Traitement des langages

BE : Compilation d'un petit langage (sous Linux)

1. Compilation du langage Niklaus :

On souhaite écrire une grammaire ANTLR pour un petit langage appelé Niklaus puis générer des arbres de syntaxe abstraite (AST) pour enfin compiler ce langage vers le langage d'assemblage du petit micro-processeur qui a été présenté en cours.

1.1 Ecriture de la grammaire ANTLR pour le langage Niklaus

L'objectif de cette partie est de générer des arbres de syntaxe abstraite (AST) à partir de la grammaire ANTLR. On utilise les fichiers de type .niklaus pour mieux comprendre la grammaire associé à ce langage puis on établit les différentes règles de grammaires. On obtient la grammaire suivante :

```
grammar Niklaus;

options { output = AST ;        }

expr    :       term (ADDOP^ term)*;

term    :       factor (MULTOP^ factor)*;

factor  :       ID | INT | ( '('! expr ')'! );

read_var        :       READ^ ID';'! ;

write_exp       :       WRITE^ expr';'! ;

instruction:            (write_exp|read_var|condition|affectation|comparaison|loop)*;

affectation
        :       ID EGAL^ expr';'! ;
program
        :       PROGRAM^ ID ';'!  declaration?  '{'! instruction '}'! ;

comparaison
        :        expr SIGNE^ expr;

condition       :       IF^ '(' comparaison ')' '{'! instruction '}'! ELSE '{'! instruction '}'!;

loop    :       WHILE^ '('! comparaison ')'! '{'! instruction '}'!;

declaration     :        VAR^ ( ID ','! )* ID ';'! ;

SIGNE   :       '<' | '>' | '<>' | '<='|'>=' |'=';

READ    : 'read';
VAR     :       'var';

WRITE   : 'write';

EGAL    :':=';

WHILE   :'while';

IF      :'if';

ELSE    :'else';

INT :   '0'..'9'+
    ;
    PROGRAM
        :       'program';

COMMENT
    :   '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
    ;

WS  :   ( ' '
        | '\t'
        | '\r'
        | '\n'
        ) {$channel=HIDDEN;}
    ;

ADDOP   :       '+' | '-' | 'mod' ;

MULTOP  :       '*' | '/';

ID  :   ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*;
```

On peut donc générer les fichiers suivants à partir d'ANTLR :

1.2 Ecriture du programme Java et parcours de l'arbre AST

Il s'agit ici de parcourir l'arbre AST généré précédemment par ANTLR. On utilise un programme Java qui va faire office de compilateur et produire un code en assembleur. On ne modifie pas les fichiers générés par ANTLR et on crée une nouvelle classe Java (NiklausMain.java) avec une fonction main qui va permettre de manipuler le parseur.

La génération du code se fait à l'aide de la classe NiklausMain. On définit alors les attributs suivants :

- writer (Printwriter) : utilisé pour l'écriture du fichier .arm
- workdir : utilisé pour définir le répertoire de travail
- programName : utilisé pour définir le programme .niklaus que l'on veut compiler

La compilation début avec la fonction treatProgram(Tree t) appelé par la fonction main puis le parcours de l'arbre se fait de manière récursif en utilisant les fonction treatBlock, treatDecalaration, et treatInstruction.

- **treatBlock** : fait appel à treatInstruction ou treatDeclaration si il s'agit d'une déclaration de variable en début de programme
- **treatInstruction** : redirige vers la fonction correspondant à l'instruction trouvé pour les instructions suivantes (read, write, :=, if, while)
- **treatDeclaration** : génére le code correspondant à une déclaration de variables

De plus, la fonction ajoutLibrairie() permet de écrire dans le fichier .arm la libraire lib.arm utilisé pour définir les fonction readInt et writeInt.

Enfin, j'ai également choisi de rajouter une classe ParserException.java afin de mieux gérer les messages d'erreurs lorsqu'il y a une erreur syntaxique lors du parcours d'un fichier .Niklaus par le parseur.

1.3 1er niveau de complétude

Pour ce niveau, les problèmes de définition de variables sont traités., par exemple si une variable n'est pas déclarée (undef.Niklaus) ou déclarée 2 fois (redef.Niklaus) et également pouvoir afficher des nombres avec l'instruction write (basic.Niklaus).

