TP de compilation – L3 Informatique

Nicolas Bedon, Arnaud Lefebvre

TP 1

Exercice 1 - Modifiez les fichiers `ex1.l` et `ex1.y` pour écrire une petite calculatrice, capable d'évaluer, sur les entiers, des expressions de la forme "(1+2)*(3-5)". Les expressions seront évaluées en utilisant une pile, implantée par un tableau défini dans `ex1.y`. Le principe de l'évaluation est le suivant :

- si l'expression est un nombre, alors on l'empile ;
- sinon, si par exemple l'expression est de la forme $e_1 + e_2$, en supposant que e_1 et e_2 aient déjà été évaluées et que la valeur v_2 de e_2 est en sommet de pile et celle de v_1 de e_1 juste en dessous, on dépile v_2 puis v_1, on en fait la somme et on l'empile.

Quand l'analyseur rencontre la fin de ligne, alors si tout s'est bien passé la pile a taille 1 et contient la valeur de l'expression, qu'il ne reste alors plus qu'à dépiler et afficher.

Exercice 2 - Si ça n'a pas déjà été fait, ajoutez le traitement de la division par 0 dans la calculatrice de l'exercice précédent. Pour celà, chaque règle produira un attribut synthétisé : `OK` pour dire que l'expression a correctement été évaluée, `ERR` sinon.

TP 2

Exercice 3 - On souhaite maintenant intégrer dans la calculatrice l'évaluation des expressions booléennes en plus des expressions arithmétiques. Par exemple, pouvoir évaluer des expressions de la forme "not (1+2 == 4-1 and true)". L'évaluation se fera en utilisant la même pile que pour les exercices précédents. Comme la pile ne contiendra que des entiers, les booléens pourront être codés dans la pile par 0 pour `false`, une valeur strictement positive pour `true`. Vous observerez que dans ce cas, la conjonction revient à un produit et la disjonction à une somme. Modifiez votre analyseur lexical pour prendre en compte les nouvelles unités lexicales. Modifiez votre grammaire. Votre analyseur syntaxique devra maintenant produire, pour chaque forme d'expression donnée par une production de la grammaire, un attribut synthétisé indiquant

- si l'expression est de type booléen ;
- si l'expression est de type arithmétique ;
- si l'évaluation de l'expression a produit une division par 0 ;
- si une erreur de typage s'est produite (comme par exemple pour l'évaluation de "1+true").

Pour celà, vous définirez un type énuméré `type_synth` dans un fichier `typesynth.h`, que vous inclurez dans l'analyseur lexical et dans l'analyseur syntaxique, et, du coté de l'analyseur syntaxique, vous déclarerez que les types des attributs synthétisés ou des attributs des classes d'unités lexicales sont soit des entiers, soit des valeurs du type énuméré `type_synth` par

```
%union {
  int integer;
  type_synth type;
}
```

Exercice 4 - Récupérez et compilez SIPro et ASIPro. Compilez `hanoi.asm`, et exécutez-le. Ecrivez un programme "Hello world !" en ASIPro, que vous compilerez et exécuterez.

TP 3

Dans les exercices qui suivent vous pourrez avoir besoin de générer des étiquettes (labels) uniques, pour réaliser des sauts. Les labels sont des chaînes de caractères, et pour ne pas générer deux fois le même vous pourrez utiliser un compteur que vous augmenterez à chaque fois que vous générez un nouveau label. Les fonctions suivantes pourraient vous être utiles :

```
static unsigned int new_label_number() {
  static unsigned int current_label_number = 0u;
  if ( current_label_number == UINT_MAX ) {
    fail_with("Error: maximum label number reached!\n");
  }
  return current_label_number++;
}

/*
 * char buf1[MAXBUF], char buf2[MAXBUF];
 * unsigned ln = new_label_number();
 * create_label(buf1, MAXBUF, "%s:%u:%s", "loop", ln, "begin"); // "loop:10:begin"
 * create_label(buf2, MAXBUF, "%s:%u:%s", "loop", ln, "end");   // "loop:10:end"
 */
static void create_label(char *buf, size_t buf_size, const char *format, ...) {
  va_list ap;
  va_start(ap, format);
  if ( vsnprintf(buf, buf_size, format, ap) >= buf_size ) {
    va_end(ap);
    fail_with("Error in label generation: size of label exceeds maximum size!\n");
  }
  va_end(ap);
}
```

