

Rapport projet de compilation

I Lexage

Le lexage n'a pas posé de problème particulier. Cependant, la gestion des commentaires nous a demandé une attention toute particulière, notamment dans la distinction entre commentaire multi-ligne et commentaire simple ligne.

La gestion des caractères spéciaux a été l'objet d'un soin tout particulier : on peut lire par exemple dans le code Caml la chaîne "\\\\" pour rattraper le caractère "\\" dans le code C.

La gestion des erreurs au niveau du lexer est relativement restreinte : seules les erreurs "Caractère prohibé" et "Commentaire non fermé" sont remontées par le lexer.

II Parsage

Le parser produit un arbre de syntaxe abstraite (ast). Un ast est constitué de types mutuellement récursifs, dont certains sont des types enregistrement contenant d'une part la donnée d'une structure de sous ast, et d'autre part une éventuelle localisation.

Dans l'ast, les objets porteurs d'une localisation sont ceux qui représentent des types, des opérateurs binaires, des expressions, instructions, déclaration d'instructions, de fonctions, et leurs paramètres.

Pour obtenir la localisation d'un lexème dans le code C, on utilise les fonctions `$startpos` et `$endpos` fournies par `menhir`.

III Typage

On définit un nouveau type `typ` qui va uniquement servir à vérifier le bon typage de l'ast récupéré à l'issue du parsage. Il contient les types du langage PetitC, auxquels on ajoute un type qui représente les fonctions (constructeur `Cliste`).

1 La relation $\equiv$

Le fichier typage commence par la fonction "congruence", dont l'objectif est de vérifier si deux types sont congruents au sens $\equiv$ introduite dans le sujet. Cette fonction nous a causé de nombreux déboires. En effet, nous avons oublié d'implémenter la réflexivité de la relation $\equiv$, ce qui a entraîné de nombreuses erreurs de compatibilité de types.

2 Fonctions imbriquées

La gestion des fonctions imbriquées a été particulièrement épineuse. En effet, à l'issue d'une déclaration de fonction imbriquée dans une autre, il faut réaliser deux opérations sur les environnements.

La première consiste à ajouter le nom de la nouvelle fonction à l'environnement du bloc sus-jacent. La seconde consiste à produire un nouvel environnement, qui contient, en plus de ce qui précède, les arguments de la nouvelle fonction. Ce nouvel environnement a vocation à être utilisé dans le bloc contenu dans la nouvelle fonction. Ceci nous permet notamment d'autoriser les fonctions récursives.

On utilise une variable booléenne nommée `inloop` qui garde en mémoire le fait qu'on soit dans une boucle ou non. L'intérêt de cette variable est de n'autoriser l'utilisation de `break` et `continue` qu'à l'intérieur d'une boucle.

3 Gestion des conflits

Lorsque l'on déclare une fonction ou variable à l'intérieur d'un bloc, une liste de string `h` est passée en argument, elle correspond à l'ensemble des noms de variables et de fonctions déclarées précédemment au sein du bloc. L'intérêt de cette manœuvre est d'interdire de déclarer deux variables (ou fonctions) avec le même nom.

4 Type de retour

Lors d'une déclaration de fonction, instruction, variable ou bloc, une variable `t0` est passée en opérande, elle correspond au type (s'il existe) de retour de la fonction déclarée précédemment, la plus proche. Elle sert à vérifier que les valeurs renvoyées sont du bon type : elle est utilisée dès qu'une expression `return` apparaît, ou bien à la fin de la fonction.

5 Après typage

Après typage, on renvoie une structure (dans le module `astt`) similaire à l'ast précédent, dans laquelle on supprime les localisations, et lorsque c'est nécessaire (typage des expressions ou déclarations de fonctions, de paramètres et variables), on indique un type de retour (typé `ttypec` dans le programme OCaml) en vue de faciliter la production de code.

IV Production de code

À titre informatif, nous traitons les fonctions imbriquées.

1 Modifications du travail précédent

On a réalisé des modifications du lexer, à savoir la manière d'échapper les caractères `\\`, `\\'`, `\n`, `\t`. On a également corrigé quelques légères erreurs de parsing (au niveau des opérateurs de comparaison). On a exécuté des modifications mineures de l'Ast, notamment des noms, dans un but d'homogénéisation. On a fusionné les précédents modules contenant chacun un AST en un seul module en contenant plusieurs.

2 Pré-calcul et allocation

On crée également un nouvel arbre de syntaxe abstraite qui ne servira que pour la production de code (dans les fonctions `alloc`). Il ne contient plus les types, mais contient l'information booléenne pour les expressions "être un pointeur", qui nous est utile dans l'exécution car l'arithmétique des

pointeurs est traitée légèrement différemment de celle des entiers (multiplications par 8).

On maintient un environnement courant qui contient l'ensemble des fonctions définies, la profondeur à laquelle on se trouve (elle augmente de 1 à chaque imbrication de fonction). On renvoie les informations suivantes nécessaires à la production de code :

- Un identifiant unique pour chacune des fonctions, boucles `for` et `while`, utilisations de `If`. Il s'agit d'une chaîne de caractères formant un identifiant unique, composé de mots clés parmi `"For"`, `"While"`, `"Ifvrai"`, `"Ifaux"` couplés avec des identifiants numériques (fournis par la fonction `nom`).
- Chaque variable est localement identifiée par un couple : la différence de profondeur entre le lieu de l'utilisation de la variable, et l'endroit où se trouve le tableau d'activation associé à la variable.
- L'information booléenne `"suis-je un pointeur?"` pour chaque expression.

3 Production du code x86-64

- Pour les fonctions imbriquées : On utilise une fonction `voyage_mem` qui prend en argument les identifiants de positions d'une variable, relatifs à notre position dans le code (p est la différence de profondeur entre la déclaration et l'utilisation de la variable). Celle-ci place la valeur recherchée dans `r13` et le `rbp` parent dans `r14`.
- Utilisation de la pile : Chaque expression compilée est déposée sur la pile. Au moment des évaluations de fonctions, c'est légèrement différent : au sein de la fonction, un `return` a pour effet d'envoyer le résultat dans `r9`. À la sortie de l'appel de fonction, on push `r9` sur la pile, sauf si la fonction est de type `void`.
- Les `And` et `Or` sont paresseux : si on arrive à une expression du type `e1||e2`, on compile d'abord `e1`. Si `e1` est `true`, on renvoie directement `true` sans compiler `e2`. De même pour les expressions du type `e1&&e2`.
- Nous n'avons pas trouvé dans le module `x86-64` fourni de fonctions pour écrire des `cmov` (move conditionnels). Nous avons donc utilisé directement la fonction `inline` qui permet d'écrire exactement ce qu'on veut.
- L'arithmétique de pointeurs est gérée légèrement différemment de l'arithmétique habituelle. En effet, quand on effectue un calcul du type `pointeur + entier`, il faut multiplier l'entier par 8 (taille de l'espace mémoire utilisé pour une variable) avant de faire l'addition. Par exemple, `p[1]` va chercher non pas à l'espace mémoire `p+1` mais à l'espace mémoire `p+8`. Des règles similaires sont appliquées pour les autres opérations.