Un des problèmes rencontrés avec la fonction write est le fait d'utiliser une adresse différente pour chaque appel de cette fonction. La solution pour pallier ce problème est de rajouter des variables globales de type int (endprint_id) que l'on incrémente à la fin de l'instruction et que l'on ajoute à une chaine de caractères adresse qui reste inchangé par exemple (« endprint »).

Enfin, pour gérer une double déclaration de variable ou vérifier qu'une variable n'est pas déclaré, il a fallu crée une variable globale de type *Vector<String>* pour stocker les variables déclarées :

```
private static Vector<String> listeVariable = new Vector<String> ();
```

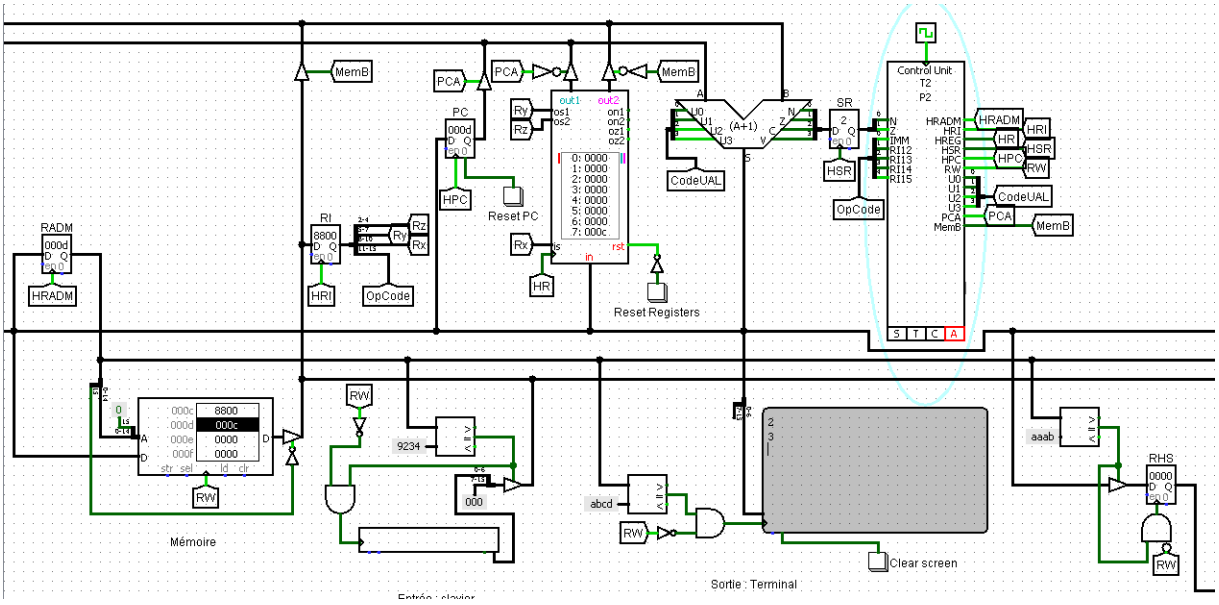
Ainsi, les erreurs sont bien détectées pour les fichiers redef.Niklaus et undef.Niklaus

Pour le fichier basic.niklaus on obtient le programme assembleur suivant :

```
@main
mov r7, #endprint1
mov r0, #2
b printInt
@endprint1
mov r7, #endprint2
mov r0, #3
b printInt
@endprint2
b endprint2
@amem rmw 1
@bmem rmw 1
```

On utilise alors le fichier python assembler.py pour compiler le programme assembleur. Cela génère plusieurs fichiers dont un fichier basic.mem.

