POLYTECH NICE SOPHIA

SI4 / 2021-2022

ERICK GALLESIO

Compilation

Contrôle final

08 décembre 2021

Durée: 3 heures.

Introduction — Consignes

Vous devez travailler dans le répertoire `RENDU`.

Ce dernier comporte deux sous-répertoires principaux:

- `RENDU/Toy` contient une version du compilateur avec les dernières extensions vues en TD. Le compilateur qui vous est donné est organisé comme celui qui a été utilisé en TD. En particulier, la construction et le test se font de la même manière qu'en TD (`make` et `make tests`).
- `RENDU/Reponses` contient **vos réponses aux questions de ce sujet**. Pour chaque question, il vous sera indiqué dans quel fichier vous devez répondre.

Important:

- L'épreuve comporte des questions *indépendantes* qui peuvent être donc traitées dans n'importe quel ordre.
- Les questions sont données dans un ordre de difficulté (plus ou moins) croissante.
- **Vous devez répondre dans les fichiers du dossier `RENDU/Reponses`**
- **Vous serez noté sur les fichiers précédents principalement**. Ce sont eux qui constituent votre copie finale et pas seulement le programme que vous allez construire.
- Les tests entrent pour une **faible proportion** dans la note finale.
- Essayez de ne pas trop modifier la structure des fichiers de réponses. Vous pouvez copier/coller les parties *significatives* de vos modifications (n'y copiez pas des fichiers entiers!!!).
- Le code c'est bien, mais s'il est expliqué c'est mieux. Idéalement, les réponses attendues sont du type:

```
Avec l'introduction du type float, j'ai rajouté un test permettant de combiner
un entier et un flottant (et vice-versa) dans la fonction compatible_types:

if ((t1 == int_type   && t2 == float_type) ||
    (t1 == float_type && t2 == int_type))    return true;
```

Lorsque vous avez terminé, vous devez:

- faire un `make distclean` (qui nettoie plus que `make clean`) dans le dossier `RENDU/Toy`;
- construire une archive de la **totalité de votre répertoire** **`RENDU`**. L'archive devra s'apeller `RENDU-xxxx.tar.gz` (ou `RENDU-xxxx.zip`) avec `xxx` qui correspond à votre nom.

Cette archive devra **IMPERATIVEMENT** être rendue sur Slack (en **MESSAGE PRIVÉ**). Par sécurité, vous devez aussi me l'envoyer par mail à `eg@unice.fr` (ne vous inquiétez pas si vous avez un message qui dit que votre message n'a pas pu être délivré, ça devrait être bon si vous avez bien fait un `make distclean` comme indiqué plus haut.

L'envoi de vos deux messages (Slack + mail) doit se faire avant que vous ne quittiez la salle.

1 Correction de bug

Le corrigé qui vous avait été distribué pour l'ajout du type `float` en *Toy* comporte une erreur. Cette erreur est mise en évidence par le programme `ok-float-bug.toy`. Ce programme (qui est **correct**) ne compile pas avec la version actuelle du compilateur.

Le programme `ok-float-bug.toy` est dans fichier `tests/IGNORE`. On rappelle que les fichiers de tests dont le nom est présent dans `test/IGNORE` ne sont pas compilés par la commande `make tests`.

Activez la compilation du test `ok-float-bug.toy` et vérifiez que la correction que vous avez apportée au compilateur permet de le compiler et de l'exécuter sans erreur.

Répondre dans le fichier `RENDU/Reponses/question1.md`

1. Indiquer sommairement les modifications que vous avez apportées au compilateur pour corriger l'erreur sur les flottants.
2. Pour chacun des fichiers que vous avez modifiés, indiquez la (ou les) modification(s) apportée(s).

2 Nombres avec séparateurs

Plusieurs langages de programmation permettent de placer des caractères `'_'` dans des nombres pour en simplifier la lecture. Par exemple, Python ou JavaScript, entre autres, permettent l'utilisation de caractères `'_'` dans un nombre. Ces caractères peuvent être placés à l'intérieur d'un nombre entier pour regrouper des «paquets» de chiffres. En général, les langages n'imposent pas de règle sur la taille de ses paquets. Les seules règles que l'on a sont que l'on ne peut pas avoir deux caractères `'_'` successifs et qu'un nombre ne peut commencer ou se terminer par un tel caractère.

