École Nationale Supérieure des Sciences Appliquées et de Technologie
LSI2 2014/2015

Compilation - Projet

Compilateur NNP

Jean-Baptiste Louvet

Laurent Laluque

Javid Mougamadou

Ulrich Audouard-sadou

Edern Rivoallan

Antoine Nourry

12 juin 2015

Chargé de cours :

Damien Olive

Table des matières

1 Introduction

Pendant notre cursus en Logiciels et Systèmes Informatiques à l'Enssat, nous avons suivi un cours de Compilation. Au cours de ce module, nous avons étudié le langage *NilNovi*, langage adapté car existant à la fois en tant que langage Algorithmique, Procédural et Objet. Ainsi nous avons pu voir les différences de compilation et d'exécution de ces trois paradigmes. Au terme de ce cours, nous avons eu un projet à réaliser pour mettre en œuvre les connaissances acquises au cours du module.

1.1 Sujet du projet

Notre projet consistait en la réalisation d'un compilateur pour le langage *NilNovi Procédural*. Nous avions des documents de départ pour le faire, à savoir un document explicatif de la grammaire composée de diagrammes syntaxiques, un analyseur lexical et un analyseur syntaxique se contentant de vérifier la conformité du programme d'entrée à la grammaire *NilNovi*. Il nous restait alors à réaliser l'analyseur sémantique, pour vérifier l'intégrité sémantique des programmes, la génération de la table des identificateurs et la génération du code *NASM*.

2 Conduite du projet

2.1 Répartition des tâches

Le projet étant divisé en trois parties, nous nous sommes divisés en trois groupes pour le réaliser.

Le premier groupe, chargé de réaliser la génération de la table des identificateurs, était constitué de Didier Fromont et Edern Rivoallan. Le second groupe, chargé d'implémenter l'analyseur sémantique, était constitué de Javid Mougamadou et Ulrich Sadou-Audouar. Le dernier groupe, chargé de réaliser la génération de code assembleur était constitué de Laurent Laluque, Jean-Baptiste Louvet et Antoine Nourry. De plus, Jean-Baptiste Louvet assurait la responsabilité de chef de projet.

3 Analyse de l'existant

Pour réaliser ces différentes parties, il nous fallait savoir comment fonctionnaient les deux programmes existants.

3.0.1 L'analyseur lexical

La première partie à analyser était l'analyseur lexical. Il s'agissait d'un utilitaire de parcours d'un fichier d'entrée, qui servait à découper le fichier en une suite de mots-clés, éliminant tout espace ou retour chariot superflu, éliminant également les commentaires. Cette partie, utilisée dans l'analyseur syntaxique, en fournissait l'entrée.

3.0.2 L'analyseur syntaxique

La seconde partie, et le fichier principal du projet, était l'analyseur syntaxique. Cette partie se servait de l'analyseur lexical, en consommant ses mots-clés au fur et à mesure. L'analyseur suit les diagrammes syntaxiques de *NilNovi*, un diagramme correspondant à une fonction. Ainsi, avec ce fonctionnement, l'ajout de points de générations était rendu très simple.

4 Analyseur sémantique

A l'issue de l'analyse syntaxique, nous devons procéder à une analyse sémantique pour pouvoir poursuivre sur la base de génération de code. Dans cette partie, nous allons implémenter la partie analyseur sémantique de notre compilateur *NNP*. Il réalise la gestion des erreurs en comparant les types des différentes unités lexicales par rapport aux autres. Pour cela, nous avons identifié les différents diagrammes syntaxiques qui doivent retourner un résultat afin de permettre les vérifications nécessaires,ainsi nous avons effectué une génération d'erreurs sémantiques, et nous avons définie des types pour nos balises. Nous allons maintenant voir les différentes modifications ajoutées.

<opUnaire> : Nous avons commencé par cette balise car, avant de définir le type de tout le monde, il faut d'abord définir le type des deux opérations élémentaires (*+/-* et *not*).Ainsi, pour les opérandes *+/-* nous retournons le type "*integer*" et pour l'opérande *not*, nous retournons le type "*boolean*". Ensuite,on doit à présent consulter la règle du dessus.

<prim> : celle-ci fait appel à la règle de <**elemprim**>, qui a son tour fait appel à une autre règle :

<valeur> : On constate à ce moment qu'une valeur peut être soit un entier, soit une valeur booléenne. Lorsqu'une valeur est entière, elle doit retourner un type "*integer*" et lorsqu'elle est booléenne, elle retourne un type "*boolean*".

Voici ci-dessous un schéma récapitulatif de notre cheminement, qui montre comment nous retournons nos types en ce qui concerne : <*+/-*>, <*not*>, <*entier*> et <*valBool*>. Cependant, pour avoir le type d'un identificateur, nous utilisons la table des identificateurs comme mentionné ci-dessous. Les éléments <*appelFonct*> et <*expressions*> font respectivement appel aux éléments <*ident*> et <*prim*> (déjà typé).

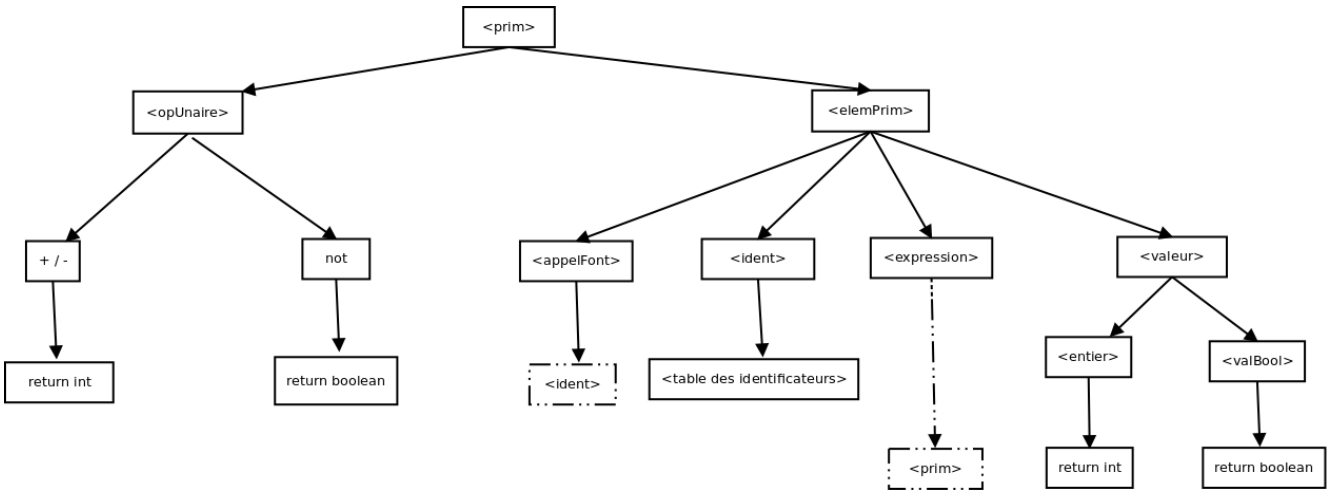

FIGURE 1 – definition des types.

