GREFFARD Brice

MARANDON Lorenzo

Compte-rendu ALGO-RAM Compiler(ARC)

Sommaire

Introduction

Des approfondissements de nos connaissance sur les compilateurs et sur le fonctionnement de Flex et Bison en générale ont été trouvée sur https://web.mit.edu/ /Stackoverflow / youtube / chatgpt.

Il y'a eu beaucoup de discussion avec FAUCON Léo sur le compilateur, la gestion des arguments plus précisément qui a été inspirée de son travail .

Contenu de l'archive :

L'archive contient les fichiers suivants :
bin : un dossier qui va contenir le fichier arc exécutable
include : un dossier qui contient les fichier .h
obj : un dossier vide qui contiendra les fichiers objet (.o)
src : un dossier qui contient les fichiers source du programme
test : un dossier qui contient tout les fichier pour tester le compilateur
a.out : un fichier qui contient le code ram produit par l'exécution de arc
Makefile : le fichier a exécuté pour compiler tous les projets
README.pdf : le fichier d'explication du Projet

Utilisation du programme :

```bash
make
./bin/arc test/{nom_du_test}.algo
```

Le code RAM sera produit dans le fichier a.out et pourra être utiliser directement dans la machine RAM de M.Zanotti.

/ !\ Les commentaires à droite des instruction RAM on principalement servie d'outils de débogage durant le projet mais n'ont pas forcément été mise à jour et peuvent ne pas avoir de sens / ne pas être compréhensible sans explication des développeurs

Structure d’un Programme :

Lexique :

Les commentaires :

Les commentaires sur une ligne sont les mêmes qu'en Python et en C (// et #).

Les commentaires sur plusieurs lignes sont pris en compte avec :

/*

Ligne commentaire 1

...

Ligne commentaire x

*/

/ !\ Attention : N'oubliez pas de fermer le commentaire (*/), sinon une erreur sera générée.

Les mots réservés au langage :

"VAR" , "ALGO” , "MAIN" , "DEBUT" , "FIN" , "RENVOIE" , "ECRIRE" , "LIRE" , "TQ" , "FAIRE" ,"FTQ" , "POUR" , "ALLANT" , "DE" , "A" , "FPOUR" , "SI" , "ALORS" , "SINON" , "FSI" , "NON" ,"ET" , "OU" , "VRAI" , "FAUX"

Tout mot réservé utilisé en dehors de son contexte renverra une erreur.

Les opérateurs :

`+` , `-` , `*` , `/` , `%` , `(` , `)` , `<-` , `=` , `!=` , `<=` , `>=` , `,` , ‘ ;’

Les identificateurs :

/ !\ Attention : le ‘A’ ne peut pas être utiliser comme id car il est dans les mots réservé

Début obligatoire : L'identifiant commence obligatoirement par une lettre (a-z, A-Z) ou un souligné (_).

Suite optionnelle : Après ce premier caractère, l'identifiant peut continuer avec des lettres, des chiffres ou des soulignés, avec une longueur maximale de 32 caractères(définit en cours).

Les nombres :

L'ensemble des entiers relatifs.

Les espaces blancs (whitespace):

Les espaces (‘ ‘) , les tabulation (‘\t’) et les retour chariot (‘\n ‘) sont ignorés dans le code mais compte pour la localisation des erreurs.

Tout caractère qui ne rentre dans aucune de ces catégories renverra une erreur lexicale.

Grammaire :

Ecriture d’un programme :

Premièrement : Déclaration des variables globale :

Ce sont les variables qui vont être commune à chaque fonction , il peut ne pas y’en avoir.

Elles sont déclarées de la manière suivante : VAR toto ;

Elles peuvent être déclarée en liste (VAR toto,titi,tata ;).

Elles peuvent être initialiser lors de la déclaration (VAR toto<-1 ;)

Et on peut initialiser une liste de variable (VAR toto,tutu,tata<-10 ;)dans ce cas, toutes les variables prennent la valeur.

