TME3 : Réalisation d'un analyser syntaxique pour le format de fichier ".vst"

Objectifs

L'objectif de ce TME est de vous familiariser avec les outils **flex** et **bison**, qui sont des outils de la distribution GNU permettant de générer automatiquement des *scanners* (un scanner est un analyseur lexical) et des *parsers* (un parser est un analyseur syntaxique) pour différents formats de fichiers.

Vous allez développer un analyseur lexical (scanner), puis un analyseur syntaxique (parser) pour le sous-ensemble de la grammaire du langage VHDL utilisé par la chaîne de CAO Alliance pour décrire la vue structurelle (format *.vst*). La structure de donnée construite en mémoire est la structure de données *lofig* présentée en cours.

Nous ne vous donnons pas la grammaire du format *.vst* mais vous devrez la construire en analysant la structure du fichier exemple.vst.

Vous pourrez également utiliser le fichier signal.vst qui ne décrit pas un composant complet mais simplement une liste de déclarations de signaux.

Ces deux fichiers sont disponibles dans le répertoire:

```
/users/enseig/encadr/cao/tme3/vst
```

Le sous-ensemble de VHDL visé permet de décrire des composants possédant des ports d'entrée/sortie, des signaux, et d'instancier ces composants en les connectants par les signaux. On fait l'hypothèse que tous les signaux sont de type bit, et il n'est donc pas demandé de traiter les vecteurs de bits.

Etape 1 : Ecriture de l'analyseur lexical

L'outil **'flex** est un générateur d'analyseur lexical. Il prend en entrée un fichier vst.l contenant la définition des "token" à reconnaître, et génère en sortie un fichier vst.yy.c qui contient le code C de l'analyseur lexical. On appelle *token* une suite de caractères lus dans le fichier texte qu'on cherche à analyser, et correspondant à une expression régulière : mot-clef, identificateur, etc... Le but de cette première étape est d'écrire le fichier *vst.l* définissant les expressions régulières du format *.vst*.

On rappelle qu'en VHDL, les lettres majuscules et minuscules ne sont pas différenciées. Cherchez dans le manuel de **flex** une manière simple de générer un analyseur lexical qui ne fasse pas la différence entre majuscules et minuscules.

1. Etablir la liste des mots clefs du format *.vst*, et en déduire les expressions régulières en langage **flex** permettant de les reconnaître.
2. Etablir la liste des opérateurs du format *.vst*, et écrire les règles en langage **flex** permettant de les reconnaître. On utilisera une classe pour les opérateurs possédant un seul caractère.

3. Etablir la règle de reconnaissance d'un identificateur, sachant qu'un identificateur du VHDL commence toujours par une lettre, suivied'un nombre quelconque de lettres, chiffres ou *underscore*, avec la contrainte que l'on ne doit pas avoir 2 *underscore* successifs.
4. Etablir la règle permettant d'absorber les commentaires, sachant qu'un commentaire commence par "--" et s'achève en fin de ligne.
5. Etablir la règle permettant d'absorber les caractères d'espace et les caractères de tabulation.

On donne ci-dessous une partie du fichier *vst.l* que vous devez écrire.

```
%{
#include <string.h>
#include <stdio.h>
int yylineno = 1; /* compteur de numero de ligne */
%}
définitions de macro
%%
[ \t]           { /* Rien de rien */ }
\n              {yylineno++;}
regle1          {printf("TOKEN1: %s\n", yytext);}
regle2          {printf("TOKEN2: %s\n", yytext);}
regle3          {printf("TOKEN3: %s\n", yytext);}
%%
int main(void)
{
   yylex();
   return 0;
}
```

Modifiez ce fichier en introduisant les règles manquantes, et définissez les commandes d'affichage qui permettent d'afficher la chaîne de caractères correspondant à chaque token reconnu.

La compilation du fichier *vst.l* s'effectue comme suit, à charge pour vous d'intégrer cela dans un Makefile:

```
flex -t vst.l > vst.yy.c
gcc -Wall -Werror -o scanner vst.yy.c -lfl
```

flex déclare une macro non utilisée dans le cas présent, ce qui déclenche un avertissement justifié au moment de la compilation. Pour éviter cela, il faut définir dans le prologue du fichier *vst.l* la macro YY_NO_UNPUT.

Etape 2 : Modification de l'analyseur lexical

On cherche maintenant à modifier l'analyseur lexical pour préparer la communication entre l'analyseur lexical et l'analyseur syntaxique. On va demander à l'analyseur lexical d'afficher non plus la chaîne de caractères associée à chaque token, mais un entier représentant le type du token reconnu.

Pour cela:

- Définir dans le prologue du fichier *vst.l* un type énuméré qui associe à chaque token un entier définissant son type. On utilise en général des valeurs supérieures à 255 pour les token correspondant à une chaîne de plusieurs caractères, de façon à pouvoir directement utiliser le code ascii du caractère dans le cas d'un token constitué d'un seul caractère.
- Modifier les actions associées aux règles afin qu'elles retournent le type du token. Par exemple :

```
entity   {return T_ENTITY;}
```

- Modifier la fonction main() pour qu'elle affiche le type des tokens reconnus. On rappelle que *yylex* retourne 0 lorsque la fin de fichier est lue.

