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

Licence 3 - 2020/2021

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. On pourra s'inspirer de la table de symbole donnée au TP7.

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. La première chose à faire est d'essayer de produire le code RAM correspondant à l'exemple donné dans le fichier `somme.algo`.

Grammaire du langage algo

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

- commence par le mot clé `DEBUT` et se termine par le mot clé `FIN`;
- le séparateur d'instructions simples est le point virgule;
- les espaces et les tabulations sont ignorées;
- l'affectation se fait avec les symboles `<-`
- les variables sont déclarées une par une (mot clé `VAR`);

Exemple:

```
DEBUT
  VAR toto;
  toto <- 100;
  TQ (toto != 1) ET (toto != 0) FAIRE
    SI (toto % 2) = 0 ALORS
      toto <- toto / 2;
    SINON
      toto <- toto * 3 + 1;
    FSI
  FTQ
  AFFICHER toto;
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/ALG`

Par 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:

- une zone de **pile** qui servira à l'allocation des variables temporaire dont le compilateur à besoin pour dérouler les calculs;
- une zone de **mémoire statique** qui servira à stocker les données des variables déclarées par l'utilisateur.

Gestion des autres opérations arithmétiques

Modifier le programme pour produire du code pour toutes les expressions arithmétiques parenthésées utilisant les opérateurs reconnus par la machine RAM.

Exemple de programme

```
DEBUT
  (2 + 3*4-(2-7)*3 ) % 5;
FIN
```

Gestion des symboles

En langage algorithmique, on considère que toute variables doit être déclarée avant son utilisation. Il y a donc trois contextes dans lesquels peuvent apparaitre un identificateur:

- la déclaration de variable utilisant le mot clé `VAR` (ex:`VAR toto`)
- l'affectation de variable (ex: `toto = 2+3+5`)
- l'utilisation dans une expression (ex: `3+toto*2`)

Modfier la grammaire et l'analyseur lexical, pour prendre en compte c'est trois cas de figures, créer les noeuds correspondant dans l'arbre abstrait. La gestion des adresses mémoires via la table de symbole se fera lors de la génération de code.

Exemple de programme

```
DEBUT
  VAR toto;
  VAR titi;
  toto <- (2 + 3*4-(2-7)*3 ) % 5;
  titi <- toto*toto + 1;
FIN
```

Gestion des opérations de comparaisons

Modifier le programme pour gérer les opérations de comparaisons. Comme en C, une expressions de comparaisons sera considèrée comme fausse si elle vaut 0 et vraie sinon. Seule les comparaisons utilisant un seul opérateur seront autorisées (`1<toto<2` est interdit).

Les opérateurs booléens `ET, OU, NON` sont autorisés.

Exemple de programme

```
DEBUT
  VAR toto;
  VAR titi;
  toto <- 5 * 2;
  titi <- -8;
  titi <- (toto = 1) ET (titi > 4);
FIN
```

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`, la lecture avec le mot clé `LIRE`.

Exemple de programme

```
DEBUT
  VAR toto;
  VAR titi;
  LIRE toto;
  LIRE titi;
  AFFICHER toto*titi;
  AFFICHER toto > titi;
FIN
```

Gestion de la structure conditionnelles

Ajouter la gestion de la structure `SI-ALORS-SINON-FSI` au compilateur.

Exemple de programme

```
DEBUT
  VAR toto;
  VAR titi;
  LIRE toto;
  LIRE titi;
  SI toto >= titi ALORS
    AFFICHER toto;
  SINON
    AFFICHER titi;
  FSI
FIN
```

Gestion de la structure de répition

Ajouter la gestion de la structure `TQ-FAIRE-TQ` au compilateur.

Exemple de programme

```
DEBUT
  VAR n;
  VAR x;
  VAR y;
  VAR temp;
  LIRE n;
  x <- 1;
  y <- 1;
  TQ n > 0 FAIRE
    temp <- x;
    y <- y+x;
    x <- temp;
    n <- n-1;
  FTQ
  AFFICHER y;
FIN
```

Gestion des tableaux

Ajouter la gestion des tableaux d'entiers. La declaration d'un tableau se fera sous la forme `VAR ID [ EXP ]` , ce qui signifie que l'on accepte la déclaration dynamique de tableau.

```
DEBUT
  VAR n;
  LIRE n;
  VAR tab [n];
  VAR i;
  VAR max;
  i <- 0;
  max <- tab[i];
  i <- 1;
  TQ n > i FAIRE
    SI tab[i] > max ALORS
      max <- tab[i];
    FSI
    i <- i+1;
  FTQ
  AFFICHER max  ;
FIN
```

Gestion des commentaires

Un commentaire en algo est une chaîne de caractères commençant par le symbole `#`. Seuls les commentaires sur une seule ligne sont considérés.

Gestion des procédures

On étend notre langage à la définition de procédures (ou fonctions). Un programme est alors constitué d'une suite de définitions de procédures ainsi qu'une procédure principale appelée `MAIN`. La fonction `MAIN` prend en argument deux paramètres deux variables: une de type entierpour le nombre de données à lire dans l'entrée standard et une de type tableau où seront stocké les entrées.

```
MaFonction(x)
DEBUT
  VAR n <- x*2;
  RENVOYER n*n ;
FIN

MAIN(ARGC, ARGV)
DEBUT
  VAR x;
  SI ARGC > 0 ALORS
    x <- ARGV[0];
  SINON
    x <- 0;
  FSI
  AFFICHER MaFonction(x);
FIN
```

Optimisations et extensions de la grammaire

Pour les plus motivés, on pourra améliorer le compilateur avec les ajouts suivants:

1. autoriser les déclarations de variables multiples (ex: `VAR x, t[2]`)
2. autoriser l'initialisation des variables pendant la déclaration (ex: `VAR x = 0` et `VAR t[2] = {12,-2}`)
3. utiliser les instructions spécifiques de la machine RAM pour produire un code plus simple dans le cas d'une addition de constante (ex: `ADD #5`)
4. idem pour les instructions de type incrémentation ou décrémentation (ex `x <- x+1`)

5. idem pour les opérations de comparaisons de type `x < y`

6. *"sky is the limit"*