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

SWAILEM Abdullah – BA GUBAIR Emad

28/12/2023

Introduction

- Nous avons réussi à compiler le programme contenu dans le fichier `example12.algo`, dans lequel nous indiquons ce qui ne fonctionne pas encore.
- Dans le répertoire test, il y a des exemples numérotés de 1 à 12.
- Pour faciliter l'exécution du code, nous récupérons le code de la machine RAM de M. Zanotti. Ensuite, pour compiler notre algorithme, nous utilisons la commande :
  - `/bin/arc "./test/exemple{file_number}.algo"`
- Vous pouvez l'exécuter avec la commande :
  - `node ./ram/ram.js out.ram ./ram/input`
- Dans le fichier "input", nous mettons les valeurs des données d'entrée.
- Pour faciliter les tests de code, vous pouvez utiliser la commande :
  - `./test -{option} file_number`
- Les options sont les suivantes :
  - "-l" lire
  - "-c" compiler
  - "-e" compiler et exécuter
  - "-h" liste de example
- Dans les sections suivantes, nous expliquerons en détail chaque fichier individuel du projet.

Analyse lexicale (`lexer.lex`) - Reconnaissance du vocabulaire

Il s'agit du découpage du flot de données en unités lexicales, en accord avec le code portant du sens. pour l'exemple 6:

```
VAR g<-9
PROGRAMME()
VAR taille,t[5], @p
DEBUT
    ALLOUER( p, 10 )
    taille <- 5+3
    t[0] <- 1
    p[0] <- 5
FIN
```

- `<PROGRAMME, "PROGRAMME">`
- `<(, "(">`
- `<), ")">`
- `<VAR, "VAR">`
- ..

Analyse syntaxique (`parser.y, asa.h, asa.c`)

Il prend les unités lexicales et produit une structure reflétant la grammaire du langage.

Affichage avec DOT

Pour faciliter la visualisation et la gestion de la structure arborescente de notre programme, nous avons utilisé l'outil DOT. Cette technique s'est avérée être une solution claire et intuitive pour représenter l'arbre syntaxique généré par le compilateur. Pour le dernier exemple, l'image ci-dessous représente cette structure .

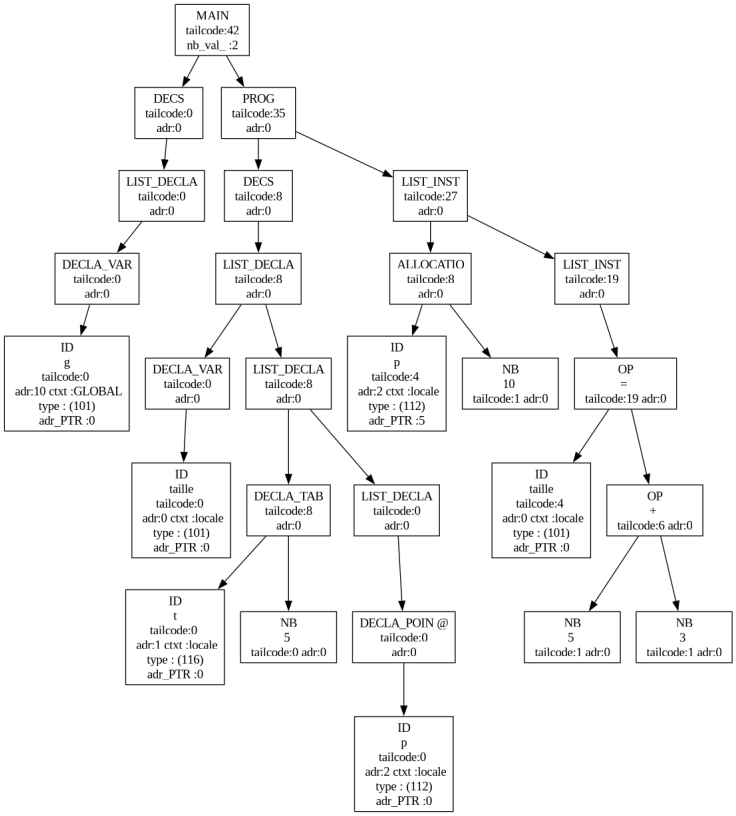

Figure 1: Arbre syntaxique généré par le compilateur pour example 6

Analyse sémantique (`Semantic.c, ts.c`)

Elle utilise l'arbre abstrait pour vérifier le sens des instructions.

- Elle permet de détecter les erreurs:
  1. Mauvais typage
  2. Incompatibilité de type
  3. Déclaration de variables et ajout dans la table des symboles
  4. Dépassement de mémoire
- Nous avons ajouté une fonction : `int error_semantic(const char *s)` pour gérer les erreurs sémantiques.
- Les fonctions `semantic_decla_..` gèrent la déclaration de variables, qu'elles soient locales ou globales.
  1. Pour chaque déclaration nous vérifions s'il a déjà été déclaré, que ce soit dans la portée globale ou locale. Si oui, une erreur est affichée.
  2. Pour chaque appel nous vérifions s'il a déjà été déclaré, que ce soit dans la portée globale ou locale. Si non , une erreur est affichée.

