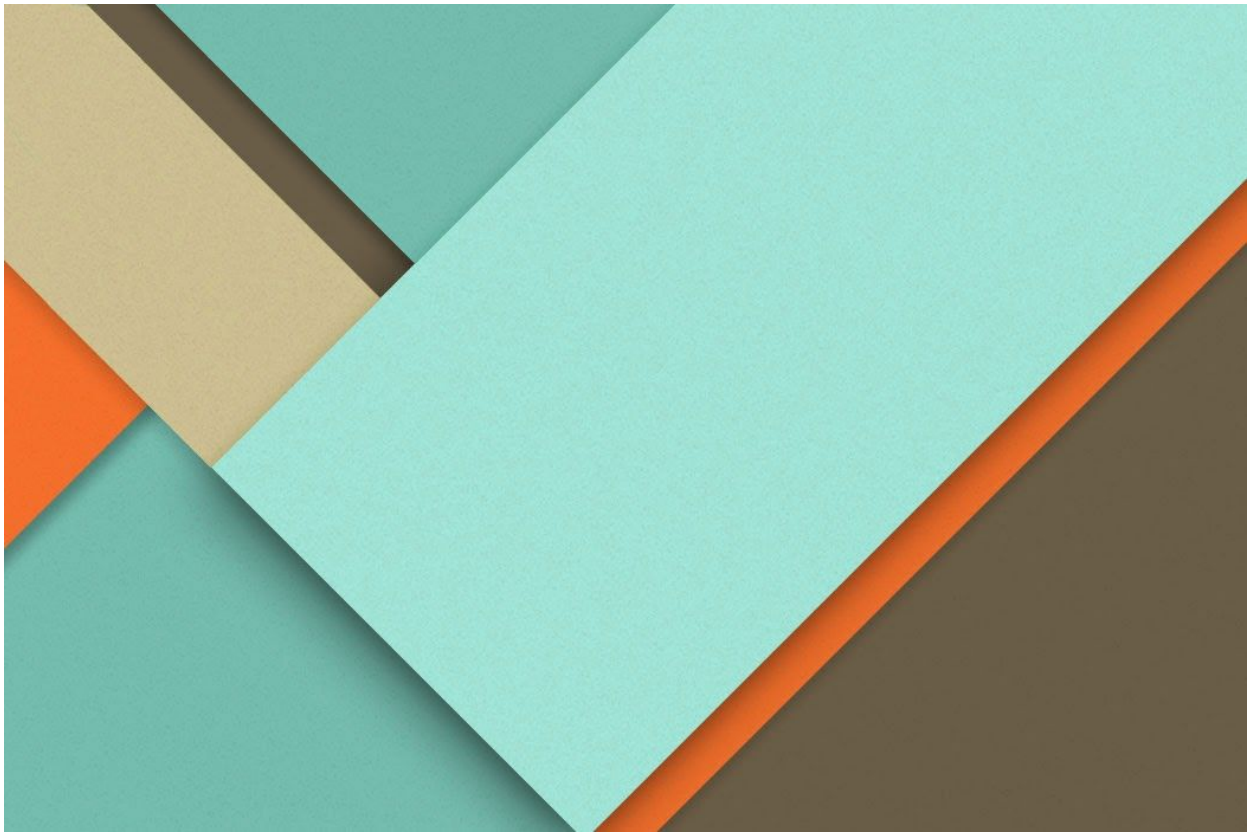

APS

27.02.2020 - 29.05.2020

— Pierre-Octave VELLY 3682130 - Quentin PIOTROWSKI 3407333

Introduction

Dans le cadre de l'UE APS, nous avons développé notre propre langage de programmation respectant les spécifications *aps0* (langage fonctionnel basique) et *aps1* (ajouts de traits impératifs au langage) qui nous ont été fournies. Le langage est prévu pour être interprété, ou bien compilé vers du Prolog. L'ensemble est développé en Java, à l'exception du *typer* réalisé en Prolog. Nous avons été au bout de l'implémentation et l'ensemble des tests fournis est **compilé, typé et interprété correctement**.

I. Exécuter notre code

I. Compilation des sources

La compilation des sources se fait grâce au MakeFile présent dans le dossier *src* (il appelle celui du dossier *src/parser*).

II. Scripts de test

3 scripts sont présents dans le dossier *src* et s'exécutent conformément à ce qui a été établi dans les consignes :

- *eval*
- *prologTerm*
- *typrog*

Les 3 scripts acceptent en argument soit *aps0*, soit *aps1*, soit directement un fichier de test spécifique (ex : *aps0/prog000.aps*). Tous les programmes APS testés sont dans le dossier *samples* et sont ceux fournis dans le cadre de l'UE.

Les scripts affichent leur résultat sur la sortie standard ; *prologTerm* sauve également sa sortie dans un fichier.pl.