Le typage des autres éléments de notre grammaire sera fondé sur ce même principe. Nous allons maintenant détailler les différents cas que nous avons implémentés pour générer nos erreurs :

Expressions scalaires : Lors des opérations arithmétiques ou booléennes, les opérandes et leurs opérateurs doivent être du même type (compatible), par exemple : l'expressions "$3+a=x$ " and "b" est valide si a et x sont des entiers et b un booléen.

Pour les opérateurs d'addition et de soustraction (+|-), nous modifions la règle <*exp3*> pour pouvoir comparer le type des deux opérandes et dans le cas où les deux types sont égaux nous renvoyant le type *integer*. Cela permet à la règle antérieur (ici <*exp2*>) de pouvoir manipuler le type de son opérande grâce à l'enboitement des règles. Pour les opérateurs de multiplication et de division (*|/), nous effectuons également une modification sur la règle <*exp4*> pour la comparaison des deux opérandes.

Pour les opérateurs de comparaison, nous distinguons 2 cas particulier. Si l'opérateur est = ou /= alors nous verifions si les opérandes sont de même type. Si l'opérateur fait partie du groupe (=>,>,<,<=) alors il faut que les opérandes soit du type *integer*. Ensuite, nous retournons dans les 2 cas le type *booleen* pour la règle antérieur.

De même pour les opérateurs booléen and/or, nous modifions les règles <*expression*> et <*exp1*>.

Methode put et get : Lors de l'appel de l'une de ces 2 méthodes, le paramètre formel doit être du type *integer*. Nous effectuons des comparaisons dans la règle <*es*>. De plus pour la foncion *get*, si la donnée fournie est un paramètre alors nous verifions s'il est du mode *in out*.

Affectation : Lors d'une affectation, nous verifions si la variable a bien été déclaré auprès de la table des identificateurs puis nous comparons si le type de la variable et le type de la valeur sont égales. Pour cela nous modifions la règle <*instr*>. De plus, s'il s'agit d'un paramètre, il doit être du mode *in out*. Dans le contraire, nous générons une erreur sémantique. Par exemple :

$a : integer; - - - > a := true$; Affectation Invalide

Boucle *while* et *if* : Le type de l'expression attendue dans la condition doit être booléenne. Pour cela, nous modifions les règles <*altern*> et <*boucle*>.

$val : integer - -- > if\ val + 12\ then$, Condition Incorrecte.

Compatibilité entre modes : Lors de l'appel d'une fonction, si celle-ci a été déclarée avec un paramètre formel *integer* et qu'elle est appelée cette fois avec un paramètre effectif "*boolean*", le compilateur nous génère une erreur.De même, si le paramètre formel est de mode "*in out*", et que le paramètre effectif est de mode *in*, le compilateur doit aussi nous générer une erreur.

Double Déclaration : A chaque fois qu'une variable ou un paramètre est déclaré nous verifions s'il n'existe pas auprès de la table des identificateurs. Nous vérifions également qu'il n'y a pas de conflits entre les variables locales et les paramètres formels de la fonction/procédure.

Instruction return : Lors de la déclaration d'une fonction, nous verifions qu'elle possède au moins une instruction *return* et que le type de valeur retournée soit égale au type du retour déclaré dans la règle <*fonction*>. Pour cela, nous effectuons des comparaisons dans les règles : <*suiteInstr*>, <*instr*>, <*return*>

Appel de fonction ou de procédure : Lors de l'appel d'une fonction ou d'une procédure, nous construisons une table temporaire comportant tous les paramètres effectifs utilisés lors de l'appel. La table temporaire est très similaire à la table des identificateurs et celle-ci est détruite à la fin de l'appel.

Nom	Type	Mode	Statut
i	*integer*	*in*	*Paramètre*
5	*integer*	*None*	*Constante*
a	*integer*	*None*	*Variable*

La création de cette table permettra d'effectuer des comparaisons entre les paramètres effectifs appelées lors de la fonction et les paramètres formels utilisés lors de la définition de la fonction en comparant avec la table des identificateurs uniquement les lignes concernant la fonction.

Tout d'abord, nous vérifions si le nombre d'argument donné lors de l'appel correspond avec le nombre de paramètre attendus par la fonction. Pour cela, nous comparons simplement si le nombre de ligne de la table des identificateurs (concernant les paramètres de la fonction) et le nombre de ligne de notre table temporaire sont égales. Ensuite, nous verifions pour chaque ligne de ces deux tables si les paramètres effectifs et formels possèdent le même type. Dans le cas où le paramètre formel est de mode *in out*, le paramètre effectif doit lui aussi être du même mode sauf si c'est une variable globale/locale.

Pour réaliser ce cas, nous avons du modifier les règles : <*listePe*>, <*instr*>, <*elemPrim*> (partie *AppelProc* et *AppelFonct*). Si l'un des arguments d'entré e est une expression alors nous n'affichons uniquement le type de l'expression en tant que constante.

En outre, si lors de la compilation, aucun de ces cas n'a été détecté, elle se passe normalement, sinon on génère une erreur avec un message spécifié pour le cas détecté et en affichant la ligne et la colonne concernée par cette erreur. Ces informations sont récupérés grâce à l'analyseur lexicale.

