Projet de Compilation des Langages
PCL2
2A TELECOM Nancy - 2022-2023

Rapport d'activité final

Génération de code

Membres de l'équipe projet :
BÉNÉ Tom, RAPS Hugo, RICARD Guillaume, SIMON Damien

Professeurs référents :
COLLIN Suzanne / DA SILVA Sébastien

Table des matières

1 Schémas de traduction - choix faits

Pour la partie génération de code du projet de compilation, nous avons choisi le langage ARM32 comme langage cible de la traduction du code tiger.
Nous utiliserons le processeur d'une Raspberry pi pour la compilation et l'exécution du code ARM généré par notre compilateur.
Le programme est découpé en 2 sections :

— La section `.data` dans laquelle on stocke les données connues à la compilation comme le display et certaines chaînes de caractères

— La section `.text` dans laquelle le programme ARM32 sera écrit

1.1 Choix d'implémentation

1.1.1 Utilisation des registres

R0	Registre de travail
R2	Registre de travail
R3	Registre de travail
R4	Registre de travail
R5	Registre de travail
R6	Registre de travail
R7	Registre d'appel système
R8	Stocke le dernier résultat calculé dans le programme,aussi utilisé pour les retours de fonctions
R9	Registre tampon pour les opérations arithmétiques
R10	Display pointeur
R11	Base pointeur
R12	Chaînage statique
R13	Stack pointeur
R14	LR
R15	PC

1.1.2 Retour de fonction

Le retour des fonctions se fait via le registre R8. Cela implique donc que tous les types doivent tenir dans 4 octets.

1.1.3 Représentation des types de données

1.1.4 `int`

Les entiers sont des entiers signés codés sur 32 bits en complément à 2. Cela implique que la valeur d'un `int` est comprise dans l'intervalle $[2^{31}, 2^{31} - 1] = [-2147483648, 2147483647]$

1.1.5 Chaînes de caractères

Une chaîne de caractères est représentée par son adresse mémoire (codée sur 32 bits) et elle se termine par un null byte \0.
Elles sont directement dans la section `.data` du code assembleur lorsqu'elles sont explicitement déclarées dans le programme. Sinon, dans le cas ou elles sont le résultat de fonctions telles que `getchar` ou `chr`, elle sont réservé dans le tas grâce à la fonction `malloc`.
Dans notre implémentation du langage `tiger` les chaînes de caractères sont des données non mutables.

1.1.6 Arrays et records

On utilise la même représentation pour les arrays et records de taille N :
Une adresse mémoire codée sur 32 bits va pointer vers un espace mémoire de taille $4 * (N + 1)$ octets, où le premier octet sert à stocker la valeur de N et les autres stockent les valeurs des éléments.

L'allocation dynamique de mémoire s'effectue avec la fonction `malloc`.

1.1.7 Chaînage statique

Le chaînage statique s'effectue via un display, dont l'adresse est chargée au début du programme dans le registre R10.
Le display est mis à jour au début de chaque scope, c'est-à-dire au début des déclaration de fonctions et des `Let in`. Il est ensuite rétabli à la fin de ces scopes.
Le chaînage statique permet à n'importe quel scope d'accéder aux valeurs des variables des scopes précédents en lisant directement la case du display correspondant au numéro d'imbrication du scope recherché. Ainsi pour charger dans R8 la variable de déplacement D situé dans le scope d'imbrication N, on génère le code assembleur suivant :

```
1     LDR     R12,[R10,#<N>*4]
2     STR     R8,[R12,#<D>*-4]
```

