Compilation des langages

Compte rendu de projet

Locust2520 ; <Redacted> Erwan Kessler ; Yann Meyer

29 avril 2020

VERSION FINALE

Table des matières

1 Introduction

1.1 Document

Ce document présente les avancements de la grammaire ainsi que de la création de l'arbre syntaxique abstrait. Ce rapport concerne donc le travail effectué de début octobre jusqu'à mi-décembre. Durant cette période nous avons rendu LL1 une grammaire sous forme BNF puis nous avons créer la structure d'AST associée.

2 Grammaire

2.1 Généralités

La grammaire LL(1) complète a été rédigée à partir de la spécification BNF ALGOL 60.

CODE SOURCE 1 – Configuration ANTLR pour la grammaire

```
grammar algol60;

options {
    k            = 1     ; /* LL(1) */
    backtrack    = false ;
    output       = AST   ;
    language     = Java  ;
}
```

2.2 Symboles terminaux

La liste des symboles terminaux est d'abord établie.
Pour simplifier les règles suivantes, on définit deux fragments alphanumériques.

CODE SOURCE 2 – Définition des fragments alphanumériques

```
fragment DIGIT
    : '0'..'9'
    ;

fragment LETTER
    : 'A'..'Z'
    | 'a'..'z'
    ;
```

On définit ensuite les identificateurs.

CODE SOURCE 3 – Définition des identificateurs

```
IDF
    : LETTER (DIGIT | LETTER)*
    ;
```

La définition des entiers naturels et des réels positifs est plus complexe car notre configuration ANTLR n'effectue pas de retour sur trace. Ainsi, pour l'analyseur syntaxique, les deux règles sont ambiguës. On définit donc uniquement les réels positifs (donc indirectement les entiers naturels).

CODE SOURCE 4 – Définition des réels positifs

```
UREAL
    : (DIGIT+ ('.' DIGIT+)? | '.' DIGIT+) ('e' ('+' | '-')? DIGIT+)?
    ;
```

On définit les chaînes de caractères comme symboles terminaux, sans respecter l'imbrication spécifiée du langage. Par exemple, «'TELECOM Nancy'» est reconnu, mais «' 'TELECOM ' 'Nancy' '» n'est pas reconnu.

CODE SOURCE 5 – Définition des chaînes de caractères

```
STR
: '‘' (~('‘' | '\''))* '\''
;
```

2.3 Dérécursivation à gauche

Pour rendre la grammaire LL(1), une dérécursivation à gauche des règles est effectuée.

2.4 Factorisation à gauche

Pour rendre la grammaire LL(1), une factorisation à gauche des règles est effectuée.

2.4.1 Expressions

Les expressions arithmétiques, booléennes et d'appellation [1] posent un problème de factorisation car elles peuvent admettre comme première étiquette un identificateur (par exemple pour $x + 1$; $x \wedge true$; x).
De plus, la structure des comparaisons arithmétiques entraîne de nombreuses difficultés pour simplifier la grammaire.
Pour remédier à ces problèmes, l'équipe propose de définir les expressions sans tenir compte des règles sémantiques associées aux opérations. Ces règles seront vérifiées par l'analyseur sémantique.

2.4.2 Affectations multiples

Si une grammaire est supposée LL(1), la reconnaissance correcte d'affectations multiples est impossible. Par exemple, $x := y := 1 := 0$ est nécessairement reconnu par la grammaire. En effet, seul le dernier membre de l'affectation multiple peut être un réel positif; mais l'analyseur syntaxique est incapable de situer la dernière affectation car il ne peut lire qu'un symbole à la fois et ne peut pas retourner dans un état précédent.
Pour remédier à ce problème, l'équipe propose de définir les affectations multiples sans tenir compte des règles sémantiques associées aux opérations. Ces règles seront vérifiées par l'analyseur sémantique.

2.5 Simplification

La grammaire est finalement simplifiée au maximum (moins d'une cinquantaine de règles) pour être facilement compréhensible.

1. «designational expression»

3 Arbre de syntaxe abstraite

L'AST a été commencé le 12 novembre 2019, il suit la construction de la grammaire.

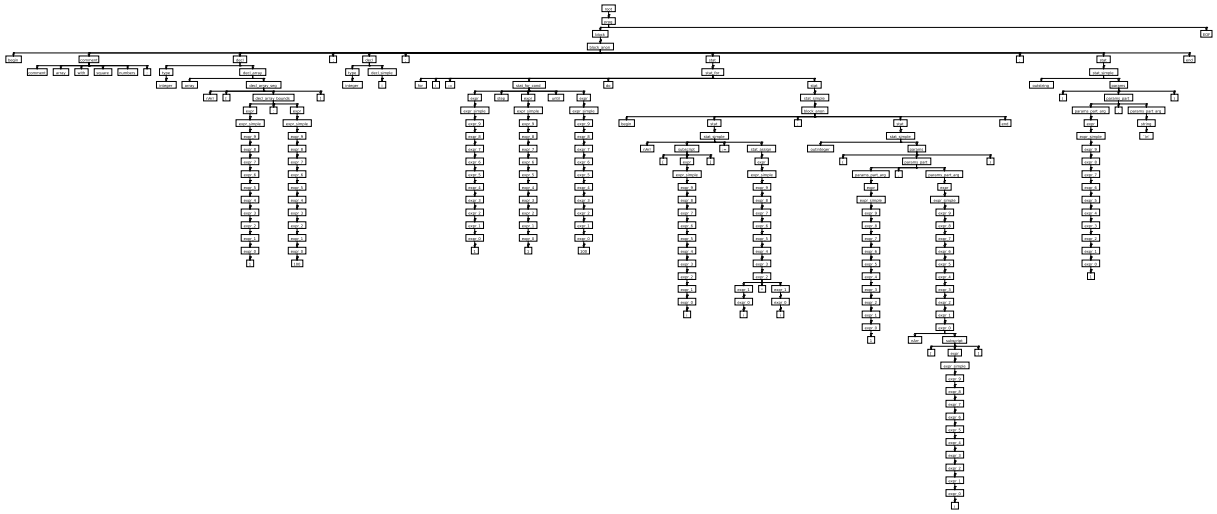
3.1 Étiquettes (Tokens)

Nous avons donc choisi des étiquettes reflétant chaque règle.

