

Compilateur pour le langage COREC

But du projet : Le but du projet est de réaliser un compilateur pour le langage *COREC*, depuis le code de haut niveau jusqu'à un code exécutable MIPS ou RISC-V (au choix).

1 Description du langage COREC

Le langage COREC (pour « COMputed RECurrences ») est un « langage métier » (en anglais *Domain Specific Language*, DSL), dont l'objectif est de faciliter, pour des mathématiciens, la programmation de calculs effectués sur des tableaux de toutes dimensions, ces calculs étant définis par des relations de récurrence.

1.1 Structure d'un programme en COREC

On donne en figure 1 un exemple de programme qui implémente une simulation numérique de type «MADNESS». D'autres exemples de programmes en COREC sont fournis en annexe.

Un programme COREC débute par un entête utilisant le mot clé **prog** suivi du nom du programme, et d'accolades encadrant tout le reste du code source. Celui-ci comporte une suite de définitions de fonctions, débutant chacune avec le mot clé **def**, suivi d'un nom de fonction, et d'accolades encadrant le reste du code de la fonction. Il est obligatoire que le programme contienne une fonction dont le nom est *Main*. Celle-ci peut éventuellement être la seule fonction du programme.

Il est possible d'intégrer des commentaires dans le programme source, délimités par `/*` et `*/`, comme dans le langage C.

1.2 Les fonctions en COREC

Les fonctions, délimitées par **def** *nom* { ... }, sont structurées en 4 parties principales :

1. La déclaration des arguments d'entrée, de sortie et d'entrée/sortie.
2. La section *Loc* pour les déclarations des variables locales à la fonction.
3. La section *Dom* définissant les espaces de récurrences, qui sont des intervalles de valeurs d'indices.
4. La section *Rec* qui contient les calculs récurrents à effectuer.

Arguments des fonctions : La liste des arguments d'entrée est préfixée par le mot clé **in**, celle des arguments de sortie par le mot clé **out**, et celle des arguments d'entrée/sortie par le mot clé **inout**. Ces arguments peuvent être des tableaux ou des variables simples. Les tableaux sont implicitement de type flottant et leurs indices ont pour valeur minimum zéro, et les variables simples peuvent être des flottants ou des entiers. Dans le cas des variables simples de type flottant, celles-ci doivent être déclarées comme des tableaux à une dimension et de taille 1, de la forme (**id**,1). Toutefois, de tels arguments pourront ensuite être référencés simplement en utilisant le nom (**id**) ou la référence à l'élément de tableau (**id**[0]).

Tout argument dans la liste **out:** est implicitement déclaré par la fonction, et pourra apparaître ensuite dans une liste **in:** ou **inout:** d'une ou de plusieurs autres fonctions. Une des conséquences est que cet argument ne peut pas figurer dans la liste **out:** d'une autre fonction, et que sa portée est définie par les fonctions appelantes de la fonction qui le déclare. De plus, le même nom de variable ou de tableau doit être utilisé pour toutes les fonctions ayant accès à la donnée.

Une fonction ne peut accéder qu'à ses arguments, à ses variables locales, et aux arguments de sortie des fonctions qu'elle appelle.

```

/* Multiresolution ADaptive NumErical Scientific Simulation (MADNESS) */
prog doitgen {

  def Inita { [in: NR,NQ,NP; out: (a,NR,NQ,NP)]
    Dom: i in [0..NR-1], j in [0..NQ-1], k in [0..NP-1]
    Rec: a[i,j,k]=((i*j + k) % NP) / NP
  }

  def Initc4 { [in: NP; out: (c4,NP,NP)]
    Dom: i in [0..NP-1], j in [0..NP-1]
    Rec: c4[i,j]=(i*j % NP) / NP
  }

  def Eq1 { [in: (sum,NP), r,q,NR,NQ,NP; inout: (a,NR,NQ,NP)]
    Dom: k in [0..NP-1]
    Rec: a[r,q,k] = sum[k]
  }

  def Eq2 { [in: (c4,NP,NP), NR,NQ,NP; inout: (a,NR,NQ,NP)]
    Loc: r,q, /* déclaration de variables locales entières */
          (sum,NP) /* déclaration de tableau local */
    Dom: i in [0..NR-1], j in [0..NQ-1], k in [0..NP-1], l in [0..NP-1]
    Rec: {
      l==0?sum[k]=0: ;
      sum[k] += a[i,j,l] * c4[l,k];
      k==NP-1?l==NP-1?
      {
        r=i;
        q=j;
        Eq1() /* appel de fonction dans une fonction */
      }::
    }
  }

  def Main {
    Loc: NR,NQ,NP
    Dom:
    Rec : {
      printstr("Entrez une valeur entière NR : ");
      read(NR);
      printstr("Entrez une valeur entière NQ : ");
      read(NQ);
      printstr("Entrez une valeur entière NP : ");
      read(NP);
      printstr("Le résultat est : ");
      print(Eq2(Initc4(),Inita()))
    }
  }
}

