Université de Strasbourg

Master Informatique
Compilation
2017 / 2018

Projet : Compilateur pour le langage StenC

Le but du projet est de réaliser un compilateur pour le langage StenC, depuis le code de haut niveau jusqu'à un code exécutable MIPS.

1 Langage StenC

Le langage StenC (prononcer « Stène-Ci ») est un *domain specific language* (ou DSL) basé sur un langage C très simplifié qui a pour but de faciliter l'écriture de codes à stencils. Les codes à stencils sont présentés en Section 1.1 et les détails du langage StenC sont présentés en Section 1.2.

1.1 Codes à Stencils

Les codes à stencils sont une classe particulière de noyaux de calcul où des éléments de tableaux sont mis à jour en fonction de leur voisinage selon un motif particulier appelé *stencil*. On trouve souvent de tels codes dans les applications de traitement d'image, de simulation, et de calcul scientifique (équations aux dérivées partielles, méthodes itératives de Jacobi, de Gauss-Seidel etc.).

Un bon exemple est le filtre Sobel, qui permet d'extraire les contours dans les images. Ce filtre calcule chaque pixel de l'image finale à partir du pixel correspondant de l'image d'origine et de ses pixels voisins. Plus précisément, à partir d'un pixel d'origine p, le filtre Sobel calcule deux pixels intermédiaires g_x et g_y appelés *gradients*, et obtient le pixel final g en calculant la norme des deux pixels intermédiaires. Chaque pixel intermédiaire est obtenu par la somme pondérée du pixel d'origine et de ses voisins selon les formules suivantes :

$$g_x = \begin{array}{|c|c|c|}\hline +1 & 0 & -1 \\ \hline +2 & 0 & -2 \\ \hline +1 & 0 & -1 \\ \hline \end{array} \$\, p, \quad \text{et} \quad g_y = \begin{array}{|c|c|c|}\hline +1 & +2 & +1 \\ \hline 0 & 0 & 0 \\ \hline -1 & -2 & -1 \\ \hline \end{array} \$\, p.$$

Où les tableaux, appelés *stencils*, donnent les poids pour le pixel d'origine et ses voisins directs (la cellule centrale correspond au poids du pixel d'origine p, celle au dessus à droite correspond au poids du pixel directement au dessus et à droite du pixel d'origine, etc.), et où \$ est l'*opérateur de stencil*, qui calcule la valeur d'un pixel en faisant la somme pondérée de ce pixel et de ses voisins avec les poids indiqués par le stencil. Une fois les pixels intermédiaires calculés, le pixel final est donné par la formule :

$$g = \sqrt{\frac{{g_x}^2 + {g_y}^2}{4}}$$

Un code C possible (sans lecture ni affichage de l'image) qui implémente ce calcul est présenté en Figure 1. Dans ce programme, les images sont encodées par des tableaux à deux dimensions contenant le niveau de gris de chaque pixel sous la forme d'un entier. Un exemple d'application de ce filtre sur une image est donné en Figure 2. Par exemple supposons que l'image soit le tableau de pixels ci-après et que l'on souhaite calculer la nouvelle valeur du pixel en ligne 2 et colonne 3 (cellule entourée qui contient le niveau de gris 23). La zone d'application du stencil serait alors formée par les 9 cellules entourées et les calculs seraient les suivants :

	0	1	2	3	4
0	0	1	2	3	4
1	10	11	12	13	14
2	20	21	22	23	24
3	30	31	32	33	34
4	40	41	42	43	44

$g_x = 1*12+0*13-1*14+2*22+0*23-2*24+1*32+0*33-1*34$
$g_y = 1*12+2*13+1*14+0*22+0*23+0*24-1*32-2*33-1*34$
$\texttt{sobel}[2][3] = \sqrt{\frac{{g_x}^2+{g_y}^2}{4}}$

```c
#include <math.h>
#define HEIGHT 128
#define WIDTH  128
int main() {
  int i, j;
  int gx, gy;
  int image[HEIGHT][WIDTH]; // Image originale en niveaux de gris.
  int sobel[HEIGHT][WIDTH]; // Image transformée.

  // Filtre Sobel
  for (i = 1; i < HEIGHT - 1; i++) {
    for (j = 1; j < WIDTH - 1; j++) {
      gx = 1 * image[i-1][j-1] + 0 * image[i-1][j] - 1 * image[i-1][j+1] +
           2 * image[i  ][j-1] + 0 * image[i  ][j] - 2 * image[i  ][j+1] +
           1 * image[i+1][j-1] + 0 * image[i+1][j] - 1 * image[i+1][j+1];

      gy = 1 * image[i-1][j-1] + 2 * image[i-1][j] + 1 * image[i-1][j+1] +
           0 * image[i  ][j-1] + 0 * image[i  ][j] - 0 * image[i  ][j+1] +
          -1 * image[i+1][j-1] - 2 * image[i+1][j] - 1 * image[i+1][j+1];

      sobel[i][j] = sqrt((pow(gx, 2) + pow(gy, 2)) / 4.);
    }
  }

  return 0;
}
```

FIGURE 1 – Exemple de code C naïf du filtre Sobel

1.2 Description du langage StenC

Le langage StenC est pour l'essentiel un sous-ensemble du langage C auquel on a ajouté un type stencil et un opérateur d'application de stencil.

