Université de Lorraine

Projet de compilation (Module PCL1)

Auteurs :

Julien GIET
Emilien JACQUES
Jean-Baptiste RENAULT
Arthur SCHMIDT

Superviseurs :

Sébatsien DA SILVA
Gérald OSTER
Suzanne COLLIN

Table des matières

1 Introduction

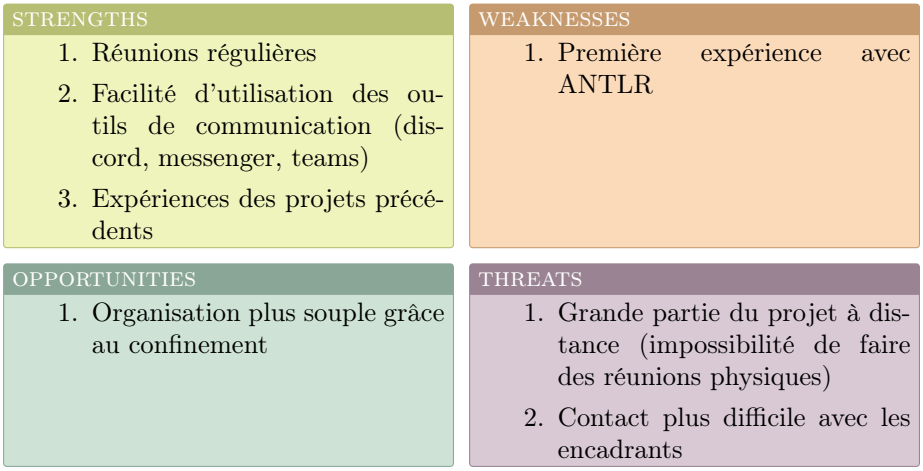
L'objectif global du projet est de créer un compilateur capable d'analyser un code écrit dans un langage spécifique (le langage BLOOD) et de produire un code assembleur en sortie.

Ce travail peut être divisé en grandes parties :

- écriture de la grammaire
- construction de l'arbre abstrait
- création de la table des symboles
- réalisation des contrôles sémantiques
- génération du code assembleur

2 Gestion de projet

2.1 Matrice SWOT



La matrice SWOT permet de mettre en évidence les points faibles et les points forts du groupe. Il faut essayer au maximum de capitaliser sur ses forces et de minimiser les faiblesses.

Dans notre cas, notre principal force est la bonne organisation du groupe. Les réunions sont régulières et efficaces. De plus, nous avons acquis de l'expérience grâce aux précédents projets et il faut essayer de s'en servir au maximum pour celui-là.

A contrario la plus grande menace pour le bon déroulement de ce projet est liée au fait qu'une grande partie de ce projet se déroule à distance en raison de la situation sanitaire. Cependant, le groupe a su s'adapter, notamment grâce aux outils de communication à distance. De plus, un projet ayant déjà été mené à distance l'année dernière, il a fallu se servir de cette expérience pour que celui-ci se déroule au mieux.

