

TELECOM Nancy

Projet de compilation

Compilation

Rapport final

Réalisé par :
HANTZ Simon
KESLICK Malaury
LASSAGNE Guillaume
RACOUCHOT Maïwenn

Professeurs :
COLLIN Suzanne
DA SILVA Sébastien
OSTER Gérald

2020

Table des matières

1 Introduction **3**

2 Table de symboles **4**

2.1 Structure de la table 4

2.2 Remplissage de la table des symboles 4

2.3 Gestion des déplacements mémoire 4

3 Contrôles sémantiques **5**

3.1 Parcours de l'arbre 5

3.2 Contrôles effectués 5

4 Génération de code **7**

4.1 Gestion de la mémoire 7

4.2 Contrôles sémantiques dynamiques 7

4.3 Gestion et utilisation des registres 7

4.4 Schémas de traduction 8

4.4.1 L'accès à une valeur 8

4.4.2 Les comparaisons 8

4.4.3 Les *IF* 9

4.4.4 Les *FOR* 10

4.4.5 La gestion des procédures 11

5 Jeux d'essais **12**

5.1 Jeux d'essais sur la TDS 12

5.2 Jeux d'essais sur les CS 14

6 Gestion de projet **16**

6.1 Diagramme de Gantt 16

6.2 Tableau de répartition des tâches 17

6.3 Comptes-rendus de réunion 17

7 Annexe **18**

7.1 Compte rendu de réunion numéro 1. 18

7.2 Compte rendu de réunion numéro 2. 20

7.3 Compte rendu de réunion numéro 3. 21

7.4 Compte rendu de réunion numéro 4. 22

7.5 Compte rendu de réunion numéro 5. 23

7.6 Compte rendu de réunion numéro 6. 24

1 Introduction

L'objectif de ce projet est d'écrire un compilateur du langage Algol 60 à partir de la spécification fournie sur le site Algol60.org ainsi que celle fournie par les responsables du tutorat. Ce projet est à réaliser par groupe de 4 étudiants. Il est séparé en deux modules : PCL1 et PCL2.

Ce rapport traite de la partie PCL2 pour laquelle il faut présenter la structure de la TDS, les différents contrôles sémantiques mis en place ainsi que les jeux d'essais mettant en évidence les limites et les réalisations que permettent notre compilateur. Il est aussi question d'expliciter la gestion de projet mise en place afin de gérer les ressources humaines et temporelles que nous avions à notre disposition.

2 Table de symboles

2.1 Structure de la table

La table des symboles est représentée par une LinkedHashMap, une table d'association de clés (identificateur de la variable) et de valeurs (un symbole). Ces LinkedHashMap sont liées à un et un seul bloc afin de représenter son espace des noms. À chaque entrée dans un nouveau bloc, une nouvelle LinkedHashMap est créée. Un symbole peut décrire différentes choses :

- Une variable possédant un identificateur (qui est en fait le même argument que la clé), un type, une taille mémoire et un paramètre définissant l'index d'ajout à la HashMap.
- Un tableau possédant un identificateur, un type, une borne inférieure, une supérieure et une dimension.
- Une procédure possédant un nom permettant de l'identifier, le type de retour, le nombre de paramètres ainsi qu'une liste regroupant ces mêmes paramètres.
- Les itérateurs de boucles sont définis comme des variables à la différence près qu'un champ particulier permettant de les distinguer est mis en place.

2.2 Remplissage de la table des symboles

Le remplissage de la table se fait grâce à l'AST. Elle est remplie au fur et à mesure en fonction des différents tokens rencontrés. À chaque nouvelle région, une nouvelle table est créée avec un niveau d'imbrication incrémenté puis elle est ajoutée en tant que fils de la table précédente. Grâce à cela, on conserve un lien de parent-enfant afin de naviguer dans toutes les tables du programme. À chaque déclaration dans le programme, un symbole est créé avec les informations fournies par le programme (nom, type etc...). Lorsque l'on rencontre une définition de fonction, le prototype de celui-ci est déclaré dans la table père mais toutes les informations qui la concerne sont envoyées dans sa table fils.

2.3 Gestion des déplacements mémoire

Les déplacements mémoire concernent les symboles et sont calculés en fonction de l'ordre de déclaration des paramètres. Ainsi, lorsqu'un bloc se termine, les variables mettent leurs déplacements à jour pour pouvoir être utilisées correctement dans la phase de génération du code assembleur. Par exemple, les variables de type entier ont une taille de 2 alors que les réels ont une taille de 4. Il suffit donc de sommer les variables déclarées précédemment pour définir le déplacement mémoire nécessaire pour chacune.

