Université de Strasbourg

Master Informatique 1ère année
Compilation - Projet 2022

Compilateur pour le langage **SoS**

But du projet : Réaliser un compilateur pour le langage SoS décrit ci-dessous. SoS est un langage impératif simple qui utilise une syntaxe et des fonctionnalités issues d'un sous-ensemble de langage shell unix (Sous-Shell). Ce compilateur produira en sortie du code assembleur MIPS qui devra s'exécuter sans erreurs à l'aide d'un simulateur de microprocesseur MIPS.

1 Présentation du langage

1.1 Considérations lexicales

Tous les mots clés de SoS sont en minuscules. Les mots clés et les identificateurs sont sensibles à la casse. Par exemple, **if** est un mot clé, alors que **IF** est un identificateur ; **foo** et **FOO** sont deux noms différents qui font référence à deux identificateurs distincts.

Les mots réservés sont :

if then for do done in while until case esac echo read return exit local elif else fi declare test expr

Les commentaires commencent avec # et se terminent avec la fin de ligne.

Les blancs apparaissent entre les tokens. Un blanc est défini comme étant un ou plusieurs espaces, tabulations, caractères de saut de ligne ou commentaires.

Les chaînes de caractères sont des suites de caractères. Si elles contiennent des blancs, les constantes doivent obligatoirement être délimitées par des doubles quotes (") ou des simples quotes (').

Les nombres en SoS sont des entiers signés sur 32 bits, c'est-à-dire des valeurs décimales entre -2147483648 et 2147483647. Notons que la vérification d'une valeur représentable est effectuée plus tard. Une longue suite de chiffres (par exemple 1234567891234567890) est bien reconnue comme un token unique.

Un caractère est tout caractère ASCII imprimable (les valeurs ASCII entre les valeurs décimales 32 et 126, ou octales entre 40 et 176), autres que double quote ("), simple quote ('), ou backslash (\), mais incluant les suites de deux caractères "\" " pour spécifier la double quote, "\' " pour spécifier la simple quote, "\\" pour spécifier le backslash, "\t" pour spécifier la tabulation, et "\n" pour spécifier le saut de ligne.

En SoS, toute valeur d'argument, d'opérande, de constante ou de variable est considérée par défaut comme étant une chaîne de caractères. Cependant, si la chaîne est constituée uniquement de chiffres, éventuellement préfixés par le signe plus (+) ou le signe (-), alors celle-ci pourra servir d'opérande lors d'opérations entières arithmétiques et logiques.

Dans la grammaire de la section suivante :

- **mot** est un symbole terminal désignant toute chaîne de caractères sans espace, tabulation ou retour à la ligne. Cette chaîne ne peut pas être égale à un mot clé du langage, ni à un point-virgule (;), ni à un crochet ouvrant ([) ou fermant (]), ni à une parenthèse ouvrante (() ou fermante ()), ni au signe égal (=), ni à l'opérateur "différent" (!=), ni à la barre verticale (|), ni au point d'exclamation (!), ni au symbole $, ni au symbole *, ni aux accolades ouvrantes ou fermantes ({ ou }), ni à un opérateur arithmétique, logique, ou conditionnel.
- **chaîne** est une chaîne de caractère pouvant inclure des espaces, des tabulations, ou des retours à la ligne.

1.2 Grammaire du langage

Notation :

<foo>	signifie que foo est un symbole non terminal
foo	(en gras) signifie que **foo** est un symbole terminal
|	est le séparateur d'alternatives

<programme> → <liste_instructions>

<liste_instructions> → <liste_instructions> **;** <instruction>
| <instruction>

<instruction> → **id** = <concaténation>
| **id[**<opérande_entier>**]** = <concaténation>
| **declare id[entier]**
| **if** <test_bloc> **then** <liste_instructions> <else_part> **fi**
| **for id do** <liste_instructions> **done**
| **for id in** <liste_opérandes> **do** <liste_instructions> **done**
| **while** <test_bloc> **do** <liste_instructions> **done**
| **until** <test_bloc> **do** <liste_instructions> **done**
| **case** <opérande> **in** <liste_cas> **esac**
| **echo** <liste_opérandes>
| **read id** | **read id[**<opérande_entier>**]**
| <déclaration_de_fonction>
| <appel_de_fonction>
| **return** | **return** <opérande_entier>
| **exit** | **exit** <opérande_entier>