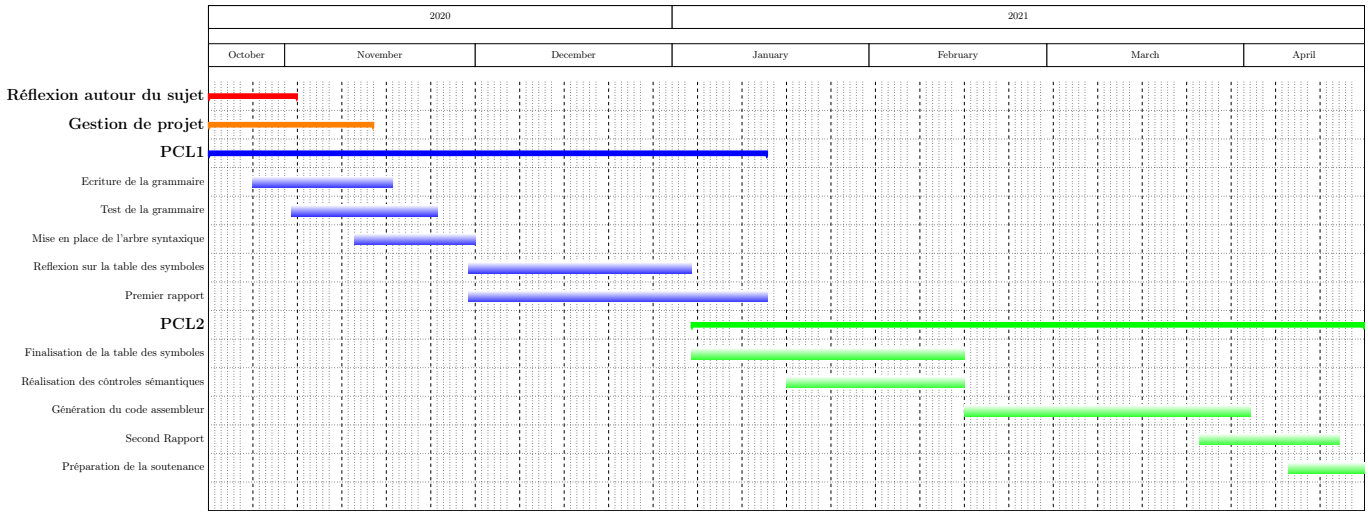
2.2 Matrice RACI

	Arthur	Emilien	Jean-Baptiste	Julien
écriture de la grammaire	RACI	RACI	RACI	RACI
rendre la grammaire LL(1)	ACI	ACI	RACI	ACI
test de la grammaire	RACI	ACI	ACI	RACI
construction de l'arbre abstrait	RACI	RACI	RACI	RACI
test de l'arbre abstrait	ACI	RACI	ACI	ACI
rédaction du rapport	RACI	RACI	RACI	RACI

La matrice RACI permet de situer le rôle de chacun dans les différentes étapes du projet, le R désigne le fait de réaliser la tache, le A d'en être responsable, le C d'être consulté et le I d'être informé. Le groupe étant composé de seulement 4 personnes et la communication étant aisée, tous les membres du projet sont informés et consultés pour chaque tâche. De plus, le A

(de Accountable) a été donné à chacun car chacun est responsable/concerné par chaque tâche pour la bonne avancée du projet.

2.3 Gantt



Ce projet a la particularité de se dérouler sur 2 semestres. Bien que ce rapport rende compte de la première partie, le diagramme de Gantt a été réalisé pour l'intégralité du projet dans le but d'avoir un maximum de visibilité.

Le diagramme de Gantt est un outil permettant la gestion temporelle du projet. Celui-ci permet de jalonner le travail à réaliser et de savoir à quel rythme le projet avance. Il est intéressant de noter à la cloture du projet le temps réél nécessaire à chaque étape et de comparer avec le Gantt initial. Cela permet de voir la différence entre le temps prévu et le temps réél pour chaque étape et d'acquérir de l'expérience à réutiliser dans de futurs projets.

Pour notre projet, toutes les parties semblent à peu près équivalentes en temps de travail. Cependant, il est parfois possible de réaliser ces actions simultanément pour avancer plus rapidement en répartissant les tâches.

3 Grammaire

Nous avons séparé la construction de notre grammaire en deux grandes étapes :

- Tout d'abord nous avons écrit les règles de la grammaire afin qu'elle accepte et reconnaisse entièrement le langage imposé par le sujet sans trop se préoccuper de l'aspect LL(1) de la grammaire.
- Nous avons ensuite modifié ces règles pour que la grammaire soit entièrement LL(1) afin de respecter les consignes du sujet.

Nous avons utilisé l'outil ANTLRWORKS dans ces deux étapes car ce logiciel est capable de vérifier les règles que nous avons implémentées et indique celles qui ne permettent pas d'avoir une grammaire LL(1) tant qu'il en existe.

3.1 Création de la grammaire

Afin de construire cette grammaire, nous avons tout d'abord implémenté les deux règles de base :

- Une première régle sert à avoir une grammaire augmentée (cette règle est indispensable au bon fonctionnement d'ANTLR).
- La deuxième impose la syntaxe générale du langage : il faut un ensemble (éventuellement vide) de définition de classes puis un bloc.