Etape 3 : Ecriture d'un premier analyseur syntaxique

L'outil **bison** est un générateur d'analyseur syntaxique. Il prend en entrée un fichier *vst.y* contenant la définition des règles de la grammaire correspondant au format *.vst*, et génère la fonction yyparse() constituant l'analyseur syntaxique, dans un fichier vst.tab.c.

Dans ce TME, il ne sera pas question d'analyse sémantique ni même d'actions à effectuer lorsqu'une règle est activée. On se borne donc à effectuer l'analyse de syntaxe en s'assurant que les structures grammaticales définies permettent effectivement d'analyser le fichier *exemple.vst*.

Cette étape vise à analyser le fichier *signal.ex*, qui ne constitue qu'une partie du fichier *exemple.vst*. On cherche principalement à traiter le problème général des règles permettant de reconnaître un nombre variable d'arguments (une liste de signaux dans notre cas).

1. Définir les règles de grammaire correspondant a la déclaration des signaux en analysant le contenu du fichier *signal.ex* qui vous est fourni, et implanter ces règles dans le fichier *vst.y*.
2. Dans le fichier *vst.y*, compléter la déclaration des différents token utilisés par le parser susceptibles d'être reconnus par

   le scanner.

1. Comme aucune action n'a été associée à chacune des règles de grammaire, vous utiliserez le mode debug de **bison** pour vérifier que l'analyse se passe bien. Pour activer ce mode, il faut utiliser l'option -t sur la ligne de commande de **bison** et affecter une valeur non-nulle à la variable globale yydebug dans la fonction main().
2. Il faut par ailleurs que **bison** fournisse à **flex** la liste des tokens qu'il utilise. Pour cela, il faut utiliser l'option -d dans la ligne de commande de **bison**, pour lui demander de générer un fichier *vst.tab.h' contenant ces définitions. Il faut également modifier le fichier* vst.l *pour inclure le fichier* vst.tab.h *soit inclus dans le fichier* vst.l *et supprimer les définitions de token existantes dans* vst.l.
3. Il faut enfin modifier la fonction main(), qui est maintenant définie dans le fichier *vst.y*, et doit appeller la fonction yyparse().

Voici une ébauche du fichier *vst.y* que vous devez modifier et compléter:

```
%{
#include <string.h>
#include <stdio.h>

extern int      yylex(void);
extern int      yylineno;
extern FILE     *yyin;

int yyerror(char *s)
{
   fprintf(stderr, "%s line %d\n", s, yylineno);
   return 1;
}
%}

%token ...

%%
prod : regle TOKEN
     | ...
     ;
%%
```

```
int main(int argc, char *argv[])
{
   if(argc != 2) {
        printf(usage : %s filename\n", argv[0]);
        exit(1);
   }
   yyin = fopen(argv[1],"r");
   if(!yyin) {
        printf(cannot open file : %s\n", argv[1]);
        exit(1);
   }
   yydebug = 1;
   yyparse();
   return 0;
}
```

La compilation s'effectue comme suit, à charge pour vous d'intégrer cela dans le Makefile. Nous vous rappellons que la compilation de **flex** doit dépendre du fichier d'entête généré par **bison**.

```
bison -t -d vst.y
gcc -Wall -Werror -o parser vst.tab.c vst.yy.c -lfl
```

Etape 4 : Fin de l'écriture de l'analyseur syntaxique

Une règle de grammaire est construction grammaticale utilisant des token reconnus par *flex*... ou d'autres constructions grammaticales, plus simples. La grammaire a donc une structure arborescente.

Dessinez explicitement le graphe représentant la grammaire associée au format *.vst*. Dans ce graphe, qui a - presque - une structure d'arbre, chaque noeud correspond à une construction grammaticale (c'est à dire une règle de grammaire), et un arc orienté entre deux noeuds X et Y signifie : "La construction grammaticale Y est contenue dans la construction grammaticale X". Les constructions grammaticales élémentaires sont les tokens, et constituent les "feuilles" de l'arbre.

La syntaxe de la construction grammaticale correspondant à la déclaration d'un signal a déjà été définie. Il reste donc à :

1. Définir la construction grammaticale correspondant à la délaration d'un port
2. Définir la construction grammaticale correspondant à la déclaration d'un composant
3. Définir la construction grammaticale correspondant à la déclaration d'une entité
4. Définir la construction grammaticale correspondant à la déclaration d'une instance
5. Définir toutes les règles nécessaires à l'analyse du fichier complet.

Vous pouvez faire des validations intermédiaires simplement en utilisant le symbole racine:

```
%start regle
```

Vous pouvez considérer que le TME est terminé lorsque le parser construit à partir des deux fichiers *vst.l* et *vst.y* est capable de lire et de reconnaitre la totalité du fichier *exemple.vst*.

Compte-Rendu