<else_part> → **elif** <test_bloc> **then** <liste_instructions> <else_part>
| **else** <liste_instructions>
| ϵ

<liste_cas> → <liste_cas> <filtre> **)** <liste_instructions> **; ;**
| <filtre> **)** <liste_instructions> **; ;**

<filtre> → **mot** | **"chaîne"** | **'chaîne'**
| <filtre> '|' **mot**
| <filtre> '|' **"chaîne"**
| <filtre> '|' **'chaîne'**
| ***

<liste_opérandes> → <liste_opérandes> <opérande>
| <opérande>
| **${id[*]}**

<concaténation> → <concaténation><opérande>
| <opérande>

<test_bloc> → **test** <test_expr>

<test_expr> → <test_expr> **-o** <test_expr2>
| <test_expr2>

<test_expr2> → <test_expr2> **-a** <test_expr3>
| <test_expr3>

```
<test_expr3>              →  ( <test_expr> )
                          |  ! ( <test_expr> )
                          |  <test_instruction>
                          |  ! <test_instruction>

<test_instruction>        →  <concaténation> = <concaténation>
                          |  <concaténation> != <concaténation>
                          |  <opérateur1> <concaténation>
                          |  <opérande> <opérateur2> <opérande>

<opérande>                →  ${id} | ${id[<opérande_entier>]}
                          |  mot | $entier | $* | $ ?
                          |  "chaîne" | 'chaîne'
                          |  $( expr <somme_entière> )
                          |  $( <appel_de_fonction> )

<opérateur1>              →  -n | -z

<opérateur2>              →  -eq | -ne | -gt | -ge | -lt | -le

<somme_entière>           →  <somme_entière> <plus_ou_moins> <produit_entier>
                          |  <produit_entier>

<produit_entier>          →  <produit_entier> <fois_div_mod> <opérande_entier>
                          |  <opérande_entier>

<opérande_entier>         →  ${id} | ${id[<opérande_entier>]} | $entier
                          |  <plus_ou_moins> ${id}
                          |  <plus_ou_moins> ${id[<opérande_entier>]}
                          |  <plus_ou_moins> $entier
                          |  entier | <plus_ou_moins> entier
                          |  ( <somme_entière> )

<plus_ou_moins>           →  + | -

<fois_div_mod>            →  * | / | %

<déclaration_de_fonction> →  id ( ) { <décl_loc> <liste_instructions> }

<décl_loc>                →  <décl_loc> local id = <concaténation> ;
                          |  ϵ

<appel_de_fonction>       →  id <liste_opérandes>
                          |  id
```

1.3 Structure d'un programme en SoS

Un programme en SoS est une suite d'instructions séparées par des points-virgules (;), et se terminant par une instruction **exit**. Une déclaration de fonction peut se situer n'importe où dans le code du programme principal, ou dans le code d'une autre fonction. Mais pour qu'une fonction puisse être appelée, son identificateur doit d'abord être connu. L'exécution d'un programme se faisant séquentiellement, il s'ensuit qu'une fonction ne pourra être appelée que lorsque l'exécution sera passée par la déclaration de la fonction.

Un programme, une fois qu'il a été compilé en assembleur MIPS, peut être lancé avec des arguments séparés par des espaces. Ces paramètres sont alors accessibles dans le programme à travers les références **$1**, **$2**, ..., pour le 1er argument, 2ème argument, ... respectivement.