```

FIGURE 1 – Exemple de code source en langage COREC

Une fonction ne peut appeler une autre fonction que si elle peut lui fournir ses arguments **in :** et **inout :**. Ainsi, une suite d'appels de fonctions s'effectue obligatoirement en respectant la chaîne de dépendances entre les fonctions. Par exemple, dans le programme de la figure 1, l'instruction `Eq2(Initc4(), Inita())` représente les exécutions des fonctions `Initc4` et `Inita` dans un ordre quelconque, suivies par l'exécution de la fonction `Eq2`. Cet ordre d'exécution respecte les dépendances car la fonction `Initc4` et `Inita` fournissent en sortie les tableaux `c4` et `a` respectivement, qui sont utilisés respectivement en entrée et en entrée/sortie de la fonction `Eq2`. De plus, la fonction appelante `Main` déclare les 3 variables locales `NR`, `NQ` et `NP` également référencées en entrée des fonctions appelées. La fonction `Main` n'a accès qu'à l'argument d'entrée/sortie de la fonction `Eq2`, à savoir le tableau `a`.

La valeur retournée d'une fonction est l'ensemble de ses arguments de sortie et d'entrée/sortie. Cependant, cette valeur ne peut être affectée à une variable simple que si la fonction possède un seul argument de sortie ou d'entrée/sortie qui est une variable simple. Il n'y a pas d'affectation directe, en une seule instruction, entre tableaux de taille et de dimension supérieures à 1.

La fonction principale `Main` est particulière et ne possède pas d'arguments¹.

La section *Loc* : Toute fonction peut déclarer des variables locales, qui peuvent être des tableaux ou des variables simples. Comme pour les arguments de fonction, les tableaux sont implicitement de type flottant avec des indices commençant à zéro, et les variables simples peuvent être des flottants ou des entiers. Cependant, les variables locales simples de type flottant doivent être déclarées comme des tableaux à une dimension et de taille 1, comme pour les arguments simples de type flottant. Ils pourront ensuite également être référencés simplement par leur nom, où par la référence au seul élément du tableau. Toute variable locale pourra être référencée comme arguments d'entrée, ou d'entrée/sortie, par les fonctions appelées par cette fonction.

La section *Dom* : Cette section permet de définir l'espace de récurrence, c'est-à-dire les valeurs des indices pour lesquelles les calculs définis dans la section *Rec* devront être exécutés. De plus, l'ordre dans lequel les intervalles de valeurs pour chaque indice sont déclarés définit l'ordre d'exécution des calculs de la section *Rec*. Par exemple, pour le programme de la figure 1 et sa fonction `Inita`, la section *Dom* :

Dom: i in [0..NR-1], j in [0..NQ-1], k in [0..NP-1]

implique un ordre d'exécution du calcul `a[i,j,k]=((i*j + k) % NP) / NP`, comme si ce calcul était situé à l'intérieur de 3 boucles imbriquées de la forme :

```
for (i=0; i<NR; i++)
  for (j=0; j<NQ; j++)
    for (k=0; k<NP; k++)
      a[i,j,k]=((i*j + k) % NP) / NP;
```

Les bornes dans les intervalles peuvent être des constantes entières, ou des variables entières. Elles peuvent également être des indices dont les intervalles ont été définis précédemment dans l'ordre des définitions des intervalles. De plus, ces bornes peuvent être des expressions arithmétiques utilisant ces éléments, constantes, variables ou indices, comme dans : Dom: i in [0..N-1], k in [0..i-2] (voir `SystTriang.corec`).

La section *Rec* : Cette section contient les instructions de la fonction qui sont soit des affectations, soit des instructions conditionnelles, soit des appels de fonction. Ces instructions accèdent uniquement aux arguments de la fonction, aux variables locales ou aux arguments de sortie des fonctions appelées précédemment à l'instruction.

