Rapport projet ARC

Programmeur : CROS Ivan

Description et fonctionnement du programme

Notre compilateur traduit du pseudo-code en langage de machine RAM selon la définition formelle de Jean-Pierre Zanotti (https://zanotti.univ-tln.fr/ALGO/I31/MachineRAM.html).

Il se découpe en 2 parties fondamentales:

Nos fichiers Bison parser.y et lexer.y pour l'analyse sémantique, qui génère avec asa.c un arbre abstrait.

De l'arbre abstrait, à l'aide de semantic.c et codegen.c avec la table de symbole, on génère du code RAM.

On définit les identificateur dans le fichier Flex, lexer.lex.

Ces fichiers se trouvent dans le sous-dossier `src` et leurs en-tête dans le dossier `include`.

`lexer.lex` permet de reconnaître les identificateurs et de les transformer en token pour notre parser.

`parser.y` contient les règles de grammaire pour définir la structure syntaxique de notre pseudo-code. Ces règles permettent de construire des noeuds de notre arbre abstrait définit dans asa.h et asa.c.

`semantic.c` permet de mettre à jour la table de symbole et son contexte, et de vérifier la cohérence de l'analyse sémantique des différents types. En outre, il permet de fournir quelques indications au contexte et à la gestion mémoire pour la partie codegen.

`codegen.c` permet de générer le code pour la machine RAM à partir de la table de symboles et de l'arbre syntaxique.

Ce processus est une première étape de bootstrapping pour générer un premier compilateur avec le langage C de bas niveau et de nos outils Bison et Flex.

Commandes

```
Projet ARC$ make
Projet ARC$ ./bin/arc test/exemple1.algo
```

Note : Le makefile gère avec `-verbose -Wcounterexamples` les avertissements de conflits dans la grammaire.

La liste des caractéristiques du compilateur et du langage reconnu

L'analyseur est capable de reconnaître :

- des déclarations de variables et liste de déclarations
- des affectations
- des affectations booléennes (exemple : toto← VRAI ou titi←FAUX)
- des fonctions
- des boucles

- des conditions
- des opérateurs binaires arithmétiques et opérateurs logiques
- des expressions de comparaisons

Choix de conception

Soit deux règles: PROGRAMME_ALGO et PROGRAMME

La règle PROGRAMME_ALGO définissant l'algorithme comme étant une liste de déclarations
de variables suivis du programme principal.

La règle PROGRAMME définit la structure du programme principal et encapsule l'ensemble
des directives de déclarations de variables, fonctions, instructions etc.

Idée: Rendre la grammaire plus expressive pour faciliter des ajouts et son maintien par de petits ajustements.

3 types de noeuds sont implémentés pour séparer mieux structurer et séparer la représentation des différentes composantes dans notre codegen:

- noeudOP pour représenter les opérations arithmétiques
- noeudOPL pour représenter les opérations logiques
- noeudCMP pour représenter les comparaisons.

Il en va de même pour les feuilles, nous en avons 3 types:

- feuilleID pour stockes les identificateurs (variables).
- feuilleNB pour stocker les nombres.
- feuilleBOOL pour stocker les valeurs booléennes.

L’objectif est de garder une cohérence pour le programmeur et de compartimenter chaque élément de notre langage.

Par exemple, une valeur autre que 0 ou 1 dans un noeud feuilleBOOL n’a pas besoin d’être relié à un contexte pour être analysé et peut être traité de manière indépendante.

Convention

On peut instaurer la convention dans semantic.c que toute valeur supérieure à 1 dans un noeud feuilleBOOL peut être acceptée par défaut comme VRAI soit remplacé par 1. Un warning peut néanmoins notifier cette interprétation du traducteur.

(Ceci n’a pas été implémenté par faute de temps.)

Limitations du programme

Le programme ne peut générer du code RAM pour l’instant.

Exemple de fichier

N’ayant pu terminer correctement mon compilateur, les exemples fournis de base sont suffisants.