5 La table des identificateurs

5.1 Structure

Cette étape préliminaire à l'analyseur lexical permet de distinguer dans le code source, tous les identificateurs de toute nature (procédure, variables...). Il prend en entrée ce fichier, et en parcourant le programme qui y est contenu, on remplit un tableau de la forme :

Nom	**Type**	**Adresse**	**Parent**	**Statut**	**Mode**
None	*None*	*None*	*None*	*None*	*None*

Nom : Appellation de l'identificateur

Type : Donne le type de la variable/paramètre ou du type du retour de la fonction
- ◇ Boolean
- ◇ Integer
- ◇ None

Adresse : Adresse de l'identificateur en question (entier positif)

Parent : Donne le nom de la procédure/fonction parente de l'identificateur
- ◇ Si l'identificateur est une procédure/fonction, on renseigne le nom de la procédure principale.
- ◇ Si c'est une variable /paramètre, c'est le nom de la procédure/fonction qui la contient.

Statut : Renseigne une information supplémentaire sur le qualificatif de l'identificateur
- ◇ Procédure
- ◇ Fonction
- ◇ Variable
- ◇ Paramètre

Mode : Si l'identificateur est un paramètre, on renseigne son mode. Sinon on ne met rien (*none*).
- ◇ in
- ◇ inout
- ◇ None

5.2 Construction

Une classe *TableIdent* a été créée. Elle contient des variables modifiées tout au long du programme (détaillé plus bas) ainsi qu'une liste de listes qui vont représenter les lignes du tableau. Toute ligne est initialisée avec ses champs à *None* par défaut. Le code est parcouru selon les règles qui définissent la grammaire du langage *NILNovi* Procédural. En python, notre table est une liste de listes. Celles-ci représentent les lignes du tableau. En parcourant le code, à chaque fois que l'analyseur syntaxique lit un terme, il va accéder à la fonction correspondante au terme dans le package « *analsyn.py* ». La fonction *acceptKeyword()* va permettre de consommer ces termes.

Le code nécessaire pour remplir le tableau est injecté dans ces fonctions. Il est donc rempli au fur et à mesure du parcours du code. Par exemple, le mécanisme pour remplir une ligne renseignant une procédure est le suivant :

On accède à la définition de « *procedure* » dans le package, car on sait que l'analyseur accède à cette définition quand le « *lexical_analyser* » dans le code se trouve sur une procédure, le code aura une syntaxe de la forme « *procedure <ident> <partieFormelle> is <corpsProc>* ».

On peut donc déjà renseigner :

Nom : *<ident>* : Le code fournit, récupère déjà le nom/identifiant du terme en question avec la ligne « *ident = lexical_analyser.acceptIdentifier()* ».

Type : *None* : c'est une procédure.

Adresse : On va compter ici le nombre de terme déclarer dans la procédure principale (de 0 à n) incluant procédures/fonctions et variables. Pour connaître l'adresse de la procédure que l'analyseur est en train de lire, une variable de l'objet *tableIdent* (*cpt_varprocglobal*) est incrémentée à chaque fois qu'il en rencontre une.

Parent : <procédure principale> : c'est un paramètre qui est sauvé (*nomProcPrinc*) quand l'analyseur débute le programme.

Statut : Procédure : on peut l'écrire avec certitude.

Mode : *None* : on est sûr que c'est une procédure.

Tous ces champs sont récupérés ou renseignés dans « *def procédure(lexical_analyser) :* », et on insère une nouvelle ligne dans la table des identificateur. Le mécanisme décrit ci-dessus pour une procédure est identique que pour les autres types d'identifiants avec, évidemment, le code qui ajoute une ligne pour les autres identificateurs (*fonction*, *mode*, *nnpType*) qui est écrit dans leurs définitions respectives dans le package « *analsyn.py* ».

On retrouve alors la tendance générale :

Nom : récupéré par « *ident = lexical_analyser.acceptIdentifier()* ».

Type : traité par la définition de «*nnpType*» avec la vérification du type (*lexical_analyser.isKeyword("integer/boolean")*). Il vaudra None si c'est une procédure.

Astuce : Lorsqu'une procédure/fonction a plusieurs variables/paramètre du même type, celles-ci sont séparées par une virgule avant le type. L'analyseur récupère seulement le type de la dernière variable/paramètre. Pour palier à ce problème, on utilise une boucle qui crée autant de lignes que de variables/paramètres et lorsque le code « *: integer ;* » est rencontré, on édite le champ type des lignes précédemment créés avec le bon type. (fait dans « *def declaVar* »).

Adresse :

- ◇ Une variable *cpt_varprocglobal* compte le nombre de procédures/fonctions dans la procédure principale.
- ◇ La variable *cpt_varspecif* compte le nombre de paramètres dans la fonction actuelle.
- ◇ Une variable *cpt_var* compte le nombre de variables dans la procédure/fonction actuelle.

En fonction de l'endroit où se situe le curseur de l'analyseur syntaxique, une de ces variables est incrémentée ou non (par exemple, *cpt_varprocglobal* est incrémenté lorsqu'on passe à une seconde procédure).

Parent : A chaque fois qu'on entre dans une procédure/fonction, une variable *nomProcActu* est mise à jour avec le nom de la procédure/fonction dans laquelle on vient d'entrer (*Ident*).

Statut : En fonction de la définition du terme auquel on accède, on sait avec certitude le statut du terme en question :

- ◇ *def listeIdent* -> variable (*<declaVar> : := <listeIdent> : <type>*)
- ◇ *def specif* -> paramètre (*<listeIdent> :<mode> <type>* | *<liteIdent> : <mode>*)
- ◇ *def fonction* - > fonction
- ◇ *def procedure* -> procédure

Mode : Est traité par la définition du terme «*mode*» avec la vérification (*if lexical_analyser.isKeyword("*
Vaut *None* si ce n'est pas un paramètre.

Lorsque le code est parcouru dans sa totalité, la table des identificateurs est complètement générée. Il est prêt à être utilisé pour les étapes suivantes comme la vérification d'erreurs syntaxiques du code.

6 Génération de code

Pour simplifier l'élaboration du compilateur, nous l'avons réalisé en deux étapes. Nous avons commencé par développer un compilateur *NilNovi* Algorithmique, plus simple, puis l'avons modifié pour réaliser le compilateur *NilNovi* Procédural. Nous reprendrons ici l'ensemble des poinrs de génération, avec l'emplacement de leur insertion et leur nom, ce dernier se rapportant à celui indiqué dans le fichier *codegen.py*.

