MINIML INTERPRÉTEUR

Victor ROBERT LAMBRECHT

Adam SEKKAT

Professeur: Thibaut BALABONSKI

Matière: Compilation

Groupe: LDD3 Maths-Info

Date: *08/01/2023*

Date: 08/01/2023.

Abstract. Nous présentons ci dessous le déroulé du projet MiniML du cours de compilation qui consistait à créer un interpréteur pour un langage Mini-Caml, une version incomplète de Ocaml. Ce projet réunissait presque toutes les connaissances que nous avions apprises au cours du semestre. Il n'était pas très long en nombre de lignes écrites, mais éxigeait une bonne capacité de raisonnement. Il existait deux stratégies pour faire le projet, une consistait à écrire chaque fichier séparément, et l'autre consistait à implémenter chaque cas de figure, expression par expression. Nous avons alterné entre ces deux stratégies en fonction du besoin. Au début nous avons écrit chaque fichier séparement, puis sur la fin lorsqu'il fallait implémenter les vérifications de types sur les structures et l'interpretation des fonctions récursives, la difficulté était plus élevée nous sommes passés à une stratégie de cas par cas.

Le sujet avait une complexité croissante, les difficultés principales que nous avons encontrées étaient:

-Savoir par où commencer, souvent le professeur avait déjà donné un exemple d'une implémentation "facile", qui nous guidait pour la suite

-Comment les différentes parties et les différents fichiers du programme communiquaient

-Garder le schéma global du programme en tête

-Debugger une erreur: principalement pour le typechecker et l'interpreteur puisque qu'on avait pas beaucoup d'information sur la provenance de l'erreur et la ligne où ça bloquait, et on ne pouvait pas se servir d'un "Printf.printf" dans le programme pour débugger.

-Implémenter les structures, trouver qu'elles étaient automatiquement mémorisées par le prog.types, et faire le lien entre Tstrct, strc, Strct, Vstrct.

Le projet a été compilé en utilisant ocamlbuild.

Nous avons décidés dans ce rapport de ne pas commenter spécifiquement chaque ligne de code que nous avons écrite, mais plutôt de parler des choix et stratégies que nous avons pris au cours du projet, des difficultés que nous avons encontrées, et comment nous les avons résolues. Un peu paradoxalement ce projet a été écrit en Ocaml, on peut se demander si il serait possible d'écrire le compilateur entier d'Ocaml en Ocaml sans qu'il y ait de circularité, le projet pourrait t-il se compiler lui même ? Le projet a été traité dans son intégralité, quelques extensions sont décrites en dernière partie.

Contents

Part 1. Analyse syntaxique

L'objectif de cette partie était de compléter les fichiers de lexeur et de parseur afin d'avoir un programme qui puisse lire un fichier de code et produire un arbre de syntaxe abstraite.

1. Le lexeur

Il n'y avait pas de difficulté particulière, parfois il était nécessaire de réflechir un peu pour savoir par où commencer, mais une fois lancé la progression était assez fluide.

1.1. **Fonction keyword_or_ident.** Nous avons complété la fonction keyword_or_ident en associant chaque string à son mot clef.

1.1.1. *Difficultés.* Il n'y a pas eu de difficultés particulières.

1.2. **Règles pour les tokens.** Nous avons complété cette partie comme pour la fonction keyword_or_ident en associant des symboles avec des tokens.

1.2.1. *Difficultés.* Il n'y a pas eu de difficultés particulières.

2. Le parseur

2.1. **La grammaire.** Nous avons simplement appliqué les règles de grammaire qui nous étaient données dans le sujet du DM.