II. Structure du projet

I. Détails du Lexer et du parser

Nous avons généré le parser du langage APS à l'aide de JFlex pour l'analyse lexicale (cf *lexer.lex*) et BYacc/j (cf *parser.y*) pour l'analyse syntaxique, en respectant scrupuleusement la spécification fournie par P. Manoury.

II. *APS0*, *APS1* et interfaces

Le coeur du langage APS est réparti sur les packages *aps0*, *aps1* (correspondant respectivement aux deux versions du langages) et *interfaces* qui contient l'ensemble des interfaces Java nécessaires au langage (comme par exemple *IASTexpression* ou *IASTtype*) ; certaines sont dispensables mais ont été laissées avec pour but de rendre le code plus lisible.

III. Le *Compiler* vers Prolog

Le package *compiler* contient uniquement la classe *Compiler* qui implémente un pattern de visiteur qui pour chaque noeud de l'AST va retourner une chaîne de caractères correspondant à un programme Prolog équivalent.

IV. Le *typer*

Le typechecker, écrit en Prolog, permet l'analyse de type d'un code prolog et retourne *ok* si le code est correctement typé, *ko* sinon. Notre implémentation suit rigoureusement les différentes règles détaillées dans la spécification.

V. Détails de l'interpréteur pour APS0, APS1 et les exceptions

Enfin, le package *interpreter* contient toutes les classes nécessaires à interpréter un programme APS.

La classe *Interpreter* implémente un visiteur pour chaque noeud de l'AST ; *Context* représente un contexte d'exécution sous la forme d'une liste chaînée. Le package contient également trois classes implémentant l'interface *ExpressionEvaluator* qui en fait des visiteurs spécifiques pour les noeuds correspondant à une expression : *BooleanEvaluator*, *IntegerEvaluator* et *PartialEvaluator*. Ce dernier permet d'ajouter le support pour l'évaluation partielle de variables dans des applications. Nous reviendrons sur les deux autres dans la dernière partie concernant nos choix d'implémentation.

Le package exception contient les exceptions que peut lever l'interprétation d'un programme APS : *ArityException* (quand une fonction est appelée avec le mauvais nombre d'arguments), *TypeException* (quand le type des arguments d'une fonction ne correspondent pas à sa signature) et *UnboundVariableException* (appel d'une variable non définie).

III. Choix d'implémentation

I. La mémoire gérée comme un contexte

Nous avons fait ce choix d'implémentation car nous avons visualisé la mémoire comme une structure similaire a celle utilisé par le contexte.

Nous avons beaucoup hésité au sujet de ce choix - c'est pourquoi nous avions demandé un review de notre code, malheureusement resté sans réponse - puis finalement décidé de laisser tel quel. Avec le recul, il nous semble qu'il aurait été plus judicieux de gérer la mémoire autrement car notre implémentation pose un problème lors de l'appel d'une procédure (*CALL*) ; nous aurions eu besoin d'aide sur ce point précis.

II. Interprétation de programmes type (add 1 bool)

Un programme tel que *(add 1 bool)*, bien que incorrect du point de vue du typage est valide au regard des règles de syntaxes telles que définies par la spécification *aps0*. Nous avons donc fait le choix de supporter de tels programmes en ajoutant deux nouvelles classes sur le modèle de l'interface *ExpressionEvaluator* : *BooleanEvaluator* et *IntegerEvaluator*. Ces deux visiteurs retournent pour chaque type d'expression rencontré un integer ou bien un boolean équivalent.

III. Analyse de type a l'interpretation

En plus du typer en Prolog, nous avons choisi d'implémenter une analyse de type à l'interprétation qui permet de vérifier si les arguments d'une application ou d'un appel de procédure étaient correctement typés et lève une exception dans le cas contraire. C'est un autre visiteur : *TypeChecker* qui s'en charge, une fois encore sur le modèle d'*ExpressionEvaluator*.

IV. Absence des opérateurs primitifs dans le contexte

Nous avons choisi de représenter le typeChecking des opérateurs primitifs par des fonctions plutôt que de les mettre dans le contexte initial, que nous avons donc laissé vide. En nous éloignant légèrement de la spécification, il nous semble que nous avons gagné en lisibilité.

Conclusion

Ce projet nous a permi d'avoir une bonne vision du développement d'un langage de programmation fonctionnel ; nous avons notamment pu découvrir comment mettre en place les analyses lexicales et syntaxiques pour générer notre parser. Bien que nous avions déjà étudié l'interprétation dans l'UE DLP au premier semestre, l'aspect fonctionnel du langage était une vraie nouveauté pour nous. Enfin, compiler notre code vers du Prolog, et l'utiliser pour ajouter une vérification de type, était une bonne manière de découvrir un nouveau langage, très différent de ce que nous avions étudié jusqu'alors.