1.2.1 Restrictions par rapport à C

Les restrictions par rapport à C sont les suivantes :
— Le seul type possible est l'entier `int`, ainsi que les tableaux multidimensionnels de `int`. Toutes les variables sont locales et statiques. Il est également possible de déclarer des constantes entières.
— Les opérateurs possibles sont `+`, `-` (unaire et binaire), `*`, `/`, `++` et `--`.

FIGURE 2 – Exemple d'une image originale puis transformée par le filtre Sobel

— Les structures de contrôle possibles sont :
  — les conditionnelles `if` avec ou sans `else`,
  — les boucles `while`,
  — les boucles `for` qui fonctionnent comme des boucles itératives (celles du C sont en fait beaucoup plus générales) : la première partie correspond à l'initialisation d'un itérateur, la seconde à la condition d'arrêt de la boucle et la troisième à la mise à jour de l'itérateur.
— Les appels de fonctions, y compris récursivement, sont possibles. Cependant il vous est demandé de vous concentrer sur le reste avant de tenter d'ajouter le support des fonctions. Dans un premier temps, seule la fonction `main()` sans argument sera présente.
— La bibliothèque standard ne fournit qu'une fonction *printf()* simplifiée et uniquement capable d'afficher une chaine de caractères ASCII. On dispose aussi d'une fonction additionnelle *printi()* qui prend un entier en argument et l'affiche. Aucune inclusion de bibliothèque n'est nécessaire pour cela.

1.2.2 Spécificités liées aux stencils

Les spécificités liées aux stencils sont les suivantes :
— StenC dispose du type `stencil`. Ce type permet de déclarer des tableaux de poids entiers. Pour déclarer un stencil, le nom d'identificateur doit être précédé du mot clé `stencil` et suivi de deux entiers entre accolades et séparés par une virgule :

  1. Le premier entier est un nombre positif ou nul qui indique la distance de voisinage couverte par le stencil. Par exemple, la distance de voisinage des stencils du filtre Sobel présenté en Section 1.1 est de 1, car le pixel le plus éloigné du pixel central est à une distance de 1.
  2. Le second entier est un nombre strictement positif qui indique le nombre de dimensions du stencil. Par exemple le nombre de dimensions des stencils du filtre Sobel présenté en Section 1.1 est de 2, car le voisinage va dans deux directions.

  Une déclaration de stencil est obligatoirement suivie de l'initialisation du tableau de poids, à la manière des initialisations de tableaux en C. Par exemple, les déclarations de stencils suivantes sont correctes :

```
stencil s1{2,1} = { 1, 2, 3, 2, 1 };
stencil gx{1,2} = {{  1,  0, -1 }, {  2,  0, -2 }, {  1,  0, -1 }};
```

— StenC dispose d'un opérateur d'application de stencil. Cet opérateur est noté `$`. Il est binaire, commutatif, et s'applique entre un stencil et un élément de tableau. Son résultat est le calcul de la somme pondérée de l'élément du tableau et de ses voisins. Le voisinage étudié dépend de la taille du stencil, et les poids correspondent aux entrées

du stencil. Le comportement du programme si le voisinage défini par le stencil est en dehors du tableau est indéfini. Par exemple, les expressions suivantes sont correctes :

```
int tab[5] = { 42, 21, 7, 314, 127 };
int somme, moyenne;
stencil s{2,1} = { 1, 1, 1, 1, 1 };
somme = tab[2] $ s;
moyenne = (s $ tab[2]) / 5;
```

Une version StenC du filtre Sobel est donnée en Figure 3, à comparer à la version originale en Figure 1 pour comprendre l'intérêt (voire l'élégance ;) !) de ces extensions [1].

```
#define HEIGHT 128
#define WIDTH  128
int main() {
  int i, j;
  int gx, gy;
  int image[HEIGHT][WIDTH]; // Image originale en niveaux de gris.
  int sobel[HEIGHT][WIDTH]; // Image transformée.
  stencil gx{1,2} = {{  1,  0, -1 }, {  2,  0, -2 }, {  1,  0, -1 }};
  stencil gy{1,2} = {{  1,  2,  1 }, {  0,  0,  0 }, { -1, -2, -1 }};

  // Filtre Sobel
  for (i = 1; i < HEIGHT - 1; i++) {
    for (j = 1; j < WIDTH - 1; j++) {
      sobel[i][j] = sqrt((pow(gx $ image[i][j], 2) +
                          pow(gy $ image[i][j], 2)) / 4.);
    }
  }

  return 0;
}
```