Nous avons ensuite mis en place les règles relatives aux définitions de classes :

- La composition de la définition de la classe
- La composition du bloc de classe
- La définition d'un constructeur
- La définition d'une méthode
- La déclaration d'une variable dans la classe (attribut/statique ou non, avec initialisation ou non)

Nous avons alors implémenté les règles concernant les expressions, en définissant :

- les opérateurs de comparaison : =, <, >, <>, <=, >=
- les opérateurs binaires (avec leur ordre de priorité) : +, -, *, /, &
- la structure atomique d'une expression : instanciation, expression paranthèsée, Entier (signé ou non), Chaine de caractère, identifiant, identifiant de classe et expression "as".
- les règles relatives aux sélections et envois de messages.

La dernière étape a donc consisté en l'implémentation des règles relatives aux instructions :

- Définition d'un bloc
- Définition d'un "return"
- Définition de l'instruction "if"
- Définition de l'instruction "while"
- Définition d'une affectation
- Définition spéciale : expression + " ;"

Une fois ces étapes terminées, nous avions une grammaire qui reconnaissait le langage du sujet mais dont certaines parties n'étaient pas LL(1) : dans certains cas l'analyseur n'arrivait pas à décider quelle règle appliquer avec un seul token d'avance en lecture.

3.2 Modifications pour la rendre LL(1)

Tout d'abord, il nous a fallu identifier les parties de notre grammaire qui n'étaient pas LL(1), l'outil ANTLRWORKS (lorsque l'on met l'option k=1) nous a permis d'avoir une visualisation rapide de ces problèmes : les règles ayant des alternatives sont soulignées en rouge et l'onglet "syntax diagram" permet de voir quel est le chemin emprunté pour arriver au moment où l'analyseur n'arrive plus à se décider avec un seul token d'avance.
Ainsi nous avons pu constater que la partie définition de classes était LL(1) et ne posait pas de problème, tandis que la partie instructions/expressions n'était pas LL(1).

Afin de résoudre ces problèmes, il nous a fallu factoriser à gauche certaines règles :

- Un bloc peut commencer par une instruction ou par une déclaration, or certaines instructions et certaines déclarations commencent de la même manière (ID,CLASSID,...). Nous avons ainsi dû factoriser cette expression en faisant la dichotomie des différents cas possibles. Nous avons également eu besoin de différencier les expressions atomiques commençant par un ID ou CLASSID de celles ne commençant pas ainsi afin de pouvoir les factoriser dans le bloc. (A noter que ce problème ne se posait pas dans le bloc de classe car une déclaration y commence par le mot clef "var" contrairement à une déclaration dans un bloc classique).
- Il nous a également fallu factoriser par le token '(' dans les règles atomiques car l'analyseur n'arrivait pas à déterminer s'il fallait appliquer la règle atom -> '(' expression ')' ou atom -> '(' 'as' CLASSID ' :' expression ')' lorsqu'il tombait sur un symbole '('.

Une fois notre grammaire LL(1), il ne nous manquait plus qu'à créer l'AST correspondant pour terminer cette première partie du projet compilation.

4 Arbre Syntaxique Abstrait

Une fois la grammaire LL(1), il nous a été assez facile de construire un AST (essentiellement grâce aux indications/instructions présentes dans le TP de Traduction qui y était consacré).
Ainsi nous avons en priorité écrit les tokens correspondant aux opérateurs binaires des expressions.
Nous avons ensuite créé un certain nombre de tokens imaginaires ayant pour but d'avoir une visualisation de certaines instructions/expressions : IF,WHILE,BLOCK,CLASS,...

La plupart de ces tokens se comprennent facilement grâce à leur nom mais certains sont moins évidents que d'autres :

- INSTEXPR désigne une instruction du type : expression ';'
- ATOMID contient une suite de sélections/messages appliqués à un identifiant, un identifiant de classe, un entier ou une chaîne de caractères.
- SIGNEDINTEGER désigne un entier signé, c'est à dire précédé d'un '+' ou d'un '-' unaire. (ceci a été fait après la soutenance suite à une remarque de Mme COLLIN à ce sujet).
- THEN/ELSE : nous avons rajouté ces tokens à l'intérieur du token IF afin de faciliter l'étape de lecture de l'AST. (cela a également été réalisé après la soutenance suite à une remarque de Mme COLLIN à ce sujet).
- Les tokens LOCALVARIABLEINIT/VARIABLECLASSEINIT qui décrivent respectivement une déclaration avec initialisation dans un bloc normal ou dans un bloc de classe. (A l'origine, on scindait une déclaration avec initialisation en une déclaration immédiatement suivie d'une affectation, suite à des remarques de Mme COLLIN à ce sujet, nous avons créé ces tokens afin de faciliter la compréhension de l'AST quand il sera parcouru de manière automatique).