Pour illustrer, cela regardons une session en node:

```
Welcome to Node.js v16.11.1.
Type ".help" for more information.
> 1_000_000
1000000
> 1_2_3_4_5
12345
> 12_34
1234
> 1000__00
      ^
Uncaught SyntaxError: Only one underscore is allowed as numeric separator
> 1000_000_
Uncaught SyntaxError: Numeric separators are not allowed at the end of numeric literals
> _1000       // En fait, _1000 n'est pas un nombre en JavaScript,  mais un
        identificateur
Uncaught ReferenceError: _1000 is not defined
>
```

La dernière erreur indique que la variable `_1000` n'est pas définie. Ici, le symbole `_1000` est vu comme une variable puisque le caractère `'_'` est considéré comme une lettre en JavaScript.Notons qu'en *Toy*, nous n'avons pas ce problème puis le caractère underscore ne peut pas apparaître au début d'un identificateur.

Introduire la possibilité d'avoir des nombres comprenant éventuellement un ou plusieurs caractères `'_'` avec les règles suivantes:

- on peut mettre des `'_'` n'importe où dans un nombre
- pas de `'_'` au début ou à la fin d'un nombre
- on ne peut pas avoir deux ' `_'` successifs dans un nombre.

Pour simplifier, on n'autorisera les `'_'` que dans les nombres entiers exprimés en décimal.

Pour tester votre extension, vous pourrez utiliser les deux fichiers de tests `ok-underscore.toy` et `fail-underscore.toy`.

Ces programmes sont présents dans fichier `tests/IGNORE`. On rappelle que les fichiers de tests dont le nom est présent dans `test/IGNORE` ne sont pas compilés par la commande `make tests`.

Répondre dans le fichier `RENDU/Reponses/question2.md`

1. Expliquer sommairement le principe de fonctionnement de cette extension.
2. Pour chacun des fichiers que vous avez modifiés, indiquez la (ou les) modification(s) apportée(s).
3. Si votre extension ne marche pas, ou pas complètement, indiquer ce qui d'après vous en est la cause.

3 Qualificatif const sur les variables *Toy*

On veut permettre d'ajouter le qualificatif `const` lors de la déclaration de variables. Ce qualificatif joue le même rôle qu'en C: permettre d'interdire qu'une variable soit modifiée.

Les variables peuvent être déclarées comme constantes en utilisant le mot clé `const` avant le type de données de la variable. Les variables constantes peuvent être initialisées une seule fois, au moment de leur déclaration. Leur valeur ne peut plus être changée par la suite. Un exemple d'utilisation est donné ci-dessous:

```
void test(int p) {
  const int save_p = p+1;
  p = 10;
  // save_p = 10; ⟹ erreur de compilation car save_p est constant
  printf("save_p = %d  p = %d\n", save_p, p);
}

int main(int argc, char *argv[]) {
  test(42);
  return 0;
}
```

Le code produit par votre extension **ne doit pas produire de code C contenant le mot clé `const`**. Cela veut dire que **c'est le compilateur *Toy* qui doit détecter une erreur** lorsqu'une variable déclarée avec le qualificatif `const` est modifiée.

Notes:

1. Dans la version du compilateur qui vous est distribuée, la représentation des identificateurs comprend un champ supplémentaire permettant de savoir si un identificateur est constant ou non. Ainsi, dans `ast.h`, la représentation interne d'un identificateur est donc maintenant:

   ```
   struct s_identifier {
     ast_node header;      ///< AST header
     char *value;          ///< value of the identifier (a string)
     bool is_const;        ///< default to "false"; "true" if identifier is  a const
   };

   #define IDENT_VAL(p)            (((struct s_identifier *) p)->value)
   #define IDENT_IS_CONST(p)       (((struct s_identifier *) p)->is_const)
   ```

2. Pour la réalisation de cette extension, les identificateurs qualifiés de `const` devront bien sûr être rangés dans la table des symboles avec leur champ `is-const` affecté à `true`.

3. Comme cela été dit précédemment, vous ne devez pas produire de code contenant le mot clé `const` (c'est inutile puisque vous aurez détecté en *Toy* les accès interdits en écriture).

Pour tester cette extension du compilateur *Toy* vous pouvez utiliser les programmes suivants:

- `ok-const1.toy`
- `ok-const2.toy`
- `fail-const1.toy`
- `fail-const2.toy`

Ces programmes sont présents dans fichier `tests/IGNORE`. On rappelle que les fichiers de tests dont le nom est présent dans `test/IGNORE` ne sont pas compilés par la commande `make tests`.

