Auteur:
Daniel El-Masri (ID: 20096261)
Philippe Marsan-Loyer (ID : p1054077)

IFT2035 Petit-Interp. – Rapport

1) Explication du programme :

Le programme est divisé en deux grande partie, l'analyseur syntaxique et lexical (fonction parse) et l'interpréteur (fonction execute) :

Analyseur lexical et syntaxique :

Lexical : Il s'agit de la partie du programme qui s'occupe de lire le fichier donné en paramètre au programme. Lorsque cette partie du programme est appelée elle parcourt le fichier dans le but d'aller chercher le prochain symbole le reconnu par l'analyseur. Les symboles reconnue sont soit un des symboles suivant : '(' , ')', ';' , '+' , '-' , '*', '/', '%', '<', '>', '!', '=', '{' et '}', un des mots spécifiques suivant : "print", 'if', 'else', 'while' et 'do', un entier, ou sinon il construit un nom de variable composé uniquement de lettre minuscule. Une fois un symbole reconnue la fonction renvoie le symbole scheme utilisé pour représenter ce symbole et le reste du fichier après ce symbole.

Syntaxique : L'analyse syntaxique reçoit en paramètre les symboles lus par l'analyse lexical et déclenche ensuite le traitement désirer selon le symbole reçu. Ce traitement est défini selon la grammaire fournit dans l'énoncé du devoir. En d'autres mots selon le type de symbole des appels récursifs seront fait dans le but de construire un énoncé pour le symbole reçu. C'est au cours de cette partie du code que les erreurs de grammaire seront repérées puisque c'est dans le traitement définie pour un symbole que l'on test si les symboles attendus apparaissent dans l'ordre prévue.

Interpréteur :

L'interpréteur reçoit l'arbre de syntaxe abstraite construit par l'analyseur lexical et syntaxique et le parcours de façon récursive pour interpréter le programme passé en paramètre. Durant le parcours de l'ASA l'interprète génère un environnement dans lequel les expressions sont évalué. Cette environnent correspond à une liste de pair ou la première partie de la paire est le nom de la variable et la deuxième la valeur qui lui est associé. L'interprétation consiste donc à garder à jour cette environnent dans le but d'exécuter les différents énoncés en s'assurant que les variables utilisées est la bonne valeur. Les énoncés 'print' s'occupe de créer la sortie du programme dans une 'string' qui sera renvoyé comme valeur de retour de la partie interprétation du programme.

2) Problème de programmation :

a) Comment se fait l'analyse syntaxique :

L'analyse syntaxique à son plus haut niveau l'analyse d'un statement. A ce niveau l'analyseur vérifie si le symbole obtenu peut débuter un statement, si c'est le cas il appel la fonction qui gère le cas spécifique pour le symbole sinon il essayera de générer une expression. A c'est fonction il passera comme continuation celle qu'il a reçu en paramètre. La création d'un statement ou d'une expression est toujours identique, plusieurs appels seront faits sur les inputs qui suivent le symbole de début pour aller chercher les valeurs qui se doivent de suivre dans le but de respecter la grammaire. À chaque fois qu'un symbole une expression ou un statement est nécessaire la fonction correspondante est appelée et on lui passe en paramètre comme continuation la suite de la fonction dans laquelle le statement ou l'expression est construite. Une fois toute les symbole, expression et statement nécessaire obtenue on peut construire le nœud désiré et ensuite le passer en paramètre à la continuation qui était obtenue au départ pour la construction du statement ou de l'expression. Les erreurs de syntaxes sont détectées lorsqu'un symbole attendu n'est pas trouver, souvent dans le cas où une expression est attendu le symbole qui ne sera pas trouvé sera un '('.

b) Comment se fait l'interprétation du programme :

Les expressions : Pour les expressions il y a trois cas, une expression arithmétique ou un test, un entier ou une variable. Dans tous les cas on renvoie avec la continuation utiliser l'environnement, l'output et la valeur obtenu pour le traitement.