6.1 Génération NNA

La première partie du développement était consacrée à NNA. Nous allons décrire ici chaque point de génération ajouté pour *NilNovi* Algorithmique sur les diagrammes syntaxiques originaux.

6.1.1 programme

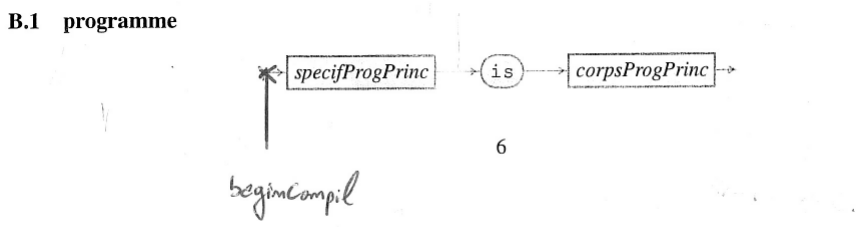

FIGURE 2 – Diagramme syntaxique de programme avec les points de génération correspondants

beginCompil	Le premier point de génération a été ajouté dès le lancement de la compilation pour ajouter les instructions d'initialisation. Il sert à déclarer la procédure principale, à lier les fonctions d'entrée-sortie printf et scanf et à déclarer les formats d'entrée et de sortie pour l'utilisation des deux primitives.

6.1.2 corpsProgPrinc

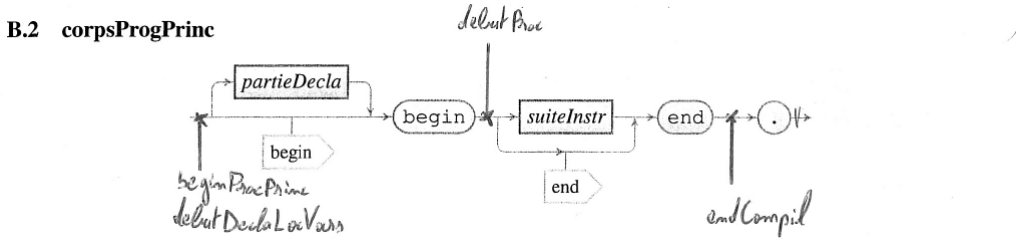

FIGURE 3 – Diagramme syntaxique de corpsProgPrinc avec les points de génération correspondants

beginProgPrinc	Au début de la section d'instructions du programme principal, on ajoute un point de génération pour générer les lignes d'assembleur correspondantes.
endCompil	À la fin du programme, on ajoute aussi un point de génération pour générer les instructions de sortie du programme.

6.1.3 partieDecla

declaVarGlobSection	Au début du programme, avant le début des instructions, on déclare une partie *.bss* servant à l'adressage des variables.

6.1.4 declaVar

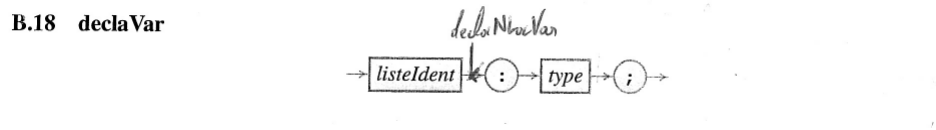

FIGURE 4 – Diagramme syntaxique de declaVar avec les points de génération correspondants

declaAVarGlob	Après avoir récupéré une ligne de déclaration de variables dans le source original, on déclare toutes les variables dans le segment *.bss* avec leur identifiant original, pour les récupérer plus facilement dans le programme.

6.1.5 listeIdent

buildListeIdent	À chaque identifiant rencontré sur une ligne de déclaration, on met à jour une liste temporaire d'identifiants dans le générateur pour réserver tout l'espace ensuite.

6.1.6 instr

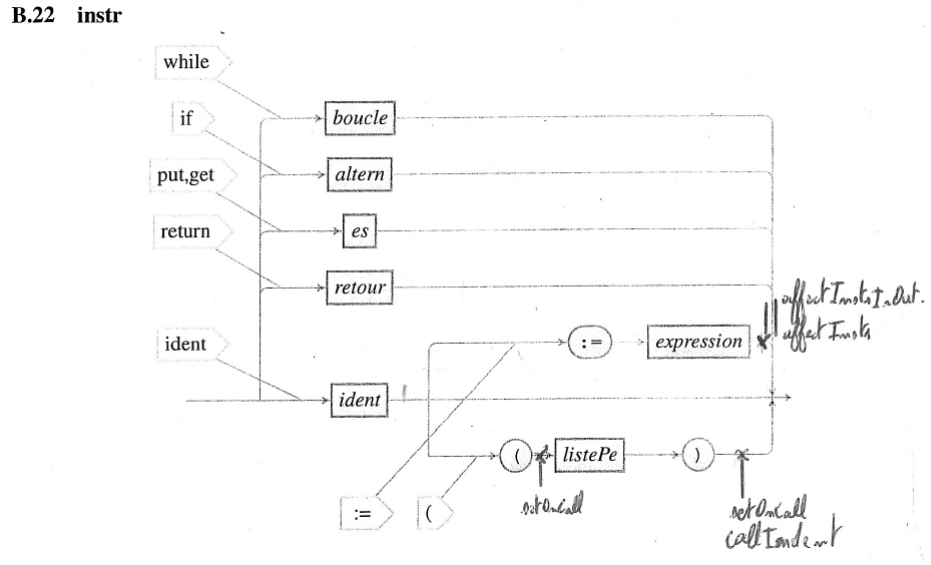

FIGURE 5 – Diagramme syntaxique de instr avec les points de génération correspondants

affectInstr	Après avoir rencontré une affectation et évalué l'identifiant affecté ainsi que l'expression à affecter, on appelle ce point de génération pour faire l'affectation. Pour notre compilateur, nous fonctionnons comme sur une machine à pile : chaque expression évaluée est poussée sur la pile, et n'est récupérée dans un registre que pour faire une opération.