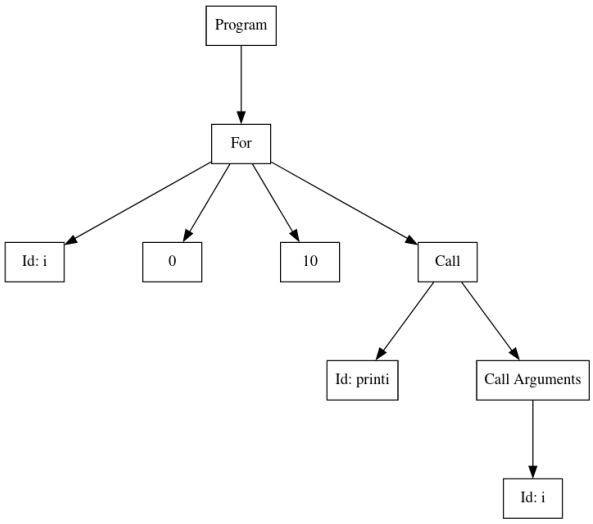
1.2 Desugaring

Le *desugaring* est une étape de la compilation permettant de transformer l'arbre syntaxique abstrait afin de faciliter l'étape de génération de code. Deux structures de l'AST sont modifiées lors de cette étape : les boucles `for` et les accès aux champs d'un type record.

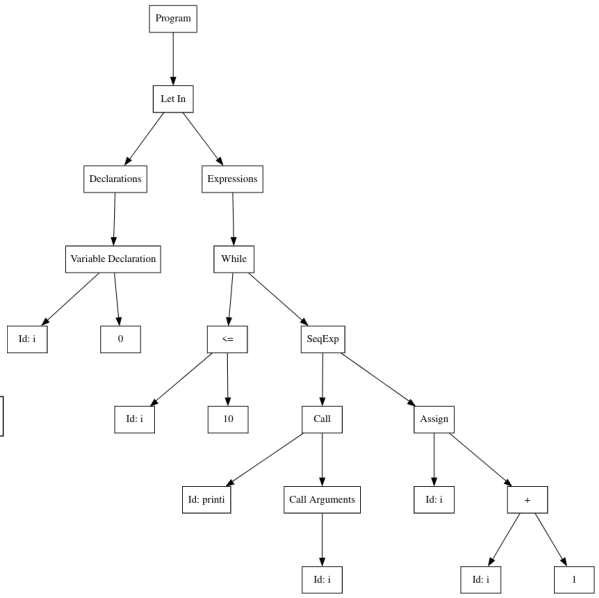
1.2.1 Boucles `for`

Afin de faciliter la génération de code, toutes les boucles `for` du programme sont transformées en boucles `while`. Pour cela, il faut :

1. Déclarer la variable d'itération avant la nouvelle boucle
2. Initialiser cette variable comme dans la boucle d'origine
3. Générer la condition de sortie de la nouvelle boucle
4. Générer une expression pour incrémenter la variable de boucle
5. Générer le corps de la nouvelle boucle, qui correspond à une séquence composée du corps de la boucle for et de l'incrémentation de la variable



(a) Boucle d'origine



(b) Boucle après desugaring

1.2.2 Accès aux records

Les records et les tableaux étant représentés de la même façon en mémoire, tous les accès à des champs d'un record sont transformés en accès par index à la donnée.

Les champs des records sont stockés dans la TDS grâce à une `LinkedHashMap`, ce qui permet de conserver d'ordre d'ajout des champs. Il est donc possible de récupérer l'index d'un champ pour un record et transformer les `FieldExp` en `Subscript`

1.3 Fonctions

Les fonction sont toutes déclarées à la fin de la section `.text` et identifiées par un label unique de la forme : `<NOM>_<SCOPE_ID>`
Où `SCOPE_ID` correspond à l'identifiant dans la table des symboles du scope où est déclaré la fonction. On peut ainsi permettre la redéclaration de fonction sur des scopes différents.

1.3.1 Déclaration de fonctions

La déclaration de fonction correspond à l'ouverture d'un nouveau scope.
Il faut donc empiler le chaînage statique, le chaînage dynamique et l'adresse de retour de la fonction, puis mettre à jour le base pointer ainsi que le display. Les valeurs empilées sont ensuite rétablies à la fin de la déclaration de la fonction, et l'adresse de retour est inscrite dans PC.