Les **appels de fonctions** peuvent prendre deux syntaxes différentes, qui peuvent être combinées :

— L'écriture fonctionnelle, de la forme :

$$fonc(fonc_1, func_2, \dots, func_i(fonc_{i1}, func_{i2}, \dots, func_{ij}(\dots), \dots), \dots)$$

1. On peut aisément imaginer une extension du langage COREC où la fonction `Main` possède des arguments issus de la ligne de commande.

- L'écriture par références relatives d'appels, de forme :

```

fonc11()
fonc12()
fonc1(%2,%1)
fonc(%1)

```

où %1 représente l'appel de fonction précédent, %2 représente le deuxième appel précédent dans l'ordre inverse des appels, etc. L'écriture fonctionnelle (à peu près) équivalente de cet exemple est : `fonc(fonc1(fonc11, fonc12))`

- L'écriture combinée, où l'exemple précédent pourrait être écrit de la manière suivante :

```

fonc11()
fonc12()
fonc(fonc1(%2,%1))

```

L'ordre des appels doit **respecter les dépendances de données entre les appels**. Dans l'exemple précédent, `fonc1` doit impérativement préciser de quelles fonctions, appelées précédemment, elle utilise les résultats en entrée. En effet, on peut imaginer qu'un autre appel, produisant des sorties compatibles, a également pu être exécuté précédemment.

Dans le cas d'une écriture fonctionnelle des appels, la fonction appelante n'aura accès qu'aux sorties de la fonction appelée en dernier. Lors d'appels successifs par références relatives d'appels, chaque sortie des fonctions appelées devient accessible pour l'appelant.

De plus, un appel de fonction peut être affecté à une variable simple (`var = fonc...`), si et seulement si la fonction possède un seul argument de sortie, ou d'entrée/sortie, qui est aussi une variable simple. Comme indiqué plus haut, il n'y a pas de possibilité d'affectation directe, en une seule instruction, entre tableaux de dimension et de taille supérieures à 1.

Il est possible d'écrire des **fonctions récursives** (voir les exemples `Fact.corec` et `Fibonaci.corec`).

On dispose de **3 fonctions prédéfinies** : `read`, `print` et `printstr` :

- La fonction `read`, qui prend le nom d'une variable simple en argument, permet de demander la saisie au clavier d'une valeur entière ou réelle, selon le type de la variable en argument.
- La fonction `print` prend en argument soit le nom d'une variable simple, soit un tableau, soit un appel de fonction. Elle permet d'afficher à l'écran des valeurs réelles ou entières, selon le type de l'argument. Le format de sortie s'adapte automatiquement aux dimensions de l'argument (tableau). De plus, dans le cas d'un argument qui est un appel de fonction, et que cette fonction produit plusieurs sorties, toutes les sorties sont affichées dans l'ordre de leurs déclarations en tant qu'arguments.