Répondre dans le fichier `RENDU/Reponses/question3.md`

1. Expliquer sommairement le principe de fonctionnement de cette extension.
2. Pour chacun des fichiers que vous avez modifiés, indiquez la (ou les) modification(s) apportée(s).
3. Si votre extension ne marche pas, ou pas complètement, indiquer ce qui d'après vous en est la cause.

4 Traces automatiques

Cette extension consiste à ajouter un mécanisme de trace aux programmes *Toy*. Ce mécanisme permet d'ajouter automatiquement

- une trace à l'entrée et la sortie d'une fonction
- l'affichage de la valeur renvoyée pour une fonction non `void`

Pour ajouter les traces aux fonctions, il faut compiler le programme avec l'option `-t` (pour **trace**). Cette option met à `true` la variable globale booléenne `trace_mode` (qui est autrement initialisée à `false`).

Ainsi, l'exécution du programme *Toy* suivant:

```
int fact(int n) { return (n <= 1) ? 1: n * fact(n-1); }

void boucle() {
    for (int i=0; i < 10; i++)
        { print("i = ", i, "\n"); if (i == 5) return; }
}

int main() {
    int f = fact(5);
    print("Fact(5) =", f, "\n");
    boucle();
    return 0;
}
```

produit les traces suivantes sur la sortie standard:

```
>> Enter function 'main'
 >> Enter function 'fact'
  >> Enter function 'fact'
   >> Enter function 'fact'
    >> Enter function 'fact'
     >> Enter function 'fact'
     << Leave function 'fact' <-1
    << Leave function 'fact' <-2
   << Leave function 'fact' <-6
  << Leave function 'fact' <-24
 << Leave function 'fact' <-120
Fact(5) =120
 >> Enter procedure 'boucle'
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
 << Leave procedure 'boucle'
<< Leave function 'main' <-0
```

Pour réaliser ces traces vous pouvez utiliser le support d'exécution qui est disponible dans `toy-runtime.h`. Il propose 6 macros:

- `ENTER_VOID` qui doit être placée au début du corps d'une fonction `void` ;
- `LEAVE_VOID` qui doit être placée à la fin du corps d'une fonction `void` ;
- `RETURN` qui doit être utilisé à la place de `return` dans une fonction `void` ;
- `ENTER_FUNC(t)` qui doit être placée au début du corps d'une fonction dont le type de retour est `t` ;
- `LEAVE_FUNC(t)` qui doit être placée à la fin du corps d'une fonction dont le type de retour est `t` ;
- `RETURN_VALUE(t, v)` qui doit être utilisé à la place de `return v` dans une fonction de type `t`.

Ainsi, la fonction `fact` de l'exemple précédent sera traduite en (si on utilise l'option `-t` du compilateur):

```
int fact(int n){
  ENTER_FUNC(int);
  {
    RETURN_VALUE(int, (n <= 1) ? 1 : n * fact(n - 1));
  }
  LEAVE_FUNC(int);
}
```

et la fonction `boucle` sera plus ou moins traduite en (*plus ou moins* car le `for` est traduit en un `while` et le `print` en plusieurs `printf`):

```
void boucle(void){
  ENTER_VOID;
  for (int i=0; i < 10; i++)
    { print("i = ", i, "\n"); if (i == 5) RETURN; }
  LEAVE_VOID;
}
```

Important:

Vous ne devez ajouter les macros de trace que si l'option `-t` est activée (le traitement de l'option et l'affectation dans la variable globale `trace_mode` sont déjà faits dans le compilateur qui vous est distribué). Cela veut dire que lorsque la commande `toy` est appelée avec l'option `-t`, la variable `trace_mode` est mise à la valeur `true`.

Indication: Pour cette commande, vous pouvez utiliser la fonction (définie dans `prodcode.c`)

```
    char* toy_type_to_string(ast_node *type);
```

qui renvoie, sous la forme d'une chaîne C, le type contenu dans le nœud `type`. Voir, par exemple, `produce_code_print_statement` pour une utilisation de cette fonction.