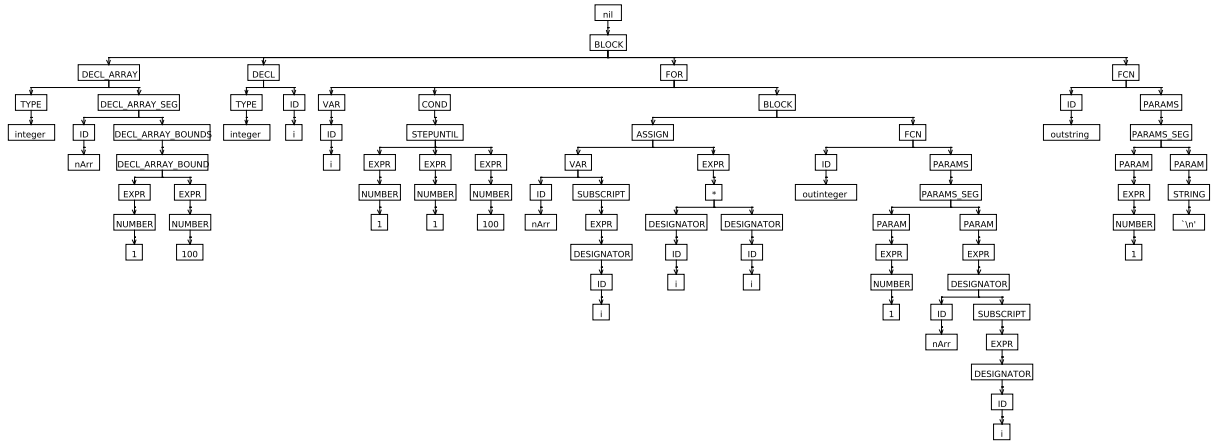
```
/*
ASSIGN              -> assignation
BLOCK               -> déclaration de blocs
COND                -> expression de type while ou step until
DECL                -> une déclaration simple
DECL_ARRAY          -> une déclaration d'un tableau
DECL_ARRAY_SEG      -> une déclaration d'un segment d'un tableau
DECL_ARRAY_BOUND    -> une déclaration d'une borne inférieure ou
   supérieure
DECL_ARRAY_BOUNDS   -> une déclaration d'une suite de bornes
DECL_SWITCH         -> une déclaration d'un switch
DECL_FCN            -> une déclaration d'une fonction
DECL_FCN_PARAMS     -> une déclaration de multiples paramètres
DECL_FCN_PARAMS_SEG -> une déclaration d'une liste de paramètres
DECL_FCN_PARAM      -> une déclaration d'un paramètre
DECL_FCN_SPECS      -> une déclaration d'une liste de spécifications
   dans une fonction
DECL_FCN_SPEC       -> une déclaration d'une spécification dans une
   fonction
DECL_FCN_VALUES     -> une déclaration d'une liste de valeurs dans une
   fonction
DESIGNATOR          -> un identificateur
DUMMY               -> une feuille nécessaire à ANTLR
EXPR                -> une expression
FCN                 -> une fonction
FOR                 -> une boucle for
GOTO                -> un saut vers un label
ID                  -> le nom d'un identificateur
IF                  -> début d'une condition si
LABEL               -> label permettant un saut
LOGICAL             -> booléen
NUMBER              -> nombre
OWN                 -> déclaration statique
PARAMS              -> liste de paramètres
PARAMS_SEG          -> segment de paramètres
PARAM               -> un paramètre
STEPUNTIL           -> boucle à pas fixe
STRING              -> chaine de caractères
SUBSCRIPT           -> début déclaration d'un tableau
TYPE                -> type (entier, booléen ou réel)
VAR                 -> variable
WHILE               -> boucle tant que
*/
```

3.2 Généralités

L'arbre abstrait à été construit de manière à être facile à analyser en Java, ainsi nous avons rajouté un maximum d'étiquettes permettant de nous situer au sein des déclarations tout en supprimant la plupart des noeuds inutiles, notamment dans les dérivations arithmétiques et booléennes.



(a) Arbre d'analyse



(b) Arbre abstrait

FIGURE 1 – Exemple d'analyse syntaxique et de génération de l'arbre abstrait associé

4 Table des symboles

La table des symboles comprend toutes les déclarations de variables, procédures, tableaux, switchs et labels. Elle est construite comme un arbree comportant des portées comme noeuds et feuilles, par conséquent elle est construite récursivement.

Pour cela elle utilise notre parcours d'AST récursif mis en place précédemment qui permet de facilement accéder à l'ensemble des informations des noeuds avec des noms clairs et précis (et non plus les indices de antlr). Chaque portée possède un nom unique qui est son chemin dans l'arbre, un type pour préciser si c'est un bloc, une procédure ou le chargement du programme, à cela s'ajoute la liste des noeuds fils ainsi que la liste des déclarations de cette portée.

Chaque déclarations peut donc être de l'un des cinq types précédent, chacune ayant sa propre spécificité. Cependant dans un souci de structure, elle hérite tous d'une structure commune qui va regrouper :

— L'identificateur unique
— Le type primaire (Booléen, entier, flottant, label, chaîne de caractère ou aucun)
— Le genre permettant facilement de cast sur le sous type de déclaration
— La ligne et le caractère où est défini l'identificateur,
— Le déplacement
— La taille de la donnée (soit 4 pour les flottants sinon 2),

La table contient aussi dans la toute première portée, celle du programme, l'ensemble des fonctions disponibles nativement, à savoir outinteger, outreal, outstring permettant les sorties, de même la fonction heapInfo permet d'obtenir des informations sur l'utilisation actuel du tas (nécessaire pour les tableaux dynamiques).

Lors de la génération de la table, une erreur peut être levée si deux déclarations dans une même portée possède un même identifificateur.