Le code de la fonction `fail_with`, qui sert à sortir du programme sur erreur dont le message est passé en paramètre de la fonction à la manière de `printf`, est le suivant :

```
void fail_with(const char *format, ...) {
  va_list ap;
  va_start(ap, format);
  vfprintf(stderr, format, ap);
  va_end(ap);
  exit(EXIT_FAILURE);
}
```

Exercice 5 - Reprenez votre calculatrice. Vous allez la modifier pour qu'à la place de réaliser l'évaluation de l'expression, elle génère un programme en ASIPro qui réalise l'évaluation de l'expression. Pour celà, il suffit, plutôt que de dépiler deux entiers pour en faire la somme et empiler le résultat, de générer le code ASIPro qui :

- charge la valeur en sommet de pile dans un registre de SIPro ;
- retire cette valeur du sommet de la pile ;
- charge la valeur en nouveau sommet de pile dans un autre registre de SIPro ;
- retire cette valeur du sommet de la pile ;
- réalise la somme des deux registres ;
- place le résultat, qui se trouve dans un registre, en sommet de pile.

Vous vous limiterez, dans un premier temps, aux expressions arithmétiques, puis vous ajouterez les opérateurs logiques. Le code sera généré sur la sortie standard, par exemple, ou dans un fichier texte. Le préambule sera :

```
; Calculette

        const ax,debut
        jmp ax

:debut
; Préparation de la pile
        const bp,pile
        const sp,pile
        const ax,2
        sub sp,ax
```

L'épilogue sera :

```
; Pour afficher la valeur calculée, qui se trouve normalement en sommet de pile
        cp ax,sp
        callprintfd ax
        pop ax
        end

; La zone de pile
:pile
@int 0
```

TP 4

Exercice 6 - On souhaite maintenant rajouter la notion de variable (globale pour l'instant). Toute variable doit être déclarée (ou définie) avant d'être utilisée. Une variable ne peut pas être déclarée deux fois. L'effet de la déclaration (ou de la définition) est de réserver la place nécessaire au stockage de la variable, et d'ajouter à la table des symboles le nouveau nom de variable et les informations associées. Pour la table des symboles, vous téléchargerez le code sur la page web de l'enseignement de compilation qui contient une implantation simple de table des symboles par liste chaînée (améliorable de beaucoup). Ce code contient des fonctionnalités qui ne vous seront pas toutes nécessaires pour l'instant, mais qui seront utiles par la suite. Ajoutez à votre grammaire le nécessaire pour prendre en compte la déclaration des variables et leur utilisation dans les expressions. Générez le code associé aux nouvelles productions. Votre programme devra être capable d'évaluer les lignes suivantes, composées d'une suite de déclarations/définitions de variables et d'une expression :

```
int a=5;
int b=3;
2*b+4*a+1
```

Par exemple, l'analyse de la première ligne a pour effet

- de générer le code associé à l'évaluation de l'expression à droite du `=` (vous supposerez que la valeur de l'expression se trouve en sommet de pile après exécution de ce code),
- de réserver de la place pour y stocker la valeur de la variable `a` en produisant par exemple le code SIPro suivant (qui initialise `a` à 0)

  ```
  :var:a
  @int 0
  ```
- d'ajouter `a` avec les informations associées dans la table des symboles,

— de générer du code qui va mettre la valeur en sommet de pile dans l'espace mémoire réservé pour `a`.

Faites attention aux contrôles des types.