6.1.7 expression

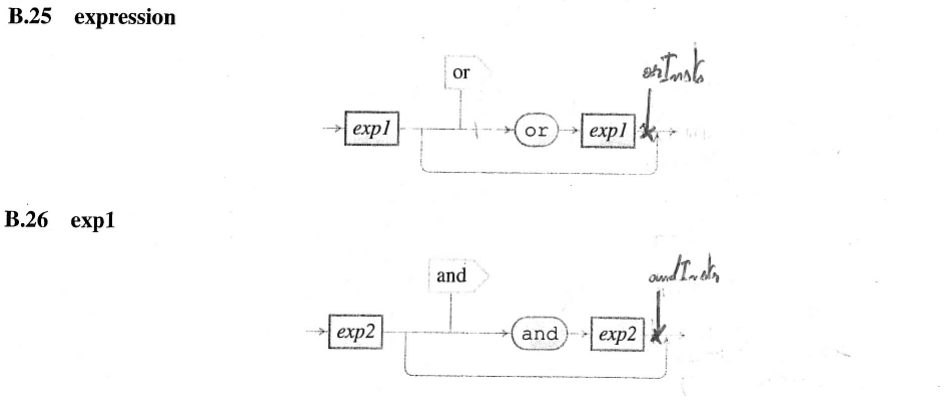

FIGURE 6 – Diagramme syntaxique de expression avec les points de génération correspondants

orInstr	Lors de la rencontre d'une instruction *or*, on évalue les deux expressions concernées, on effectue une addition de ces deux expressions, normalement égales à 0 ou 1. Si le résultat est supérieur ou égal à 0, on place 1 dans la pile.

6.1.8 exp1

andInstr :

andInstr	Lors de la rencontre d'une instruction *and*, on évalue les deux expressions concernées, on effectue une multiplication de ces deux expressions, normalement égales à 0 ou 1. Le résultat de cette multiplication est directement placé sur la pile.

6.1.9 exp2

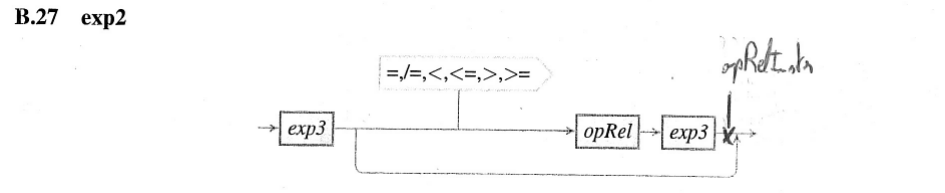

FIGURE 7 – Diagramme syntaxique de exp2 avec les points de génération correspondants

opRelInstr :

opRelInstr	Si l'on rencontre une instruction d'opération relationnelle, on commence par évaluer les deux expressions concernées, puis on appelle le point de génération avec comme paramètre l'opération rencontrée. Ensuite, une simple condition est faite en assembleur, et si la condition est vraie, on place la valeur 1 sur la pile.

6.1.10 exp3

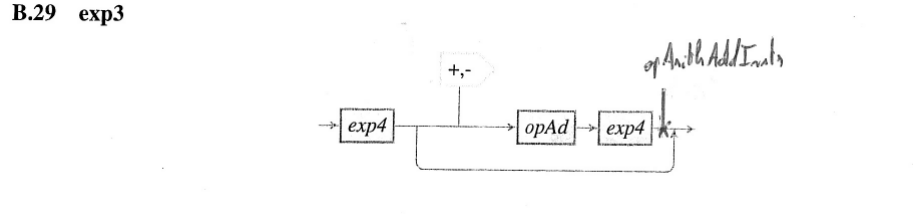

FIGURE 8 – Diagramme syntaxique de exp3 avec les points de génération correspondants

opArithAddInstr	Si l'on rencontre une instruction d'opération arithmétique d'addition ou de soustraction, on commence par évaluer les deux expressions concernées, puis on appelle le point de génération avec comme paramètre l'opération rencontrée. Le point génère alors les instruction nécessaires à l'exécution de l'opération et au placement du résultat sur la pile.

6.1.11 exp4

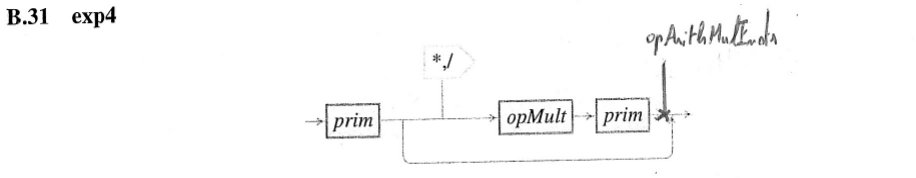

FIGURE 9 – Diagramme syntaxique de exp4 avec les points de génération correspondants

opArithMulInstr	Si l'on rencontre une instruction d'opération arithmétique de multiplication ou de division, on commence par évaluer les deux expressions concernées, puis on appelle le point de génération avec comme paramètre l'opération rencontrée. Le point génère alors les instruction nécessaires à l'exécution de l'opération et au placement du résultat sur la pile.

6.1.12 prim

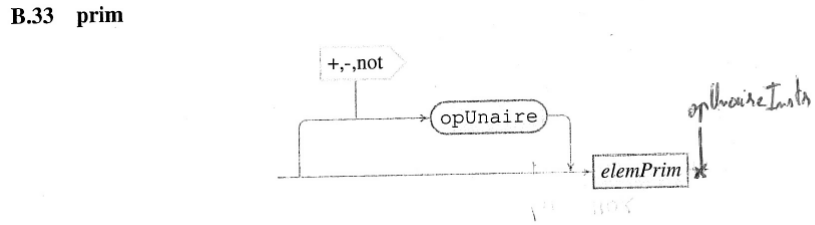

FIGURE 10 – Diagramme syntaxique de prim avec les points de génération correspondants

opUnaireInstr	Le point de génération d'opération unaire est divisé en trois parties, une par opération possible. En cas d'opération *+*, on ne fait rien, la valeur concernée étant déjà positive. En cas d'opération -, on place l'instruction *neg* pour obtenir l'opposé de la valeur concernée. Enfin, en cas d'opération *not*, on teste la variable booléenne associée pour mettre la valeur opposée sur la pile (1