Afin de visualiser plus facilement la structure, il est possible d'utiliser la commande : `./gradlew DisplayTable`. Voici un exemple d'une telle table :

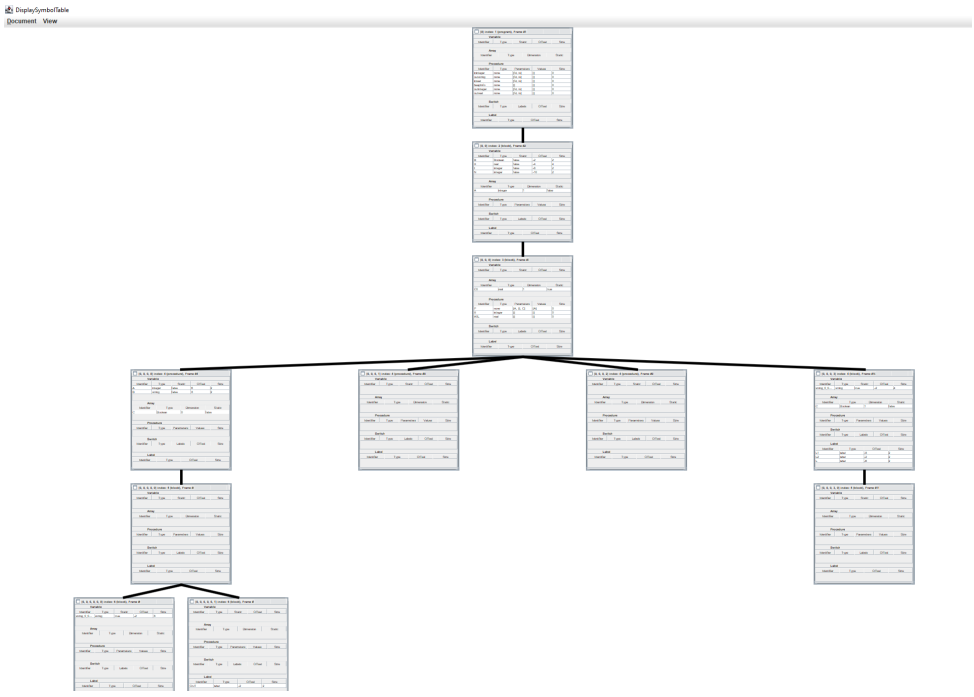


Figure 2 – TDS

5 Contrôles sémantiques

5.1 Généralités

Pour cette partie, le groupe a convenu de couvrir l'intégralité des contrôles sémantiques spécifiés par le *Modified report on the Algorithmic Language Algol 60*. Ce document sert donc de référence pour les choix a effectuer et les contrôles à implémenter durant l'intégralité de cette partie du projet.

5.2 Structure de données

Afin de pouvoir travailler efficacement durant le reste du projet, le groupe a convenu de commencer par implémenter une structure de données efficace.
L'AST fourni par ANTLR est donc analysé afin de créer une structure d'arbre en Java où chaque noeud est une instance d'une classe représentant ses propriétés.
Afin de pouvoir efficacement parcourir cet arbre, un design pattern de type *visitor* est mis en place.

5.3 Principe de travail

Devant la nature du problème et de la structure de données crée, nous avons convenu que la grande majorité des contrôles sémantiques seront effectués de manière récursive. Il faut alors prêter une attention particulière à l'interopérabilité des contrôles écrits par les différents membres du groupe.

5.4 Optimisations effectuées

Durant cette partie, nous avons décidé d'implémenter une optimisation de compilation appelée le *constant folding*. Cette opération revient à calculer lors de la compilation l'intégralité des opération arithmétiques ou booléennes dont la valeur est connue à la compilation. De cette manière, la contrainte d'arrondis spécifiée à la section 4.2.4 du *Modified report on the Algorithmic Langauge Algol 60* peut être traitée pour tous les cas statiques.
De plus, ce traitement permet d'éliminer du code non atteint dans certains branchements *if*. Grâce à ce traitement, les deux programmes suivants résultent en le même code assembleur.

CODE SOURCE 6 – Exemple de programme avant constant folding

```
/*
begin
    real x;
    Boolean b;
    x := (1 + 2) * 3;
    b := (1 <= 2) => (true /\ false);
    x := if (true \/ false) then 1 + 1 else -2;
    if (false <=> false) then
        outstring(1,'THEN')
    else
        outstring(1,'ELSE')
end
*/
```

CODE SOURCE 7 – Programme équivalent optimisé avec constant folding

```
/*
begin
    real x;
    Boolean b;
    x := 9;
    b := false;
    x := 2;
    outstring(1,'THEN')
end
*/
```

Toutefois, cette optimisation est en fait présente sous la forme d'un pré traitement et nécessite donc un parcours de l'AST supplémentaire. Elle peut légèrement rallonger la durée de compilation.

5.5 Restriction

Lors de cette partie, nous avons décidé de restreindre les instructions *goto* à des labels déclarés dans le bloc courant.
Cette restriction permet plus de simplicité dans la génération du code associé à un mot-clé *goto* et fera l'objet d'une possible amélioration du compilateur.

6 Génération de code

6.1 Généralités

Dès le début de cette partie, nous avons décidé de travailler sur le long terme afin de minimiser la quantité de code à modifier lors du passage au niveau suivant.
Ce faisant, nous avons décidé de construire l'intégralité de la génération de code en accord avec la structure nous permettant de gérer les nombre réels. De la même manière, la génération de code relative aux appels de fonction est directement pensée pour pouvoir supporter les fonctions récursives.

En terme d'organisation chronologique, nous avons commencé par implémenter l'allocation des registres, les opérations arithmétiques simples, les zones de liaison et les accès et stockages de variables. Le reste des fonctionnalités à été implémenté une fois que l'ensemble de ce travail réalisé.
Une librairie en assembleur regroupant plusieurs fonctions utiles (affichage des nombres, gestion des réels) a été développée en parallèle et intégrée au fur et à mesure de son avancement. On peut la trouver dans `asm/lib.src`, et le code assembleur généré par le compilateur y est par la suite intégré pour produire le `src` final.

6.2 Allocation de registres

Les registres sont alloués au fur et à mesure de la compilation, un registre est immédiatement marqué comme libre lorsqu'il n'est plus nécessaire. Afin de faciliter la gestion des registres qui sont au nombre seulement de 16, nous avons écrit une classe RegisterAllocator, celle ci définit plusieurs registres spéciaux, notamment on a R14 pour le base pointer, R15 pour le pointeur de pile (stack pointer), R0 pour le retour de fonction entière, R1 pour le retour de fonction réelle, R2 comme registres temporaire pour toutes les opérations fixes et le reste est alloué (de R3 à R13) pour la gestion au cours de la compilation. Sachant qu'en moyenne seulement 4 registres sont utilisé au même moment dans un bloc ou sous bloc

donné, les registres R3 à R9 peuvent aussi être utilisé avec l'optimisation des registres fixes comme registre qui transmettent l'information en dehors d'un bloc, afin de garantir l'intégrité des registres, il est nécessaire de faire un coloriage de graphe avec les durée de vies des registres. Cette fonctionnalité est implémentée avec l'algorithme de Chaitin et peut être activé lors de la compilation avec l'option `-opti`. Afin de mieux observer l'attribution de registre la commande `./gradlew DisplayGraph` permet de visualiser cette attribution.