Pour évaluer une expression arithmétique ou un test il suffit d'aller évaluer les deux enfants du nœud et ensuite d'y appliquer le traitement désiré selon le test ou l'expression arithmétique du nœud.

Pour évaluer un entier on fait simplement retourner la valeur de l'enfant du nœud

Pour évaluer une variable, on parcourt l'environnent passer en paramètre pour essayer de trouver la variable, si celle-ci est trouvé on renvoi la valeur qui lui est associé, si elle n'est pas trouvée on renvoie une liste vide (équivalent a null) et on créer la variable dans la liste en lui assignant la valeur null.

Le cas ASSIGN est un cas spécial discuté en c)

Les statements : Selon la valeur d'un nœud une séquence d'appel pour obtenir d'autre statement ou d'autre expression est lancé en utilisant la partie de l'ASA désiré. La continuation est appelée avec le nouvel environnent et le nouvel output construit lors de ces appels.

c) Comment se fait l’interprétation des affectations aux variables et la gestion de l'environnement :

Pour une affectation à une variable, le tout commence avec le cas ‘ASSIGN dans la fonction exec-expr. La première partie de la fonction consiste à aller évaluer le deuxième enfant du nœud, ainsi une fois évalué la suite de la fonction aura reçu en paramètre la valeur qui devra être assigner à la variable. Ensuite on lance une recherche avec assoc sur l’environnement pour voir si la variable existe déjà. Si ce n’est pas le cas ou ajoute la variable avec ça variable au début de l’environnent autrement on donne la nouvelle paire et l’environnement à la fonction update-env qui parcours à l’aide d’un map l’environnement, et construit le nouvel environnement en reprenant telle quelles les paires de l'environnement qui n’ont pas le même nom de variable et en changeant pour la paire passé en paramètre si c’est le cas.

On note ici qu’il est assumé qu’il existe uniquement un environnent globale puisqu’il est impossible de déclarer explicitement des variables, pour dire exactement dans quel contexte celles-ci doivent être considéré.

d) Comment se fait l’interprétation des énoncé print :

Pour un énoncé print, le tout commence avec le cas ‘PRINT dans la fonction exec-stat.
La première partie consiste à aller évaluer la valeur de l’enfant du nœud avec la fonction exec-expr en lui donnant comme continuation le reste la fonction. Une fois le reste de la fonction appelé, elle aura reçu en paramètre la valeur de l’expression qui était son enfant. La fin de la fonction consiste à appeler la continuation reçue en paramètre en retournant l’environnent obtenu, et en retournant le nouvel output correspondant à l’ancien output concaténer avec la valeur de l’enfant du nœud qui vient d’être évalué.

e) Comment se fait le traitement des erreurs :

Le traitement des erreurs syntaxique est fait à l’aide de la fonction ‘syntax-error()’ qui est appelée lorsque l’analyseur syntaxique rencontre un symbole non définie, où lorsqu’un symbole attendu est manquant. Dans ce cas, la fonction est appelée avec le code d’erreur correspondant à l’erreur et celle-ci renvoie le message d’erreur.

Pour les erreurs d'exécution, le processus est le même que pour les erreurs syntaxique. La fonction ‘execution-error()’ est définie et lorsque l'exécution rencontre une erreur comme une variable non définie où bien une division par 0, l'interprète appel la fonction avec le code correspondant. Le traitement d’erreur d'exécution cependant n'interrompt pas la totalité de l'exécution. Le output généré avant l’erreur est renvoyée à la console.

À noter qu’une erreur détecté interrompt l’analyse syntaxique en retournant un string directement, sans passer par une continuation donné, ce qui met fin au programme.

3) Expérience de Développement:

Comparé à notre petit-comp.c qui fait 787 lignes de codes, notre petit-interp.scm en fait 704, ceci étant dit le petit-interp à un langage moins développé et un implémente à la fois un compilateur et un interprète, il est donc difficile de tirer une quelconque conclusion selon le nombre de ligne.