On appliquera le formatage suivant lors de l'affichage du contenu de tableaux, quelque soient les dimensions : les éléments seront affichés l'un après l'autre, dans l'ordre des dimensions (ligne par ligne en dimension 2), séparés par un espace, avec 2 chiffres après la virgule (format `%0.2f` en C), et avec un affichage de 20 éléments par ligne. Ainsi, l'affichage du résultat du programme en figure 1, pour `NQ=8`, `NR=10` et `NP=12` devra être :

```

0.00 3.51 2.78 2.38 2.06 2.74 1.50 2.76 1.94 2.12 2.22 1.99 0.00 3.51 2.78 2.38 2.06 2.74 1.50 2.76
1.94 2.12 2.22 1.99 0.00 3.51 2.78 2.38 2.06 2.74 1.50 2.76 1.94 2.12 2.22 1.99 0.00 3.51 2.78 2.38
2.06 2.74 1.50 2.76 1.94 2.12 2.22 1.99 0.00 3.51 2.78 2.38 2.06 2.74 1.50 2.76 1.94 2.12 2.22 1.99
0.00 3.51 2.78 2.38 2.06 2.74 1.50 2.76 1.94 2.12 2.22 1.99 0.00 3.51 2.78 2.38 2.06 2.74 1.50 2.76
1.94 2.12 2.22 1.99 0.00 3.51 2.78 2.38 2.06 2.74 1.50 2.76 1.94 2.12 2.22 1.99 0.00 3.51 2.78 2.38
2.06 2.74 1.50 2.76 1.94 2.12 2.22 1.99 0.00 3.06 2.36 2.00 1.72 2.61 1.25 2.81 1.94 2.25 2.47 2.36
0.00 2.68 2.11 1.88 1.72 2.90 1.50 2.43 1.61 2.12 2.56 2.65 0.00 2.39 2.03 2.00 2.06 2.61 1.25 2.64
1.94 1.75 2.47 2.86 0.00 2.18 2.11 2.38 1.72 2.74 1.50 2.43 1.94 2.13 2.22 2.99 0.00 2.06 2.36 2.00
1.72 2.28 1.25 2.81 1.61 2.25 1.81 3.03 0.00 2.01 2.78 1.88 2.06 2.24 1.50 2.76 1.94 2.12 2.22 2.99
0.00 2.06 2.36 2.00 1.72 2.61 1.25 2.31 1.94 1.75 2.47 2.86 0.00 3.51 2.78 2.38 2.06 2.74 1.50 2.76
1.94 2.12 2.22 1.99 0.00 2.68 2.11 1.88 1.72 2.90 1.50 2.43 1.61 2.12 2.56 2.65 0.00 2.39 2.03 2.00
1.72 2.74 1.50 2.43 1.94 2.13 2.22 2.99 0.00 2.01 2.78 1.88 2.06 2.24 1.50 2.76 1.94 2.12 2.22 2.99
0.00 2.18 2.11 2.38 1.72 2.40 1.50 2.43 1.61 2.12 2.56 2.65 0.00 2.68 2.11 1.88 1.72 2.24 1.50 2.43
1.94 2.12 2.22 1.99 0.00 3.51 2.78 2.38 2.06 2.74 1.50 2.76 1.94 2.12 2.22 1.99 0.00 2.68 2.11 1.88
1.72 2.90 1.50 2.43 1.61 2.12 2.56 2.65 0.00 3.51 2.78 2.38 2.06 2.74 1.50 2.76 1.94 2.12 2.22 1.99
0.00 2.39 2.03 2.00 2.06 2.61 1.25 2.64 1.94 1.75 2.47 2.86 0.00 2.01 2.78 1.88 2.06 2.24 1.50 2.76
1.94 2.12 2.22 2.99 0.00 2.39 2.03 2.00 2.06 2.61 1.25 2.14 1.94 2.25 2.47 2.36 0.00 3.51 2.78 2.38
2.06 2.74 1.50 2.76 1.94 2.12 2.22 1.99 0.00 2.39 2.03 2.00 2.06 2.61 1.25 2.64 1.94 1.75 2.47 2.86
0.00 2.01 2.78 1.88 2.06 2.24 1.50 2.76 1.94 2.12 2.22 2.99 0.00 2.39 2.03 2.00 2.06 2.61 1.25 2.14
1.94 2.25 2.47 2.36 0.00 3.51 2.78 2.38 2.06 2.74 1.50 2.76 1.94 2.12 2.22 1.99 0.00 2.18 2.11 2.38
1.72 2.74 1.50 2.43 1.94 2.13 2.22 2.99 0.00 2.18 2.11 2.38 1.72 2.40 1.50 2.43 1.61 2.12 2.56 2.65
0.00 3.51 2.78 2.38 2.06 2.74 1.50 2.76 1.94 2.12 2.22 1.99 0.00 2.18 2.11 2.38 1.72 2.74 1.50 2.43