5 Test de l'arbre abstrait

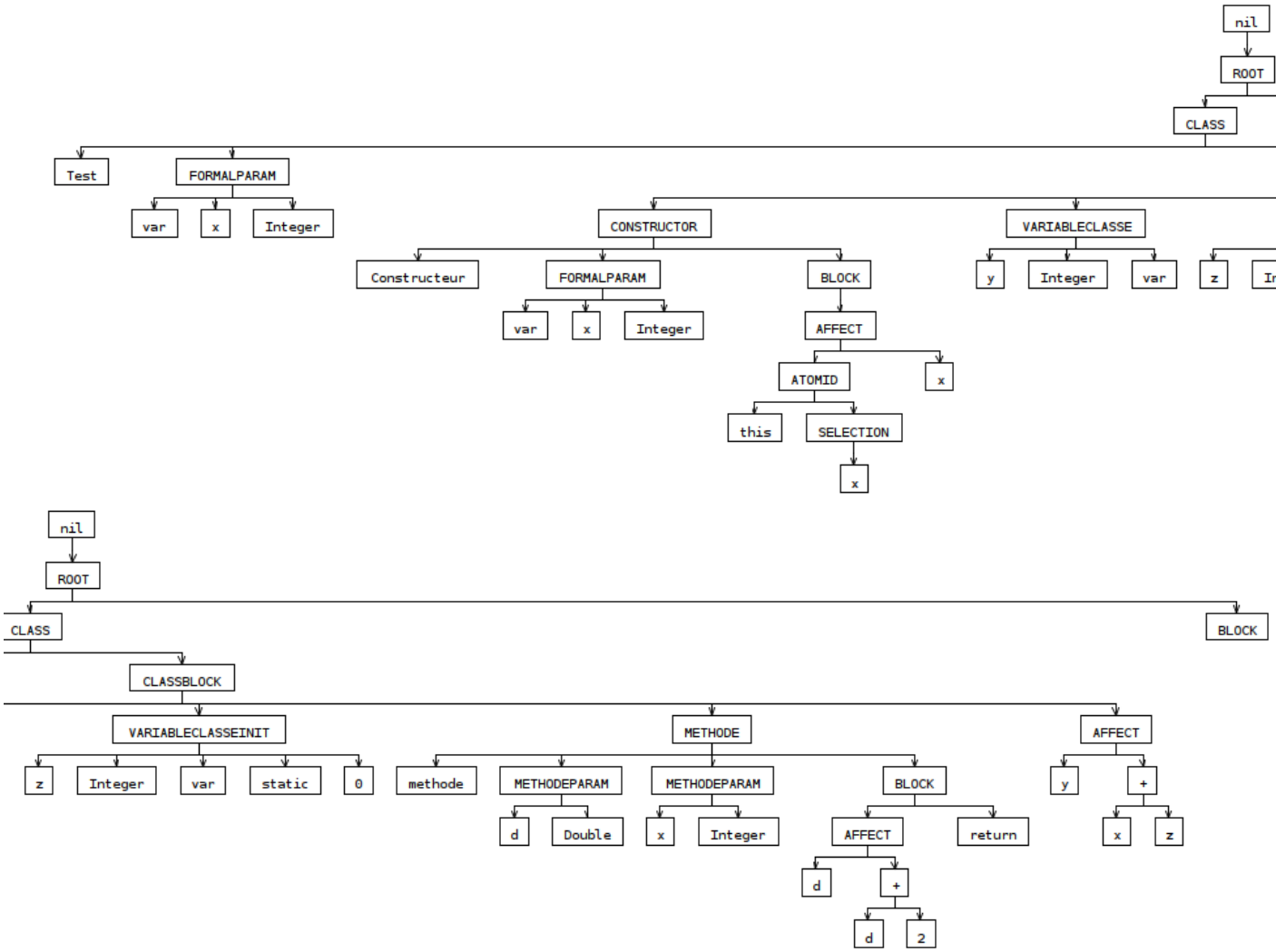
Nous avons fait le choix d'illustrer notre AST à l'aide des jeux de tests présentés lors de la soutenance. Entre temps, ils ont été légérement modifiés pour tenir compte des remarques faites lors de cet entretien. Ils couvrent assez bien les différentes lignes de code reconnues par notre grammaire.