1.4 Types

Les variables ne sont pas typées et sont essentiellement des chaînes de caractères, mais, selon le contexte, les opérations arithmétiques et logiques sont permises, à condition que les valeurs des opérandes ne contiennent que des chiffres, éventuellement préfixés par le signe plus (+) ou le signe (-). Dans le cas contraire, le compilateur, ou le programme en cas de valeurs d'opérandes inconnues à la compilation, provoquera une erreur.

Les tableaux sont indicés à partir de 0, uni-dimensionnels et statiques. Un tableau indicé est créé à l'aide du mot clé **declare** en spécifiant sa taille, comme dans `declare tableau[10]`. Une telle déclaration peut se situer n'importe où dans le code du programme principal, ou dans le code d'une fonction. Mais son utilisation ne peut se situer qu'après sa déclaration. Un tableau est toujours global à tout le programme, quelque soit l'endroit où il est déclaré.

1.5 Règles de portée

Une variable est déclarée implicitement lors de sa première affectation. Toute variable est globale à tout le programme, sauf si elle est déclarée en tout début du corps d'une fonction, dans la partie <decl_loc>, avec le mot clé **local**. Dans ce dernier cas, elle est locale à la fonction où elle est déclarée. Toute utilisation d'une variable doit être précédée dans le texte du programme par au moins une affectation de cette variable, à l'intérieur ou en dehors de la fonction où elle est utilisée.

Dans le cas particulier de l'instruction de boucle **for**, la variable compteur de boucle **id** est déclarée implicitement dans l'entête de la boucle, mais elle peut avoir été déclarée précédemment.

Comme indiquée plus haut, toute utilisation de tableau ou de fonction doit être précédée par la déclaration du tableau correspondant ou de la fonction correspondante. Un tableau ou une fonction, dès leur déclaration, sont utilisables globalement en n'importe quel point successif du programme. Il n'y a pas de tableau local ou de fonction locale, même si celle-ci est déclarée à l'intérieur d'une autre fonction.

Une fonction peut donc accéder à toutes les variables globales du programmes, et à celles qui lui sont locales (suite à une affectation avec **local**). Elle peut aussi disposer de paramètres d'entrée auquels elle accède à travers les références **$1**, **$2**, ..., pour le 1er argument, 2ème argument, ... d'appel. Par exemple, un appel de la forme : `ma_fonction truc ${machin} "la chose"`, aura pour effet la substitution de **$1** à la chaîne de caractères "truc", de **$2** à la valeur de la variable `machin` (qui est une chaîne de caractères) et de **$3** à la chaîne de caractères "la chose".

Une fonction retourne deux valeurs différentes : un résultat et un statut (une valeur entière). Un résultat est renvoyé à travers une instruction d'affichage (**echo**), alors qu'un statut est renvoyé à l'aide de l'instruction **return entier**. Le résultat d'une fonction peut être affecté à une variable : `id = $( <appel_de_fonction> )`. Si la fonction n'exécute pas d'instruction **echo**, alors sont résultat est une chaîne vide (""). Si le corps de la fonction contient plusieurs utilisations de l'instruction **echo**, c'est la dernière utilisation qui sert de résultat. Si l'instruction **echo** est utilisée avec plusieurs arguments, le résultat de la fonction est unique et est la chaîne de caractères composée de tous les arguments, séparés par un espace. Le résultat d'une fonction est donc toujours unique, mais une fonction peut modifier n'importe quelle variable globale du programme. Le statut d'une fonction est accessible par **$?**, qui contient le statut de la dernière fonction exécutée (0 si celle-ci n'a pas renvoyée de statut).

1.6 Emplacements

Comme les tableaux possèdent des tailles fixées à la compilation, ils peuvent être alloués dans la zone de données statiques du programme, et n'ont pas besoin d'être alloués sur le tas. Par contre, certaines chaînes de caractères pouvant être découvertes à l'exécution (entrée à travers **read**, ou comme paramètre de ligne de commande, ou résultante d'une concaténation), celles-ci nécessitent une allocation sur le tas (appel système `sbrk` en assembleur MIPS).

