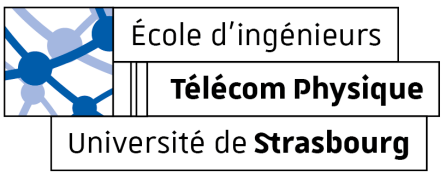
Projet Compilation 2022

Louise Couture SDIA, Aymane Elmahi SDIA,
Félix Lusseau RIO, Oussama Taki Amrani SDIA
2A IR

January 8, 2023



Contents

Introduction

Ce projet a été réalisé en 2022 dans le cadre de nos études d'ingénieurs en Informatique et Réseaux à Télécom Physique Strasbourg et du Master 1 de l'UFR Math-Info. Il a pour but de créer un compilateur pour un programme écrit en langage SoS.

Le dossier "projet compilation" est composé de dossiers regroupant les différents types de fichiers :

- **src** : contient les sources .c utilisées dans le programme
- **include** : contient les .h des fichiers de src
- **lex-yacc** : contient les fichiers lex et yacc
- **obj** : contient les .o créés lors de la compilation
- **bin** : contient l'exécutable, nommé sos
- **tests** : contient les différents programmes à tester

Il y a également un **Makefile** et un fichier **mips.asm** utilisé par Mars ou Spim pour compiler les tests.

Les principaux fichiers dans src regroupent les fichiers contenants les fonctions utilisées :

- **main.c**
- **quads.c** : chargé de convertir la lecture du yacc en langage intermédiaire sous forme d'une liste de quads, appelée Lglobal
- **tos.c** : chargé au cours de la lecture de remplir la table des symboles, appelée tos (Table of Symbol)
- **mips.c** : chargé de traduire la liste des quads en langage MIPS qui sera écrit dans le fichier mips.asm
- **errors.c** : contient la fonction "raler" qui renvoie une erreur pour les primitives systèmes à l'aide de la maccro CHK

1 Exécution

Une fois le projet téléchargé, le compiler avec la commande :

```
$ make
```

Pour exécuter le compilateur, lancer :

```
$ ./bin/sos < "programme SoS"
```

avec un fichier source SoS à compiler. Il est également possible de le saisir directement dans l'entrée standard.

Lancer :

```
$ ./bin/sos -h
```

pour visualiser l'aide suivante :

```
Usage: ./bin/sos [--version | -v] [--tos | -t] [(--output | -o) <name>]
    [--help | -h] < "SoS\_program"
```

Pour lancer les tests, taper :

```
$ make test
```

ou

```
$ make test_mips
```

pour exécuter également Spim s'il est installé.

Un pipeline Gitlab a été mis en place pour réaliser ces tests lors de chaque push de code. Celui-ci s'exécutait dans un environnement Docker dont nous avons personnalisé l'image.

Notre compilateur se compile en C avec les flags sans warnings ou erreurs.
Tous les tests de compilation de SoS en MIPS s'exécutent sans souci.
Les tests d'exécution du code MIPS eux ne sont pas tous fonctionnels...

2 Lex

2.1 Lexique

Dans le fichier lex.lex, la syntaxe est définie par :

- une liste de symboles et mots clés qui ont la priorité pour être reconnus : “-“ “/” “+” etc… et “if” “for” “while” etc…
- tout autre mot ne contenant pas de caractères spéciaux est lu comme étant un “ID”, à ce stade ID n’a aucune signification particulière et ne fait référence qu’à un mot qui n’est pas un mot clé du langage
- toute phrase encadrée par des guillemets, simples ou doubles, est lue comme une CHAÎNE.
- La tabulation et les commentaires sont ignorés.

Durant la lecture, le lex affiche les mots lus précédés d’une indentation.
Exemple:

```
while
x
=
0
```

3 Yacc

3.1 Syntaxe

Les règles de grammaire sont basées sur celles données dans le sujet du projet. Avec elles, le yacc est capable de lire sans erreur tout programme utilisant le langage SoS. Certaines règles ont été ajoutées pour effectuer des actions particulières à des endroits précis du programme.