```

```

1.94 2.13 2.22 2.99 0.00 2.18 2.11 2.38 1.72 2.40 1.50 2.43 1.61 2.12 2.56 2.65 0.00 3.51 2.78 2.38
2.06 2.74 1.50 2.76 1.94 2.12 2.22 1.99 0.00 2.18 2.11 2.38 1.72 2.74 1.50 2.43 1.94 2.13 2.22 2.99
0.00 3.51 2.78 2.38 2.06 2.74 1.50 2.76 1.94 2.12 2.22 1.99 0.00 2.06 2.36 2.00 1.72 2.28 1.25 2.81
1.61 2.25 1.81 3.03 0.00 2.68 2.11 1.88 1.72 2.24 1.50 2.43 1.94 2.12 2.22 1.99 0.00 2.39 2.03 2.00
2.06 2.61 1.25 2.64 1.94 1.75 2.47 2.86 0.00 2.18 2.11 2.38 1.72 2.40 1.50 2.43 1.61 2.12 2.56 2.65
0.00 3.06 2.36 2.00 1.72 2.61 1.25 2.81 1.94 2.25 2.47 2.36 0.00 2.01 2.78 1.88 2.06 2.24 1.50 2.76
1.94 2.12 2.22 2.99 0.00 3.06 2.36 2.00 1.72 2.28 1.25 2.31 1.61 1.75 1.81 1.53 0.00 3.51 2.78 2.38
2.06 2.74 1.50 2.76 1.94 2.12 2.22 1.99 0.00 2.01 2.78 1.88 2.06 2.24 1.50 2.76 1.94 2.12 2.22 2.99
0.00 3.51 2.78 2.38 2.06 2.74 1.50 2.76 1.94 2.12 2.22 1.99 0.00 2.01 2.78 1.88 2.06 2.24 1.50 2.76
1.94 2.12 2.22 2.99 0.00 3.51 2.78 2.38 2.06 2.74 1.50 2.76 1.94 2.12 2.22 1.99 0.00 2.01 2.78 1.88
2.06 2.24 1.50 2.76 1.94 2.12 2.22 2.99 0.00 3.51 2.78 2.38 2.06 2.74 1.50 2.76 1.94 2.12 2.22 1.99
0.00 2.01 2.78 1.88 2.06 2.24 1.50 2.76 1.94 2.12 2.22 2.99 0.00 3.51 2.78 2.38 2.06 2.74 1.50 2.76
1.94 2.12 2.22 1.99 0.00 2.06 2.36 2.00 1.72 2.61 1.25 2.31 1.94 1.75 2.47 2.86 0.00 2.68 2.11 1.88
1.72 2.90 1.50 2.43 1.61 2.12 2.56 2.65 0.00 2.39 2.03 2.00 2.06 2.61 1.25 2.14 1.94 2.25 2.47 2.36
0.00 2.18 2.11 2.38 1.72 2.74 1.50 2.43 1.94 2.13 2.22 2.99 0.00 3.06 2.36 2.00 1.72 2.28 1.25 2.31
1.61 1.75 1.81 1.53 0.00 2.01 2.78 1.88 2.06 2.24 1.50 2.76 1.94 2.12 2.22 2.99 0.00 3.06 2.36 2.00
1.72 2.61 1.25 2.81 1.94 2.25 2.47 2.36 0.00 3.51 2.78 2.38 2.06 2.74 1.50 2.76 1.94 2.12 2.22 1.99
0.00 2.18 2.11 2.38 1.72 2.40 1.50 2.43 1.61 2.12 2.56 2.65 0.00 2.18 2.11 2.38 1.72 2.74 1.50 2.43
1.94 2.13 2.22 2.99 0.00 3.51 2.78 2.38 2.06 2.74 1.50 2.76 1.94 2.12 2.22 1.99 0.00 2.18 2.11 2.38
1.72 2.40 1.50 2.43 1.61 2.12 2.56 2.65 0.00 2.18 2.11 2.38 1.72 2.74 1.50 2.43 1.94 2.13 2.22 2.99
0.00 3.51 2.78 2.38 2.06 2.74 1.50 2.76 1.94 2.12 2.22 1.99 0.00 2.18 2.11 2.38 1.72 2.40 1.50 2.43
1.61 2.12 2.56 2.65 0.00 3.51 2.78 2.38 2.06 2.74 1.50 2.76 1.94 2.12 2.22 1.99 0.00 2.39 2.03 2.00
2.06 2.61 1.25 2.14 1.94 2.25 2.47 2.36 0.00 2.01 2.78 1.88 2.06 2.24 1.50 2.76 1.94 2.12 2.22 2.99
0.00 2.39 2.03 2.00 2.06 2.61 1.25 2.64 1.94 1.75 2.47 2.86 0.00 3.51 2.78 2.38 2.06 2.74 1.50 2.76
1.94 2.12 2.22 1.99 0.00 2.39 2.03 2.00 2.06 2.61 1.25 2.14 1.94 2.25 2.47 2.36 0.00 2.01 2.78 1.88
2.06 2.24 1.50 2.76 1.94 2.12 2.22 2.99 0.00 2.39 2.03 2.00 2.06 2.61 1.25 2.64 1.94 1.75 2.47 2.86

```

— La fonction *printstr* affiche la chaîne caractères donnée en argument.

Les **instructions conditionnelles** sont de la forme :

condition?instructions_si_vrai : instructions_si_faux

La condition se limite à une expression de comparaison de deux opérandes avec un opérateur relationnel. Les opérandes peuvent être des expressions arithmétiques, des constantes, des variables simples ou des éléments de tableau. Il n'y a pas d'opérateurs logiques. Ainsi, un «et» pourra être mis en œuvre par une imbrication de conditions (voir `k==NP?l==NP?` figure 1), et un «ou» par une succession de lignes d'instructions conditionnelles (voir `floyd-warshall.corec`).