3 Contrôles sémantiques

3.1 Parcours de l'arbre

Après le remplissage de la table des symboles, on parcourt l'AST pour réaliser les contrôles sémantiques. Ces tests se font à travers la classe *newAnalyseur.java*. Le parcours se base sur la fonction *analyseExp*. Cette fonction prend en paramètre un arbre et détermine la fonction d'analyse à appliquer à sa racine. Si la racine ne donne pas directement lieu à un contrôle sémantique, sa fonction associée appelle analyseExp sur ses fils. Ainsi, on passe par tous les noeuds de l'arbre. Les fonctions d'analyse renvoient des String. Cela permet de récupérer le type des fils quand cela est nécessaire pour vérifier la cohérence au sein de l'arbre. Si une erreur est détectée, une ligne est affichée en console. Elle indique la ligne et la position du problème dans le programme en Algol60 et un intitulé qui explique l'erreur.

3.2 Contrôles effectués

Dans cette partie, nous avons effectué des contrôles sémantiques statiques de différents types :

Définition :

Ce contrôle se concentre sur les définitions de variables. L'identifiant doit suivre le format suivant : une chaîne alphanumérique commençant par un caractère alphabétique.

Step/Until :

Dans le cas où l'itérateur est un entier, ce contrôle permet de vérifier que les bornes sont bien de type entier. Dans le cas où l'une des bornes ne respecte pas cette condition, le message d'erreur précise le type trouvé.

Puissance :

Ce contrôle analyse la nature des deux fils d'une puissance et renvoie une erreur si l'un des deux est de type booléens et non réel ou entier.

Opérateurs mathématiques (+,-,*,/, **) :

Ces contrôles suivent le même schéma. Ils commencent par regarder les types de leurs enfants. Cela peut donner lieu à des appels récursifs dans les cas où plusieurs opérations se suivent. Ensuite, on vérifie qu'aucun des fils n'est un boolean. Si ce n'est pas le cas, on regarde les types présents pour déterminer le type de retour de l'opération. Il est à noter que l'opérateur moins à un cas supplémentaire. En effet, le

même symbole sert pour l'opération et le moins unaire, symbole du négatif. Pour le moins unaire, le contrôle est similaire, on vérifie que le fils n'est pas de type boolean et renvoie le type.

Identifiant :

Ce contrôle vérifie si un identifieur a bien été défini plus haut dans le programme. Pour cela, on utilise la TDS.

For :

Les contrôles relatifs aux boucles for vérifient que l'itérateur est bien de type integer.

Comparateurs (<, <=, =, ...) :

Ce contrôle fonctionne sur le même principe que les opérateurs mathématiques. On vérifie que les fils ont bien un type boolean (en appelant une analyse récursive s'il le faut). L'erreur renvoie le type du fils problématique s'il y a lieu.

Opérateurs logiques :

Ce contrôle vérifie si au moins un des deux fils n'est pas de type booléen. Dans ce cas, une erreur est affichée.

Action :

Le noeud *Action* peut apparaître dans différentes situations et donne donc lieu à plusieurs types de contrôles.

- Tableaux : le noeud Action peut être impliqué pour récupérer les éléments d'un tableau. Il est donc nécessaire de vérifier si l'élément demandé se situe dans les bornes connues du tableau. Ce contrôle ne fonctionne que si l'accès est statique.
- Affectations : le noeud peut aussi être utiliser pour affecter une valeur à une variable. Dans ce cas, on vérifie si la valeur donnée est du type correspondant à celui déclaré pour la variable.
- Procédure : enfin, le noeud action intervient dans l'appel de procédure. Dans ces cas, le contrôle vérifie que la procédure est appelée avec le nombre de paramètres déclaré dans la table des symboles.

4 Génération de code

La fin d'une analyse sémantique sans erreur déclenche alors la génération de code. Celle-ci se fait via la classe *CodeGenerator* et repose sur les mêmes principes de fonctionnement que l'analyseur sémantique, à savoir un parcours de l'AST noeud par noeud. Chaque noeud appelle une fonction *genere*, permettant d'écrire dans un fichier .src les commandes ASM nécessaires.

Nous allons maintenant vous présenter le fonctionnement de notre traduction du code en langage assembleur.

4.1 Gestion de la mémoire

Tout d'abord, nous avons divisé la mémoire en deux parties : la pile et la zone des instructions. Nous n'avons pas eu recours au tas dans notre travail, puisque nous ne sommes pas allés assez loin pour avoir à prendre en charge des chaînes de caractères et des tableaux. Nous avons fait démarrer notre pile à 0x1000 parce que c'est l'adresse de début de pile par défaut de l'outil MicroPIUPK. Nous avons alors choisi l'adresse de chargement du programme à 0x2000. Bien que l'espace entre les zones ne soit pas nécessaire puisque la pile se remplit par adresses décroissantes et inversement pour la zone des instructions, ce choix a été fait pour faciliter les moments de debogage puisque nous pouvions ainsi différencier facilement chacune de ces parties via l'interface du simulateur.