1.7 Invocation de fonction et retour

L'invocation d'une méthode implique (1) éventuellement le passage de valeurs d'arguments de l'appelant vers l'appelé, (2) l'exécution du corps de l'appelé, et (3) le retour vers l'appelant, éventuellement avec un résultat et/ou un statut (voir ci-dessus).

Le passage d'arguments se fait en donnant une liste d'arguments séparés par des espaces lors de l'appel, à la suite de l'identificateur de la fonction. Les arguments sont évalués de gauche à droite et accédés dans le corps de l'appelé à travers les références **$1**, **$2**, ... Ceux-ci peuvent également être accédés en une seule chaîne de caractères, qui comprend tous les arguments séparés par un espace, à travers la référence **$***. Le corps de l'appelé est exécuté en exécutant ses instructions en séquence.

Une fonction redonne le contrôle à l'appelant lorsque **return** est exécuté, ou lorsque sa fin textuelle est atteinte.

L'exécution par la fonction de l'instruction **return** provoque le retour à l'appelant. De plus, si cette instruction est utilisée avec un argument **entier**, alors cette valeur entière correspond à un statut retourné par la fonction. Ce statut peut être obtenu par l'appelant avec **$?**, qui contient le statut de la dernière fonction exécutée. Si cette fonction ne retourne pas de statut, alors **$?** contient la valeur 0.

1.8 Structures de contrôle

test

L'instruction **test** permet de comparer des chaînes de caractères et des variables :

- <concaténation1> = <concaténation2> est vrai si les chaînes <concaténation1> et <concaténation2> sont égales, faux sinon.
- <concaténation> **!=** <concaténation> est vrai si les chaînes <concaténation1> et <concaténation2> sont différentes, faux sinon.
- **-n** <concaténation> est vrai si <concaténation> n'est pas une chaîne vide.
- **-z** <concaténation> est vrai si <concaténation> est une chaîne vide.
- Pour la forme <opérande> <opérateur2> <opérande>, les opérandes doivent obligatoirement être des chaînes contenant exclusivement des chiffres, éventuellement préfixés par le signe plus (+) ou le signe (-), sinon l'instruction provoque une erreur :
  - **-eq** : égal
  - **-ne** : non égal
  - **-gt** : strictement supérieur
  - **-ge** : supérieur ou égal
  - **-lt** : strictement inférieur
  - **-le** : inférieur ou égal

Les expressions suivent des règles normales d'évaluation. En l'absence d'autres contraintes, les opérateurs sont évalués de la gauche vers la droite. Les parenthèses peuvent être utilisées pour modifier cet ordre, ainsi que les opérateurs logiques **-a** («et», le plus prioritaire) et **-o** («ou»).

if

L'instruction **if** possède la sémantique habituelle. Tout d'abord, <test_bloc> est évalué. Si le résultat est vrai, alors la branche **then** est exécutée. Sinon, la branche **else** ou **elif** est exécutée, si elle existe.

for

Le symbole **id** est la variable d'index qui supplante toute variable de même nom déclarée précédemment, si elle existe. Elle est locale que si elle a été déclarée comme telle précédemment.

Cette variable prend une à une les valeurs de <liste_opérandes> à chaque itération de la boucle, et termine avec la valeur du dernier élément de cette liste. Le symbole terminal **${id[*]}** désigne la liste de tous les éléments du tableau **id**. Ainsi, tous les éléments d'un tableaux peuvent être parcourus un à un avec une boucle **for**.