2.1.1. *Difficultés.*

- On a eu un peu de difficultés à gérer le cas où le signe "-" était une opération unaire. Le problème se situait au niveau des priorités: le signe "-" lorsque c'est une opération binaire a une priorité plus faible que la multiplication, mais lorsque c'est une opération unaire a la même priorité que la multiplication. Nous nous sommes emmêlés les pinceaux, et avons ajouté un nouveau token "NEG" (que nous avons aussi ajouté dans le lexeur) qui permettait de gérer le cas où il le signe "-" était une opération unaire. Ça n'a pas fonctionné et finalement nous avons simplement utilisé le token MINUS à la place, et ça a fonctionné: le parenthésage de 2*-3 était bien (2*(-3)) mais nous ne savons pas pourquoi. On peut tout de même remarquer que de toute façon l'ordre n'était pas très important puisque la multiplication est une opération commutative: 2*-3=-(2*3)=(-2)*3=2*(-3).
- Aussi nous avions oublié d'implémenter le sucre syntaxique pour le Fix dans les règles grammaticales. Puisque l'analyse syntaxique ne renvoyait pas d'erreur, nous n'avons pas remarqué qu'il y avait un problème. C'est seulement lors de l'interpretation du Fix, et suite à de nombreuses recherches, que nous nous sommes rendus compte qu'on avait fait une erreur dans le parser.

2.2. **Les priorités et associativité.** Pour les priorités et l'associativité nous nous sommes aidés du site officiel d'Ocaml: règles, que nous avons appliquées. Finalement on a été "trop" précis et lors de la compilation il y avait plusieurs messages de "warning" nous indiquant qu'il y avait des priorités qui n'étaient pas utiles, que nous avons donc enlevées. Nous avons remarqué plus tard, lors de l'implémentation de certaines extensions, qu'il aurait été plus sage de les garder, puisqu'on avait finalement besoin de certaines de ces priorités.

2.2.1. *Difficultés.* On n'a pas eu de difficultés particulières pour les règles de précédence, mais nous avons eu un peu de mal à trouver l'associativité des token.

Part 2. Vérification de types

Dans cette partie du projet l'objectif était d'implémenter la vérification des règles de typage des expressions du langage et de renvoyer le type d'une expression.

3. TYPECHECKER

On devait compléter la fonction type_expr en suivant les règles qui nous étaient données dans le sujet du projet. Nous avons ajouté une fonction get_mutable qui nous permettait d'accéder au dernier élément d'un triplet.

3.0.1. *Difficultés.* Il n'y avait pas vraiment de difficultés sur les opérations binaires et unaires, cette partie a commencée à devenir plus difficile lorsqu'on a dû manipuler l'environnement de typage, par exemple sur les Let(s,e1,e2) et les initialisations de fonctions, Fun(s,typ,e). Parfois nous faisions des erreurs sur le parenthésage des match, ce qui faisait que le prochain token n'était pas lu correctement. C'est une petite erreur, mais qui nous a cependant pris pas mal de temps à trouver.

Ensuite la très grande difficulté était l'implémentation du typage des structures: En effet on a passé la majorité du temps à écrire le code pour GetF(e,x), SetF(e1,x,e2) et Strct(lst).

Au début nous avions pas vu qu'il existait "prog.types" dans le fichier mml.ml et donc nous ne comprenions pas comment avoir accès au types de structures. Initialement nous pensions que les structures étaient "stockées" dans l'environnement de typage, mais puisqu'on ne trouvait pas à quel moment elles y étaient stockées nous avons essayé d'écrire nous même le code qui mémoriserait les structures dans l'environnement de typage.Ce fût un processus très long qui n'aboutissait à rien, donc on était bien soulagés quand on a vu que en plus du "prog.code" il existait aussi un "prog.types" qui un peu magiquement contenait toute l'information sur les types de structure.

Ensuite ce qui a été particulièrement compliqué fût la gymnastique mentale entre les différents modes d'accès aux structures: Tstrct,Strct, strct, VStrct et Vptr, et le fait de manipuler des Hashtbl embriquées.

Lors du typage de "Strct(lst)" nous étions face à un choix:

-Lorsqu'on initialise un type: type t = {mutable a: int; b: int;}

-Et qu'ensuite on appelle: z = {a=3; b=7;} on se demandait si l'ordre des arguments avait une importance, c'est à dire si on pouvait aussi appeler: z = {b=7; a=3;}

-On pouvait dans le code matcher les arguments 1 à 1 et dans ce cas l'initialisation d'un structure de type t devait avoir la forme: z = {a=3; b=7;} mais pas la forme z = {b=7; a=3;}

-Mais on pouvait aussi accepter toutes les permutations de l'ordre des arguments et dans ce cas: z = {a=3; b=7;} serait équivalent à: z = {b=7; a=3;}

-Finalement nous avons opté pour la première option qui nous semblait plus naturelle pour Ocaml (on fait pas du Python ici !)