6.1.13 elemPrim

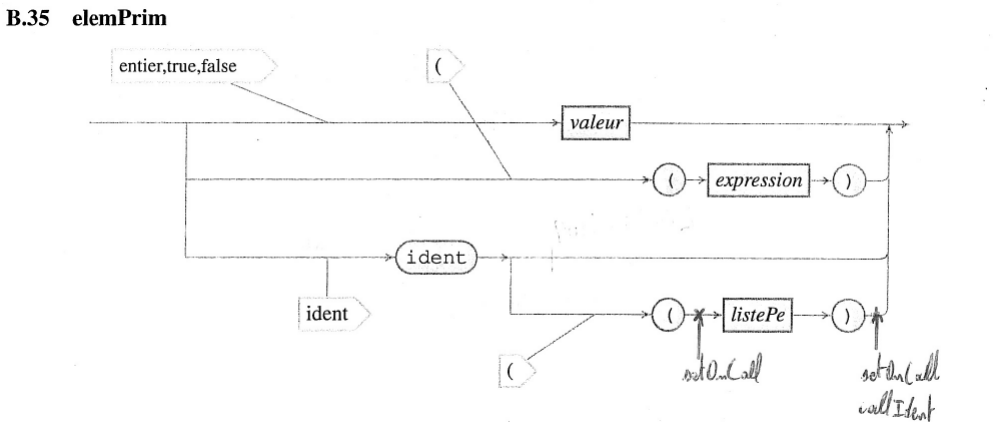

FIGURE 11 – Diagramme syntaxique de elemPrim avec les points de génération correspondants

identInstr	Si un identifiant est rencontré, on appelle un point de génération permettant de mettre la valeur de l'identifiant dans la pile.

6.1.14 valeur

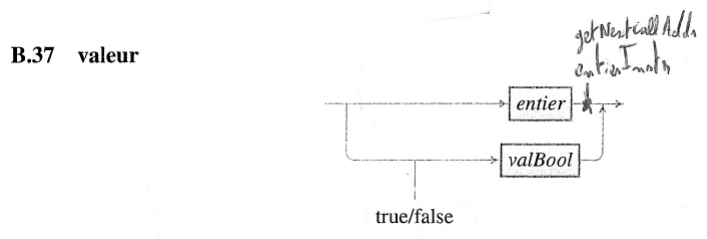

FIGURE 12 – Diagramme syntaxique de valeur avec les points de génération correspondants

entierInstr	Lors de la rencontre d'un entier, on place simplement ce dernier au sommet de la pile.

6.1.15 valBool

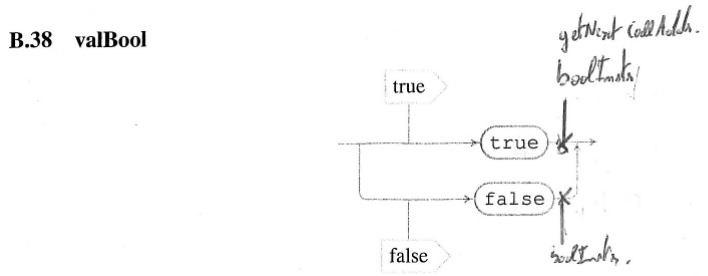

FIGURE 13 – Diagramme syntaxique de valBool avec les points de génération correspondants

boolInstr	Lors de la rencontre d'un booleen, on place simplement ce dernier au sommet de la pile. La valeur *true* est codée par un 1, la valeur *false* est codée par un 0.

6.1.16 es

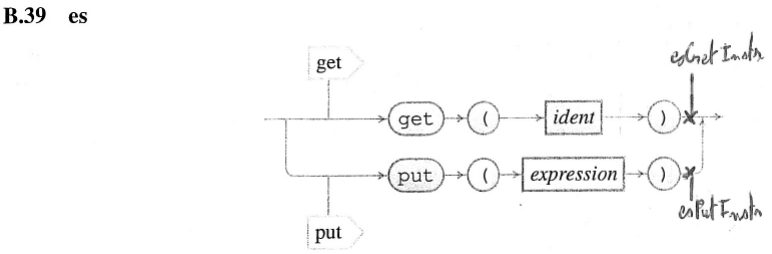

FIGURE 14 – Diagramme syntaxique de es avec les points de génération correspondants

esGetInstr	Lors de l'appel à *get*, on place les instructions assembleur permettant un appel à scanf.
esPutInstr	Lors de l'appel à *put*, on évalue l'expression à afficher, puis on place les instructions assembleur permettant un appel à printf.

6.1.17 boucle

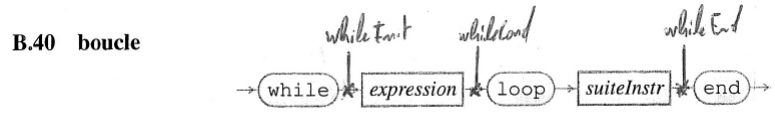

FIGURE 15 – Diagramme syntaxique de boucle avec les points de génération correspondants

Les boucles sont gérées en trois parties : une initialisation de boucle, une partie d'évaluation de la condition et une terminaison de boucle.

whileInit	L'initialisation de la boucle correspond au placement d'un label pour revenir à cette instruction en fin de boucle.
whileCond	Après l'initialisation, on évalue la condition de boucle, et on place le résultat dans la pile. Ensuite, ce point de génération ajoute les instructions d'évaluation du résultat et de saut vers la fin de boucle si la condition n'est plus respectée.
whileEnd	Enfin, en fin de boucle, on place l'instruction de saut vers le début de boucle, et on place un label de fin de boucle.

6.1.18 altern

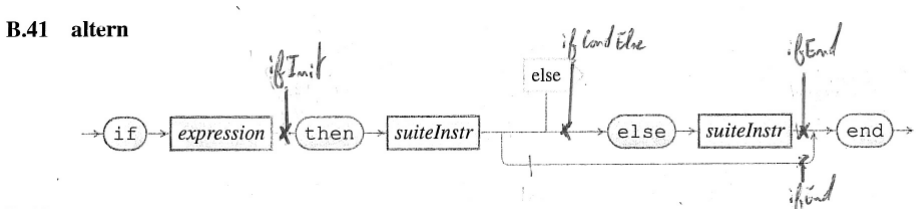

FIGURE 16 – Diagramme syntaxique de altern avec les points de génération correspondants

Les instruction de condition sont gérées, elles aussi, en trois parties. Tout d'abord, une initialisation servant à l'évaluation de condition, puis un bloc en cas de else, et une conclusion de bloc.

ifInit	Lors de l'initialisation de l'alternative, on évalue la condition d'alternative. Si elle n'est pas vraie, on place l'instruction de saut vers la fin de boucle, ou le bloc *else* le cas échéant.
ifCondElse	Le point de génération du *else* place un saut vers la fin de l'alternative pour la fin du premier bloc d'alternative, et un label pour le début du bloc *else*.
ifEnd	Enfin, le point de génération de conclusion place simplement un label pour signifier la fin d'alternative.