5.1 Class

Ce code montre la composition d'une classe. Elle est déclarée par le mot clé class, a un nom commençant par une majuscule, peut contenir un ou plusieurs paramètres, suivi du mot clé is et d'un block de classe. Dans celui-ci, on peut y trouver des déclarations de variables (mot clé var), des affectations et des déclarations de constructeur et de méthodes (mot clé def). Ces 2 dernières structures sont similaires et ne se distinguent que par la majuscule débutant le nom du constructeur. On y retrouve des paramètres (optionnnels), le mot clé is ainsi qu'un block de classe contenant des instructions et éventuellement un return. De plus, seule une méthode peut être déclarée static.

```
class Test (var x : Integer) is {

    var y : Integer;
    var static z : Integer := 0;

    y := x+z;

    def Constructeur (var x : Integer) is {
        this.x := x;
    }
    def methode (d : Double, x : Integer) is {
        d := d + 2;
        return;
    }

}

{

}
```

Voici l'arbre en résultant (étant assez large, il a été découpé en 2 parties) :

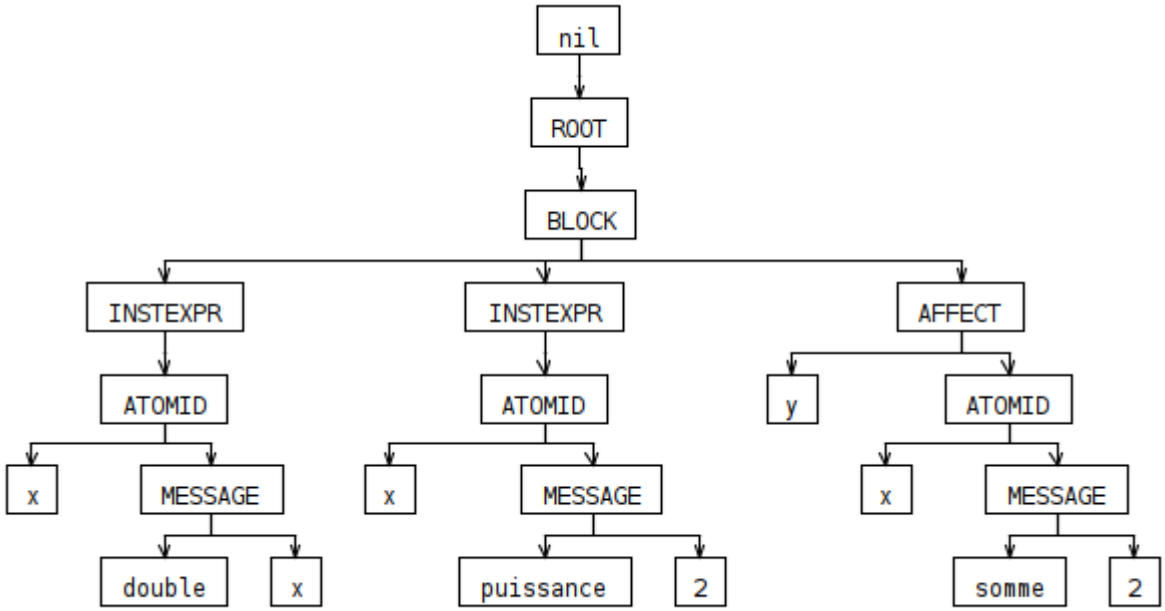


5.2 Envoi de messages

Dans le bloc principal, on peut faire appel à des méthodes définies. Pour cela, on utilise le point. On appelle la méthode (pouvant avoir ou non des paramètres, éventuellement de différents types) à droite du point et l'objet sur lequel appliquer cette méthode à gauche. Le résultat de cette méthode (s'il existe) peut ensuite faire partie d'une affectation.

```
{
x.double(x);
x.puissance(2);
y := x.somme(2);
}
```

Voici l'arbre des différentes manières d'envoyer un message :



5.3 Boucles If et While

Ce code présente les boucles if et while qui peuvent très bien être imbriquées les unes dans les autres.