Part 3. Interprétation

L'objectif de cette dernière partie du projet était d'évaluer la valeur des expressions et la valeur finale du programme, qui serait affichée à l'écran. Il fallait aussi expliciter et renvoyer les erreurs appropriées pour permettre de debbuger les "Anomaly" efficacement.

4. Interpreteur

Nous avons complété la fonction (eval e env), et avons ajouté deux fonction d'aide (evalb e env) et (evali e env) qui évaluent des expressions et renvoient un bool et un int respectivement.

4.0.1. *Difficulté: Différence entre environnement et mémoire globale.* Même si la différence entre environnement et mémoire globale, et leur utilisation dans les différentes expressions était expliqué dans le sujet du projet nous avons été un peu confus par moments. Par exemple, nous pensions initialement à tort que les champs de structure se comportaient comme des variables et devaient donc être mémorisées dans l'environnement.

4.0.2. *Cas des opérations binaires.* Dans les opérations binaires l'opérande de droite devait être évaluée avant l'opérande de gauche. Nous ne savions pas comment Ocaml évalue les opérations binaires et donc si il fallait changer l'ordre d'évaluation. Par exemple dans: "| Bop(Add, e1, e2) ->" ,faut t-il renvoyer: "-> VInt(evali e2 env + evali e1 env)" ou "-> VInt(evali e1 env + evali e2 env)" ?

Si Ocaml évalue déjà les opérations binaires d'abord à droite il suffit d'écire: "-> VInt(evali e1 env + evali e2 env)" sans se faire de soucis, mais si Ocaml évalue d'abord les opérations à gauche alors il faut inverser l'ordre d'évaluation. Nous avons décidé de laisser l'ordre d'évaluation "normal" car nous avons vu que Ocaml évaluait d'abord les opérandes de droite, mais on aurait aussi pu créer des variables: "let res1 = evali e2 env in let res2 = evali e1 env in Vint(res1 + res2)".

4.0.3. *Les fonctions récursives.* Au delà de la difficulté inhérente de l'évaluation des fonctions récursives nous avions fait une erreur dans le parseur, et donc même lorsque notre code dans l'interppreteur était correct il produisait quand même une erreur.

Part 4. Extensions

5. Boucle for

Ce n'était pas une extension proposée par le projet mais on trouvait qu'un interpreteur ne pouvait pas être complet sans boucles.

5.0.1. *La forme d'une boucle for.* Une boucle for est écrite dans un programme de cette manière: "for x=e1 to e2 do e3 done;"

Le lexeur va trouver les mots clefs suivants: "FOR", "TO","DO","DONE", et le parseur va détecter une boucle for de la manière suivante: FOR x=IDENT EQUAL e1=expression TO e2=expression DO e3=expression DONE {For(x,e1,e2,e3)}

5.0.2. *Typage.* For(x,e1,e2,e3) avec x un string et e1,e2,e3 des expressions.

On s'assure que e1 et e2 sont de type TInt et e3 est de type TUnit.

5.0.3. *Interpretation.* Concernant l'evaluation et l'interpretation : on evalue d'abord e1 qui sera la valeur initiale de "x", et e2, qui sera la valeur finale de "x", et ensuite chaque tour de boucle on evalue e3 et incrementant et décrementant la valeur de "x". Quand "x" prend la valeur de e2, la boucle s'arrête et on renvoie un VUnit. On avait le choix d'éxecuter e3 une dernière fois quand "x" prend la valeur de e2, càd que "for x=10 to 10 do e3", e3 serait évaluée une fois, mais nous avons décidé contre ce choix.

5.0.4. *La direction de l'argument.* Une petite touche supplémentaire qu'on a ajoutée permet au programme de traverser une boucle dans l'ordre inverse. Lorsque l'expression initiale est supérieure à l'expression d'arrivée, par exemple si on a "for i=10 to 0 do ..." au lieu d'incrémenter de 1 le i à chaque tour de boucle (et d'avoir au passage une boucle infinie) nous avons décider de décrementer la valeur de i dans ce cas là.