Si <liste_opérandes> n'est pas présent dans l'entête de la boucle, alors **id** prend successivement les valeurs des paramètres **$1**, **$2**, ... La boucle se termine lorsque tous les éléments de <liste_opérandes> ou tous les paramètres ont été parcourus et ont donné lieu à une itération.

while

La boucle **while** suit le fonctionnement habituel d'une telle boucle : les itérations sont exécutées tant que <test_bloc> est vrai.

until

Les itérations de la boucle **until** sont exécutées jusqu'à que <test_bloc> soit vrai.

case

La structure case offre la possibilité de sélection d'un des cas à condition que <opérande> corresponde à <filtre>. Ainsi, la <liste_instructions> associée est exécutée et les autres cas sont ignorés. Le métacaractère ***** peut être utilisé comme filtre pour définir un cas par défaut. Plusieurs alternatives peuvent composer un filtre à l'aide de l'opérateur "|".

1.9 Expressions

Les expressions suivent des règles normales d'évaluation. En l'absence d'autres contraintes, les opérateurs de mêmes priorités sont évalués de la gauche vers la droite. Les parenthèses peuvent être utilisées pour modifier cet ordre.

Une emplacement dans une expression est évalué à la valeur contenue dans l'emplacement.

Les invocations de fonctions dans les expressions sont discutées ci-dessus (voir «Invocation de fonction et retour»).

Les opérateurs arithmétiques (<plus_ou_moins>, <fois_div_mod> et le moins unaire) ont les significations et priorités habituelles, ainsi que les opérateurs relationnels **-a** et **-o**. L'opérateur **%** calcule le reste de la division de ses opérandes. La division / calcule la division entière de ses opérandes.

Les expressions avec opérateurs conditionnels **-a** et **-o** sont évaluées en court-circuit : Le deuxième opérande n'est pas évalué si le résultat du premier opérande détermine la valeur de toute l'expression, c'est-à-dire si le résultat est faux pour **-a** et vrai pour **-o**.

Priorités des opérateurs, de la plus forte à la plus faible :

Opérateurs	Commentaires
-	moins unaire
***** / **%**	multiplication, division, reste
+ -	addition, soustraction
	concaténation
-eq -ne -gt -ge -lt -le	relationnel
= **!=**	égalité
-n -z	chaîne non vide, vide
!	non logique
-a	conditionnel «et»
-o	conditionnel «ou»

1.10 Entrées/Sorties

echo

L'instruction **echo** permet d'afficher à l'écran la suite des chaînes de caractères correspondante aux arguments séparés par des espaces.

read

L'instruction **read** permet de lire au clavier une chaîne de caractères affectées à la variable **id** spécifiée en arguments.

1.11 Règles sémantiques

Ces règles s'ajoutent aux contraintes imposées par la grammaire du langage SoS. Un programme qui est grammaticalement bien formé et qui ne viole aucune des règles qui suivent est un programme légal. Un compilateur robuste doit vérifier explicitement chacune de ces règles, et doit générer un message d'erreur décrivant chaque violation qu'il est capable de détecter. Un compilateur robuste doit générer au moins un message d'erreur pour chaque programme illégal, et aucun message pour un programme légal.

1. Aucune fonction ou tableau ne doit être utilisé avant d'avoir été déclaré.

2. Le symbole terminal **entier** dans une déclaration de tableau doit être supérieur à 0.

3. Le nombre d'arguments dans un appel de fonction doivent être les mêmes que le nombre de paramètres utilisés dans la fonction (**$1**, **$2**, ...). Ce nombre d'arguments est fixe, pour tous les appels de la fonction.

4. Si un appel de fonction est utilisé dans une expression, la fonction doit retourner un résultat (via **echo**).

5. Pour tous les emplacements de la forme **id[<opérande_entier>]** :

   (a) **id** doit être une variable tableau, et

   (b) La valeur de <opérande_entier> doit être composé que de chiffres, sans préfixe - ou +. Dans le cas contraire, le compilateur, ou le programme en cas de valeurs d'opérandes inconnues à la compilation, provoquera une erreur.

6. Les valeurs des opérandes des opérateurs <plus_ou_moins>, <fois_div_mod> et moins unaire doivent être des chaînes composées uniquement de chiffres, éventuellement préfixés par - ou +. Dans le cas contraire, le compilateur, ou le programme en cas de valeurs d'opérandes inconnues à la compilation, provoquera une erreur.