On charge ce dernier dans la mémoire du processeur et on lance la machine, on obtient les résultats suivant une fois la simulation lancée :



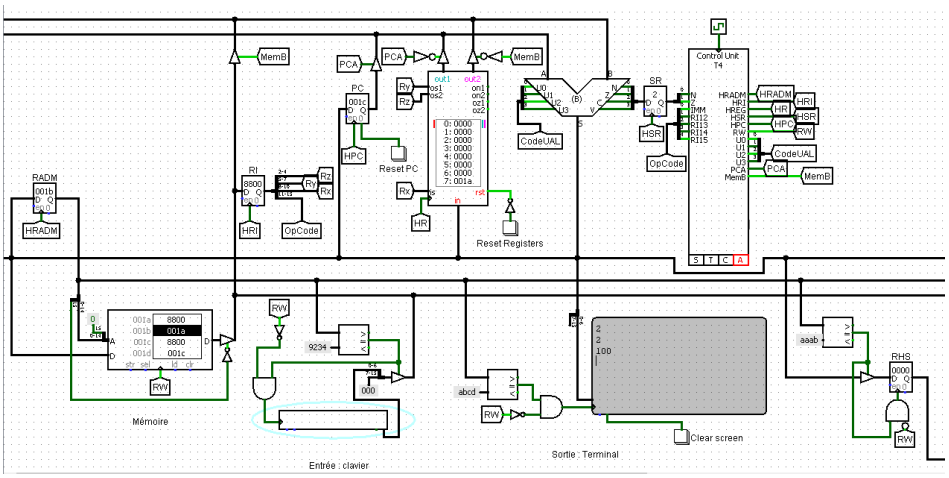
Le compilateur fonctionne bien, le processeur exécute bien le programme basic.niklaus (soit l'affichage du chiffre 2 puis affichage du chiffre 3).

1.4 2ème niveau de complétude

Pour ce niveau, il s'agit de traiter les variables, les assignations ainsi que l'instruction read.

Pour bien écrire l'instruction, il convient de rajouter la variable globale endread_id pour avoir les bonnes adresses tout comme ce qui a été fait pour endprint_id précédemment.

De plus, on rajoute une fonction parserExpression(Tree tree) qui permet de gérer pour l'instant le cas ou une expression après une affectation\read\write est une variable ou un int.
Après génération du fichier var.arm puis var.mem, on charge ce dernier dans la mémoire du processeur et on lance la machine, on obtient le résultat suivant une fois la simulation lancée :



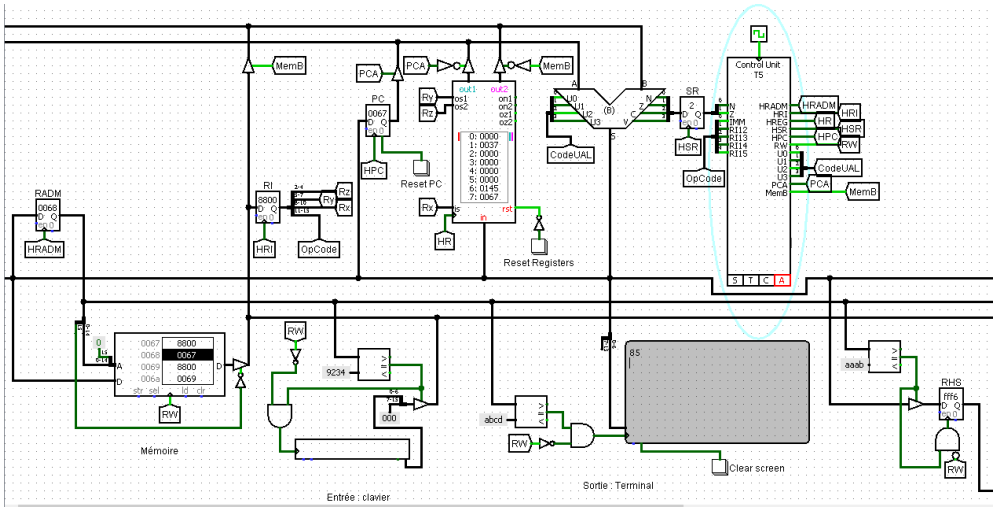
Le comportement du programme correspond bien au résultat attendu.

1.5 3ème niveau de complétude

Dans cette partie, l'objectif est de traiter les expressions génériques. On s'inspire du fichier calculations.niklaus pour compléter notre compilateur java.