4.2 Contrôles sémantiques dynamiques

La gestion de code nous a permis de mettre en place un nouveau contrôle sémantique, cette fois dynamique. Il s'agit des divisions par 0. En effet, si le dénominateur d'une fonction est nul, alors le programme passe dans une trappe d'arrêt du programme.

4.3 Gestion et utilisation des registres

Les registres nous auront été très utiles lors de ce travail, afin d'éviter l'empilement systématique des variables temporaires. Notre convention de base est la suivante : nous avons choisi R15 pour être notre sommet de pile (SP), R0 pour être notre pointeur de base (BP) et R1 pour être notre registre de travail courant (WR), où tous les résultats de recherche de valeur associée à un identificateur ou d'opérations arithmétiques et logiques se stockent avant d'être soit réutilisés dans une autre opération, soit rangés dans les cases mémoires voulues.

Le registre R2 sert régulièrement de registre tampon, permettant de retenir une valeur avant de la réutiliser dans un autre calcul, comparaison, etc.. Le registre R3 est

ponctuellement utilisé pour accueillir des valeurs de comparaison, au même titre que R2 (les sections *Les comparaisons* et *Les IF* expliciteront ce fonctionnement).

Les registres R11, R12 et R13 sont utilisés dans le cadre de la gestion des boucles for, gestion qui sera explicitée dans la section *Les FOR*.

Finalement, le registre R14 est celui qui permet de garder le BP courant en tampon lors des différentes remontées de chaînages nécessaires à la localisation d'une variable ou d'un paramètre.

4.4 Schémas de traduction

4.4.1 L'accès à une valeur

La méthode *genereIdf* est la méthode appelée par défaut lorsque l'on arrive sur un noeud ou une feuille de l'arbre qui n'est pas l'un des tokens imaginaires définis dans la grammaire. Nous distinguons alors plusieurs cas :

- Si l'on arrive sur un entier, alors cette valeur sera directement chargée dans le registre de travail WR.
- Si l'on arrive sur un identificateur de procédure lors d'un appel de celle-ci (l'appel étant modélisé par un booléen), alors les paramètres effectifs sont empilés, puis le *Jump* à l'adresse du sous-programme est effectué, avant de finalement dépiler les paramètres. Le fonctionnement interne à la procédure sera explicité quatre sous-parties plus loin.
- Sinon, on arrive sur un identificateur de variable, itérateur, ou paramètre : alors on commence par localiser le bloc de déclaration de cette variable, et on remonte jusque celui-ci à l'aide du chainage statique stocké dans BP. Une fois dans le bloc souhaité, on calcule le déplacement correspondant, qui nous permet de stocker une valeur à cet emplacement ou de renvoyer celle s'y trouvant (le paramètre booléen *save* nous permettant de discerner le comportement à adopter). On termine en revenant dans le bloc appelant grâce au registre R14 ayant servi de tampon pour le BP.

4.4.2 Les comparaisons

Lorsque l'on arrive sur un élément de comparaison (<,<=,>,>=,<>,=) lors du parcours de l'AST, la méthode *genereComparateur* est appelée. Tout d'abord, en fonction du comparateur donné en Algol60, nous déduisons le comparateur adéquat en ASM (< : JLW, <= : JLE, > : JGT, >= : JGE, <> : JNE, = : JEQ). Ensuite, nous lançons

la méthode mère *genereExp* sur le fils gauche (son résultat final se range alors directement dans WR). Après avoir stocké le résultat dans R2, nous lançons le même procédé sur le fils droit. Il ne nous reste plus qu'à comparer les contenus des registres R2 et WR. Pour ce faire, nous avons recourt aux étiquettes en assembleur. Le fonctionnement est le suivant :