Ainsi pour la déclaration d'une fonction `NOM`, de numéro d'imbrication N dans un scope d'identifiant `SCOPE_ID` dont le corps est `EXP` Le schéma de traduction est le suivant :

```
<NOM>_<SCOPE_ID>:
    PUSH    {R11,LR}
    MOVl    R11,R13
    LDR     R12,[R10,\#<N>*4]
    PUSH    {R12}
    STR     R11,[R10,\#<N>*4]
    GENERECODE(<EXP>)
    POP     {R12}
    STR     R12,[R10,\#<N>*4]
    MOVl    R13,R11
    POP     {R11,PC}
```

1.3.2 Appel de fonctions

L'appel d'une fonction s'effectue grâce à l'instruction ARM `BL` qui correspond à un branchement qui met à jour le registre `LR` en lui donnant l'adresse dans le code de la prochaine instruction à exécuter. `LR` sera ensuite sauvegardé dans la pile dans le code de déclaration de la fonction.
Pour effectuer le branchement, il faut donc connaître le nom de la fonction ainsi que l'identifiant du scope dans lequel elle est déclarée. On récupère ensuite ses données à l'aide de la TDS.
Avant d'effectuer le branchement vers la fonction, l'appelant va d'abord empiler la valeur de ses arguments.
Une fois l'appel effectué ceux ci devront être dépilés en incrémentant R13. Ainsi pour l'appel d'une fonction `NOM`, déclaré dans un scope d'identifiant `SCOPE_ID` dont les valeurs des arguments sont `EXP1,EXP2,EXP3,EXP4`
Le schéma de traduction est le suivant :

```
    GENERECODE(<EXP1>)
    PUSH    {R8}
    GENERECODE(<EXP2>)
    PUSH    {R8}
    GENERECODE(<EXP3>)
    PUSH    {R8}
    GENERECODE(<EXP4>)
    PUSH    {R8}
    BL      <NOM>_<SCOPE_ID>
    ADD     R13,R13,#4*4
```

1.4 Structures de contrôle

On utilise des labels uniques, pour ce faire chaque structure de contrôle possède un identifiant unique noté `ID`.

1.4.1 Boucle `while`

Une boucle `while` comporte un label de début et un label de fin
ces labels sont de la forme :

- `_LOOP_<ID>`
- `_END_LOOP_<ID>`

Ainsi la génération du code tiger :
`while <COND> do <EXP>` donne :

```
_LOOP_<ID>:
    GENERECODE(<COND>)
    CMP     R8,#0
    BEQ     _END_LOOP_<ID>
```

```
    GENERECODE(<EXP>)
    B       _LOOP_<ID>
```

1.4.2 `if then else`

La structure de contrôle `if then else` comporte 3 labels :

- `_IF_<ID>`
- `_ELSE_<ID>`
- `_END_IF_<ID>`

Le schéma de traduction du code tiger :
`if <COND> then <EXP1> else <EXP2>` est :

```
    GENERECODE(<COND>)
    CMP     R8,#0
    BNE     _IF_<ID>
    BEQ     _ELSE_<ID>
_IF_<ID> :
    GENERECODE(<EXP1>)
    B       _END_IF_<ID>
_ELSE_<ID> :
    GENERECODE(<EXP2>)
    B       _END_IF_<ID>
_END_IF_<ID> :
```

On assume ici qu'une expression `if then` classique suit le même schéma avec `GENERE(<EXP2>)` qui ne génère rien.