1.2.1 Grammaire du langage COREC

Les symboles entre quotes (' ') sont des symboles terminaux du langage.

```

PROG -> 'prog' ID '{' LFONC 'def' 'Main' '{' LOC DOM REC '}' '}'

LFONC -> LFONC FONC | epsilon

FONC -> 'def' ID '{' ARGS LOC DOM REC '}'

LOC -> 'Loc:' LDECL | 'Loc:' | epsilon

LDECL -> LDECL ',' DECL | DECL

DECL -> ID | ID '=' E | ARRAY

ARRAY -> '(' ID ',' DLIST ')'

DLIST -> DLIST ',' E | E

ARGS -> '[' SECTIONARGS LARGS ']' | '[' ']'

LARGS -> LARGS ';' SECTIONARGS | epsilon

SECTIONARGS -> 'in:' ARGLIST | 'out:' ARGLIST | 'inout:' ARGLIST

ARGLIST -> ARGLIST ',' ELTARG | ELTARG

ELTARG -> ARRAY | ID

DOM -> 'Dom:' DOMLIST | 'Dom:' | epsilon

```

```

DOMLIST -> DOMLIST ',' D | D

D -> ID 'in' '[' E '..' E ']'

REC -> 'Rec:' BLOCKINST

BLOCKINST -> '{' LISTEI '}' | I

LISTEI -> LISTEI ';' I | I

I -> COND
| ARRAYREF OP '=' E
| ID OP '=' E
| CALL // appel de fonction
| 'read' '(' ID ')'
| 'print' '(' ID ')'
| 'print' '(' CALL ')'
| 'print' '(' PRECINST ')'
| 'printstr' '(' CHAINE ')'

OP -> '+' | '-' | '*' | '/' | epsilon

CALL -> ID '(' LCALL ')' | ID '(' ' ' ')'

LCALL -> LCALL ',' CALL | LCALL ',' PRECINST | CALL | PRECINST

PRECINST -> '%' ENTIER

ARRAYREF -> ID '[' ELIST ']'

ELIST -> ELIST ',' E | E

E -> E '+' T | E '-' T | E '-' E | T
T -> T '*' F | T '/' F | T '%' F | F
F -> '(' E ')' | ID | CALL | ARRAYREF | ENTIER | FLOTTANT

COND -> E OPREL E '?' B ':' B

B -> BLOCKINST | epsilon

OPREL -> '<' | '>' | '<=' | '>=' | '=='

```

2 Détection des erreurs

On désire apporter un soin particulier à la détection des erreurs de programmation, en affichant des messages les plus explicites possibles. Les erreurs devant être détectées impérativement sont :

- Dépendances non respectées entre appels successifs de fonctions.
- Utilisations d'indices dans les bornes des intervalles, pour lesquels les intervalles ne sont pas définis au préalable dans la même section *Dom*.
- Utilisations de variables ou de tableaux non accessibles pour la fonction.
- Utilisations de mêmes noms de variables ou de tableaux dans plusieurs arguments de sortie de fonctions différentes, ou dans des variables locales et des arguments de la même fonction.

2.1 Assembleur MIPS ou RISC-V

Le code généré devra être en assembleur MIPS R2000² ou RISC-V³, au choix. L'assembleur est décrit dans les documents fournis. Le code assembleur devra être exécuté à l'aide du simulateur de processeur MIPS *SPIM*⁴ (il existe un package debian/ubuntu) ou *Mars*⁵, ou du simulateur de processeur RISC-V

2. https://pages.cs.wisc.edu/~larus/SPIM/spim_documentation.pdf

3. <https://github.com/TheThirdOne/rars>

4. <http://spimsimulator.sourceforge.net>

5. <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, 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 « corec »
- 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 en assembleur 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 » !

Il est impératif d'implémenter la fonction `print`, afin de pouvoir vérifier que vos programmes s'exécutent correctement et produisent des résultats justes !

5 Précisions concernant la notation

- Si votre projet **ne compile pas, ou plante directement, ou ne génère aucun code MIPS ou RISC-V correct, 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 : **même s'il est incomplet, votre compilateur doit pouvoir générer des programmes MIPS ou RISC-V exécutables**.
- 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.

6. https://github.com/TheThirdOne/rars/releases/download/v1.6/rars1_6.jar