- M: %empty
  sert à renvoyer la taille de **Lglobal** (donc le nombre de quads actuel dans Lglobal) à l'endroit où est placé M, très utilisé pour les branchements IF, WHILE, CASE…
- id: ID
  utilisé lorsque l'ID lu est censé être un identificateur et doit être ajouté (s'il n'existe pas déjà) dans la **ToS** (Table of Symbol)
- for_id_liste_branchement : %empty
  utilisé dans les règles contenant FOR afin d'insérer ses branchements dans Lglobal
- dec_fct: %empty
  utilisé pour la règle declaration_de_fonction pour insérer la déclaration de fonction dans Lglobal et modifier la profondeur actuelle pour la ToS

Durant la lecture, le yacc affiche les règles utilisées.
Le yacc doit construire la liste **Lglobal** qui contient le code intermédiaire. Toutes les règles ont été traitées pour produire ce code intermédiaire et il est correct. Cependant comme il a été fait avant le MIPS et dû au fait que nous ne connaissions pas le langage avant ce projet, il s'est révélé (trop tard) que le code intermédiaire n'est pas toujours optimal pour une traduction en MIPS.

4 Quads

Nous décrirons ici le contenu des fichiers quads.c et quads.h

4.1 Structures

Dans quads.h sont déclarées les structures permettant de créer la liste de quads ainsi que d'autres structures utilisées dans le yacc afin de faire circuler certaines informations entre les règles.

- listQ, quads et quadOP :

```
typedef struct listQ {
   struct listQ *next;
   struct quads *quad;
   int taille;
}listQ;
```

listQ est une liste chainée du langage intermédiaire de quads.
Chaque quad est une structure composée d'un quadOP, d'un op1, d'un op2 et d'un res. **quadOP** est une énumération contenant toutes les opérations possibles. op1 et op2 sont des pointeurs vers d'autres quads ou vers une ToS, et res est un pointeur vers une ToS.

```
/* quad / instruction à 3 addresse*/
typedef struct quads {
   enum{Q_ADD=100, // +
        Q_LESS,    // -
        ...
        Q_MUL,     // *
        Q_DIV,     // ./.
        }kind;
   quadOP *op1,*op2,*res;
} quads;
```

quads est la structure contenant l'instruction 3 adresses du code intermédiaire, elle contient un élément appelé kind pour faire référence au type d'instruction qu'elle contient et 3 opérateurs.

```
/* opérande d'une instruction à 3 addresse*/
typedef struct quadOP {
   enum{QO_CST=1, // constante (int)
        QO_STR,  // chaine/mot (string)
        QO_ID,   // identificateur (string)
        QO_FCT,  // fonctions (string)
        QO_ADDR, // addresse (goto) (int)
        QO_BOOL, // booléen
        QO_TAB   // tableau (string)
```

```
      }kind;
   int id_type;
   union{int cst;char *name;}u;
} quadOP;
```

quadOP stocke les opérandes des quads, elle possède un argument kind qui fait référence à son type (identificateur, fonction, chaîne de caractères etc…), un argument id_type qui qualifie le type de la variable (string, int…) si kind=QO_ID, enfin elle possède un argument u qui peut soit être une chaîne de caractère soit un entier selon le type de la variable.

- embranchement :

```
typedef struct embranchment {
   listQ *True; // goto dans le cas ou le bool=true
   listQ *False; // goto dans le cas ou le bool=false
} embranchment;
```

embranchement est une structure utilisée lors des IF, ELSE IF, CASE,WHILE etc… Elle contient 2 listeQ qui stockent tous les branchements (Q_GOTO, Q_IF_GOTO) qui sont vides (pas d'adresse pour les goto). La fonction void complete(listQ *listGT, int addresse) est chargée de remplir ces listQ avec l'adresse donnée en argument.

- case_test et for_brnch :

```
/* utilisé pour CASE ESAC: contient les test + les branchements */
typedef struct case_test {
   listQ *test;
   embranchment *branch;
} case_test;

/* utilisé pour FOR contient les test + les branchements */
typedef struct for_brnch {
   int addr_goback;
   quads *Max;
   quads *GoTo;
   quads *Id;
} for_brnch;
```

case_test est une structure contenant une liste de valeurs et une liste d'adresses associées.
for_brnch est une structure contenant l'adresse de départ et d'arrivée d'un saut de boucle.