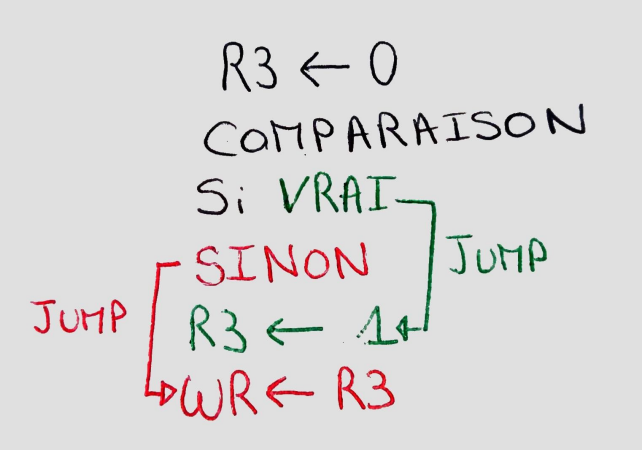

FIGURE 1 – Schéma de fonctionnement de la comparaison

Ainsi, c'est le registre R3 qui est utilisé pour stocker le résultat final de la comparaison (il contiendra 1 si la comparaison est vraie, 0 sinon). En effet, si la comparaison est vraie, alors on saute l'instruction suivante et on met 1 dans le registre R3. Puisque les instructions en ASM sont lues dans l'ordre d'écriture, alors on passera tout de même dans l'instruction chargeant la valeur de R3 dans WR. En revanche, si la comparaison est fausse, le programme se poursuit et passe dans le *jump* du sinon, qui saute la partie du chargement de la valeur 1 dans R3, et stocke donc 0 dans WR.

4.4.3 Les *IF*

Si l'on arrive sur un noeuf *If* dans l'AST, alors le procédé est le suivant :

- On récupère le booléen résultant de la condition dans le registre de travail WR.
- On charge 1 dans le registre R2, et on compare WR et R2. La façon de procéder ensuite est différente s'il y a un 'else' ou non.
- S'il y a un 'else', alors on cherche à savoir si les contenus des deux registres sont égaux. S'ils le sont, alors on effectue un *jump* au code assembleur du bloc *Alors*, puis un *jump* au code suivant l'appel du *If*. Sinon, on saute directement

au code assembleur du bloc *Sinon*, et la suite s'enchaîne naturellement.

— En revanche, s'il n'y a pas de 'else', alors on vérifie si les contenus de WR et R2 sont différents. S'ils le sont, on effectue un *jump* à la suite du code, et sinon on passe par le bloc du *Alors*.

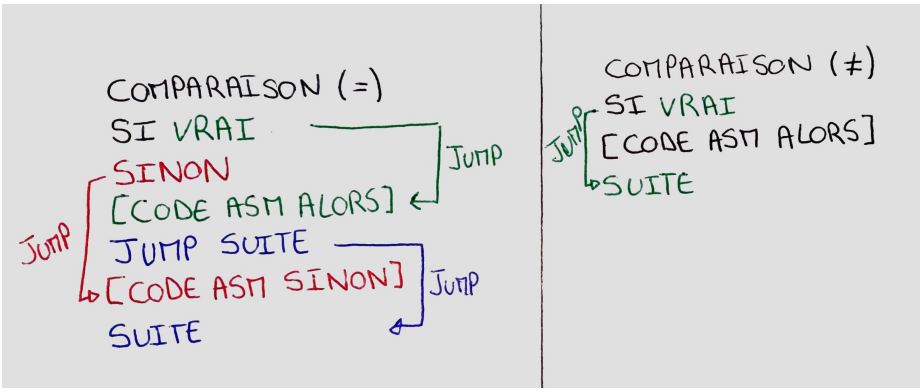

FIGURE 2 – Schéma de fonctionnement du IF (avec else à gauche, sans else à droite)

4.4.4 Les *FOR*

Les méthodes *genereForCondition* et *genereForstatement* permettent de générer le code correspondant aux blocs *for*. Le fonctionnement est le suivant :

— La valeur de l'itérateur est stockée dans R11, celle de la borne supérieur dans R12 et celle du pas dans R13.

— On compare la valeur de l'itérateur à celle de la borne supérieur, et tant que la première est strictement inférieure à la seconde, on effectue un *jump* à l'étiquette *LOOP*, à partir de laquelle le code inclut dans le for est exécuté avant d'incrémenter l'itérateur de la valeur stockée dans R13.

— On réitère tant que la borne supérieure n'est pas atteinte. On sort de la boucle lorsque l'itérateur devient supérieur ou égal à cette borne. Si elle est égale, alors on passe une dernière fois dans le code de la boucle for, puis on termine, sinon on termine directement.