L'allocation générale des registres se faisant par étape, il est nécessaire de garder en mémoire quel registre est utilisé à quel endroit, c'est pour cela que dans notre visiteur pour la compilation (AsmCompilerVisitor), nous gardons les registres entier et flottant dans des variables globale, et nous faisons attention à les relâcher une fois que leur résultat a été utilisé.

6.3 Zone de liaison

Une zone de liaison est placée entre chaque environnement de la pile et permet de garantir le bon fonctionnement des blocs et procédures, notamment récursives. Toute zone de liaison est consitutée de trois mots (six octets) :

— Une adresse de retour permet au système de poursuivre l'exécution du programme
— Un lien statique permet au compilateur d'identifier les symboles sans erreurs
— Un lien dynamique permet au système de chaîner les environnements de la pile

Un schéma de la zone de liaison est présenté figure 1 (page 10).

...
chaînage statique
chaînage dynamique
adresse de retour
...

TABLEAU 1 – Schéma d'une zone de liaison

Pour simplifier la génération de code, nous avons traité les blocs et le procédures de façon similaire. Les blocs sont considérés comme des procédures sans paramètres, sans retour, et sans adresse de retour. À l'entrée d'un bloc, l'adresse de retour est remplacée par un rembourrage de deux octets, car le flot d'exécution d'un bloc est connu à la compilation : utiliser des instructions « JSR » et « RTS » n'est pas nécessaire. Pour une procédure, l'adresse de retour est empilée par le micro-code du processeur pour l'instruction « JSR ». La valeur du compteur ordinal est alors sauvegardée pour poursuivre l'exécution au retour de la procédure.

Le lien dynamique est toujours égal à la base de l'environnement courant.

Le lien statique est composé en deux étapes :

— Pour chaque appel de procédure, la différence de profondeur entre les environnements appelant et appelé est calculée, l'adresse du lien est calculée puis copiée dans le registre spécial « R0 ».
— À l'exécution de la procédure, le lien statique empilé est égal à la valeur de « R0 ».

Ce procédé est obligatoire car le calcul du lien statique dépend de l'appelé et de l'appelant.

6.4 Appels de fonctions

L'organisation de la pile lors de l'appel d'une fonction s'est effectuée conformément aux acquis du module de traduction selon le schéma suivant :

...
variables locales
registres sauvegardés
adresse de retour
paramètres
...

TABLEAU 2 – Schéma simplifié d'une pile

6.5 Génération des étiquettes

= Les étiquettes générées automatiquement sont créés à partir des identifiants uniques des noeuds traités en Java. Ainsi le code généré pour une condition `if` suivra le schéma suivant :

CODE SOURCE 8 – Exemple d'implémentation de if

```
        /* condition de saut */
            /* code exécuté si la condition est vraie */
            jmp #ifend_1e965684 - $ - 2
        else_1e965684
            /* code exécuté si la condition est fausse */
        ifend_1e965684
```

L'identifiant du noeud `if` traité est donné par la représentation hexadécimale de son hashcode : `Integer.toHexString(node.hashCode())`.

Le même principe est employé pour les boucles `for`. Cela est également utilisé pour l'appel de fonctions, mais aussi l'affichage de chaînes de caractères spécifiques lors de la gestion d'exceptions :

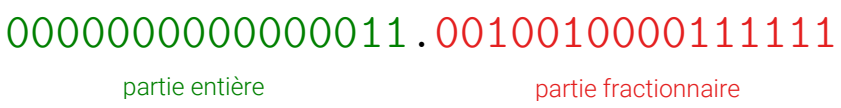
CODE SOURCE 9 – Exemple d'une gestion d'exception, ici une division par zéro

```
        // STATIC ZONE
          str_39aeed2f
          STRING "/!\ Division by zero at line 14, character 8"

        main
        /* Code généré pour la division à la ligne 14, caractère 8 */
            /* si le diviseur est nul */
                // Aficher la chaîne de caractère à l'adresse str_39aeed2f
                jsr @exit_
```

6.6 Gestion des réels

Les réels sont représentés par des nombres à virgule fixe sur deux registres :

FIGURE 3 – Représentation de 3,1415863



Cela rend la conversion entiers / réels plus simple qu'avec une représentation à virgule flottante.

Les fonctions de bases permettant de manipuler les réels ont été écrites dans la librairie :
— La négation (renvoie l'opposé d'un réel)
— L'addition
— La multiplication : cette opération a nécessité l'écriture d'une subroutine appelée `big_mul`, permettant de stocker sur deux registres le résultat d'une multiplication de deux registres, ce qui permet de récupérer les éventuels débordements de la multiplication. Cela a été rendu possible en effectuant plusieurs multiplications quartet par quartet.
— La division : le résultat est obtenu en effectuant une recherche dichotomique du quotient. Initialisé à zéro, les 32 bits du résultat sont modifiés du plus fort au plus faible de sorte que la multiplication du résultat avec le diviseur soit toujours plus proche de la dividende.
— L'affichage : il suffit d'afficher la partie entière (déjà implanté) puis la partie fractionnaire. Celle-ci utilise la subroutine `big_mul` pour effectuer des multiplications successives par 10.
Exemple pour afficher 0.142 :
- $0.142 \times 10 = 1.42 \Rightarrow$ on affiche 1
- $0.42 \times 10 = 4.2 \Rightarrow$ on affiche 4
- $0.2 \times 10 = 2.0 \Rightarrow$ on affiche 2
- partie fractionnaire nulle $\Rightarrow$ terminé
— L'égalité
— La comparaison *strictement supérieur*
— La comparaison *supérieur ou égal*

Lors de la génération de code, le type des expressions générées est constamment vérifié. Dans le cas des réels, un registre supplémentaire est utilisé afin de stocker la partie fractionnaire.

6.7 Gestion des tableaux

Les tableaux sont gérés dynamiquement par conséquent lors de la compilation il n'est pas connu les bornes supérieures et inférieures, ni la taille totale du tableau, seul le type et l'opération effectuée sont connues. Cependant comme à la compilation il est connu le nombre de tableaux déclarés, il est possible de construire une table statique permettant de retrouver facilement la position d'un tableau lorsqu'il est demandé. Cette table contient deux entiers qui sont le nombres de tableaux déclarés et la position actuelle du dernier octet du dernier tableau alloué, à la suite se trouve l'ensemble des labels des tableaux qui peuvent être déclaré avec chacun 4 entiers associés donnant le type (connu statiquement), le nombre de dimensions (connu statiquement), la position dans le tas du tableau (connu dynamiquement) et la taille du tableau (connu dynamiquement).