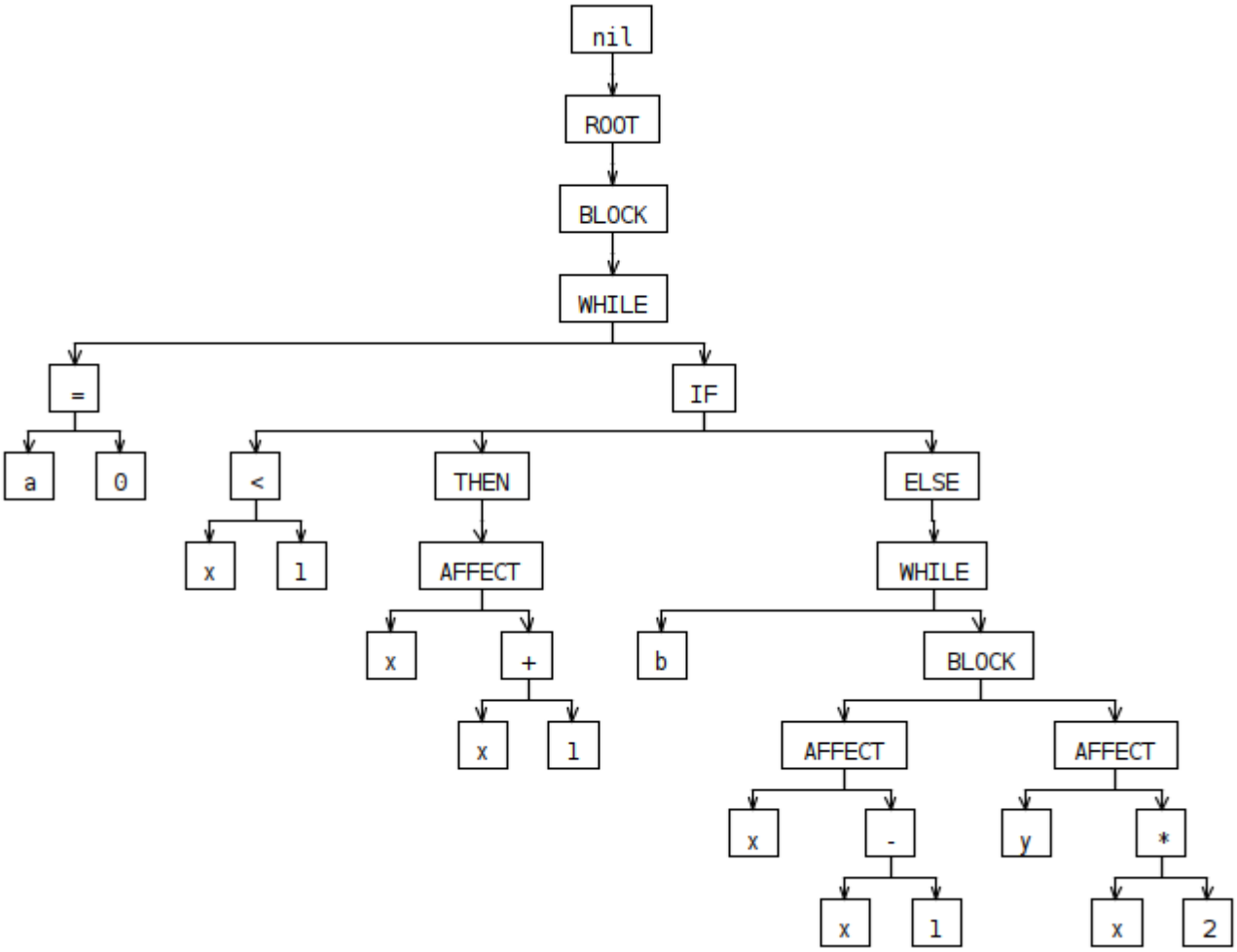
La boucle while a pour mots clés while et do. Le while est suivi d'une expression pouvant être vraie ou fausse (cela peut aussi être un entier) puis le do est suivi d'un bloc contenant les instructions à effectuer.