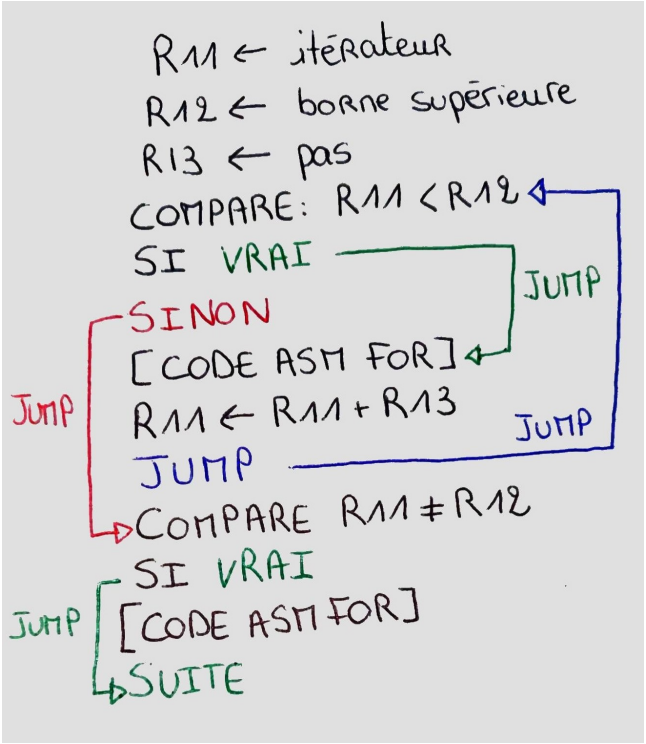

FIGURE 3 – Schéma de fonctionnement du FOR

4.4.5 La gestion des procédures

Nous allons expliciter dans cette partie la manière dont nous avons géré la création d'une procédure (ce qu'il se produit lors de l'appel à une procédure est décrit dans le partie *L'accès à une valeur* un peu plus haut). Lorsque l'on arrive sur un noeud "TYPARPROCDEC", synonyme de déclaration, on vérifie si c'est une procédure. Si c'est bien le cas, alors on entre dans la méthode *genereProc*. Celle-ci permet de générer le code de la sous-fonction après l'étiquette appropriée, afin que l'on puisse y faire un *jump* lorsque l'appel est fait. Lorsque l'on entre dans le bloc de cette procédure, la méthode *genereEntrerBloc* est appelée. Elle permet d'actualiser la table des symboles courante, le numéro de bloc, de créer une nouvelle zone de liaison et d'empiler les variables locales au bloc. Ensuite, le code de la fonction est deroulé, puis la sortie du bloc se fait en revenant aux BP, SP, table des symboles et numéro de bloc précédents. Les paramètres sont à leur tour retirés de la pile après l'appel via la fonction *genereIdf*.

5 Jeux d'essais

5.1 Jeux d'essais sur la TDS

Afin de valider nos productions, nous avons réalisé un certain nombre d'essais avec des programmes types représentant tous les identificateurs que nous devions intégrer à la TDS et vérifier les différents numéros d'imbrication ainsi que la cohérence entre les TDS. Voici un exemple avec le code source (tiré du programme Matrix7 des Lego pieces) ainsi que le résultat renvoyé par notre programme.

```
begin
comment random matrix generation;
        real array aa[1:100,1:100];
        integer n;

        procedure outmatrix(channel, a, dim);
        integer channel; real array a; integer dim;
        begin
                integer i, j;
                for i := 1 step 1 until dim do
                begin
                        for j := 1 step 1 until dim do outreal (channel, a[i,j]);
                end
        end ;

        procedure matrRandom(ab, dim, seed);
        real array ab; integer dim, seed;
        begin
                integer i, j;
                integer m, a, b, s, r;
                m := 32768; a := 805; b := 6925;
                s := seed;
                for i := 1 step 1 until n do
                begin
                        for j := 1 step 1 until n do
                        begin
                                s := s*a+b; r := entier(s/m); s := s - r*m;
                                ab[i,j] := (entier((s+1)/(m+1)*10000)) / 10000
                        end
                end
        end ;

        n := 10;
        matrRandom(aa, n,  0);
        outmatrix(1, aa, n)

end
```