Dans une liste de variable, toutes les variables doivent être des ID valide et s’il y’a une affectation, elle doit être à la fin de la déclaration. Donc pas de VAR toto<-10,tata+1 ;

Deuxièmement : Déclaration de fonction qui ne sont pas le MAIN :

Ce sont les fonctions que l’on va pouvoir appeler depuis le MAIN, il peut ne pas y’en avoir.

Une fonction doit commencer par le mot clé ALGO puis un ID puis une liste de paramètre entre parenthèse , il peut ne pas y’en avoir .

Puis il y’a la déclaration des variables local à cette fonction, ces déclarations suivent les mêmes règles que la déclaration des variables global.

Puis il y’a le mot clé DEBUT. Passé ce mot clé on ne peut plus déclarer de variable.

Après le DEBUT, il y’a les liste de la fonction d’instruction (on y reviendra à la fin de la déclaration du programme).

Et pour finir une fonction il faut le mot clé FIN.

Une déclaration de fonction type ressemble donc à :

ALGO fonction(param1,param2)

VAR totolocal,tutulocal<-10 ;

VAR tata ;

DEBUT

Liste d'instruction

(Exemple :1+1 ;)

FIN

Dernièrement : La déclaration du MAIN :

Un programme possède obligatoirement un MAIN, c'est une fonction qui va être exécuter au lancement du programme.

La fonction main commence par le mot clé MAIN et ne prend pas de paramètre. Passé ce point la déclaration est la même que pour une fonction classique.

Un programme type ressemble donc à :

```
VAR global1,global2 ;
VAR global3<-0 ;

ALGO fonction(param1,param2)
        VAR totolocal,tutulocal<-10 ;
        VAR tata ;
DEBUT
        Liste d'instruction
        (Exemple :1+1 ;)
FIN

MAIN()
DEBUT
        fonction(1,2) ;
FIN
```

/ !\ j'utilise des retours a la ligne et des tabulations pour une meilleur lisibilité mais ils ne sont pas obligatoires , la ligne :

ALGO fonction(param1,param2) VAR totolocal,tutulocal<-10 ; VAR tata ; DEBUT Liste d'instruction ; (Exemple :1+1 ;) FIN

Est valide.

Ecriture d'une liste d'instruction :

Les listes d'instruction sont composées d'une ou plusieurs instructions qui peuvent elle-même être des expressions (il n'y a pas de liste d'instruction vide).

Les instruction / expression sont les suivantes :

Expressions (EXP) :

Calculs arithmétiques : addition, soustraction, multiplication, division, modulo, parenthèse

Comparaisons : inférieur, supérieur, égalité, inégalité, inférieur strict et supérieur strict.

Opérations logiques : ET, OU, NON.

Assignations : ID <- EXP (affectation d'une valeur à une variable).

Constantes logiques : VRAI, FAUX.

Appels de fonctions : ID_FONCTION (LIST_EXP)

Lecture d'entrée utilisateur via LIRE.

Pour des facilités grammaticales , on autorise les affectations de type :

EXP OP ID<-EXP (répétable)

(1+a<-1+b<-5)

Ce qui a du sens techniquement mais qui n'est pas plaisant à lire ni à calculer si les expressions deviennent trop grandes ou inclut des appelle de fonction

Mais cela permet aussi :

a<-b<-c<-10 ;

On autorise aussi les affectations dans les appels de fonction :

Fonction(a<-5) ;

Instruction(INSTR) :

Boucles :

TQ (tant que) : Une boucle conditionnelle avec une évaluation au début.

FAIRE…TQ : Une boucle conditionnelle avec une évaluation à la fin.

POUR : Une boucle similaire à for en Python, avec un compteur explicite.

Conditions :

SI…ALORS…FSI : Une condition simple.

SI…ALORS…SINON…FSI : Une condition avec un bloc alternatif.

