I53 - Projet de fin de semestre
Conception d'un compilateur ALGO / RAM

Licence 3 - 2022/2023

Introduction

Le but de ce TP est de construire un compilateur du langage algorithmique vers la machine RAM du cours d'algorithmique de 2ème année (`arc` pour **A**lgo-**R**am-**C**ompiler). Pour cela nous utiliserons conjointement `Flex` et `Bison` pour construire l'arbre syntaxique abstrait du programme, une table de symbole sous forme de double liste chaînée et un module chargé de gérer l'arbre abstrait et de produire le code cible.

L'archive `ProjetARC.tar` contient une version élémentaire du programme. Celui-ci peut être compilé à l'aide de la commande `make`. La structure du projet est présentée ci-dessous. Une étude du `makefile` permettra de bien comprendre la structure générale du projet. Quelques exemples de programmes ALGO que le compilateur doit être en mesure de gérer sont proposés dans le répertoire `test`.

```
>> ls *
makefile

bin:

doc:
arc_doc.tex

include:
asa.h  codegen.h  parser.h  ram_os.h  semantic.h  ts.h

lib:
RAM_OS.ram

obj:

src:
asa.c  codegen.c  lexer.lex  parser.y  semantic.c  ts.c

test:
asa  exemple1.algo  exemple2.algo  exemple3.algo  exemple4.algo  ts
```

La compilation du programme peut être décomposé en les étapes suivantes:

1. **construction de l'arbre abstrait:** on utilise Bison et Flex pour construire un analyseur syntaxique et les fonctions du fichiers `asa.h` pour générer un arbre abstrait à partir des feuilles.
2. **Analyse sémantique:** on parcours l'arbre abstrait en profondeur pour remplir la table de symbole, vérifier les règles sémantiques (déclaration avant utilisaion, valeurs min/max des entiers etc) et calculer au cours des retours arrières le nombre d'instructions correspondant à chaque noeud.
3. **Génération de code:** on parcours l'arbre abstrait en profondeur et on utilise les informations de la table de symbole et les nombres d'instructions pour produire le code RAM.

Table de symboles

Les fichiers `ts.[ch]` sont fournis complets: ce sont les seuls qui n'ont pas besoin d'être modifiés. La table de symbole est une structure de double liste chaînée. La liste principale est une liste de contextes représentant la portée d'un ensemble d'identificateurs. Chaque sous liste est la liste des identificateurs du contexte en question.

```
typedef struct symbole{
  char id[ID_SIZE_MAX];
  int type;
  int size;
  int adr;
  struct symbole *next;
} symbole;

typedef struct contexte{
  char name[ID_SIZE_MAX];
  symbole * liste_symbole;
  struct contexte * next;
} contexte;
```

Arbre de syntaxe abstrait

Afin de produire du code pour la machine RAM il est nécessaire de construire une représentation abstraite du programme analysé. Cette construction se fera à l'aide de la structure `struct asa` définie dans le fichier `asa.h`. Un noeud de l'arbre sera composé d'un type (correspondant chacun à une construction de la grammaire) et d'une structure correspondant au type en question.

Le type des noeuds est simplement un type énuméré :

```
  typedef enum {typeNB, typeOP} typeNoeud;
```

et chaque type correspondra à une structure propre:

```
typedef struct {
  int val;
} feuilleNB;

typedef struct {
  int ope;
  struct asa * noeud[2];
} noeudOP;

typedef struct asa{
  typeNoeud type;
  int memadr;
  int codelen;
  union {
    feuilleNb nb;
    noeudOp op;
  };
} asa;
```

Norme du langage ALGO

La norme d'un veritable langage est un document bien trop pour être conçu avec rigueur dans un simple sujet de travaux pratiques. Nous nous contenterons ici de décrire les prinpales caractéristiques du langage.