Figure 4 – Code source qui permet de remplir la TDS

```
------------------------------------------------------------
TDS numéro 1/ imbrication 0

Clé : aa
Symbole [nom=aa, typeElement=tableau, type=real, borneInf=1, borneSup=100, dimension=2]

Clé : n
Symbole [nom=n, typeElement=variable, type=integer, taille=2, deplacement : 0]

Clé : outmatrix
Symbole [nom=outmatrix, typeElement=procedure, typeSortie=void, nombre de paramï¿½tres=3, parametres = [channel, a, dim]]

Clé : matrRandom
Symbole [nom=matrRandom, typeElement=procedure, typeSortie=void, nombre de paramï¿½tres=3, parametres = [ab, dim, seed]]

------------------------------------------------------------
TDS numéro 1/ imbrication 1

Clé : outmatrix
Symbole [nom=outmatrix, typeElement=variable, type=integer, taille=2, deplacement : 0]

Clé : channel
Symbole [nom=channel, typeElement=parametre, type=integer, taille=2, deplacement : -2]

Clé : a
Symbole [nom=a, typeElement=tableau, type=real, borneInf=0, borneSup=0, dimension=0]

Clé : dim
Symbole [nom=dim, typeElement=parametre, type=integer, taille=2, deplacement : 0]

Clé : i
Symbole [nom=i, typeElement=iterateur, type=integer, taille=2, deplacement : 2]

Clé : j
Symbole [nom=j, typeElement=iterateur, type=integer, taille=2, deplacement : 4]

------------------------------------------------------------
TDS numéro 2/ imbrication 1

Clé : matrRandom
Symbole [nom=matrRandom, typeElement=variable, type=integer, taille=2, deplacement : 0]

Clé : ab
Symbole [nom=ab, typeElement=tableau, type=real, borneInf=0, borneSup=0, dimension=0]

Clé : dim
Symbole [nom=dim, typeElement=parametre, type=integer, taille=2, deplacement : -2]

Clé : seed
Symbole [nom=seed, typeElement=parametre, type=integer, taille=2, deplacement : 0]

Clé : i
Symbole [nom=i, typeElement=iterateur, type=integer, taille=2, deplacement : 2]

Clé : j
Symbole [nom=j, typeElement=iterateur, type=integer, taille=2, deplacement : 4]

Clé : m
Symbole [nom=m, typeElement=iterateur, type=integer, taille=2, deplacement : 6]

Clé : a
Symbole [nom=a, typeElement=iterateur, type=integer, taille=2, deplacement : 8]

Clé : b
Symbole [nom=b, typeElement=iterateur, type=integer, taille=2, deplacement : 10]

Clé : s
Symbole [nom=s, typeElement=iterateur, type=integer, taille=2, deplacement : 12]

Clé : r
Symbole [nom=r, typeElement=iterateur, type=integer, taille=2, deplacement : 14]
```

FIGURE 5 – Etat de la TDS renvoyé par notre programme

5.2 Jeux d'essais sur les CS

Pour vérifier nos contrôles sémantiques, nous avons écrit un programme en Algol60 regroupant l'ensemble des erreurs traitées.
Voici le programme en question ainsi que les affichages obtenus. En commentaires du code se trouvent les erreurs qui sont supposées être relevées.

```
begin

  real y,a,n,i,r;
  integer b,c;
  boolean b1,b2,b3;
  integer array nPT[1:10.1]; comment limite non entière;
  integer array tab[1:15];
  real procedure poly2(a0,a1,a2,x);real a0,a1,a2,a; real x;
    begin
      poly2 := a0+a1*x+a2*x*x
    end;

  y := 12.0;
  b:=y+a; comment non concordance des types;
  n:=10**(y+p); comment variable non déclarée;
  z:=poly2(0,0,1.0,2.0); comment affectation d'une valeur à une variable non déclarée;
  y:=poly2(0,0,1.0); comment appel d'une procédure avec un argument manquant;
  tab[4.2]:=12; comment accès à un emplacement de tableau non entier;
  tab[17]:=0; comment accès à un emplacement de tableau dépassant les limites;

  if c < 0 then
  begin
    if c>= -100 then b := c-100.0 else b := -100
    comment non concordance des types au sein d'un if;
  end
  else b := 0;

  for i:=1 step 1 until 10.05 do a:=2;
  comment la borne de fin n'est pas entière alors que l'itérateur commence à un entier et que le pas est entier;
  for b:=1 step 0.5 until 10 do outinteger(1, b); comment b est un entier et ne peut pas évoluer par pas non entier;
  for b3 := 0 step 0.2 until 5 do a:=7; comment iterateur non entier ou réel;

  if b1<0 then a:=4.0; comment on ne peut pas effectuer de comparaison sur un boolean et quoi que ce soit d'autre;
  b3:=b1+b2; comment on ne peut pas additioner de booleens;
  b3:=b1**b3; comment on ne peut pas mettre un booleen dans une puissance;
  b2:=b3&&y; comment on ne peut pas mettre autre chose qu'un booleen dans une op logique;

  n:=y/0 comment Erreur sem division explicite par 0;

end
```

FIGURE 6 – Code source qui permet d'afficher les CS

```
ERROR : line 6, position 20 : Boundaries can't be reals : 10.1.
ERROR : line 14, position 2 : integer expected, real found for idf b.
ERROR : line 15, position 12 : Identifier p is not defined.
ERROR : line 16, position 2 : Identifier z is not defined.
ERROR : line 17, position 2 : poly2 procedure is called with a false number of arguments (4 expected).
ERROR : line 18, position 2 : Array elements are reachable by their position in the Array (integers only), but 4.2 is real.
ERROR : line 19, position 2 : Index 17 is superior to the upper boundary 15.
ERROR : line 23, position 21 : integer expected, real found for idf b.
ERROR : line 28, position 16 : Integer iterators with integer steps must reach an integer : 10.05 found.
ERROR : line 30, position 16 : Integer iterators must evolve among integers : 0.5(STEP) found.
ERROR : line 31, position 6 : Iterators must be typed integer or real : boolean found.
ERROR : line 33, position 7 : Impossible to apply '<' between a boolean and anything else (boolean found : b1, integer found :  0)
ERROR : line 34, position 8 : Booleans can't be added.
ERROR : line 35, position 8 : Booleans can't play any role in any raising to a power.
ERROR : line 36, position 8 : '&&' operator can only be applied on booleans but real found for y.
ERROR : line 38, position 6 : Explicit division by 0.
```