Le test if a pour mots clés if, then et else dans cet ordre. Le if est suivi d'une expression (ou d'un entier) à tester. Si c'est vrai, les instructions à effectuer sont après le then sinon elles sont après le else.

Les espaces, retours à la ligne et tabulations sont là pour avoir une mise en page plus claire mais ne changent rien dans la grammaire.

```
{
while a = 0 do
    if x < 1 then
        x := x + 1;
    else
        while b do {
            x := x - 1;
            y := x * 2;
        }
}
```

Voici le code des 2 boucles while et du test if imbriqués les uns dans les autres :



5.4 Calcul et priorité opérationnelle

Ce code présente un bloc de calcul. On peut voir que la déclaration de classe n'est pas obligatoire mais le block (matérialisé par les accolades) si.

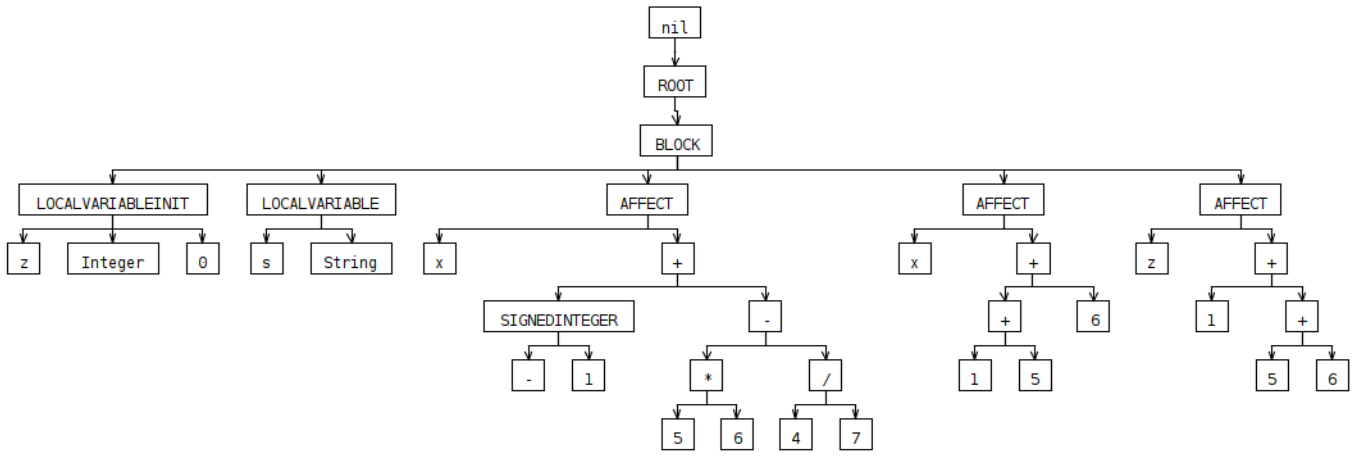
On a d'abord une déclaration avec initialisation de z suivie de la déclaration de s. Le mot clé is permet ensuite de distinguer les déclarations des affectations.

On a ensuite trois affectations. La première permet de mettre en avant les priorités opératoires (/,*,-,+ dans l'ordre de priorité décroissante). On remarque ensuite le -1, permettant de mettre en avant la possibilité d'avoir des entiers signés.

Les 2 calculs suivants permettent de mettre en évidence le rôle des parenthèses qui sont prioritaires sur les autres opérations.

```
{
z : Integer := 0;
s : String;
is
x := -1 + 5 * 6 - 4 / 7;
x := (1 + 5) + 6;
z := 1 + (5 + 6);
}
```

Voici l'arbre de ces déclarations et affectations :



6 Annexes

6.1 Comptes rendus des réunions

Compte rendu réunion n°1

Emilien JACQUES

4 novembre 2020

1 Membres

- Julien GIET, présent
- Emilien JACQUES, présent
- Jean-Baptiste RENAULT, présent
- Arthur SCHMIDT, présent

2 Ordre du jour

- Mise en commun des premières réflexions
- Mise en place des éléments de gestion de projet

3 Echanges

Mise en commun des premières réflexions

Chacun présente les premières idées qu'il a sur le projet. Une première version de la grammaire a été présentée. Celle-ci est LL(1). Elle est fonctionnelle mais incomplète. Il manque encore principalement le "if" et le "while". De plus, elle a besoin d'être testée sur des morceaux de codes supplémentaires.

