UNIVERSITÉ PIERRE ET MARIE-CURIE

MASTER 1
PROJET APS 2017

4I503 APS

réalisé par :

Hacene KASDI - 3524196

Année universitaire : 2016-2017

Table des matières

Introduction

De point de vue statique, un programme est un fichier : lignes de code lisible par le programmeur et peut être compilé par la machine, ou bien un ensemble de bits qui peut être compris et interprété que par la machine.

lors de la compilation d'un programme d'un langage compilé, le code source est donné à un programme appelé **compilateur** qui va lire le code source et le convertir dans un langage que l'ordinateur sera capable d'interpréter : c'est le langage binaire, fait de 0 et de 1. Les langages comme le C, C++, Pascal et OCaml sont des langages dits compilés, nous définissons dans les sections suivantes les trois manières d'utiliser du code source.

Dans notre projet APS, nous traitons en premier lieu APS0 notre premier langage, Il contient le noyau impératif minimal que l'on rencontre dans la plupart des langages de programmations. ensuite on passera à l'implémentation de quelques traits de langage APS1 et les nouveautés apporté à cette nouvelle version également les expressions fonctionnelles et ensuite à la fin on parlera de APS2 en intégrant les fonctions et procédures et l'ajout du mot clé RETURN et au final nous présentons APS fonctions et procédures récursives.

Chapitre 1

Les Extensions du langage APS

Dans ce Chapitre nous allons présenter le langage APS avec ses quatres extensions(APS0, APS1, APS2, APS3), chaque ajout du trait du langage apportera des modifications au niveau de la syntaxe et au niveau du typage et évaluation.

1.1 Les trois manières d'utiliser du code source

— Langage compilé : le code source est donné à un programme appelé compilateur qui va lire le code source et le convertir dans un langage que l'ordinateur sera capable d'interpréter : c'est le langage binaire, fait de 0 et de 1. Les langages comme le C, C++, Pascal et OCaml sont des langages dits compilés.

— Langage précompilé : ici, le code source est compilé partiellement, généralement dans un code plus simple à lire pour l'ordinateur, mais qui n'est pas encore du binaire. Ce code intermédiaire devra être lu par ce que l'on appelle une « machine virtuelle », qui exécutera ce code. Les langages comme le C ou le Java sont dits précompilés.

— Langage interprété : dans ce cas, il n'y a pas de compilation. Le code source reste tel quel, et si on veut exécuter ce code, on doit le fournir à un interpréteur qui se chargera de le lire et de réaliser les actions demandées. Éventuellement, pour obtenir de significatifs gains de performances, le code peut-être compilé à la volée pendant son exécution, c'est aujourd'hui ce que font la plupart des interpréteurs JavaScript.

1.2 APS0

La première version de notre langage est APS10, Il contient le noyau impératif minimal que l'on rencontre dans la plupart des langages de programmations. Ses programmes ne manipulent que des valeurs entières ou booléennes. Les composants du langages sont des opérations de base sur les entiers et les booléens ; les instr- cutions d'affectation, d'alternative et de boucle non bornée ; des déclarations de variables et de constantes ; des blocs encapsulant déclarations et instructions. [1]

1.3 APS1

La version APS1 est est une 'extension du noyau APS0 avec des **expressions fonctionnelles**. La première version du langage APS (APS0) ne fournit pas l'opérateur de comparaison «inférieur ou égal >= ». Toutefois, pour toutes expressions arithmétique, on peut calculer la valeur de cette comparaison en combinant les opérateurs primitifs lt et eq sachant que ces deux paramètres existent en APS0 : (or (lt e1 e2) (eq e1 e2))

1. Voir les fichiers aps.l et aps.y du projet dans le dossier LEXPARSER/

1.3.1 Extensions apportées à à APS1

1.3.1.1 Abstraction et Application

Expr :...		
	| '[' Args ']' Expr	{ $$ = make_abstraction($2,$4) ;}
	| '(' Exprs ')'	{ $$ = $2 ; }
Exprs :		
	Expr	{$$ = $1 ; }
	|Expr Exprs	{$$ = make_application($1,$2) ;}
Args :		
	Arg	{ $$ = $1 ; }
	| Arg ',' Args	{ $$ = make_args($1,$3) ;}
Arg :	TOKEN_IDENT ' :' Type	{$$ = make_affect_arg(strdup($1), $3) ;}
Type :	TOKEN_BOOL	{ $$ = make_type(T_BOOL) ; }
	| TOKEN_INT	{ $$ = make_type(T_INT) ; }
	| '(' Types TOKEN_GIVES Type ')'	{ $$ = make_type_func_in_out($2,$4) ; }
Types :	Type	{$$ = make_a_type($1) ;}
	| Type '*' Types	{$$ = make_type_func_in($1,$3) ;}

1.4 APS2

Dans cette section nous allons définir une fonction dont le corps n'est pas une expression, mais c'est une suite de commandes qui a une particularité, c'est que elle peut produire une valeur, en introduisant la commande RETURN comme un nouveau trait du langage.

```
Lexique   : on ajoute le mot clef RETURN
Grammaire : on ajoute un non-terminal spécifique au RETURN.

   Ret ::=
           RETURN Expr

NB : la particularité et la notion qu'on devrait retenir, tout code
     écrit après la commande RETRUN il ne sera jamais évalué dans ce
     programme, il est considéré comme du code mort.
```

1.5 APS3

Pour enrichir le langage APS2 nous allons apporter quelques modifications et ajouts au niveau de la syntaxe et la grammaire du langage, et nous allons introduire un peu de dynamicité dans les fermetures définie dans APS2 pour assigner une valeur aux procédures et fonctions n'offrent pas cette possibilité.
Nous allons utiliser quelques notions vues précédemment également la notion du point fixe " !" qui va nous aider pour représenter les fonctions récursives (mais ça ne marche pas sur les procédures récursives).

