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 analyseurs lexicaux (ou *scanner*) et des analyseurs syntaxiques (ou *parser*) pour différents formats de fichiers.

Vous allez développer un analyseur lexical, puis un analyseur syntaxique pour le sous-ensemble 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. 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. Dans ce TME3, on se limitera à l'analyse grammaticale du format*.vst*. La construction effective de la structure de données *lofig* sera réalisée dans le TME4.

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
```

Commencez par créez un répertoire *tme3*. Il est recommandé de créer un sous-répertoire pour chacune des étapes de ce TME, et à recopier vos fichiers d'un répertoire dans le suivant, de façon à conserver les résultats intermédiaires.

Etape 1 : 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, et en particulier la fonction yylex(). 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* qui définit les expressions régulières correspondant à tous les "tokens" utilisés par format *.vst*.

1. On rappelle qu'en VHDL, les lettres majuscules et minuscules ne sont pas différenciées. Cherchez dans le manuel de **flex** Flex_Manual? une manière simple de générer un analyseur lexical qui ne fasse pas la différence entre majuscules et minuscules.
2. Etablir la liste des mots clefs du format *.vst*, et en déduire les expressions régulières en langage **flex** permettant de les reconnaître.
3. 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.
4. 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.
5. Etablir la règle permettant d'absorber les commentaires, sachant qu'un commentaire commence par "--" et s'achève en fin de ligne.
6. 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. Comme vous pouvez le constater, ce fichier se termine par la définition du programme main() qui fait appel à la fonction yylex(). La variable globale yyin est un pointeur sur le fichier à analyser. La variable globale yytext est un pointeur sur la chaîne de caractère correspondant au token reconnu.

```
%{
#include <string.h>
#include <stdio.h>
int yylineno = 1;     /* numero de ligne (pour les messages d'erreur) */
#define YY_NO_UNPUT
%}

%%
[ \t]                  {}
\n                     {yylineno++;}
expression1           {printf("TOKEN1: %s\n", yytext);}
expression2           {printf("TOKEN2: %s\n", yytext);}
expression3           {printf("TOKEN3: %s\n", yytext);}
%%

int main(int argc, char *argv[])
{
  if(argc != 2) {
    printf("\n******* usage : %s filename \n", argv[0]);
    exit(1);
  }

  yyin = fopen(argv[1], "r");

  if(!yyin) {
    printf("\n******* cannot open file : %s \n", argv[1]);
    exit(1);
  }

  yylex();
  return 0;
}
```

Modifiez ce fichier en introduisant les règles correspondant aux différents "token" à reconnaître, 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
```

Validez votre travail en lançant l'analyseur lexical sur les fichiers *signal.vst* et *exemple.vst*.

```
$ scanner ../signal.vst
```

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.
- Lancer cette nouvelle version de l'analyseur lexical sur les fichiers *exemple.vst* et *signal.vst*.

```
$ scanner ../exemple.vst
```

Etape 3 : 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.

Cette étape vise à analyser le fichier *signal.vst*, qui ne constitue qu'une partie du fichier *exemple.vst*, et contient une liste de déclarations de signaux. On cherche à écrire une règle permettant de reconnaître un nombre variable d'arguments (une liste de signaux dans notre cas).

1. Déclarez dans le fichier *vst.y*, les différents token utilisés par le parser susceptibles d'être reconnus par le scanner.
2. Définissez les deux règles de grammaire correspondant a la déclaration d'une liste de signaux, et implanter ces règles dans le fichier *vst.y*. On définira successivement la règle décrivant un signal, puis la règle décrivant une liste de signaux. **Bison** permet d'associer à chaque règle une "action de compilation" qui est exécutée pendant l'analyse du fichier, au moment ou la règle est reconnue, mais dans cette première étape, on n'associera aucune action aux deux régles définies.
3. La fonction main() est maintenant définie dans le fichier *vst.y*, et doit appeller la fonction yyparse(). Il faut donc modifier la dernière partie du fichier du fichier *vst.l*, qui ne contient plus que la définition de la fonction

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

```
%{
#include <string.h>
#include <stdlib.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 ... déclarer ici les differents types de token

%%

... ecrire ici les règles grammaticales

%%

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 ces commandes dans le Makefile.

- L'option -d sur la ligne de commande de **bison** active le mode debug, qui vous permet de suivre le déroulement de l'analyse syntaxique du fichier. Il faut de plus affecter une valeur non-nulle à la variable globale yydebug dans la fonction main().
- L'option -t sur la ligne de commande de **bison** demande à **bison** de générer un fichier *vst.tab.h*. Ce fichier est utilisé par **flex**, et contient la définition des "tokens" utilisés par **bison**. Ce fichier doit être inclus dans l'en-tête du fichier *vst.l*, et la compilation de **flex** doit dépendre de ce fichier *vst.tab.h* généré par **bison**.

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

Validez l'ensemble en lançant l'analyse syntaxique du fichier *signal.vst*:

```
$ parser ../signal.vst
```

1. Lancez l'analyse du fichier *signal.vst* en utilisant les deux options - t et -d.N'associez aucune action aux deux règles de grammaire. Utilisez 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 *v' 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.

Etape 4 : Modification de l'analyseur syntaxique

Une règle de grammaire est construction grammaticale utilisant soit des token reconnus par *flex*, soit 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