FIGURE 7 – Affichage des erreurs sémantiques

6 Gestion de projet

6.1 Diagramme de Gantt

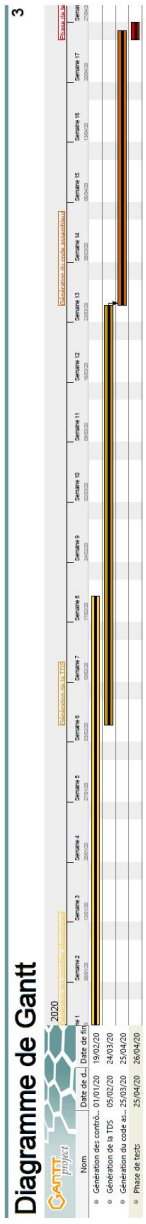

Figure 8 – Diagramme de Gantt de la deuxième partie du projet

6.2 Tableau de répartition des tâches

	HANTZ Simon	KESLICK Malaury	LASSAGNE Guillaume	RACOUCHOT Maïwenn
Génération des contrôles sémantiques	8h00	16h00	14h00	40h00
Génération de la TDS	35h00	6h00	1h00	1h00
Génération du code assembleur	8h00	50h00	15h00	10h00
Diagramme de Gantt	0h30	-	-	-
Comptes-rendus de réunion	1h00	1h00	1h00	1h00
Rédaction du rapport	2h00	2h00	2h00	2h00
TOTAL	54h30	75h00	33h00	54h00

6.3 Comptes-rendus de réunion

Tous les comptes-rendus de réunions réalisés pendant la 2ème partie de ce projet sont présents ci-dessous en annexe.

7 Annexe

7.1 Compte rendu de réunion numéro 1.

PCL2 - Compte rendu de réunion numéro 1.

Motif de la réunion : Réunion de reprise et début phase 2	Lieu : Visioconférence
Présent(s) : Simon Hantz Malaury Keslick Guillaume Lassagne Maïwenn Racouchot	Date, heure et durée : Dimanche 05/01/2020, 14h (environ 2 heures)

Prise de connaissance de la deuxième partie

La réunion débute par un retour sur le sujet de la deuxième partie à savoir :
— La création des contrôles sémantiques
— La création de la TDS
— La génération de code Assembleur

Nous discutons des différents contrôles sémantiques que nous allons mettre en place et aux différents endroits où nous pourrions trouver des exemples de programme à "contrôler" avec notre code. Nous nous servirons évidemment du site référent, Algol60.org et des jeux de tests à disposition.

Distribution du travail à effectuer sur la première partie

Maïwenn se chargera de :
— Gérer les listes de paramètre ;
— Gérer les bornes des tableaux ;
— Gérer les conditions ;
— S'assurer de la bonne définition des variables ;
— S'assurer que les paramètres soient du bon type et bien formés ;
— Gérer les affectations ;
— S'assurer du bon fonctionnement des opérateurs + et -.

Malaury se chargera de :
— Gérer les blocs if ;
— Gérer les blocs for ;
— S'assurer de la bonne forme des conditions dans les if ;
— S'assurer de la bonne forme des bornes dans les for ;

— S'assurer de la bonne forme des blocs step until ;
— Gérer le cas du else.

Guillaume se chargera de :
— Gérer les entrées dans les blocs procédure ;
— Gérer les goto ;
— Gérer les opérateurs logiques ;
— Gérer les comparaisons.

Simon se chargera de :
— S'assurer que les prototypages des fonctions sont correctement formés ;
— Gérer les procédures de déclaration des fonctions ;
— Gérer la bonne exécution des boucles for ;
— Gérer la gestion des éléments dans les boucles for.

Date de la prochaine réunion : mercredi 05/02/2020 pour discuter de la TDS .

7.2 Compte rendu de réunion numéro 2.

PCL2 - Compte rendu de réunion numéro 2.