Vu que chaque instruction prend une LIST_INSTR, elles sont répétables à l'infini et on peut les mixer au bon vouloir:

```
TQ EXP FAIRE
        POUR toto ALLANT DE 1 A tutu+1 FAIRE
                SI toto>0 ALORS
                        //liste d'instruction
                FSI
        FPOUR
FTQ
```

Une suite de SI … SINON SI … est possible (/ !\ ne pas oublier les FSI) :

```
SI a=5 ALORS
        ECRIRE (a);
SINON SI b=4 ALORS
   ECRIRE (b+1);
SINON
        ECRIRE(0);
FSI
```

FSI

Autres instructions :

RENVOIE EXP : Retourne une valeur de la fonction.

ECRIRE (EXP) : Affiche une valeur.

Une fonction n'étant pas obligé d'avoir un RENVOIE , on peut écrire des procédures mais attention : si on cherche à récupérer une valeur dans une procédure on lève une erreur :

```
ALGO procedure()
DEBUT
        1+1 ;
FIN

MAIN()
        VAR a ;
DEBUT
        a<- procedure() // Erreur
FIN
```

Fonctionnement d’un programme

1/ L’analyse lexical :

L’analyse lexical est gérée par Flex , elle génère les tocken/lexème via les règles lexicales qu’on lui a données et les transmet à Bison pour l’analyse syntaxique. Si un segment d'entrée ne correspond à aucune règle définie, Flex génère une erreur lexicale .

2/L’analyse syntaxique :

L’analyse syntaxique est gérée par Bison , Bison reconnaît les erreurs de syntaxe grâce au parser LALR(1) (comme le LR(1) mais avec plus de restriction) qu'il génère à partir de la grammaire définie dans le fichier .y. Il génère une erreur de syntaxe s’il voit :

-Un Tocken inattendu : Si le tocken lu ne correspond pas à ce qui est attendu selon les règles de la grammaire.

-Une fin prématurée de l'entrée : Si la fin du flux est atteinte alors que l'analyseur s'attend encore à d'autres tocken.

-Une règle incomplète : Si l'analyseur ne peut pas compléter une règle avec les token fournis.

Bison crée l’AST (ou ASA en français) durant les actions sémantiques

3/L’arbre syntaxique abstrait(AST) :

L’arbre syntaxique abstrait est géré par ast.[ch]

Les nœud avec une structure complexe sont commenté.

On a choisi de mettre plus de fils à un nœud qu’en TP pour permettre d’implémenter des structures plus complexes. Et on lui a rajoutée des éléments (position , nb_param, nb_declaration) pour faciliter l’analyse sémantique et la génération de code.

L’AST a été pensé pour permettre de recréer le programme , si vous avez le programme vous pouvez créer l’AST et si vous avez l’AST vous pouvez recréer le programme.

4/L’analyse sémantique :

L’analyse sémantique est gérée par semantic.[ch]. Durant l’analyse , on reconnait le type de nœud de l’AST et on exécute la fonction nécessaire. Comme dans un compilateur classique , c’est durant l’analyse sémantique que l’on :

-Vérifie les types : si on veut la valeur d’une procédure ou si on utilise une variable non initialiser.

-Gère la table de symbole : Le remplissage et la mise à jour des paramètres des symbole.

-Met à jour la taille de chaque nœud de l’AST.

5/La génération de code :

La génération de code est gérée par codegen.[ch]. On génère directement du code cible sans passer par du code intermédiaire. C'est durant la génération de code que l'on fait aussi la phase d'édition des liens (linking) qui permet de connaitre l'adresse de début d'une fonction.

Un effort a été fait pour que le code généré soit le plus court possible ,il n'est pas forcément optimal pour autant mais on a essayé d'éviter les instructions inutiles.

La table de symbole :

La table de symbole est gérée dans ts.[ch].

La table de symbole est utilisée durant :

-l'analyse sémantique où elle est remplie et mise à jour.

-la génération de code pour le linking et récupérer les symboles.

Tous les symboles sont gérés de la même façon, seul leur type change :

0 = variable.

1 = fonction.

Et possiblement

2 = pointeur

3 = liste

Et la table de symbole gère elle-même ces erreurs : le manque de mémoire (que l'on a mis a 128 en TP), la redéfinition de variable/fonction et la recherche de variable inexistante.

