COMP : Compte rendu TP2

Aurèle Barrière & Antonin Garret

9 décembre 2016

1 Introduction

Ce projet vise à implémenter la face avant d'un compilateur de VSL vers du code 3 adresses. Le but est d'arriver à traiter un langage avec des structures de contrôles, des boucles, des fonctions avec leur prototypes, des appels à des fonctions et des tableaux.

Un parser était déjà fourni et nous a permis de nous concentrer uniquement sur la génération de code 3 adresses depuis un arbre de syntaxe abstrait.

2 Description de la méthodologie

Nous avons implémenté notre compilateur progressivement, de sorte à identifier plus simplement d'éventuelles erreurs. Dans un premier temps, nous ne nous sommes occupés que de la génération de code associé aux expressions arithmétiques. Ensuite, celui du code des expressions et des blocs.

Enfin, il nous a fallu implémenter la génération de code des fonctions et prototypes, puis celle des tableaux.

À chaque étape, nous avons créé des programmes (ou des expressions, ou des instructions) pour tester notre génération. Dans un premier temps, ce code était comparé à celui des exemples dont nous disposions dans le sujet. Ensuite (quand les programmes et les instructions PRINT étaient traités), nous avons pu compiler le code 3 adresses en un code exécutable qui permettait de vérifier le comportement.

3 Bilan du travail réalisé

3.1 Génération du code d'expressions

Dans un premier, nous devions générer le code correspondant aux expressions arithmétiques . C'est l'objet de la fonction `gen_expressions`. Une partie de cette fonction était déjà présente dans le fichier d'origine, nous l'avons donc simplement complété. Elle fonctionne de façon récursive en générant le code

correspondant aux sous-expressions puis en appelant la fonction llvm correspondant à l'opération demandée. Nous n'avons dans un premier temps pas implémenté la gestion des tableaux pour s'intéresser aux fonctionnalités suivantes en utilisant uniquement les variables entières. C'est une fois ces fonctionnalités implémentées que nous avons ajouté les tableaux dans les expressions

3.2 Génération du code d'instructions

Il fallait ensuite générer le code correspondant à des instructions et suites d'instructions. Nous nous sommes là aussi reposé sur les fonctions llvm pour implémenter la fonction `gen_statement` qui gère ces instructions. Pour les assignations de variables, nous avons utilisé la fonction `build_store`. Pour créer un bloc, on génère une adresse pour marquer son entrée, puis on génère le code des différentes instructions qui le compose en prenant soin de se placer d'abord un nouveau cadre (scope) que l'on referme ensuite, afin de permettre l'utilisation de variables locales. Pour les boucles conditionnelles, un bloc est généré pour chaque partie de la boucle. Dans les cas des boucles `While`, une adresse est générée avant la boucle, avant le corps de la boucle corps de la boucle et à la fin de la boucle. Au début de la boucle est généré un branchement qui pointe vers le corps de la boucle (si la condition est satisfaite) et vers la fin de la boucle (dans le cas contraire). A la fin du corps est généré un renvoi vers le début de la boucle.

3.3 Génération du code de programmes

Une fois cela fait, nous avons pu nous occuper de la génération de programmes et de fonctions. Dans un programme, les fonctions peuvent être déclarée directement avec leur contenu ou par un prototype temporaire. Dans les deux cas, le prototype, qui associe la fonction à ses à ses arguments et indique son type, est d'abord créé est qui. Notons que la redéfinition de fonctions n'est pas acceptée, c'est pourquoi cette possibilité est testée et entraînera une erreur. Ensuite, si le corps de la fonction est spécifié on génère, après avoir changer de cadre puis créé des variables associées à ses arguments, le code correspondant à son contenu. La gestion de fonctions a aussi demandé l'implémentation des instruction Return, qui peuvent être de deux types, Void ou entières selon le type de la fonction. Dans toutes les fonctions, une instruction return (`ret void` pour les fonction Void ou `ret 0` pour les fonctions entières) est généré à la fin du code de la fonction, afin de d'éviter des erreurs du à des fonctions ne terminant pas. Un programme complet est ensuite une simple liste de fonctions. il suffit donc de générer le code de chacune d'entre elles.

3.4 Vérification de type

Enfin, nous avons implémenté un système de vérification de types pour éviter certaines erreurs. En effet, on ne veut pas qu'un programme puisse assigner un tableau à une valeur entière par exemple. Nous avons donc créé une fonction

récursive de typage d'expression. Nous avons également dû augmenter le type de la table de symboles, pour qu'un enregistrement retienne aussi son type.

4 Programme d'exemple

Nous avons implémenté un programme qui testait la plupart des fonctionnalités de notre compilateur, et tel qu'on puisse vérifier facilement la correction. Nous avons donc choisi un programme qui décrit la fractale du *tapis de Sierpinski*. Comme nous n'avions pas accès à des primitives de dessin, le programme se contente d'afficher dans la sortie standard (avec des PRINT) les coordonnées de chaque carré à colorier (un sommet du carré et une longueur). Un programme écrit en PYTHON se charge ensuite de dessiner ces carrés.

Le programme utilise des prototypes de fonctions, des fonctions avec arguments, des appels récursifs, des conditions `IF` sans `ELSE`, des expressions arithmétiques passées en paramètres de fonctions, des divisions, un tableau et un retour dans une fonction de type VOID.

