I53 - Compilation et théorie des langages
`Flex` et `Bison`

Licence 3 - 2021/2022

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 élémentaire sous forme de liste chaînée et un module chargé de gérer l'arbre abstrait et de produire le code cible.

Utilisation conjointe de `Flex` et `Bison`

Il est possible d'écrire de puissant analyseurs syntaxiques avec `Bison` tout en sous-traitant la construction de l'analyseur lexical à `Flex`. Pour cela on commence par écrire l'analyseur syntaxique puis l'analyseur lexical chargé de reconnaitre les unités lexicales (tokens) définies par `Bison`.

Les fichiers `parser.y` et `lexer.lex` fournissent un exemple élémentaire d'utilisation conjointe de `Flex` et `Bison`. La calculatrice est décrite dans le fichier `parser.y` et le programme `lexer.lex` fournit à bison l'analyseur lexical dont il a besoin pour analyser une entrée. Le programme `Flex` récupère la définition des unités lexicales grâce au fichier `.h` qui sera produit par `Bison` lors de la compilation du fichier `parser.y` avec l'option `-d`. La compilation de la calculatrice se fait alors avec la séquence de commandes suivantes :

```
$ bison -o parser.c -d parser.y
$ flex -o lexer.c lexer.lex
$ gcc -Wall -o parser parser.c lexer.c -lfl
```

Table de symboles

On souhaite étendre la gestion des identificateurs en acceptant toutes les chaînes alphanumériques commençant pas par un chiffre. La table ne sera plus un simple tableau de 26 entiers mais une liste chainée basée sur la structure suivante:

```
typedef struct ts{
  char *id;
  int adr;
  int size;
  struct ts *next;
} ts;
```

Les fichiers `ts.[ch]` sont fournis complets: ce sont les seuls qui n'ont pas besoin d'être modifiés.

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;
```

Construction du compilateur

Le programmes fournis dans l'archive `arc.tar` permet de construire un compilateur élémentaire tout juste capable de construire l'arbre abstrait d'une expression arithmétique composée uniquement d'additions.

Grammaire du langage algo

On considèrera qu'un programme en langage algo respecte les règles suivantes:

- les identifiants sont des chaînes de lettres, chiffres ou souslignés ne commençcant pas par un chiffre;
- les nombres sont écrits en décimal sans 0 superflus à gauche;
- la première ligne commence par le mot clé `ALGO` suivi d'un identidiant représentant le nom de l'algorithme;
- deuxième ligne commence par le mot clé `ENTREE` suivi d'une serie d'identificateurs séparés par des espaces;
- à partir de la troisième ligne débute le corps du programme, celui-ci commence par le mot clé `DEBUT` et se termine par le mot clé `FIN`;
- le séparateur d'instructions est le saut de ligne;
- les espaces et les tabulations sont ignorées;
- l'affectation se fait avec les symboles `<-`
- les variables sont déclarées une par une (mot clé `VAR`) et sont toutes considérées de type entier;
- les commentaires se sont sur une seul ligne et correspondent à toute chaine de caractères commençant par un `#`;
- les expressions booléennes sont à valeur entière: 0 pour `FAUX` non nul pour `VRAI`.

Exemple:

```
ALGO Syracuse
# Affiche le nombre d iterations de
# l algo de Syracuse pour atteindre
# 0 ou 1
ENTREE n
DEBUT
  VAR iteration
  iteration <- 0
  TQ (n != 1) ET (n != 0) FAIRE
    SI (n % 2) = 0 ALORS
      n <- n / 2
    SINON
```

```
      n <- n * 3 + 1
    FSI
    iteration <- iteration + 1
  FTQ
  AFFICHER iteration
FIN
```

Générer du code pour la machine RAM

Le premier travail consiste à modifier le programme pour produire du code. Cela se fera avec la fonction `fprintf`. Par défaut, la sortie se fera dans un fichier appelé `a.out`.

La description complète de la machine RAM peut être trouvée à l'adresse suivante `http://zanotti.univ-tln.fr/ALGO/I31/MachineRAM.html`.

Par exemple le code consistant à calculer l'expression $2 + 3 + 5$ et à l'afficher dans la sortie de la machine est le suivant:

```
  LOAD #2
  STORE 1
  LOAD #3
  STORE 2
  LOAD #5
  ADD 2
  ADD 1
  WRITE
```

Afin de générer un tel code il va falloir faire en sorte que le compilateur puisse définir des adresses mémoire lui servant à stoker les résultats intermédiares des calculs d'expressions comportant plus de 2 opérandes. Pour cela on découpe la mémoire de la machine RAM en deux zones:

- un segment de données servant stocker les variables déclarées et les entrées, (adresse mémoire 1 à 64);
- une zone de **pile** qui servira à l'allocation des variables temporaire dont le compilateur à besoin pour dérouler les calculs (adresse mémoire 65 à 1088).

Gestion des entrées et sorties standards

La machine RAM permet uniquement des manipulation des entrées/sorties standards en mode séquentiel (c-à-d sans retour en arrière). L'écriture dans la sortie standard se fera avec le mot clé `AFFICHER`. Pour la lecture des entrées d'un programme, l'instruction RAM `READ` permet de lire un registre de la bande d'entrée et stocker son contenu dans l'accumulateur.

Exemple de programme

```
ALGO Max                |  Code RAM:
ENTREE x y              |  --------
DEBUT                   |
                        |  READ
  SI (x > y) ALORS      |  STORE 1
    AFFICHER x          |  READ
  SINON                 |  STORE 2
    AFFICHER y          |  SUB 1
  SI                    |  JUMPG 9
FIN                     |  LOAD 1
                        |  JUMP 10
                        |  LOAD 2
                        |  WRITE
                        |  STOP
```

Optimisations et extensions de la grammaire

On pourra améliorer le compilateur avec les ajouts suivants:

1. autoriser les déclarations de variables multiples (ex: `VAR x y z`);
2. autoriser l'initialisation des variables pendant la déclaration (ex: `VAR x <- 0`;
3. autoriser la déclaration de variables de type tableau de taille statique (ex: `VAR tab[5]`);
4. utiliser les instructions spécifiques de la machine RAM pour produire un code plus simple dans le cas d'une opération arithmmétique dont une opérande est une constante (ex: `x+5`);
5. idem pour les instructions de type incrémentation ou décrémentation (ex `x <- x+1`);
6. idem pour les opérations de comparaisons de type `x < y`.