1.4.3 Logical `AND` et Logical `OR`

Évaluation fainéante en utilisant un label `skip` :

- `_OR_SKIP_<ID>` pour le OU logique
- `_AND_SKIP_<ID>` pour le ET logique

Les schémas de traduction pour les codes tiger suivants :
`<EXP1> | <EXP2>`
`<EXP1> & <EXP2>`
sont respectivement :

```
    GENERECODE(<EXP1>)
    CMP     R8,#0
    BNE     _OR_SKIP_<ID>
    GENERECODE(<EXP2>)
_OR_SKIP_<ID>:
    GENERECODE(<EXP1>)
    CMP     R8,#0
    BEQ     _AND_SKIP_<ID>
    GENERECODE(<EXP2>)
_AND_SKIP_<ID>:
```

1.5 Opérateurs infixes

On s'intéresse ici au fonctionnement des opérateurs infix autre que le ET et le OU logique : c'est-à-dire les opérations arithmétiques et les opérations de comparaison.
Dans le cadre des ces opération, on utilisera la pile ainsi que les 2 registres R8,R9 pour stocker les valeurs calculées.
Ainsi les schéma de génération de code des opérations infix de la forme
`<EXP1> op <EXP2>`
commenceront toujours par :

```
1    GENERECODE(<EXP2>)
2    PUSH    {R8}
3    GENERECODE(<EXP1>)
4    POP     {R9}
```

Les valeurs des opérandes gauches et droits sont ainsi respectivement stockées dans les registres R8 et R9

1.5.1 Opérations arithmetiques

On utilise simplement les instructions de base disponibles en ARM32 :

— Addition :
```
ADD    R8,R8,R9
```

— Soustraction :
```
SUB    R8,R8,R9
```

— Multiplication :
```
MUL    R8,R8,R9
```

— Division :
```
SDIV    R8,R8,R9
```

1.5.2 Operation de comparaison

Pour les opérations de comparaison on fait une disjonction de cas pour traiter les comparaisons d'entier et les comparaisons de chaines de caracteres.
Dans le cas de comparaisons entre des entier on génère simplement le code suivant :
```
CMP    R8,R9
```
Mais si la comparaison s'effectue sur des chaînes de caractères, on devra faire appel à la fonction `strcmp` dont le code est présent dans le fichier `Base_function.s`.
Cette fonction prend en argument deux chaînes de caractères `s1` et `s2` et retourne un entier qui prend les valeurs suivantes :

— 1 si `s1` > `s2`

— 0 si `s1` = `s2`

— -1 si `s1` < `s2`

Dans ce cas le schéma de génération de code est le suivant :

```
1    PUSH    {R8,R9}
2    BL      strcmp
3    ADD     R13,R13,#8
4    CMP     R8,#0
```

La suite du schéma est identique pour le cas des entiers ou des chaînes de caractères et est propre à chaque opération de comparaison.
On utilise pour ça la valeur des flags d'activation après leur mise à jour par l'instruction CMP

— $<$:

```
1    MOVLT   R8,#1
2    MOVGE   R8,#0
```

— $>$:

```
1    MOVGT   R8,#1
2    MOVLE   R8,#0
```

— $<=$:

```
1    MOVLE   R8,#1
2    MOVGT   R8,#0
```

— $>=$:

```
1    MOVGE   R8,#1
2    MOVLT   R8,#0
```

— $=$:

```
1    MOVEQ R8,#1
2    MOVNE R8,#0
```

— $<>$:

```
1    MOVNE R8,#1
2    MOVEQ R8,#0
```

1.6 Contrôles sémantiques à l'exécution

Les contrôles sémantiques à l'exécution sont implémentés directement en ARM.
Ils consistent essentiellement en un branchement conditionnel vers une zone du programme qui va :

1. afficher un message d'erreur dans `stder` grâce à la fonction `printer`
2. finir le programme avec 1 en sortie avec la fonction `exit`

1.6.1 Division par 0

Lors de la visite d'une division, on génère une instruction de comparaison de R9 avec 0 qui permet d'aller ou non dans la branche d'erreur.

1.6.2 Accès à un indice *out of range*

L'erreur peut avoir lieu lors d'un assignement ou dans un subscript. On effectue la comparaison de l'index avec la taille du tableau stocké dans la première case mémoire pour le cas du dépassement de taille et avec 0 dans le cas où on spécifierait un indice négatif.