La gestion est donc effectué grâce à trois fonctions définies dans notre librairie, `allocate_array`, `set_var_array` et `get_var_array`, pour permettre une meilleure gestion de la mémoire, une fonction `heap_information` (pouvant être appelée avec `heapInfo`) est définie récapitulant toutes les déclarations de tableaux, leur taille et leurs bornes respectives.

La fonction `allocate_array` permet d'allouer un tableau, elle est appelé lors de la déclaration d'un nouveau tableau avec un label qui est connu lors de la compilation, qui pointe donc vers la zone statique, un label vers un message d'erreur statique permettant de donner la ligne et le caractère exact lors d'une erreur dynamique et l'ensemble des valeurs des bornes (elles sont dynamique et sont résolus lors de l'exécution). La fonction alloue la taille du tableau suivant les formules standard, les tableaux étant compris comme borne inférieure comprise et borne supérieure exclue (donc arr[0 :100,0 :100] fait bien 100*100=10000 octet

si réél ou le double si flottant)

La fonction `set_var_array` permet de mettre une valeur dans le tableau à la case correcte, pour cela les différents index pour chaque dimensions sont empilés ainsi que le label vers la zone statique déclarant le tableau afin de pouvoir récupérer l'endroit dynamique ou devra être stocké la variable donnée, si les index sont en dehors des bornes du tableau dynamique alors une erreur dynamique est levée

La fonction `get_var_array` permet de récupérer la valeur du tableau aux index donnés, si les index sont en dehors des bornes du tableau dynamique alors une erreur dynamique est levée.

Il n'y actuellement pas de protection du tas, c'est à dire que si nous déclarons trop de tableaux, il est possible de commencer à écrire dans la pile ou même sortir de l'espace d'adressage, de plus il n'y a pas de garbage collector donc si dans une boucle nous déclarons à nouveau un tableau, nous perdons de la place indéfiniment (la méthode free n'a pas été implémentée ni la gestion de la mémoire totale nécessaire au fragmentation possible)

6.8 Bootstrapping

Une fois les principales fonctionnalités du compilateur implémentées, nous avons pu utiliser des techniques de *bootstrapping* afin d'implémenter des fonctionnalités particulières.
En effet, les calculs de puissances (pour toute combinaison de réels et d'entiers), d'exponentielles et de logarithme ont été implémentés grâce au code généré par le compilateur et ajoutés à la librairie.
Le calcul d'exponentielle s'effectue grâce à la serie $e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$
Le calcul du logarithme s'effectue grâce à la série : $ln(x) = 2\sum_{k=0}^{\infty} \frac{1}{2k-1}(\frac{z-1}{z+1})^{2k+1}$
Pour ces 2 séries, l'utilisation des 15 premiers termes fournit une précision suffisante.

Toutefois, le code compilé n'étant pas parfaitement optimisé, les performance des fonctionnalités implémentées par bootstrapping peuvent être inférieures.

6.9 Niveau atteint

Lors de la génération de code, nous avons implémenté toutes les fonctionnalités nécessaires à la validation des niveaux 1 à 4.
En plus de ces fonctionnalités, nous avons implémenté la gestion des nombre réels pour l'intégralité des opérations, les optimisations de registres fixes, le calcul d'exponentielles, de logarithmes ont été traités.

6.10 Améliorations possibles

Notre compilateur n'intègre pas certaines fonctionnalités simples et complexes.

Le mot clé « own » n'est pas utilisé pour la génération de code car nous n'avons pas eu le temps de développer la fonctionnalité. En revanche, il est assez simple de modifier le compilateur pour accepter ce mot clé : en ajoutant un cas particulier pour les méthodes « getVar » et « setVar » du compilateur, il suffit de conserver de telles variables sur un segment de mémoire statique gérée par le compilateur, et non sur la pile.

Les sauts « go to » ne permettent actuellement pas de quitter la région courante. Ce choix permet de simplifier la réalisation de cette instruction. En effet, quitter la région courante nécessite de corriger l'état de la pile (dépilage des zones de liaison, variables locales, ...).

Pour effectuer ces opérations, il faut déterminer dynamiquement le nombre d'environnements sautés et intégrer dans le code généré des repères pour garantir l'intégrité de la pile après le saut.

Le passage par nom de paramètres n'a pas été réalisé car les membres du groupe n'ont pas trouvé le temps d'étudier cette méthode originale et peu utilisée. Une alternative possible serait d'utiliser un passage par pointeurs, mais cette méthode de respecte évidemment pas la spécification.

Le passage d'étiquettes et de procédures dans les procédures n'a pas été traité par manque de temps.

7 Jeux d'essais et tests

7.1 Généralités

Afin de s'assurer du bon fonctionnement de la grammaire, l'ensemble du groupe a adopté, dès le début du projet, une gestion des tests assidue et incrémentale.

7.2 Grammaire et AST

A chaque nouvelle fonctionnalité ajoutée à la grammaire, des tests ont été effectués afin de s'assurer de son bon fonctionnement et de la compatibilité avec le reste du travail déjà effectué.
Une fois la grammaire et l'AST finis, chaque membre du groupe a écrit un programme complet visant à vérifier le bon fonctionnement de l'ensemble.

7.3 Contrôles sémantiques

Les contrôles sémantiques ont donné lieu à une série de tests assidus et répétés afin d'éviter les faux positifs ainsi que les faux négatifs. Pour s'assurer de l'efficacité des tests, ceux-ci ont été conduits sous le principe de l'évaluation par les pairs. Les membres du groupe se sont divisé en 2 groupes de 2 et chacun s'est vu assigner la responsabilité de tester les contrôles implémentés par son binôme.