5.0.5. *Le problème du point virgule.* Il est impératif dans le programme d'ajouter un point virgule après une boucle for. Si ce n'est pas fait l'interpreteur considera ce qui vient après la boucle for comme faisant partie de la boucle. Nous avons tenté de résoudre ce problème en modifiant les priorités mais n'avons pas réussi, il faut donc ajouter pour le moment un point virgule après le "done".

5.0.6. *L'affichage.* Finalement, il était aussi nécessaire de modifier le fichier mmlpp.ml afin que mmlcat.native puisse correctement afficher un programme contenant une boucle for.

6. Boucle While

Si on implémente la boucle for, il est naturel de faire aussi la boucle while !

6.0.1. *La forme d'une boucle while.* Une boucle while est écrite dans un programme de cette manière: "while e1 do e2 done;"

Le lexeur va identifier les mots clefs suivants: "WHILE", "DO", "DONE", avec DO et DONE qui sont en commun avec la boucle for.

Le parseur va détecter la présence d'une boucle while en matchant avec une expression de la forme: "| WHILE e1=expression DO e2=expression DONE {While(e1,e2)}"

6.0.2. *Typage.* On a While(e1,e2) avec e1,e2 des expressions. On s'assure que e1 est de type TBool et e2 est de type TUnit (ce n'est pas à 100% nécessaire que e2 soit de type TUnit).

6.0.3. *Interprétation.* Concernant l'evaluation et l'interpretation : à chaque tour de boucle on evalue e1 avec evalb, et si l'évaluation est vraie on évalue e2 avec eval, aussi sinon on renvoie un VUnit.

6.0.4. *Problème du point virgule.* Tout comme la boucle for, et pour les même raisons il est nécessaire d'ajouter un point virgule après le done pour indiquer la fin de l'expression.

6.0.5. *L'affichage.* Finalement, il était aussi nécessaire de modifier le fichier mmlpp.ml afin que mmlcat.native puisse correctement afficher un programme contenant une boucle while.

7. Extensions qu'on a pas réussi à implémenter

7.1. **"Missing semicolon".** Nous avons essayé d'ajouter dans le parser le cas suivant:" | e1 = expression e2=expression {missing_semicolon "Il faut ajouter un point virgule entre deux expressions!"} ", mais lors de la création de mmlcat.native nous avons obtenus une centaine de conflits, que nous avons diminués jusqu'à une vingtaine, mais qu'on arrivait pas à enlever sans modifier le fonctionnement de l'interpreteur. Nous avons donc abandonné cette extension.

7.2. **"Types algébriques ".** N'en parlons pas.

Part 5. Tests

Pour tester ce projet, nous avons éxecuter les programmes mmlcat.native et mmli.native afin de vérifier le bon parenthésage et la bonne évaluation des fichiers respectivement.

Nous avons ajouté quelques tests supplémentaires que ceux donnés par le projet, afin de vérifier le bon fonctionnement des extensions que nous avons ajoutées.

8. Boucle For

8.0.1. *Test1: for.mml.* Le premier objectif de ce test était de vérifier que le programme arrive bien à lire un fichier qui contient une boucle for sans produire d'erreurs de syntaxe ou de type.

Le deuxième objectif de ce fichier était de vérifier que la boucle for s'arrêtait bien peu importe la direction de l'incrémentation, càd "for i=0 to 10 do .." et "for i=10 to 0 do ..." s'arrêtent bien.

8.0.2. *Test2: for2.mml.* L'objectif de ce test était de tester si on pouvait bien intégrer un type avec un champs mutable dans une boucle, et que le champs serait bien modifié sans qu'il y ait de message d'erreur.

9. Boucle While

Tout comme pour la boucle for, le test ultime d'une boucle est la modification répétée d'une structure avec un champs mutable. Le résultat final est bien 11!

10. Test Unit

Il n'y avait pas de tests pour vérifier que le lexeur/parseur arrivaient à bien lire le programme suivant: "()", donc nous avons décidé de vérifier pour être prudents.

11. Test Fibonacci

Finalement nous avons décidé de tester une fonction qui comportait deux appels récursifs afin de vérifier le bon comportement des fonctions récursives.

Email address: victor.robert-lambrecht@universite-paris-saclay.fr
Email address: adam.sekkat@universite-paris-saclay.fr
URL: https://www.lri.fr/~blsk/CompilationLDD3/dm-mml.html