1.6.3 Accès au champ d'un *nil*

L'erreur peut se présenter dans un assignement ou un subscript. On effectue une comparaison de l'adresse avec 0

1.7 Fonctions de base

Toutes les fonctions de bases sont définient dans un fichier Base_function.s include au début du programme.
On utilse

1.7.1 Depuis le manuel tiger

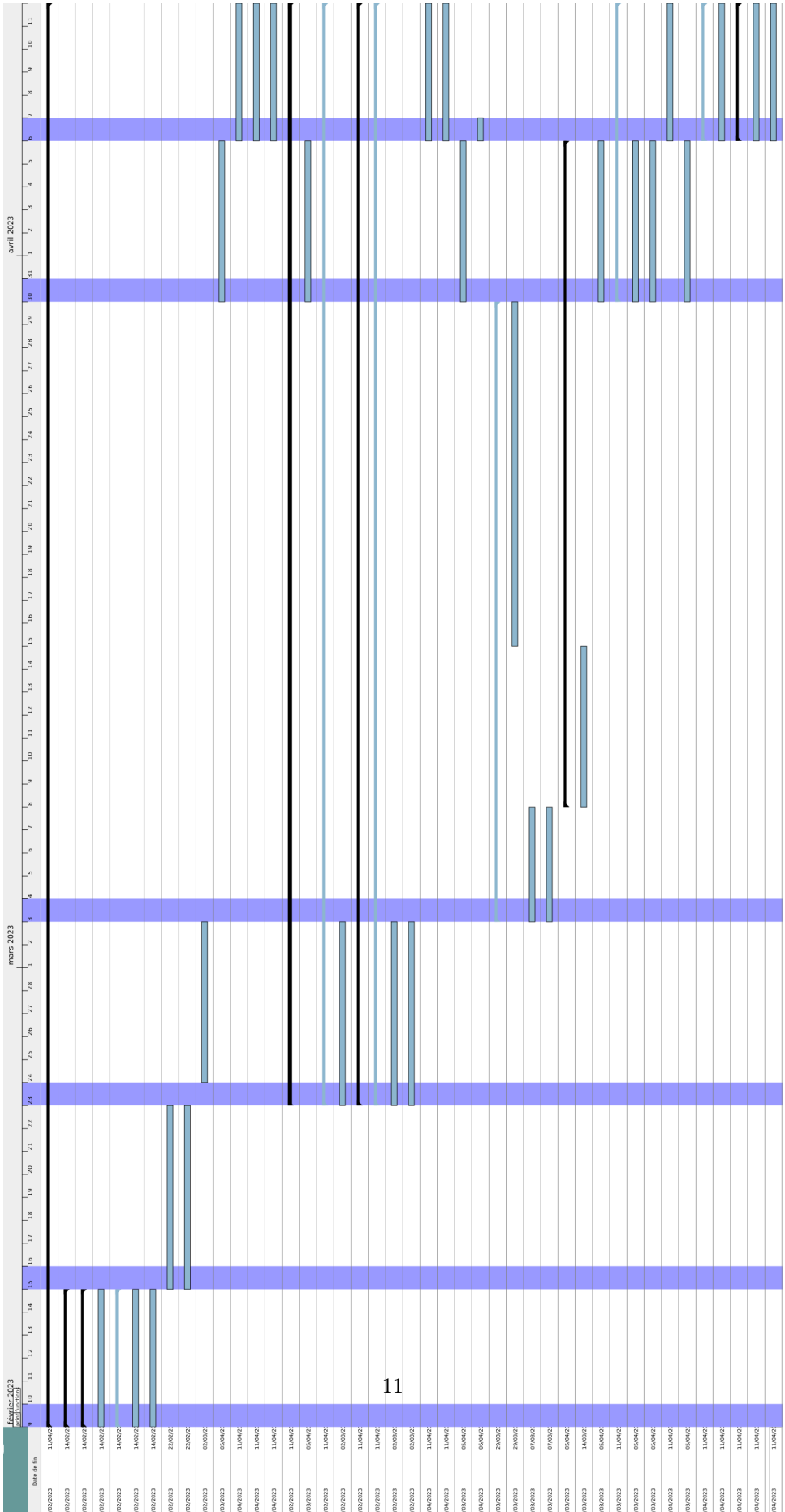
- `print` : `string -> void`
- `printi` : `int -> void`
- `getchar` : `void -> string`
- `ord` : `string -> int`
- `chr` : `int -> string`
- `not` : `int -> int`
- `exit` : `int -> void`
- `size` : `string -> int`