7.4 Génération de code

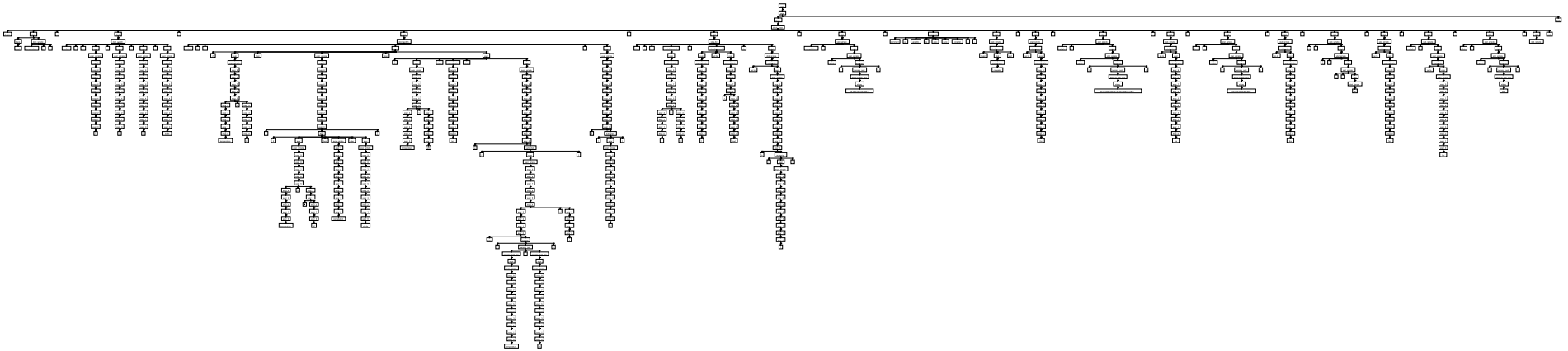
Cette partie du projet s'est révélée être la plus délicate à tester. En effet, la nature du travail rend les erreurs difficiles à traquer et encore plus à corriger.
La correction d'erreur à donc été une tâche collective, suscitant une grande quantité de travail.
En effet, aucune méthode particulière n'a pu être mise en place afin de faciliter le contrôle de qualité de cette partie du projet.

7.5 Exemples

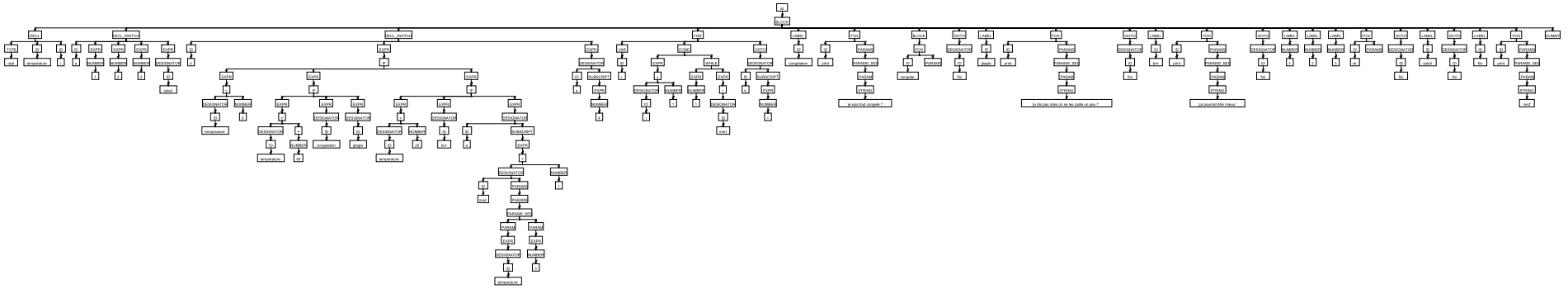
CODE SOURCE 10 – Exemple d'un programme de test écrit en ALGOL60

```
/*
begin
    integer i;
    real x, y;
    real a, b;

    real procedure ReMul(a1, b1, a2, b2);
    value a1, b1, a2, b2;
    real a1, b1, a2, b2;
    ReMul := a1*a2 - b1*b2;

    real procedure ImMul(a1, b1, a2, b2);
    value a1, b1, a2, b2;
    real a1, b1, a2, b2;
    ImMul := a1*b2 + a2*b1;

    for y := 1 step -0.125 until -1 do
    begin
        for x := -2 step 0.125 until 0.5 do
        begin
            a := x;
            b := y;
            for i := 1 while i < 50 /\ a*a+b*b < 4 do
            begin
                a := ReMul(a, b, a, b) + x;
                b := ImMul(a, b, a, b) + y;
                i := i + 1;
            end;
            outchar(1, ' #', if i < 50 then 1 else 2);
            outchar(1, ' #', if i < 50 then 1 else 2);
        end;
        outstring(1, '')
    end
end
*/
```

En utilisant ce programme en entrée, on obtient l'arbre d'analyse et l'arbre abstrait suivants :



(a) Arbre d’analyse



(b) Arbre abstrait

FIGURE 4 – Arbre d’analyse syntaxique et abstrait

8 Gestion de projet

8.1 Généralités

Dès le début du projet, les rôles ont été attribués et une charte de projet (disponible en annexe) a été réalisée afin d'officialiser l'ensemble des décision organisationnelles et techniques prises lors de la première réunion. Entre autres, le groupe a convenu de se regrouper tous les mercredi sauf impossibilité ponctuelle.

Lors de la crise sanitaire menant à la fermeture des écoles puis au confinement généralisé, le groupe s'est accordé une semaine sans réunion et sans travail imposé afin de permettre à chacun de s'adapter aux modalités de travail à distance et de vie quotidienne confinée. Suite à cette période d'adaptation, le groupe s'est réuni tous les lundi via le serveur Discord mis en place dès le début du projet.

8.2 Diagrammes de Gantt

8.2.1 Diagrammes de Gantt prévisionnels

Afin d'instaurer, dès le début du projet, un planning prévisionnel permettant une meilleure organisation du projet et une meilleur visualisation du travail à accomplir, nous avons réalisé un premier diagramme de Gantt.