On rajoute alors le cas ADDOP et MULTOP dans la fonction parserExpression(Tree tree).
Comme on utilise la pile, il faut l'initialiser au début du programme. Ainsi, on rajoute dans la fonction treatProgram() la ligne suivante : « `writer.println("mov r6, #pile_add");` »
Ainsi que le ligne : « `writer.println("@pile_add rmw 1");` » après traitement du bloc.

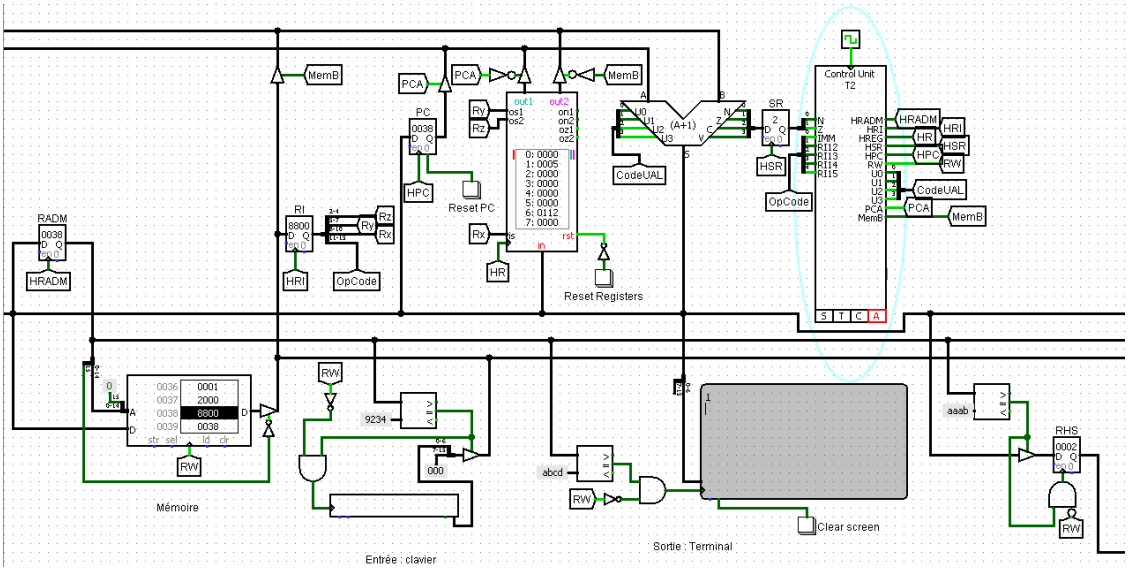
Après plusieurs tests, on remarque que si on ajoute la ligne « `writer.println("@pile_add rmw 1");` » avant les bibliothèques *PrintInt* et *ReadInt* le programme ne s'exécute pas correctement.
Il faut donc rajouter cette ligne à la fin pour que la pile ne déborde pas sur le programme.
Après génération du fichier calculations.arm puis calculations.mem, on charge ce dernier dans la mémoire du processeur et on lance la machine, on obtient les résultats suivants :



Le comportement du programme correspond bien au résultat attendu. (Affichage du nombre 85).

1.6 4ème niveau de complétude

Dans, cette partie il s'agit de rajouter la prise en compte des conditions d'un programme Niklaus.
On note le besoin de rajouter une variable globale de type int « if_id » pour le bon nommage des adresses.
Après génération du fichier comp.arm puis comp.mem, on charge ce dernier dans la mémoire du processeur et on lance la machine, on obtient les résultats suivants :



Le comportement du programme correspond bien au résultat attendu. (Affichage du nombre 1).
Pour le fichier test-if, il y a une erreur au niveau du parseur liée au « - » du nom du programme, on le remplace par « _ » pour que la génération du fichier test-if.arm ne pose pas de problème.
Le résultat obtenu est également celui attendu (à la vue de la suite des conditions).

1.7 5ème niveau de complétude

Ici, l'objectif est de rajouter la prise en compte des boucles while d'un programme Niklaus.
On a également besoin ici de rajouter une variable globale de type int que l'on nomme « while_id ».
Après avoir testé, les différents fichiers on obtient bien les résultats attendus.
Par exemple avec le fichier : fibo.niklaus on a le résultat suivant :