3. Les variables globales vont dans le segment de données, tandis que les variables locales vont dans la pile
4. `INT "VAR x" :` Toutes les variables doivent être déclarées et avoir une adresse avant leur premier usage, et elles ne peuvent pas être redéclarées .
5. `TAB " VAR T[5] :`
6. `PTR "VAR @P" : ALLOUER(P,5)` pour allocation dynamique
   (a) La taille doit être un nombre entier (pas d'expressions, même constantes)
   (b) Ils ont leur adresse dans la pile où ils stockent leur première adresse vers le tas.
   (c) Vous pouvez récupérer leur valeur par `P[5], T[i]`.

- `Fonctions` :Quand on déclare une fonction, on réserve l'espace mémoire pour ses paramètres, ses variables locales de contexte `"nom de fonction"`, et on stocke l'adresse de la première instruction de la fonction dans l'adresse de fonction dans le segment de données. Pour l'appeler, nous commencerons par remplacer les valeurs des variables`"par valeur"` ou leur adresse`"par adresse"` .

```
1              //d finies comme suit :
2              ALGO ID( L_param )
3              DEBUT
4                  //L_inst
5              FIN
```

- Le variable `int VLOCAL` : pour numéroter les variables locales, commençant par 0. Le nombre présent représente son adresse dans la pile et est ajouté à la table des symboles.
- Le variable `int VGLOBAL : :` pour numéroter l'adresse de la variable globale, commençant par le début du segment donné. Son adresse est directement présentée dans la RAM.
- Le variable `int ADRPTR` : Nous attribuons le premier pointeur qui pointera vers le tas à l'adresse 0. Cela signifie qu'il prendra la première adresse dans le tas, et ensuite nous attribuerons la suivante en ajoutant la taille du premier. Par exemple, si nous déclarons au début un pointeur de taille 5, il prendra les adresses de 0 à 4, et ensuite le suivant commencera à 5.
- Table des symboles pour gérer les identificateurs comme dans l'image suivante :

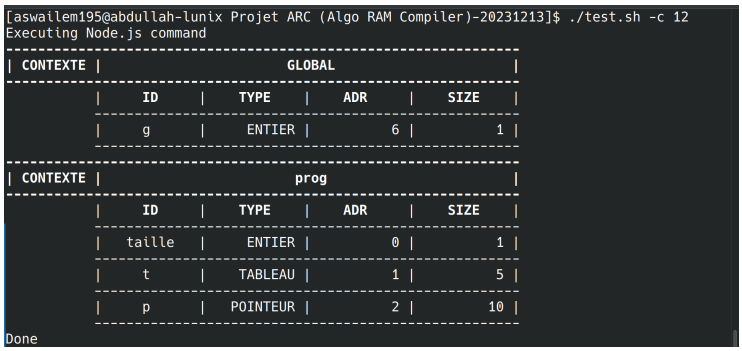

Figure 2: Table des symboles example 6

Génération de code intermédiaire

- Génère du code pour une machine RAM.
- La structure de la mémoire:
  - Registre 0: ACC
  - Registre 1: `RAM_OS_TMP_REG 1 //registre temporaire`
  - Registre 2: `RAM_OS_STK_REG 2 //le début de pile`
  - Registre 3: `RAM_OS_ADR_REG 3 //sommet de pile`
  - Registre 4: `RAM_OS_TAS_REG 4 // début de tas`
  - Registre 5: `RAM_OS_EMPILER_ADR 5 //pour le moment je l'utilise pour l'appel et le retour de fonction`
  - Registre 6: `VGLOBAL = 6 // Pour numéroter l'adresse de la variable globale, on utilise int dans semantic.c.`
  - `PILE_DEBUT_ADR 32`
  - `TAS_DEBUT_ADR 150`
- Nous considérons que la portée des variables locales reste restreinte à toute la vie du code, mais on ne peut pas les utiliser juste dans leur fonction gérée par le `semantic.c`.
- En ce qui concerne le tas, je lui donne une adresse, mais toujours, on insère à l'adresse avant. Cela pourrait dire que son adresse est décroissante.
- L'image suivante présente la structure de la mémoire pour l'exemple 6 :

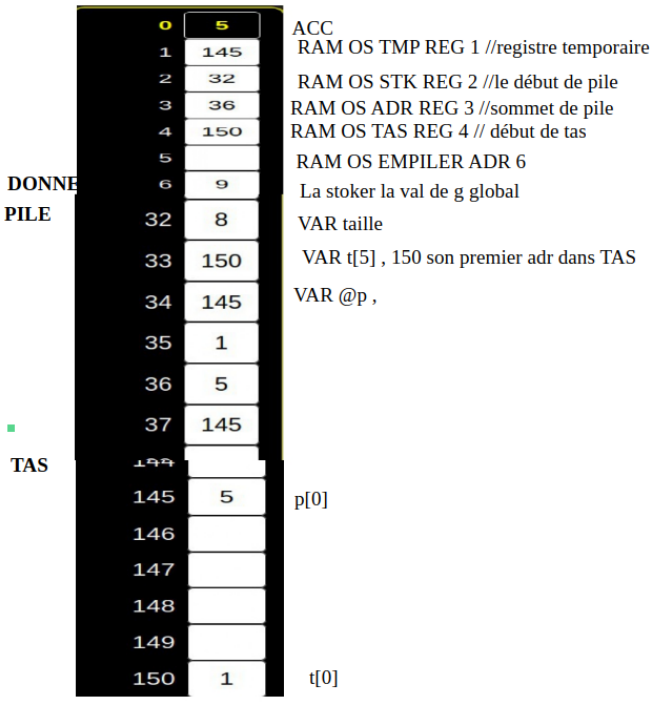

Figure 3: structure de RAM example 6