6.2 Génération NNP

La base d'un compilateur *NilNovi* Procédural est une sorte de "sur-couche" du NNA. On inclut en effet la gestion des fonctions et des procédures. Les principaux points sont autour des paramètre et des "sauvegardes" de contexte (à cause notamment du type de passage de paramètre, des variables locales et globales). Ces identificateurs doivent être déclarés avant d'être utilisés. Grâce à ceci, nous pouvons faire des opérations récursives.

6.2.1 procedure

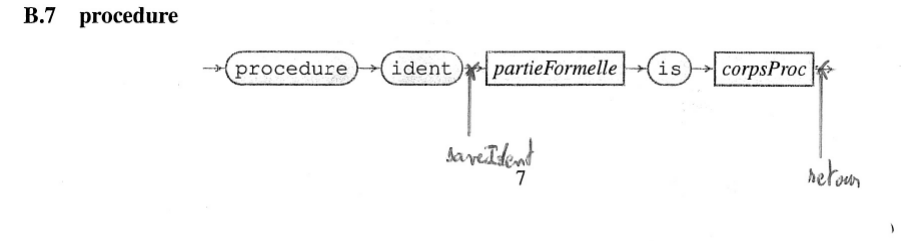

FIGURE 17 – Diagramme syntaxique de procedure avec les points de génération correspondants

retour	Destruction de la pile utilisée "localement", retour de la fonction et génération du label pour le retour.

6.2.2 fonction

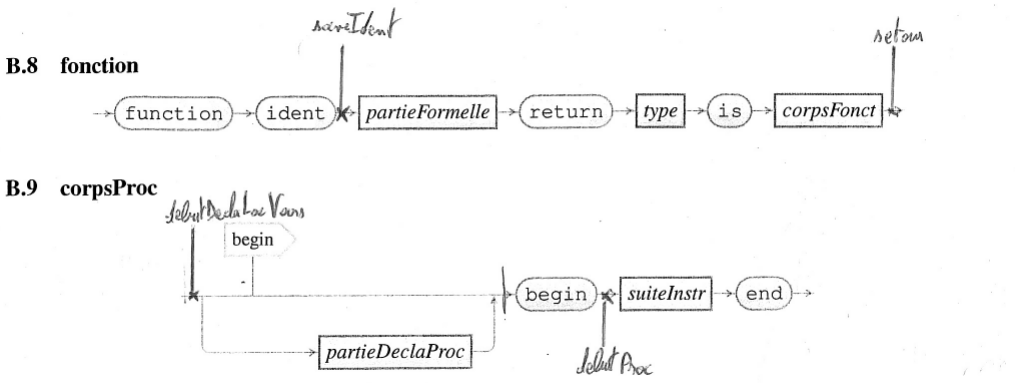

FIGURE 18 – Diagramme syntaxique de fonction avec les points de génération correspondants

saveIdent	Mise de l'identificateur avant la définition de la procédure correspondante.
retour	Même opérations que pour le retour dans procédure, avec en amont la récupération de la valeur de retour en plus.

6.2.3 corpsProc et corpsFonct

debutDeclaLocVars	Pour marquer le début de la déclaration des variables locales, on dépile l'adresse de l'exécution de retour de la fonction/procédure, que l'on stocke dans un registre.
debutProc	On remet l'adresse de retour de la fonction/procédure au sommet de la pile, après la déclaration des variables locales, et on la rempile.

6.2.4 declaVar

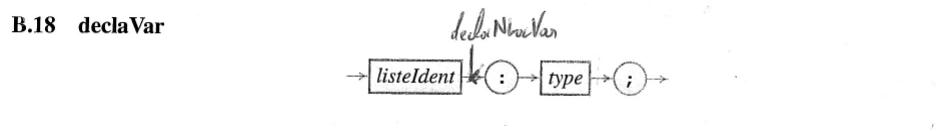

FIGURE 19 – Diagramme syntaxique de declaVar avec les points de génération correspondants

declaNLocVar	On réserve ici *nVar* emplacements pour les variables locales de la fonction/procédure courante.

6.2.5 instr

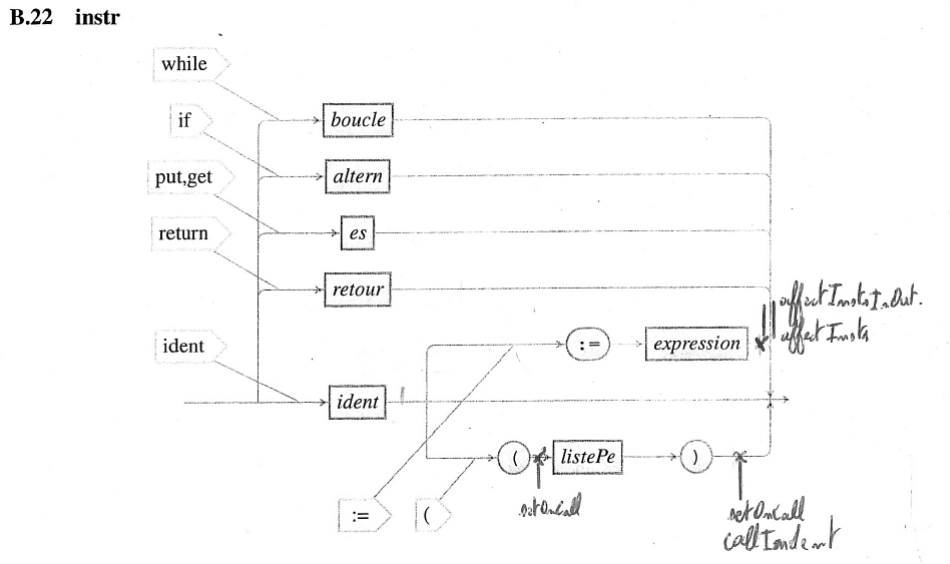

FIGURE 20 – Diagramme syntaxique de instr avec les points de génération correspondants