Exercice 7 - On va maintenant ajouter les instructions. Dans l'épilogue du code généré, supprimer le `callprintfd` et les deux instructions environnantes. Ajoutez la notion d'instruction dans la grammaire. L'entrée de l'analyseur syntaxique sera maintenant composée d'une suite de déclarations de variables (éventuellement vide) suivie d'une suite d'instructions. Les deux seules instructions (pour l'instant) sont l'affectation et l'affichage (`print`) de la valeur d'une expression.

Exercice 8 - Ajoutez le test `if` (avec sa flexion `else`, qui doit porter sur le `if` le plus proche) et la boucle `while`. Vous ferez attention à ce que le `else` porte sur le `if` le plus proche.

Exercice 9 - Lisez l'énoncé de l'exercice suivant (sans le faire bien sûr !). Modifiez votre grammaire pour que l'entrée analysée soit constituée d'une suite de définitions, une définition étant soit une définition de fonction, soit une définition de variable globale. Une définition de fonction est de la forme `type identifiant(liste de déclarations de paramètres) bloc d'instructions`, où un bloc d'instruction est de la forme

— `{` ;
— liste de déclaration de variables éventuellement vide ;
— suite d'instructions éventuellement vide ;
— `}` ;

Les instructions sont

— le `if` avec sa flexion `else` ;
— la boucle boucle `while` ;
— `print` ;
— `return` ;
— l'appel de fonction, dans le cas où elle ne retourne rien ;
— l'affectation ;
— un bloc d'instruction peut être assimilé à une instruction ;
— `;` est une instruction.

Passez votre nouvelle grammaire dans bison pour vérifier qu'elle est bien LALR(1). Modifiez votre analyseur lexical et votre analyseur syntaxique pour prendre en compte les constantes chaînes de caractères et les commentaires, de telle manière à ce que le contenu du fichier `exemple.txt` puisse être analysé.

Exercice 10 - Il faut maintenant associer du code à chacune des nouvelles productions de l'exercice précédent. La première chose à faire est de gérer la table des symboles correctement

— à chaque déclaration de variable globale, il faut regarder si le symbole est déjà dans la table. S'il y est, alors il faut produire une erreur. Sinon, il faut réserver de la place mémoire pour stocker la variable (vous associerez à cet emplacement mémoire une étiquette que vous pourrez retrouver à partir du nom de la variable), et ajouter le symbole dans la table des symboles avec les informations associées (type, classe de stockage). Il est inutile d'essayer de mémoriser dans la table des symboles l'adresse de stockage, car elle n'est connue symboliquement de votre compilateur que par son étiquette ;
— vous aurez besoin, pendant l'analyse de la définition d'une fonction, de retrouver l'entrée correspondante dans la table des symboles. Pour vous faciliter la tâche, vous déclarerez une variable globale `currentFunction` de votre compilateur qui vaudra `NULL` quand le compilateur n'est pas en train d'analyser une fonction, et qui pointera sur l'entrée de la table des symboles correspondant à la fonction en cours d'analyse sinon :

```
function:
type ID {
  Si la table des symboles contient déjà un symbole ID
    Erreur
  Sinon
    currentFunction <- nouvelle entrée de la table des symboles
    donner les informations connues jusqu'ici
} '(' lparam ')' {
  Génération du code en préambule de la fonction (à faire plus tard)
} blocinstr {
  Génération du code en épilogue de la fonction (à faire plus tard)
  Suppression des paramètres de la fonction dans la table des symboles
    (si la fonction a n paramètres, alors ce sont normalement les n
     premières entrées de la table)
  currentFunction <- NULL
}
;
```

— pour les paramètres, à chaque nouvelle déclaration de paramètre il faut ajouter un nouveau symbole dans la table. Le non-terminal `lparam` synthétisera, comme valeur, la taille de la liste ;

— pour les déclarations de variables locales, il ne faut pas vérifier si le symbole est déjà dans la table : en effet, une variable locale peut masquer une autre variable de même nom. L'implantation de table des symboles qui vous est fournie range les symboles par ordre inverse d'ancienneté, et la recherche d'un symbole dans la table suit cet ordre, afin de faciliter la gestion du masquage. Dans la production attribuée suivante

```
blocinstr :
'{' ldecl linstr '}' {
  Suppression des variables locales au bloc dans la table des symboles
    (si ldecl a taille n, alors ce sont les n premières entrées de
     la table)
}
```

vous supposerez que les non-terminaux `ldecl` et `linstr` synthétisent, comme valeur, la taille de la liste.