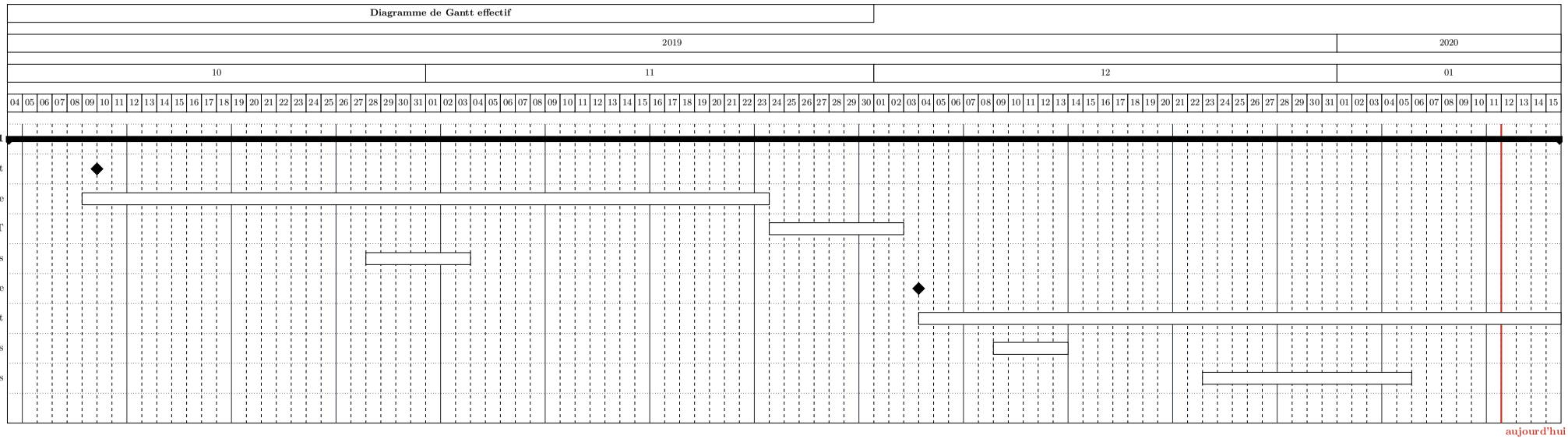


Diagramme de Gantt effectif
2019
2020
10
11
12
01
04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
Livrable PCL1
Début projet
Établissement grammaire
Construction de l'AST
Vacances scolaires
Présentation AST Grammaire
Réflexion TDS et rapport
Examens
Vacances scolaires
aujourd'hui

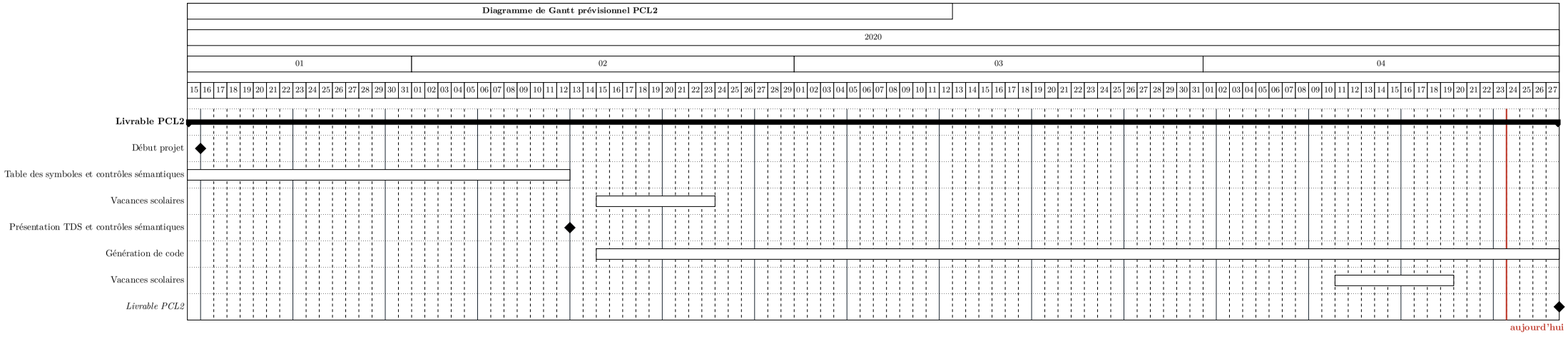


Diagramme de Gantt prévisionnel PCL2
2020
01
02
03
04
15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
Livrable PCL2
Début projet
Table des symboles et contrôles sémantiques
Vacances scolaires
Présentation TDS et contrôles sémantiques
Génération de code
Vacances scolaires
Livrable PCL2
aujourd'hui

8.2.2 Diagramme de Gantt effectif

Afin de permettre une meilleure visualisation de l'avancement effectif du projet et de mieux visualiser les modifications de planning ayant eu lieu, nous avons réalisé un second diagramme de Gantt. Cela a donc permi de comparer efficacement l'avancement réel avec l'avancement prévu.

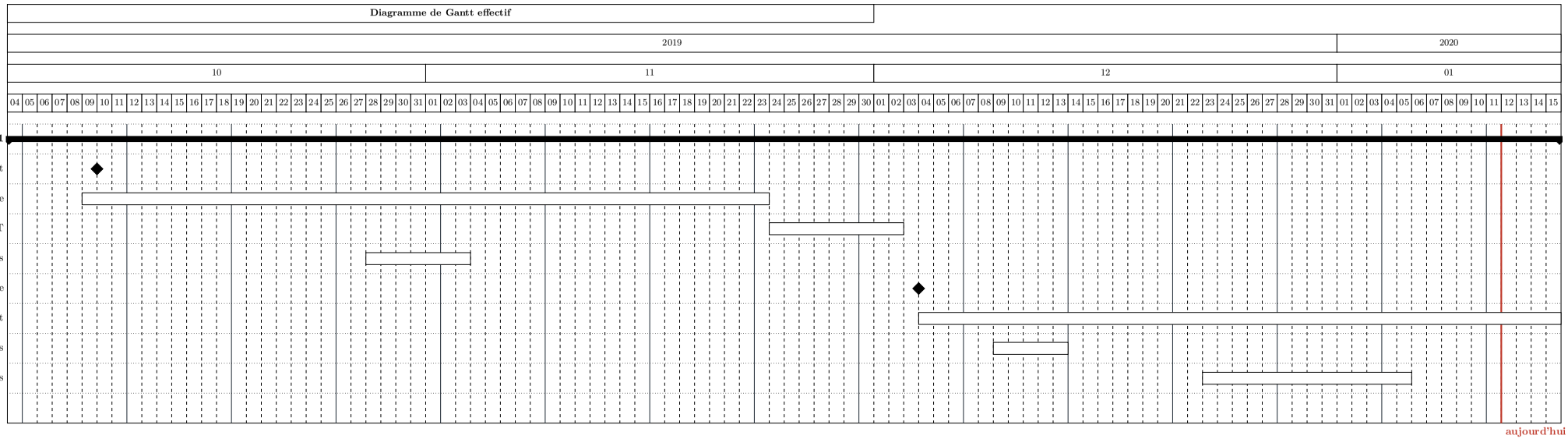


Diagramme de Gantt effectif
2019
2020
10
11
12
01
04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
Livrable PCL1
Début projet
Établissement grammaire
Construction de l'AST
Vacances scolaires
Présentation AST Grammaire
Réflexion TDS et rapport
Examens
Vacances scolaires
aujourd'hui