Pour information, seuls les fichiers dont le nom est `ok-trace-xxx.toy` sont compilés avec l'option `-t`. Pour ces fichiers, le système de test s'occupe de vérifier que le programme que vous avez produit affiche autant de traces d'entrée que de traces de sortie. Vous n'avez rien de particulier à faire pour vérifier que les traces sont bien affichées (si ce n'est pas le cas, l'exécution ne sera pas validée).

Il y a trois fichiers de test pour cette question:

- `ok-trace-simple.toy`
- `ok-trace-types.toy`
- `ok-trace-fact.toy`

Répondre dans le fichier `RENDU/Reponses/question4.md`

1. Indiquer sommairement le principe de fonctionnement de votre extension.
2. Pour chacun des fichiers que vous avez modifiés, indiquez la (ou les) modification(s) apportée(s).
3. Si votre extension ne marche pas, ou pas complètement, indiquer ce qui d'après vous en est la cause.

5 Structure de contrôle foreach

On veut ajouter à *Toy* une structure de contrôle permettant d'énumérer les différentes valeurs d'un ensemble constant. Un exemple d'utilisation de cette structure de contrôle est donnée ci-dessous:

```
int sum = 0;
foreach int i in [2, 3 , 5 , 7, 11]
   sum = sum + i;
print("Sum of the 5 first primes = ", sum, "\n");
```

Un autre exemple:

```
foreach string str in ["North", "East", "South", "West"] {
   print("str = ", str, "\n");
}
```

Cette structure de contrôle déclare toujours une variable locale à la boucle (l'entier `i` dans le premier exemple et la chaîne `str` dans le second). La variable locale de la boucle, prend en séquence toutes les valeurs de l'ensemble qui suit le mot clé `in`.

Remarque:

Telle qu'elle est définie, la structure `foreach` nécessite toujours la déclaration d'une variable de contrôle la boucle.

Le code que doit produire le compilateur pour le premier exemple est:

```
{
    int _lst[] = { 2, 3, 5, 7, 11, };                    // ← 1

    for (int _idx=0; _idx < 5; _idx++) {                 // ← 2
        int i = _lst[_idx];                              // ← 3
        sum = sum + i;
    }
}
```

Pour le deuxième exemple, le compilateur doit produire le code suivant:

```
{
    _toy_string _lst[] = { "North", "East", "South", "West", };     // ← 1

    for (int _idx=0; _idx < 4; _idx++) {                            // ← 2
         _toy_string str = _lst[_idx];                              // ← 3
         // code du print ...
    }
}
```

On voit donc ici:

1. la variable `_lst` est le tableau des différentes valeurs du `foreach`
2. la variable `_idx` est une variable entière qui parcourt les tableau `_lst`
3. la variable de contrôle de la boucle `foreach` est toujours égale à `_lst[_idx]`.

Notes:

- Comme on ne peut pas déclarer de variables commencant par le caractère `'_'` en *Toy*, les variables `_lst` et `_idx` ne peuvent pas interférer avec des variables du programme que l'on compile.
- Lors de l'analyse de la structure de contrôle `foreach`, vous aurez besoin de déclarer la variable locale de la boucle. Pour cela, vous pouvez utiliser la fonction `declare_simple_variable` (définie dans `symbol.c`). Ainsi, si on a `var` et `type` qui sont deux `ast_node *`, on peut déclarer la variable `var` de type `type` dans la portée courante en utilisant la l'appel suivant:

```
declare_simple_variable(var, type):
```

- L'implémentation de la structure `foreach` est déjà commencée dans le compilateur qui vous est distribué. Certaines parties sont à terminer pour avoir une implémentation complète de `foreach`.
- On rappelle ici que les appels à la fonction `indent` dans `prodcode.c` ne servent qu'à obtenir du code plus lisible. Il n'est pas important ici que le code produit soit bien indenté (même si cela peut, probablement, vous aider lord du développement de votre solution.

Pour tester votre compilateur, vous pouvez utiliser les fichiers de tests

- `ok-foreach1.toy`,
- `ok-foreach2.toy`,
- `ok-foreach3.toy`,
- `ok-foreach4.toy` et
- `fail-foreach.toy`

Répondre dans le fichier `RENDU/Reponses/question5.md`

1. Expliquer sommairement le principe de fonctionnement de cette extension.
2. Pour chacun des fichiers que vous avez modifiés, indiquez la (ou les) modification(s) apportée(s).
3. Si votre extension ne marche pas, ou pas complètement, indiquer ce qui d'après vous en est la cause.