Structure de la mémoire :

Adresse	Contenu
0	Accumulateur
1	REGISTRE_TMP
2	REGISTRE_TMP_VALEUR
3	REGISTRE_TMP_ADRESSE
4	REGISTRE_RETOUR
5	REGISTRE_PILE
6	REGISTRE_PILE_APPELLE
7	REGISTRE_RENVOIE
8	Réservé
9	Réservé
10 11 12 ... N	Pile (sommet au début) ... ...

Je me suis donné 9 registres réservé pour stocker des valeurs durant l'exécution d'un programme avec :

-REGISTRE_TMP = un registre pour stocker de façon temporaire

- REGISTRE_TMP_VALEUR = un registre pour stocker une valeur de façon temporaire

- REGISTRE_TMP_ADRESSE = un registre pour stocker une adresse de façon temporaire

(REGISTRE_TMP_VALEUR et REGISTRE_TMP_ADRESSE ont été créé après REGISTRE_TMP car on avait besoin de distinguer les adresses et les valeurs ce qui fait que techniquement REGISTRE_TMP ne sert plus rien mais comme certaine fonction l'utilise encore et que l'on avait assez de registre je l'ai gardé.)

- REGISTRE_RETOUR = l'adresse ou l'on doit retourner après un appel de fonction.

- REGISTRE_PILE = stocke l'adresse de la tête de pile.

- REGISTRE_PILE_APPELLE = stocke l'adresse de la teste de pile avant un appel de fonction

- REGISTRE_RENVOIE = stocke la valeur que renvoie une fonction

Mémoire lors de l'appelle de fonction :

Lorsque l'on appelle une fonction la pile est à une adresse inconnue (X) , on gère les paramètres et les déclarations comme si on était dans une nouvelle pile :

Adresse réelle	Adresse Local	Contenu
X	0	REGISTRE_TMP_ADRESSE
X+1	1	REGISTRE_PILE_APPELLE
X+2	2	Valeur de retour
X+3	3	Paramètre 1
X+4	4	Paramètre 2
X+5	5	Paramètre 3 …
X+6	6	Variable 1
X+7	7	Variable 2
X+8	8	Variable 3 …
X+9	9	Pile libre

A ce moment-là , le vrai REGISTRE_PILE_APPELLE a pour adresse X+3 alors que le REGISTRE_PILE_APPELLE qui est stocker a pour adresse le début de la fonction précédente.

A la fin de la fonction , on fait retourner la tête de pile à l'adresse X (son état avant l'appelle de fonction)

Caractéristiques du compilateur

Fonctionnalités prises en charge :

1.Expressions arithmétiques :

Supporte les opérations binaires (+, -, *, /,%,()).

Supporte les priorités de calcule et l'associativité .

2.Expressions booléennes :

Opérations logiques (ET, OU, NON).

Gestion des valeurs littérales (VRAI, FAUX).

3.Variables :

Déclaration et utilisation de variables globales et locales.

Gestion des adresses via la table des symboles.

4.Boucles :

Supporte la boucle TQ vu en cours.

Supporte la boucle POUR qui marche comme le for i in range() de python

Supporte l'imbrication de plusieurs boucles même différente(TQ et POUR)

(La boucle FAIRE TQ est implémentée mais n'est pas reconnu syntaxiquement par bison)

5.Conditions :

Structures SI , SI SINON et SI SINON SI entièrement gérées.

6.Fonctions :

Déclaration et appel de fonctions avec paramètres.

Fait la différence entre fonction et procédure

Supporte l'appel de fonction imbriqué. (fonction1(fonction2(a, b), b))

Supporte l'appel récursif et le multiple appelle récursif

Choix de conception :

On a préféré faire un compilateur qui traite bien tous les aspects de la compilation et qui respecte les phases de compilation plutôt que d'implémenter plus de structure.

On a donc travaillé sur la gestion et la localisation des erreurs, la construction d'un AST propre et la génération de code pour toutes les structures implémentées.

Avec plus de temps , on aurait implémenté les pointeurs et les listes, on aurait changé la manière d'afficher l'AST qui est très vite illisible pour le moment et on aurait amélioré l'affichage des erreurs pour essayer de faire un affichage comme gcc.