1.12 Exemples de programme

```
# Exemple 1 de programme en SoS

declare valeurs[4] ;

lecture_valeurs() # Saisie des valeurs entières
{
  local i=0 ; local nombre = $1 ;
  echo "Il faut saisir " $1 "valeurs entieres\n" ;
  while test ${i} -lt ${nombre} do
    echo "Entrez la valeur " $( expr ${i} + 1 ) " : \n" ;
    read valeurs[${i}] ;
    if test ! ${valeurs[${i}]} -gt 0 then
      echo "Les valeurs doivent etre strictement positives !\n" ;
      return 1
    fi ;
    i = $( expr ${i} + 1 )
  done ;
  return 0
} ;

moyenne() { #  Calcul de la moyenne des valeurs
  local i = 0 ;
  local somme = 0 ;
  while test ${i} -lt $1 do
    somme = $( expr ${somme} + ${valeurs[${i}]} ) ;
    i = $( expr ${i} + 1 )
  done ;
  echo $( expr ${somme} / $1 )
} ;

echo "Nombre de valeurs : " ;
read nombre ;
lecture_valeurs ${nombre} ;
if test $? = 0 then
  echo "Moyenne = " $(moyenne ${nombre} ) "\n"
else
  echo "Erreur\n"
fi ;
exit
```

```
# Exemple 2 de programme en SoS

declare mots[100] ;

echo "Entrez le nombre de mots de la phrase : " ;
read nombre ;
i=1 ;
while test ${i} -le ${nombre} do
  echo "Entrez le mot numero " ${i} " : " ;
  read mots[${i}] ;
  i = $( expr ${i} + 1 )
done ;
new_phrase="" ;
echo ${mots[*]} ; # exemple : C'est Sabine qui gagne le concours (6 mots)
for mot in ${mots[*]} do
  case ${mot} in
    Anne|Corinne|Sabine|Emilie) new_phrase = ${new_phrase}" "elle ;;
    Pierre|Paul|Jacques|Albert) new_phrase = ${new_phrase}" "lui ;;
    *) new_phrase = ${new_phrase}" "${mot} ;;
  esac
done ;
echo "Nouvelle phrase : " ${new_phrase} ; # exemple : C'est elle qui gagne le concours
exit
```

2 Génération de code

Le code généré devra être en assembleur MIPS R2000 [1]. L'assembleur est décrit dans les documents fournis. Le code assembleur devra être exécuté à l'aide du simulateur de processeur MIPS *SPIM* [2] (il existe un package debian/ubuntu) ou *Mars* [3].

Le code assembleur généré devra notamment allouer dynamiquement de la mémoire pour stocker les données non statiques, telles que les chaînes de caractères obtenues en paramètres ou issues de concaténations. Pour cela, on utilise l'appel système `sbrk` en MIPS de la manière suivante :

```
li      $a0,xxx   # on met dans $a0 le nombre d'octets à allouer (xxx)
                  # ce nombre doit être un multiple de 4
li      $v0,9     # code 9 = allouer de la mémoire
syscall           # appel système
                  # $v0 <-- contient l'adresse du premier octet
                  # du bloc mémoire alloué dynamiquement
```

Attention, il n'y a pas de moyen avec les simulateurs *SPIM* et *Mars* pour libérer de la mémoire précédemment allouée. Ce projet autorise donc exceptionnellement à allouer de la mémoire sans jamais la libérer...

Pour les chaînes de caractères issues de paramètres ou de lecture au clavier (**read**), il sera utile de déterminer leur taille afin d'allouer l'espace mémoire nécessaire à leur stockage. Dans un premier temps, une zone mémoire tampon (buffer) de taille fixée maximale pourra être utilisée dans la zone de donnée statique pour réceptionner la chaîne de caractères. Ensuite, un dénombrement des caractères octet par octet, jusqu'à obtenir la valeur 0 de fin de chaîne, permettra d'obtenir la taille.