Structure d'un programme

Un programme ALGO est constitué d'un seul fichier source l'extension `.algo`. Celui-ci doit comporter la définition de zéro, un ou plusieurs alogrithmes, zéro une ou plusieurs variables ainsi qu'une fonction principale nomée `PROGRAMME()` qui sera exécutée par la machine cible. Celle-ci renvoie une valeur entière en fin d'exécution (0 par défaut).

Types de données

1. `ENTIER`: nombre sur 16 bits (compris entre `ENTIER_MIN`$= -2^{15}$ et `ENTIER_MAX`$= 2^{15} - 1$.
2. `TABLEAU`: suite de cases consécutives en mémoires, adressé à l'aide de l'opérateur d'incidage [ ]. Le premier élément est indicé par 0.
3. `POINTEUR`: Adresse d'une donnée. Obtenu par l'opérateur de déreférencement @.

Portée des identifiants

Chaque identificateur possède une portée définie au moment de sa déclaration.

1. Un identificateur déclaré à l'intérieur d'une fonction n'est visible qu'à l'intérieur de celle-ci.
2. Un identificateur déclaré en dehors de toute fonction est considéré de porté globale et est visible depuis n'importe quelle fonction.
3. Si un identificateur de portée globale porte le même nom qu'un identificateur au sein d'une fonction, l'identificateur global est ignoré.

Langage ALGO

Lexique du langage

1. **Mots Clés**: `PROGRAMME, ALGO, DEBUT, FIN, VAR, TQ, FAIRE, FTQ, SI, ALORS, SINON, FSI, ET, OU, NON, VRAI, FAUX, LIRE, ECRIRE, ALLOUER, RENVOYER`
2. **Ponctuateurs**: `+ - * / % ( ) <- = != <= >= , [ ] @ \n`
3. **Blancs**: `' ' \t`
4. **Nombres**: entiers décimaux sans 0 superflus
5. **Identificateurs**: chaînes de caractères alpha-numériques commençcant par un caractère

Grammaire du langage

1. Un programme est suite d'algorithmes définis par le mot clé `ALGO` un identificateur est un liste de paramètres de type `ENTIER` (aucun marquage) ou `POINTEUR` (`id` précédé d'un `@`).
2. Le corps d'une fonction commence par un suite de zero, une ou plusieurs déclarations de variables (avec ou sans affectation) suivi du mot clé `DEBUT`, une série d'instructions et le mot clé `FIN` (on utilisera le ponctuateurs `\n` comme séparateur d'instructions).

La grammaire est celle manipulée en cours depuis la 1ère année. L'algorithme suivant permet de voir a peu près toutes les constructions nécessaires.

```
1 //echange les elements des cases d indices i et j
2 ALGO ECHANGER( @T,i,j)
3 VAR temp
4 DEBUT
5         temp <- T[i]
6         T[i] <- T[j]
```

```
7         T[j] <- temp
8 FIN
9
10 // Renvoie l indice de la plus petite valeur du tableau T entre i et n
11 ALGO Selection( @T, n, i )
12 VAR imin <- i
13 DEBUT
14         i <- i+1
15         TQ i < n FAIRE
16            SI T[imin] > T[i] ALORS
17               imin <- i
18            FSI
19            i <- i+1
20         FTQ
21         RENVOYER imin
22 FIN
23
24 PROGRAMME()
25 VAR taille, @T
26 VAR i, imin
27 DEBUT
28         //stockage des donnees dans un tableau dynamique
29         taille <- LIRE()
30         ALLOUER( T, taille )
31         i <- 0
32         TQ i < taille FAIRE
33            T[i] <- LIRE()
34            i <- i+1
35         FTQ
36
37         //Tri selection
38         i <- 0
39         TQ i < taille FAIRE
40            ECHANGER( T, i, Selection(T, taille, i))
41            i <- i+1
42         FTQ
43
44         //Affichage du tableau trie
45         i <- 0
46         TQ i < taille FAIRE
47            ECRIRE(T[i])
48            i <- i+1
49         FTQ
50
51         RENVOYER 0
52 FIN
```