La grammaire que l'on a faite ne change pas de celle que l'on a vu en cours et en TP juste que maintenant c'est Bison qui gère la priorité et l'associativité. Les structures rajoutées sont dans la continuité de se qu'on a fait en TP

Pour la modularité, chaque fonctionnalité est séparée dans des modules spécifiques (AST, sémantique, table des symboles, etc.). Ce qui permet une extension facile pour ajouter de nouvelles fonctionnalités.

Limitations connues

-Boucle FAIRE ... TQ : ne fonctionne pas correctement en raison de problèmes non résolus dans la grammaire Bison.

-Gestion des erreurs :

Division par zéro non vérifiée.

Localisation des erreurs syntaxiques perfectible.

Tests détaillés :

/ !\ Attention : La machine RAM réalise un troncage modulo 32768. Donc on ne peut pas avoir de valeur supérieur 32768.

1. bug_boucle.algo

Description :Montre une tentative d'implémenter une boucle FAIRE TQ, qui n'est pas reconnue correctement par le compilateur en raison de limitations dans la grammaire.

2. calcule.algo

Description :Montre la prise en charge des opérations arithmétiques simples et complexes, y compris la priorité des opérateurs, les parenthèses, et le modulo.

Pourquoi c'est important :Vérifie que le compilateur peut correctement évaluer des expressions arithmétiques et gérer les priorités des opérateurs.

3. erreur.algo

Description :Un programme conçu pour tester les différents types d'erreurs gérés par le compilateur, notamment :

- Erreurs lexicales et syntaxiques.
- Erreurs de nommage (variables/fonctions inexistantes).
- Erreurs de redéfinition.
- Erreurs de type.
- Erreurs d'arité.
- Limites de mémoire.

Pourquoi c'est important :Documente les messages d'erreur du compilateur et leur précision, montrant une robustesse dans le traitement des erreurs.

4. factorielle.algo

Description :Implémente une fonction récursive pour calculer la factorielle d'un entier.

Pourquoi c'est important :Vérifie la gestion de la récursivité. Teste le calcul sur plusieurs valeurs avec une boucle for dans la fonction principale.

5. fibo.algo

Description :Calcule récursivement la suite de Fibonacci pour un nombre donné

Pourquoi c'est important :

- Teste les fonctions récursives avec plusieurs appels imbriqués.
- Souligne le coût en performances (4188 instructions pour fibo(8)) pour des programmes lourds.
- Évalue les capacités du compilateur à gérer des appels récursifs complexes.

6. gestion_ts.algo

Description :Teste la gestion des variables globales et locales. Compare une fonction manipulant une variable globale avec une fonction utilisant une variable locale.

Pourquoi c'est important :

- Montre la capacité à gérer la portée des variables.
- Vérifie l'implémentation correcte de la table de symboles pour la résolution des noms.

7. test_global.algo

Description :Programme complet testant diverses structures de contrôle :

Conditions (SI SINON).

Boucles (POUR, TQ).

Retour de valeurs par une fonction.

Pourquoi c'est important :

- Couvre une grande variété de fonctionnalités offertes par le compilateur.
- Montre la gestion des expressions conditionnelles et des boucles.
- Évalue l'interaction entre la fonction principale et les fonctions définies par l'utilisateur.

8. warning.algo

Description :Illustre les cas où des avertissements sont générés :

- Division et modulo pour avertir des potentielles divisions par zéro.

Lecture d'entrée (LIRE) sans garantie de données disponibles.

Détection des variables ou fonctions inutilisées.

Pourquoi c'est important :

Met en lumière des aspects pratiques de l'utilisation du compilateur, comme la génération de warnings pour guider l'utilisateur.

Teste la robustesse du système pour identifier et signaler des cas problématiques.

9.bool.algo

Description :Montre la prise en charge des opérations et comparaisons booléenne simples et complexes, y compris la priorité des opérateurs et les parenthèses.

Pourquoi c'est important :Vérifie que le compilateur peut correctement évaluer des expressions booléennes et gérer les priorités des opérateurs.