1.5.1 Extensions apportées à APS3

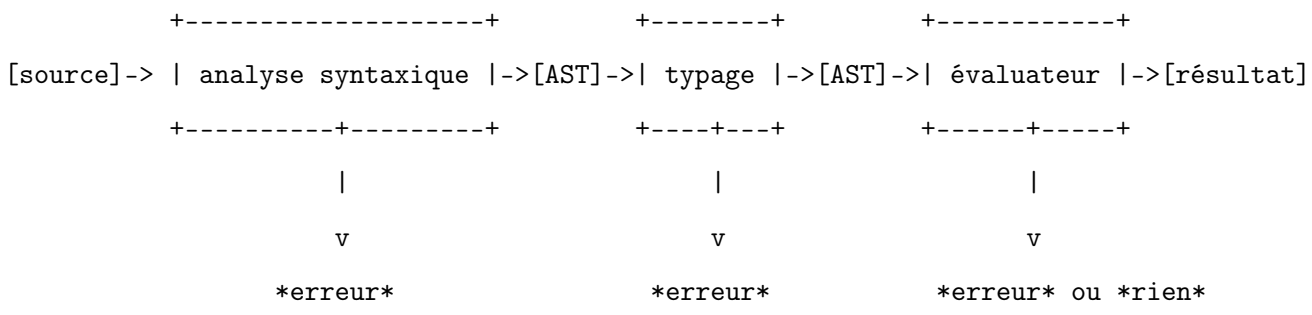
```
Lexique   : on rajoute le mot clef REC
Grammaire : on étend les possibilités de déclarations

    DEC ::= FUN ident TYPE [ARGS] EXPR
            FUN REC ident TYPE [ARGS] EXPR
            FUN REC ident TYPE [ARGS] CMDS
            FUN REC ident [ARGS] CMDS
```

Chapitre 2

Mise en oeuvre

Dans cette section je vais vous expliquer comment mettre en oeuvre les trois éléments essentiels de traitement des programmes APS, que sont l'analyse syntaxique, l'analyse de type, et l'évaluation par la réalisation des composants logiciels correspondant. le Schéma ci-dessous illustre les trois phases :

2.1 Mise en oeuvre de l'outil



2.1.0.1 Analyse syntaxique

Pour la génération d'un parser, et considérant que chaque noeud de l'AST est de type AST (voir ../AST/ast.c), généré à l'aide des fonctions du fichier ast.c, Après cette phase un arbre de syntaxe abstrait sera généré et sera passé à l'étape suivante qui est la phase de typage du programme.

Lex en C : ../LEXPARSER/aps.l

Yacc en C : ../LEXPARSER/aps.y

```
AST :          ../AST/ast.c
               ../AST/ast.h
```

2.1.0.2 Typage du programme et génération du code Prolog

Le code Prolog est généré après la phase de typage sur un fichier (programme aps), en utilisant les fonctions du de génération du code Prolog qui se trouvent dans le fichier prolog_gen.c et prolog_gen.h du dossier ../PROLOG_GEN/.
Le Typer on va lui passer en paramètre un fichier aps <program.aps> et nous génère du code prologue qui sera vérifié par la suite par l'interpréteur PROLOG si le programme est bien typé ou pas.

```
On distingue deux règles du Typer des programmes APS:

* type(Statement, Environment,Type)
* typeNamed(Statement, Environment, NewEnvironment, Type)

L'environnement est ici représenté sous la forme d'une liste de triplets
tr(NOM DE VARIABLE, CONSTANT, TYPE). La suite de trois valeurs au lieu de
deux dans l'environnement permet de connaitrel'accès et en écriture à des CONST.
```

```
Le typer :                          ../TYPER/typer.c
le générateur du code PROLOG :      ../PROLOG_GEN/prolog_gen.c
le code généré :                    ../SRC/pl_generated.pl
```

2.1.0.3 Vérification du code PROLOG généré

Les régles de typages se trouve dans le fichier PROLOG **typerProlog.pl**, le processus de vérification commence en lançant l'interpréteur SWIPL et en paramètre le fichier **typerProlog.pl**, ensuite on ouvre le fichier à vérifier [pl_generated.pl]. et le programme nous renvoie true s'il est correctement typé false si non.

$ swipl typerProlog.pl

?- [pl_generated].

?- true

2.2 Manuel d'utilisation

2.2.1 Génération du parser

— $ make parserlexer

Le parser sera généré dans le fichier LEXPARSER/

2.2.2 Génération du typeur

— $ make typer

Le fichier exécutable **typerProgram** sera généré dans le dossier courant.

2.2.3 Génération de l'évaluateur des programmes

— $ make evaluate

Un fichier exécutable **evaluator** sera généré dans le dossier courant

2.2.4 Vérification du typage d'un programme aps

Les prgrammes aps se trouvent dans le dossier PROGRAMS/, On va vérifier le typage d'un programme aps et un fichier temporaire sera généré dans le dossier SRC/ sous le nom pl_generated.pl.

— $ ./typerProgram <programAps.aps>

le code Prolog généré sera testé dans le fichier SRC/, pour voir s'il est correctement typé.

— $ cd SRC/ swipl typerProlog.pl
— ?- [pl_generated].
— true

2.2.5 Génération de l'AST et évaluation

— $ ./evaluator <programAps.aps>

2.3 L'état d'avancement du projet :

Implémentation	syntaxe	Grammaire	Typage	Evaluateur
APS 0	X	X	X	X
APS 1	X	X	X	
APS 2	X	X		
APS 3				