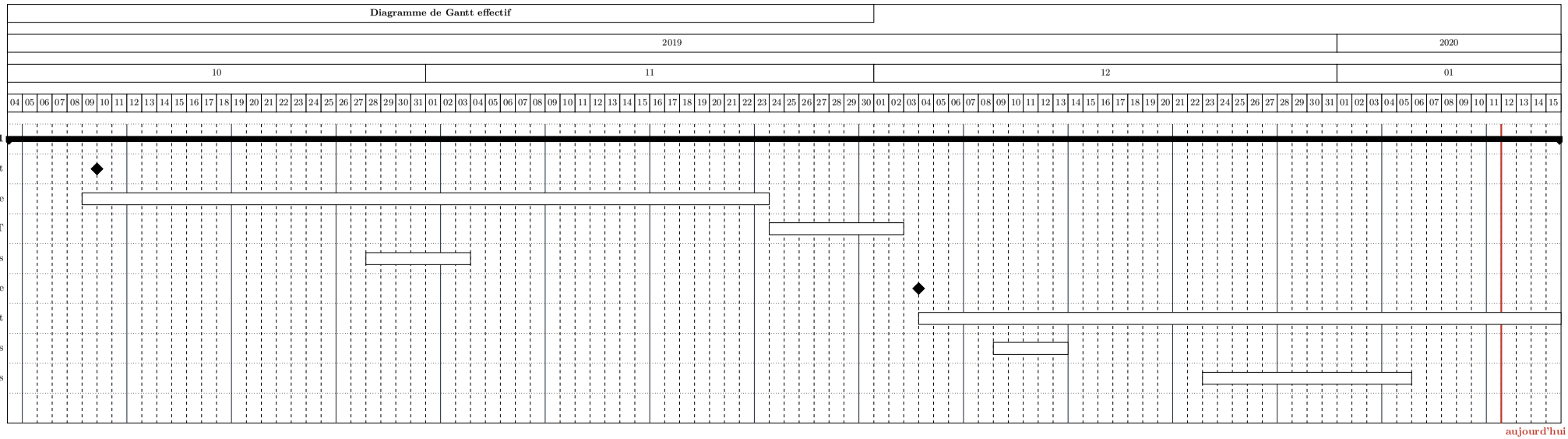


Diagramme de Gantt effectif
2019
2020
10
11
12
01
04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
Livrable PCL1
Début projet
Établissement grammaire
Construction de l'AST
Vacances scolaires
Présentation AST Grammaire
Réflexion TDS et rapport
Examens
Vacances scolaires
aujourd'hui

8.3 Matrice SWOT

	Positif	**Négatif**
Interne	Forces : Cohésion de l'équipe, bonne entente	Faiblesses : Manque d'expérience, organisation
Externe	Opportunités : Outils bien documentés, tutorats et soutien des professeurs, après-midis réservés aux projets	Menaces : Grammaire BNF longue, utilisation d'un langage ancien et non fini

FIGURE 5 – Matrice SWOT du projet

8.4 Analyse post-mortem

8.4.1 Analyse post-mortem individuelle

Locust2520

En plus d'avoir appris à créer un compilateur, le projet m'a permis de mieux m'organiser dans des plus gros projets que ceux auxquels nous sommes habituellement confrontés au cours de notre formation.
Je suis globalement satisfait de ce projet, et un peu déçu de ne pas avoir eu recours à un assembleur plus courant comme l'assembleur x86_64.

<Redacted>

La génération de code m'a permis d'appliquer en pratique des éléments de Traduction1 et de PFSI. Le projet a été complexe à réaliser et je trouve l'avancée de notre compilateur satisfaisante.

Erwan Kessler

Ce projet m'a permis de me familiariser avec la génération de code et l'assembleur MicroPIUPK. J'ai énormément apprécié faire les tableaux ainsi qu'optimiser les registres afin de les utiliser de la manière la plus efficace.
Pour moi ce projet est une réussite dans le fait qu'il m'a permis de comprendre les difficultés lors de la conception d'un langage et la nécessité d'une bonne spécification.

Yann Meyer

Ce projet m'a apporté beaucoup sur le plan technique et organisationnel. Sur le plan technique, j'ai, en effet, pu concrétiser les acquis du module de traduction et de PFSI. Sur le plan organisationnel, assumer le rôle de chef de projet sur un projet de cette envergure m'a permis de réaliser l'importance d'une bonne organisation et le degré d'investissement que cela requiert.

8.4.2 Répartition des heures PCL1

Tâches	Locust2520	<Redacted>	Erwan	Yann
Grammaire LL1	32h	40h	30h	28h
AST	5h	10h	10h	5h
Tests	4h	2h	2h	2h
Réunions/ Mise en commun	28h	28h	28h	28h
Gestion de Projet	3h	1h	3h	10h
Rapport	3h	2h	4h	3h
Total	75h	83h	77h	76h

8.4.3 Répartition des heures PCL2

Tâches	Locust2520	<Redacted>	Erwan	Yann
TDS	10h	1h	25h	1h
Contrôles sémantiques	19h	4h	5h	10h
Génération de code	65h	80h	60h	40h
Tests	8h	35h	15h	29h
Réunions/ Mise en commun	34h	34h	34h	34h
Gestion de Projet	3h	2h	3h	15h
Rapport	3h	4h	5h	7h
Total	142h	160h	147h	136h

8.4.4 Répartition des heures totales

Tâches	Locust2520	<Redacted>	Erwan	Yann
Le grand total	217h	243h	224h	212h

8.4.5 Analyse post-mortem collective

Dans son ensemble, le groupe est satisfait du travail réalisé et estime avoir produit un compilateur ALGOL60 répondant aux exigences du sujet.
Pour l'ensemble des membres du groupes, l'écriture de ce compilateur représente le plus long projet auquel nous avons pris part. Il s'agit donc d'une expérience riche, que ce soit sur le plan humain ou technique.

En effet, le niveau d'organisation et d'entente requis par un projet de ce type dépasse largement celui des projets de plus petite envergure auxquels nos avions déjà pris part.

De plus, ce projet s'est révélé être particulièrement formateur sur divers plans techniques : de l'utilisation d'*ANTLR* à une meilleure compréhension de l'assembleur, la variété de compétences techniques acquise est très large.

L'ensemble de ces éléments pousse l'intégralité du groupe à considérer ce projet comme une étape clé de leur formation d'ingénieur.