FIGURE 3 – Version StenC du filtre Sobel en Figure 1

2 But du projet

L'objectif de ce projet est d'implémenter un compilateur pour le langage StenC, à écrire en C à l'aide des outils Lex et Yacc, qui produit un code exécutable MIPS correspondant aux programmes d'entrée. Ce travail est à réaliser en binôme et à rendre à la date indiquée par vos enseignants en cours et sur Moodle. Une démonstration de votre compilateur sera faite durant la dernière séance de TP. Vous devrez rendre sur Moodle dans une archive :

— Le code source de votre projet complet dont la compilation devra se faire simplement par la commande « `make` ».
— Un document détaillant les capacités de votre compilateur, c'est à dire ce qu'il sait faire ou non. Soyez honnêtes, indiquez bien les points intéressants que vous souhaitez que le correcteur prenne en compte car il ne pourra sans doute pas tout voir dans le code.
— Un jeu de tests.

1. Cette version utilise les fonctions `sqrt()` et `pow()` qu'il n'est pas demandé d'implémenter. Un calcul plus simple du pixel final g et qui donne des résultats potables est $g = \frac{abs(g_x)+abs(g_y)}{2}$.

2.1 Assembleur MIPS

Le code généré devra être en assembleur MIPS R2000. L'assembleur est décrit dans les documents `spim_long.pdf` et `spim_short.pdf` accessibles sur Moodle. Des exemples de codes MIPS sont également fournis sur Moodle.

Le code assembleur devra être exécuté à l'aide du simulateur de processeur R2000 SPIM. Celui-ci est installé sur la machine `turing`. Pour exécuter un code assembleur, il suffit de faire : `spim <nom_du_code>.s`. Si vous désirez installer SPIM sur votre propre machine :

- Téléchargez le package `spim.tar.gz` sur Moodle
- `tar xvfz spim.tar.gz`
- `cd spim-8.0/spim`
- `./Configure` (Vérifiez qu'il n'y a pas de messages d'erreur : bibliothèque ou logiciel manquant.)
- Modifiez dans le fichier `Makefile` la ligne `EXCEPTION_DIR =` en y saisissant le chemin d'accès au répertoire contenant le fichier `exceptions.s`. Il s'agit du répertoire : `/chemin_vers_spim/spim-8.0/CPU`
- `make`

2.2 Précisions concernant la notation

- Si votre projet ne compile pas ou plante directement, la note 0 (zéro) sera appliquée : l'évaluateur n'a absolument pas vocation à aller chercher ce qui pourrait éventuellement ressembler à quelque chose de correct dans votre code.
- Si vous manquez de temps, préférez faire moins de chose mais en le faisant bien et de bout en bout : on préfèrera un support de StenC incomplet mais qui génère un code assembleur exécutable à une analyse syntaxique seule.
- Élaborez des tests car cela fait partie de votre travail et ce sera donc évalué.
- Faites les choses dans l'ordre et focalisez sur ce qui est demandé. L'évaluateur pourra tenir compte du travail fait en plus (optimisations du code généré par exemple) seulement si ce qui a été demandé a été fait et bien fait.
- Une conception modulaire et lisible sera fortement appréciée (et inversement). Une bonne démarche est de séparer les différentes étapes de la compilation comme cela a été vu en cours, en définissant des formats intermédiaires (par exemple on préfèrera une génération de code sous forme de quads génériques, suivie de la traduction des quads en assembleur, plutôt qu'une génération assembleur directe). Étant donnée l'ampleur de la tâche, on ne demande pas le passage par un arbre de syntaxe abstraite entre analyse syntaxique et génération de code, mais ceux qui le feront correctement seront des héros.

2.3 Recommandations

Conseil important : écrire un compilateur est un projet conséquent, il doit donc impérativement être construit **incrémentalement** en validant chaque étape sur un plus petit langage et en ajoutant progressivement des fonctionnalités ou optimisations (à ce propos, les optimisations du compilateur sont secondaires : si vous écrivez du code modulaire, il sera toujours possible par exemple de changer la représentation de la table des symboles en utilisant une structure de données plus efficace).

Une démarche extrême et totalement contre-productive consiste à écrire la totalité du code du compilateur en une fois, puis passer au débogage ! Le résultat de cette démarche serait très probablement nul, c'est-à-dire un compilateur qui ne fonctionne pas du tout ou alors qui reste très bogué.