Le programme VSL :

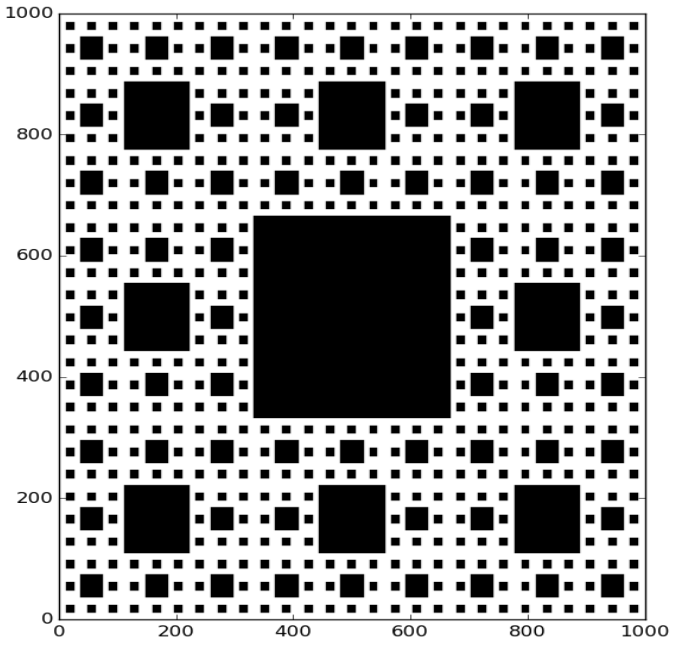
```
PROTO VOID drawsquare(x1, y1, length)

PROTO VOID sierp(depth, x1, y1, length)

FUNC VOID main()
{
  INT args[4]
  args[0] := 4
  args[1] := 0
  args[2] := 0
  args[3] := 1000

  sierp(args[0],args[1],args[2],args[3])

  RETURN 0
}


FUNC VOID drawsquare(x1, y1, length)
{
  PRINT x1, " ", y1, " ", length, "\n"
}

FUNC VOID sierp(depth, x1, y1, length)
{
  INT third, twothird

  IF depth
```

```
  THEN
  {
  third := length / 3
  twothird := (2 * length) / 3

  drawsquare(x1+third, y1+third, third)

  sierp(depth-1, x1, y1, third)
  sierp(depth-1, x1+third, y1, third)
  sierp(depth-1, x1+twothird, y1, third)
  sierp(depth-1, x1, y1+third, third)

  sierp(depth-1, x1+twothird, y1+third, third)
  sierp(depth-1, x1, y1+twothird, third)
  sierp(depth-1, x1+third, y1+twothird, third)
  sierp(depth-1, x1+twothird, y1+twothird, third)
  }
  FI
}
```

Le programme PYTHON chargé d'afficher le résultat :

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from matplotlib.patches import Rectangle

with open('data') as f:
    content = f.readlines()
    fig = plt.figure(figsize=(1,1))
    axis = plt.gca()
    axis.set_xlim([0,1000])
    axis.set_ylim([0,1000])
    for line in content:
        points = line.split()
        if (len(points) == 3):
            x = int(float(points[0]))
            y = int(float(points[1]))
            l = int(float(points[2]))
            axis.add_patch(Rectangle((x,y),l,l,alpha=1,facecolor="#000000"))
    plt.show()
    fig.savefig('sierpinski.png')
```

Et le résultat :



Nous avons également écrit un script qui compile notre programme, l'exécute en écrivant sa sortie dans un fichier, puis appelle le programme Python pour créer l'image. Pour simplifier le programme, les valeurs de rang maximal et de longueur du carré initial sont écrite directement dans le code. Si des petites irrégularités se distinguent dans la figure, elles s'expliquent par le fait que nos programmes n'utilisent que des variables entières et il y a donc quelques problèmes d'arrondi.

5 Tests effectués

Nous avons commencé par créer un script en Bash qui affichait les programmes de tests puis les compilait, puis les exécutait pour automatiser le lancement des tests.

Une fois que notre compilateur générait du code pour les programmes, nous avons pu lancer les tests fournis avec le sujet. Ceux-ci nous ont permis de vérifier que notre gestion des blocs, fonctions, prototypes, tableaux... était correcte.

Ils nous ont également permis de soulever un problème de notre implémentation : les instructions Return dans les fonctions de type Void. Dans une première version, nous avions décidé que les programmes qui essayaient de retourner une valeur dans un programme de type Void n'étaient pas corrects et ne devaient pas être compilés. Cependant, de nombreux programmes de tests le

faisaient, puisqu'il n'y pas d'instruction RETURN sans argument en VSL. Nous avons donc modifié notre compilateur pour qu'un tel appel crée une instruction `ret void` dans le code 3 adresses généré.

6 Conclusion

Dans ce projet, nous avons appris à implémenter un générateur de code 3 adresses.

Notre implémentation progressive et l'utilisation de nombreux programmes de tests nous ont permis de détecter le plus tôt possible nos erreurs.

La possibilité de compiler nos programmes nous a donné une moyen efficace de vérifier chaque fonctionnalité de notre compilateur.

Quelques problèmes se sont posés pendant notre implémentation : blocs vides en cas de `IF` sans `ELSE`, retour dans les fonctions de type VOID, blocs vides en fin de fonction, mauvaise gestion des paramètres des fonctions (pour résoudre ce problème, nous avons créé pour chaque paramètre une variable locale dans laquelle on copiait sa valeur).

Cependant, tous ces problèmes ayant étés corrigés, nous avons un compilateur qui produit du code se comportant comme attendu pour chaque problème de test correct, et qui ne compile pas pour chaque problème de test d'erreur.