affectInstr	On effectue ici l'opération d'affectation, dans le cas où l'on utilise une variable passée en *in* dans la fonction ou procédure courante. Pour cela, on dépile la valeur à affecter à notre variable, puis l'on calcule l'adresse à laquelle on va l'affecter, puis on effectue l'affectation à cette adressse.
affectInstrInOut	On effectue ici l'opération d'affectation, dans le cas où l'on utilise une variable passée en *in out* dans la fonction ou procédure courante. Le principe est similaire au point de génération précédent, sauf qu'ici, c'est l'adresse où se trouve la variable que l'on veut affecter qui est dans la variable *in out*. On utilise donc une instruction supplémentaire pour accéder à sa valeur.
setOnCall	Met la variable *onCall* à jour, pour savoir si l'on se situe dans un appel de fonction ou non, ce qui modifiera les accès aux variables, notamment.
callIdent	Permet de gérer l'appel à une fonction ou une procédure. On utilise pour cela l'instruction *CALL* en NASM, puis on réserve l'espace mémoire pour les paramètres et la valeur de retour.

6.2.6 elemPrim

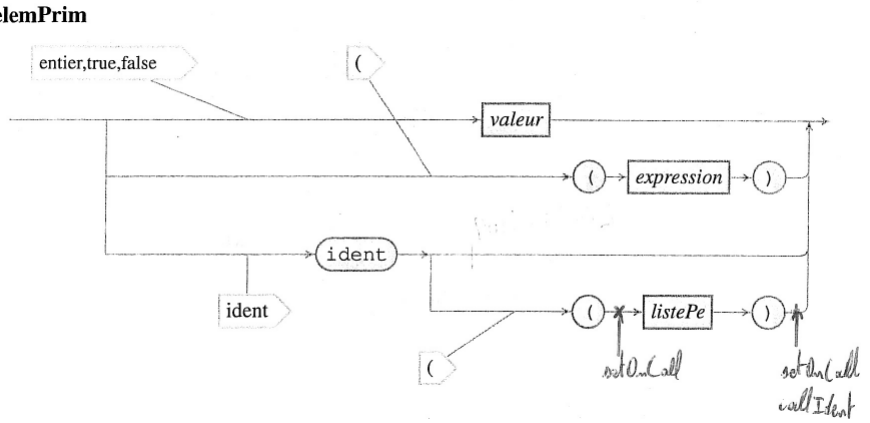

FIGURE 21 – Diagramme syntaxique de elemPrim avec les points de génération correspondants

setOnCall	Element évoqué précédemment.
callIdent	Point de génération évoqué précédemment.
identInstrInOut	Permet de gérer l'utilisation d'un paramètre lors d'un appel à une fonction, si, dans la fonction appelante, le paramètre en question est passé en mode *in out*. textitmode désigne ici le mode de destination du paramètre, c'est-à-dire si, dans l'appel de la fonction où il entre en jeu, il est en *in* ou en *in out*. Ainsi, si la fonction appelée impose de passer notre paramètre en *in out*, on lui passera la valeur du paramètre, qui est déjà l'adresse de la variable, car le paramètre est en *in out* dans la fonction appelante.
identInstrIn	Permet de gérer l'utilisation d'un paramètre lors d'un appel à une fonction, si, dans la fonction appelante, le paramètre en question est passé en mode *in*. textitmode désigne ici le mode de destination du paramètre, c'est-à-dire si, dans l'appel de la fonction où il entre en jeu, il est en *in* ou en *in out*. Ainsi, si la fonction appelée impose de passer notre paramètre en *in out*, on lui passera l'adresse du paramètre car le paramètre est, cette fois-ci, en *in* dans la fonction appelante.

7 Conclusion

Voici la conclusion e ce projet, incluant un bilan du groupe et personnel sur le projet, ainsi que les éventuelles amélioration que nous avons discernées.

7.1 Bilan du projet

Nous avons finalement un compilateur NNP fonctionnel. Nous avons divisé le travail en plusieurs parties distinctes, que nous avons ensuite rassemblées.

Tout d'abord, l'analyseur sémantique a permis, à partir de l'analyse lexicale, de déduire les instructions à générer en NASM.

Au niveau de la table des identificateurs nous avons tout de suite compris l'objectif de notre partie. Au départ, nous ne maitrisions pas le langage python et c'est pourquoi notre première itération utilisait seulement des variables globales. Dans un deuxième temps, l'utilisation d'un objet a permis la simplification du code, notamment avec la partie analyse sémantique. Nous avons en effet fournis des méthodes de tests et d'accées à la table.

Au niveau de la génération de code, nous avons décomposé notre travail en deux partie : la génération de code pour le NNA, et la génération de code pour le langage NNP. La différence majeure entre les deux réside dans l'accès aux variables, puisque *NNP* permet d'avoir des fonctions et procédures, et donc des variables locales à ces derniers. Il a donc fallu réfléchir sur l'adressage desdites variables, mais également à leur mode d'accès, notamment la différence entre les variables *in* et *in out*, les premières donnant seulement accès à la valeur de la variable, et les deuxième permettant également d'accéder à l'adresse où elles sont stockées en mémoire. Nous avons réussi à passer cette étape, et avons donc pu fournir la partie de génération de code pour la totalité du langage NNP.

7.2 Améliorations possibles

Conformément à l'enchaînement du cours, nous aurions pu poursuivre ce projet en tentant d'implémenter un compilateur pour le langage *NilNovi* Objet, en se basant sur celui du *NilNovi* Procédural. Cependant, cet objectif était assez important, dans le temps qui nous était imparti.

Nous aurions également pu nous intéresser à la gestion de la récursivité croisée, notamment en implémentant les prototypes, ce qui permettrait de compiler des programmes avec des fonctions qui ne sont pas encore définies, mais qui sont déclarées en prototypes, ce qui est nécessaire pour la récursivité croisée.