4.2 Fonctions

Toute structure possède une fonction de création et d'autres fonctions nécessaires à son implémentation. À chaque fois qu'un quadOP est créé, il est affiché dans le terminal. La fonction QOcreat_temp est chargée de créer une variable temporaire. Toutes les variables temporaires ont un nom de la forme ___TEMP___x où x est le numéro de la variable temporaire. La variable globale nb_temp est chargée de compter le nombre de variables temporaires afin de les nommer.
Pour toutes les structures, il existe:

- un Garbage Collector (GC_nom_structure) qui est un tableau stockant tout élément de cette structure une fois créé
- un Indice de garbage collector (I_nom_structure) qui compte le nombre d'éléments créés

Toutes les structures de quads.c sont libérées en même temps grâce à la fonction Lfree qui utilise les Garbage Collector. Pour listQ, quads et quadOP il existe une fonction permettant d'afficher leurs informations, elles sont appelées à la fin du programme pour afficher la liste de quads Lglobal.

Des fonctions ont été mises en place pour:

- ajouter un quad à la liste
- ajouter un saut à la liste
- ajouter une valeur et une adresse à une structure case_test
- ajouter une adresse de départ et d'arrivée à une structure for_brnch
- remplir la liste des quads avec des opérations mathématiques
- gérer les opérations booléennes

5 Table of Symbol

5.1 Structure

La table des symboles est une table de hachage de symboles chainée en profondeur. Chaque symbole est une structure contenant:

- un nom
- un type de variable (identifiant, fonction, tableau)
- la taille du tableau si applicable
- un type d'identfiant (entier, flottant, chaîne, booléen) si applicable
- une adresse mémoire vers la variable locale si elle existe

```
struct tos_entry {
  char *str;
  int used;
  int depth;
  enum { IDENTIFIER, FUNCTION, ARRAY } var_kind;
  int tab_length;
  enum { UNDEFINED, INT, FLOAT, STRING, BOOL } type;
  struct tos_entry *next_lvl[MAX_TOS_SIZE];
};
```

5.2 Fonctions

Des fonctions ont été mises en place pour:

- créer la table
- ajouter un symbole à la table
- chercher un symbole dans la table à partir de son nom
- mettre à jour le type d'un symbole à partir de son nom
- afficher le contenu de la table
- supprimer la table en libérant la mémoire

La table des symboles est chargée de stocker les noms des variables du code lu afin de leur assigner un espace lors de l'écriture en MIPS. Elle permet aussi de vérifier si les variables peuvent être utilisées selon la profondeur (code général ou fonction) où elles se trouvent. Cette table utilise une fonction de hachage afin d'accéder rapidement aux variables appelées. Lorsqu'une variable est lue dans le yacc, son type est toujours mis par défaut à UNDEFINED et est censé être changé plus tard lorsque le type est découvert. Comme SoS mélange beaucoup

les INT et les STRING il est parfois difficile de déterminer les types d'une variable, le type reste alors UNDEFINED. Ce problème a des conséquences sur la traduction en MIPS.

Comme la traduction en assembleur MIPS se passe après l'analyse syntaxique, nous sommes obligés de conserver l'entièreté de la table des symboles. Il y a alors un souci sur l'origine d'une variable locale car on ne sait pas dans quelle fonction elle a été déclarée. Encore plus dans le cas où elle est redéclarée locale dans une autre fonction car cela écrase la version précédente de la variable.

6 MIPS

6.1 Fonctions

Le fichier **mips.c** est chargé de traduire la liste **Lglobal** en assembleur MIPS qui sera écrit dans le fichier **mips.asm** (ou un fichier donné en argument) avant d'être compilé par Mars ou Spim. Il se décompose en 2 étapes :

- à la première étape, il prend la **ToS** afin de déclarer toutes les variables qui ont été rencontrées dans le champ .data.
- à la deuxième étape, la fonction prend la liste **Lglobal** et traduit chaque quad en assembleur MIPS.

Des fonctions ont été mises en place pour :

- traduire les quads en langage MIPS
- gérer les sauts
- gérer les structures de contrôle de flux (if, while, for)
- gérer les opérations mathématiques
- gérer les opérations booléennes
- gérer les affectations et les déclarations de variables
- gérer les entrées et sorties (scanf et printf) // non terminé à cause des types

6.2 Capacités

Le compilateur est capable de gérer :

- les opérations mathématiques de base (+, -, *, /, %)
- les opérations booléennes de base (&&, ||, !)
- les structures de contrôle de flux (if)
- les affectations et déclarations de variables de différents types (entier, flottant, chaîne, booléen)
- les entrées et sorties (scanf et printf) (à cause des problèmes de type, cette partie n'a pas été terminée)
- les sauts conditionnels et inconditionnels
- les déclarations et appels de fonctions (les arguments ne sont pas importés à cause du problème entre les quads et le MIPS)
- la terminaison du programme