Motif de la réunion : Etat des lieux des CS et discussion TDS	Lieu : Télécom Nancy
Présent(s) : Simon Hantz Malaury Keslick Guillaume Lassagne Maïwenn Racouchot	Date, heure et durée : Mercredi 05/02/2020, 14h (environ 1 heure)

État des lieux des CS

La réunion débute par un retour sur l'avancement des contrôles sémantiques. Ceux-ci ont bien avancé mais il reste forcément quelques ajustements à faire. Pour la mi-février, la TDS doit être en place, nous discutons donc de son format et des informations qui s'y trouveront.

Distribution du travail

Les contrôles sémantiques ayant bien avancé pour chacun, nous décidons d'attaquer la TDS. Comme Maïwenn et Malaury ont eu les contrôles sémantiques les plus difficiles, Simon et Guillaume se chargeront un peu plus de la TDS. De manière générale, nous poursuivons nos différentes tâches.

Date de la prochaine réunion : début mars après avoir eu la "soutenance".

7.3 Compte rendu de réunion numéro 3.

PCL2 - Compte rendu de réunion numéro 3.

Motif de la réunion : Retours dû à la soutenance et génération de code	Lieu : Visioconférence
Présent(s) : Simon Hantz Malaury Keslick Guillaume Lassagne Maïwenn Racouchot	Date, heure et durée : Dimanche 01/03/2020, 14h (environ 1 heure)

Retours dû à la soutenance

Les retours de la soutenance de milieu de projet furent plutôt positifs et nous en étions très contents. Quelques ajustements dans la TDS et les contrôles sémantiques ont néamoins été pointés du doigt et nous les corrigerons dès que possible.

Génération de code assembleur

Nous attaquons donc la dernière étape, la génération de code assembleur. Par chance, Malaury étant en IL, nous bénéficierons de ses cours de traduction avancée pour nous aiguiller. Nous visualisons les documents mis à disposition par Mme Collin afin de pouvoir débuter la génération.

Distribution du travail

Étude des différents documents relatifs à la génération de code. Premier tests sur du code "jouet". Ajustement de la TDS pour la gestion des déplacements mémoires.

Date de la prochaine réunion : Lorsque l'avancement aura été suffisant.

7.4 Compte rendu de réunion numéro 4.

PCL2 - Compte rendu de réunion numéro 4.

Motif de la réunion : Avancement de la génération de code	Lieu : Visioconférence
Présent(s) : Simon Hantz Malaury Keslick Guillaume Lassagne Maïwenn Racouchot	Date, heure et durée : Mercredi 25/03/2020, 14h (environ 1 heure)

Génération de code assembleur

Après nous être formés sur les différents documents, nous nous sommes répartis le travail de chacun pour la génération de code avec comme premier objectif de réussir l'étape 1 du document de génération de code.

Distribution du travail

Simon : Génération des identificateurs avec l'aide de Malaury ;
Malaury : Génération des if et des entrées/changements de blocs ;
Maiwenn : Génération des boucles for ;
Guillaume : Génération des assignations simples.

Date de la prochaine réunion : le mercredi 15/04/2020 à 14h

7.5 Compte rendu de réunion numéro 5.

PCL2 - Compte rendu de réunion numéro 5.

Motif de la réunion : Avancement de la génération de code	Lieu : Visioconférence
Présent(s) : Simon Hantz Malaury Keslick Guillaume Lassagne Maïwenn Racouchot	Date, heure et durée : Mercredi 15/04/2020, 14h (environ 30 minutes)

Avancements sur la génération de code

Réunion rapide avec les différents membres afin de statuer sur l'avancement du projet et des objectifs qui seront atteignables d'ici le 27 avril pour la date de rendu du projet. Nous nous pensons capable de produire le niveau 2 quasiment 3 puisque l'un découle de l'autre...

Distribution du travail

Pas de distribution particulière, nous continuons à travailler sur les tâches en cours et les différents problèmes rencontrés.

Date de la prochaine réunion : le Dimanche 26/04/2020 à 14h

7.6 Compte rendu de réunion numéro 6.

PCL2 - Compte rendu de réunion numéro 6.

Motif de la réunion : Bilan de l'avancement et des rendus	Lieu : Visioconférence
Présent(s) : Simon Hantz Malaury Keslick Guillaume Lassagne Maïwenn Racouchot	Date, heure et durée : Dimanche 26/04/2020, 14h (environ 30 minutes)

Bilan du projet

Réunion finale pour clôturer le projet, derniers préparatifs pour la "soutenance" et vérification des derniers scripts. Nous corrigeons les fautes du rapport et mettons tout ça en ligne. Nous avons réussi à intégrer le print à la toute fin du projet et en sommes très contents. Ce fut un projet difficile et éprouvant en plus des conditions particulières, mais nous sommes fiers de nous.