En MIPS, les paramètres de la ligne de commande donnés au lancement du programme sont accessibles via les registres $a0 et $a1. Le registre $a0 contiendra le nombre d'arguments (de manière similaire à argc en C), et les différents paramètres seront aux adresses 0($a1) (argv[1]), 4($a1) (argv[2]), etc.

D'une manière générale, il sera opportun de prévoir l'écriture d'une petite bibliothèque de fonctions utiles en MIPS, réalisant des manipulations de chaînes de caractères (comparaison de chaînes, concaténation et stockage dans le tas, taille, etc.).

1. https://pages.cs.wisc.edu/~larus/SPIM/spim_documentation.pdf
2. http://spimsimulator.sourceforge.net
3. http://courses.missouristate.edu/kenvollmar/mars

3 Aspects pratiques et techniques

Le compilateur devra être écrit en C à l'aide des outils Lex (flex) et Yacc (bison).

Ce travail est à réaliser en équipe composée de quatre étudiant.e.s dans le cadre du cours de *Compilation* et du cours de *Conduite de Projets*, et à rendre à la date indiquée par vos enseignants en cours et sur Moodle. Une démonstration finale de votre compilateur sera faite durant la dernière séance de TP. Vous devrez rendre sur Moodle dans une archive :

- Le code source de votre projet complet dont la compilation devra se faire simplement par la commande « make ». Le nom de l'exécutable produit doit être « sos »
- Un document détaillant les capacités de votre compilateur, c'est-à-dire ce qu'il sait faire ou non. Soyez honnêtes, indiquez bien les points intéressants que vous souhaitez que le correcteur prenne en compte car il ne pourra sans doute pas tout voir dans le code.
- Un jeu de tests.

Votre compilateur devra fournir les options suivantes :

- `-version` devra indiquer les membres du projet.
- `-tos` devra afficher la table des symboles.
- `-o <name>` devra écrire le code résultat dans le fichier `name`.

4 Recommandations importantes

Écrire un compilateur est un projet conséquent, il doit donc impérativement être construit incrémentalement en validant chaque étape sur un plus petit langage et en ajoutant progressivement des fonctionnalités ou optimisations. Une démarche extrême et totalement contre-productive consiste à écrire la totalité du code du compilateur en une fois, puis de passer au débogage ! Le résultat de cette démarche serait très probablement nul, c'est-à-dire un compilateur qui ne fonctionne pas du tout ou alors qui reste très bogué.

Par conséquent, nous vous conseillons de développer tout d'abord un compilateur *fonctionnel* mais *limité* à la traduction d'instructions simples. À partir d'une telle version fonctionnelle, il vous sera plus aisé de la faire évoluer en intégrant telle ou telle fonctionnalité, ou en considérant des instructions plus complexes, ou en intégrant telle ou telle structure de contrôle. De plus, même si votre compilateur ne remplira finalement pas tous les objectifs, il sera néanmoins capable de générer des programmes corrects et qui «marchent» !

5 Précisions concernant la notation

- Si votre projet ne compile pas ou plante directement, la note 0 (zéro) sera appliquée : l'évaluateur n'a absolument pas vocation à aller chercher ce qui pourrait éventuellement ressembler à quelque chose de correct dans votre code. Il faut que votre compilateur s'exécute et qu'il fasse quelque chose de correct, même si c'est peu.
- Si vous manquez de temps, préférez faire moins de choses mais en le faisant bien et de bout en bout : on préférera un compilateur incomplet mais qui génère un programme MIPS exécutable plutôt qu'une analyse syntaxique seule.
- Élaborez des tests car cela fait partie de votre travail et ce sera donc évalué.
- Faites les choses dans l'ordre et focalisez sur ce qui est demandé. L'évaluateur pourra tenir compte du travail fait en plus (par exemple des optimisations de code) seulement si ce qui a été demandé a été fait et bien fait.
- Une conception modulaire et lisible sera fortement appréciée (et inversement).