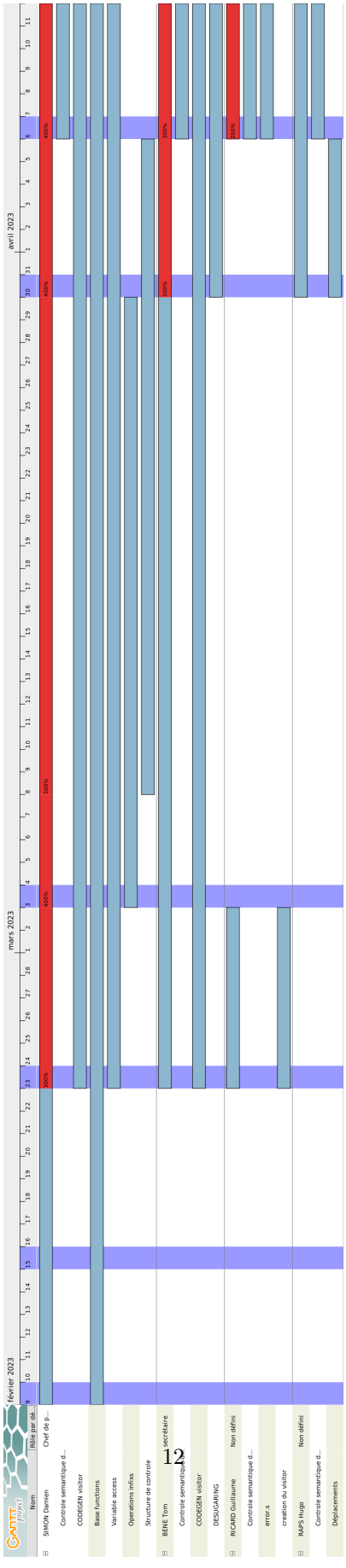
1.7.2 Fonctions bonus

- `printer` : `string -> void`
- `strcmp` : `string, string -> int`
- `malloc` : fonction interne uniquement
- `getrandom` : `int, int -> int`

2 Gestion de projet

Les réunions sur cette partie du projet ont été moins régulières que pour la première partie du projet.
La répartition des tâches s'est trouvée être plus complexe, car un seul membre du groupe est en IL et a donc eu une formation sur la génération de code.


février 2023
mars 2023
avril 2023
Date de fin


GANTT PROJECT
Nom
Rôle par dé...
SIMON Damien
Chef de p...
Contrôle semantique d...
CODEGEN visitor
Base functions
Variable access
Operations infixs
Structure de controle
BERE Tom
secrétaire
Contrôle semantique d...
CODEGEN visitor
DESIGNING
RICARD Guillaume
Non défini
Contrôle semantique d...
error.s
creation du visitor
RAPS Hugo
Non défini
Contrôle semantique d...
Déplacements
février 2023
mars 2023
avril 2023
400%
500%
400%
300%
200%
300%
200%