Nous avons découvert qu'un des avantages de Scheme est la facilité du retour en arrière. En effet, grâce au continuation, l'appel à next-sym nous permet d'aller analyser le prochain symbole tout en gardant le input de départ dans la fonction appelante, ceci nous permet d'aller voir dans le futur sans nécessairement avoir à garder dans une autre variable l'ancienne input. Cela était également particulièrement pratique pour implémenter le while, car on avait la possibilité de retourner dans l'ASA pour le ré-évalué la condition, au lieu d'avoir une gestion de pointeur à faire.

Cependant, nous avons remarqué que pour ce qui est des structures de données, C nous permet d'implémenter des structures plus intuitives sous notre control, comme par exemple les noeuds d'un arbres. Avec Scheme, nous sommes limité aux listes, donc le nombres procédures et méthodes qui sont à notre disposition est réduit. Aussi, C semble être favorable pour une programmation séquentielle comme dans ce cas. Il est beaucoup plus simple de suivre la logique du code puisque celui n'est pas qu'une imbrication de fonctions. En d'autre terme, il est plus intuitif d'avoir des valeurs de retours lorsqu'une fonction est appelée que d'avoir son retour comme les arguments d'une fonction de continuation qui suit.

Les bénéfices de la programmation fonctionnel peuvent être reconnue en regardant l'exécution de l'ASA par l'interprète où l'analyse de l'arbre ne demande pas de garder celui-ci intact. Lorsqu'une branche est exécutée, l'interprète n'as pas à re-analyse la totalité de l'arbre, mais seulement les parties non-exécuté. La possibilité de continuations nous permettent de générer du code qui peut se façonner sans avoir à implémenter concrètement une structuré spécial. C'est exactement cela qui se passe au cours de la production de notre ASA. Par exemple, la définition d'une nouvelle fonction <mult-expr> nous permet quand même de travailler sur la même série d'entrées mais en définissant un nouveau traitement, une fois ce traitement terminé on peut simplement retourner au traitement précédent des entrées grâce à la continuation donné avant l'appel.

Cependant, une des parties où la programmation fonctionnel à été détrimental fut durant la gestion de l'environnement. Cela vient du fait que les affectations de variables sont interdit, donc lors du traitement de ceux-ci, l'environnement doit être reconstruit complètement avec la modifications au lieu de pouvoir modifier directement la variable affectée. Toutefois, la portée de variable syntaxique et dynamique pourrait être plus simple a implémenter étant donné qu'il suffit de passer l'environnement globale au bloc de code lorsque ceux-ci sont exécuté, par la suite il développeront l'environnement local, et une fois l'exécution terminé avec la continuation on retourne dans la fonction globale qui elle aura toujours sont

environment inchangé, dans le cas de programmation impérative il faudrait définir un nouvel environment locale ce qui occasionnerait plus de gestion mémoire.

Les continuations sont majoritairement utilisé pour imposer une séquence spécifique lors du traitement d'une fonction. Cela nous permet d'aller analyser vers l'avant l'ASA ou les inputs donné pour ensuite revenir avec la continuation dans la fonction appelante pour ainsi renvoyer le résultat de la fonction appelé dans la fonction appelante.

Certaines fonctions récursives ne sont pas en formes itératives comme <seq-stat>, <expr>,, puisque ceux-ci appel une continuation à la fin de leur exécution après avoir faite un appel récursif.Or, il y a aussi une fonction récursive itérative, <mult-expr> où l'appel de la récursion est en position terminal. Elles est utilisée lorsque l'analyse syntaxique de la séquence de caractères demande un traitement spéciale, comme par exemple dans le traitement des opérations arithmétiques, où l'ASA doit être construit en suivant les règles des ordres d'opérations.