Exercice 11 - Générez du code SIPro pour chaque instruction. Vous ferez attention, pour `return`, à la compatibilité avec le type de retour de la fonction.

Il faut maintenant générer le code associé à l'accès à une variable globale, à une variable locale, et l'appel de fonction. Ca ne pose pas de problème particulier pour les variables globales. En revanche, pour les variables locales, il faut fixer l'organisation de l'enregistrement d'activation des fonctions. Elle est donnée par la Figure 1. Ainsi, si f appelle g qui a en tout m variables locales et n paramètres, alors pendant l'exécution de g

— la ième variable locale ($i \in [1; m]$) de g est à l'adresse $bp - (2 + 2n - 2(i-1))$;

— le ième paramètre ($i \in [1; n]$) de g est à l'adresse $bp - (4 + 2(i-1))$.

Exercice 12 - Ajoutez complètement l'appel de fonction dans les expressions (pour les fonctions retournant une valeur), et l'appel de fonction dans les instructions (pour celles ne retournant pas de valeur). Vous ferez attention à bien contrôler les types. Modifiez le préambule de programme généré par votre compilateur pour qu'il appelle toujours une fonction de nom `main` ne prenant pas de paramètre. Essayer votre compilateur.

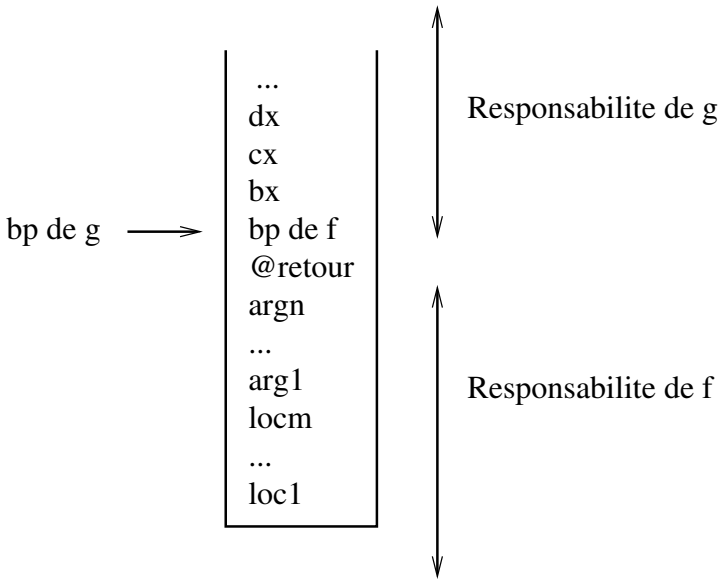

FIGURE 1 – Organisation des enregistrements d'activation des fonctions

Exercice 13 - Ajoutez ce qu'il faut au compilateur pour vérifier qu'une fonction retourne toujours quelque chose. Par exemple, pour une fonction retournant un entier, tout chemin d'exécution de la fonction doit amener à un `return` *expression entière*. Vous pourrez, par exemple, faire synthétiser un booléen à chaque instruction : `true` si elle produit à coup sûr un `return`, `false` sinon.

Exercice 14 - (facile) Une instruction est *morte* si elle ne peut pas être exécutée car il y a toujours un `return` avant dans tout chemin d'exécution. Utilisez ce que vous avez fait à l'exercice précédent pour que votre compilateur signale la présence de code mort.