Mise en place des premiers éléments de gestion de projet

Il a été décidé de faire une matrice SWOT pour avoir un meilleur aperçu des forces et faiblesses du groupe. De plus, il est important de créer un diagramme de GANTT pour planifier les avancées du projet.

4 To Do list

- Implémenter "if" et "while"
- Terminer la grammaire
- Concevoir de nouveaux jeux de tests
- Créer la matrice SWOT
- Créer le diagramme de GANTT
- Commencer à réfléchir à l'AST

5 Prochaine réunion

11 novembre 2020 à 16h

Compte rendu réunion n°2

Julien GIET

12 novembre 2020

1 Membres

- Julien GIET, présent
- Emilien JACQUES, présent
- Jean-Baptiste RENAULT, présent
- Arthur SCHMIDT, présent

2 Ordre du jour

- Mise en commun des avancées
- Réflexion sur la suite du projet

3 Échanges

Mise en commun des avancées

Certaines règles de la grammaire ont été modifiées, ces modifications sont exposées et justifiées à tout le groupe. Le diagramme de Gantt et la matrice SWOT ont été réalisés, ils sont validés par le groupe mais restent ouverts à des modifications éventuelles. Concernant la batterie de test créée, des modifications et des idées d'ajout sont proposées. D'autres tests sont réalisés en direct lors de la réunion.

Réflexion sur la suite du projet

Les tests réalisés jusque là laissent penser que l'AST est correct. D'autres tests vont tout de même être créés afin de compléter ceux déjà réalisés. La prochaine étape majeure est à présent la réalisation de la table des symboles. Un point est fait sur celle-ci, et chacun est invité à réfléchir de son côté à l'implémentation avant la prochaine réunion.

4 To Do list

- Compléter le jeu de test
- Réfléchir à l'implémentation de la table des symboles

5 Prochaine réunion

25 novembre 2020

Compte rendu réunion n°3

Arthur SCHMIDT

25 novembre 2020

1 Membres

- Julien GIET, présent
- Emilien JACQUES, présent
- Jean-Baptiste RENAULT, présent
- Arthur SCHMIDT, présent

2 Ordre du jour

- Présentation de la grammaire complet
- Exemple de tests

3 Echanges

Présentation de la grammaire complet

Les dernières modification ont été apporté à la grammaire, elle est fonctionnelle. Quelques problèmes mineurs ont été détectés est remplacé pour une meilleur utilisation de la grammaire.

Exemple de tests

Le dernier jeu de tests a été envoyé et testé. Une présentation en directe a été fait pour s'assurer de la fonctionnalité de la grammaire. L'ensemble des tests se passe comme prévu.

4 To Do list

- Commencer la construction de la TDS
- Préparer la présentation

5 Prochaine réunion

02 Décembre 2020 à 16h

Compte rendu réunion n°4

Julien GIET

13 janvier 2021

1 Membres

- Julien GIET, présent
- Emilien JACQUES, présent
- Jean-Baptiste RENAULT, présent
- Arthur SCHMIDT

2 Ordre du jour

- Rapport PCL1
- Réflexion sur la suite du projet

3 Échanges

Rapport PCL1

Un point a été fait sur ce qui est fait et ce qu'il reste à faire. Il faut rajouter les tests et les arbres obtenus, en commentant notamment ce qui a été modifié depuis la soutenance.

Réflexion sur la suite du projet

Une fois le rapport terminé, chacun est invité à réfléchir de son côté à l'implémentation de la table de symboles afin de pouvoir mener une réflexion efficace lors de la prochaine réunion, lors de la mise en commun.

4 To Do list

- Finir le rapport PCL1
- Réfléchir à l'implémentation de la table des symboles

5 Prochaine réunion

20 janvier 2021