3 Jeux d'essais

Le programme suivant teste les records/arrays, les déclarations de fonction, les boucles `for` et `while`.

```
    let
  var un_string_vide:string := ""
  var un_int_sans_type := 3
  type intArray = array of int
  type intMatrice = array of intArray
  type record_1 = {abscisse : intArray, ordonnee : intArray}
  type hashmap = {cle : int, nom : string}
  var tableau := intArray[4] of 0
  var tableau_diviseurs := intArray[4] of 0

  function arithm(a:int, b:int, c:int ,d:int) : int = a+b*c/d
  function min(x:int,y:int):int = (if x<=y then x else y)

  function nb_divis(a:intArray,b:int):intArray=(
    let var tabOfMults := intArray[4] of 0
    in
    for i:=0 to 4 do(
      let var x:=0
      in
        while a[i]>0 do(
          a[i]:=a[i]/b;
          x:=x+1
        );
        tabOfMults[i]:=x
      end
    );
    tabOfMults
    end
  )

in
  print("50+14*19/3 = ");
  printi(arithm(50,14,19,3));
  print("\n");
  print("min entre 50 et 3 = ");
  printi(min(50,3));
  print("\n");
  print("Combien de fois peut-on diviser chaque nombre du tableau [462,357,921,491]
    par 7 ? ");
  tableau[0]:=462;
  tableau[1]:=357;
  tableau[2]:=921;
  tableau[3]:=491;
  tableau_diviseurs:=nb_divis(tableau,7);
  for i:=0 to 4 do (
    printi(tableau_diviseurs[i]);
    print(" ")
  )
end
```

Ce programme teste la récursivité, les branchements `if then else` et les opérateurs logiques.

```
    let
  function min(x:int,y:int):int = (if x<=y then x else y)

  function modulo(x:int,m:int):int = (
    let
      var quotient : int := x/m
      var mult : int := x/quotient
    in
```

```
    x - quotient*mult
    end
  )

  function pgcd(a:int, b:int):int = (
    let
      var minab := min(a,b)
      var resultat:=0
    in
    for i:=1 to minab do
      if modulo(a,i)=0 & modulo(b,i)=0 then resultat:=i;
      resultat
    end
  )

  function pgcd_recurs(a:int, b:int):int = (
    if b=0 then a
    else (
      pgcd_recurs(b,modulo(a,b))
    )
  )
in
  print("\n");
  print("74 mod 3 = ");
  printi(modulo(74,3));
  print("\n");
  print("PGCD de 582 et 31 (recursif et non-recursif): ");
  printi(pgcd_recurs(582,31));
  print("   ");
  printi(pgcd(582,31))
end
```

3.1 Fonctions de la librairie standard

```
    let var a:int:=0
in
  print("Grace    chr(), on sait que le char de code ASCII 83 est : ");
  print(chr(83));
  print("\n");
  print("Grace    ord(), on sait que le code ASCII de @ est : ");
  printi(ord("@"));
  print("\n");
  print("La taille du string ViveARM32 est : ");
  printi(size("ViveARM32"));
  print("\n");
  print("Avec getrandom(1,10) on recupere un chiffre al atoire entre 1 et 10 : ");
  printi(getrandom(1,10));
  print("\n");
  print("On peut comparer deux strings avec strcmp(), par exemple 32bit et 64bit : ");
  printi(strcmp("32bit","64bit"));
  print("\n");
  print("On peut schtroumpfer un entier en 0 ou 0 en 1 avec not() : ");
  printi(not(1));
  print("\n");
  print("On peut quitter le programme avec un code d'erreur");
  exit(69)
end
```

4 Programme de démonstration

```
    let
```

```
    var N:int := 128
    var a :string := (print("fill caracter : ");getchar())
    type intarray = array of int
    var L1:=intarray [N] of 0
    var L2:=intarray [N] of 0
    function valide(i: int):int = ((i>= 0) & i<N)
    function MAj_L(L1:intarray,L2:intarray,n:int) = (
        let
            var nb_voisins:=0
        in
            for i := 0 to N-1 do (
                nb_voisins:=(valide(i-1)&L1[i-1]) + L1[i] + (valide(i+1)&L1[i+1]);
                L2[i]:= nb_voisins<3 & nb_voisins>0;
                print(if L1[i] then a else " ")
            );
            print("\n")
        end;
        if n then MAj_L(L2,L1,n-1)
    )
in
    L1[N/2]:=1;
    print("\33c");
    MAj_L(L1,L2,63)
end
```