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 Allianced

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 les fichiers d'exemples au format *.vst* qui sont disponibles dans le répertoire

```
~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

On appelle *token* une suite de caractères lus dans le fichier 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 diffÈrence 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
```

\texttt{lex} dÈclare une macro non utilisÈe dans le cas prÈsent, et il y a donc un avertissement justifiÈ au moment de la compilation. Pour Èviter cela, il faut dÈfinir dans le prologue du fichier vst.l la macro \texttt{YY_NO_UNPUT}.

Etape 2 : modification de l'analyseur lexical

On cherche maintenant à modifier l'analyseur lexical pour qu'il affiche 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.

%\small \begin{verbatim} entity {return T_ENTITY;} \end{verbatim} \normalsize

\item Modifier le \texttt{main} pour qu'il affiche le type des tokens reconnus.

Nous rappellons que \texttt{yylex} retourne 0 lorsque la fin de fichier est lue~;

\end{enumerate}

\section*{Etape 3 : dÈbut de l'Ècriture du \emph{parser}}

Il faut maintenant dÈfinir dans le fichier \texttt{vst.y} les rËgles de la grammaire correspondant au format .vst. Ces rËgles s'appuient sur les token reconnus par flex. Dans ce TME, il ne sera pas question d'analyse sÈmantique ni mÍme d'actions ? effectuer lorsqu'une rËgle est activÈe. On se bornera donc ? effectuer l'analyse de syntaxe en s'assurant que les structures grammaticales dÈfinies permettent effectivement d'analyser le fichier exemple.vst.

Cette Ètape vise ? analyser le fichier \texttt{signal.ex}, qui ne constitue qu'une partie du fichier \texttt{exemple.vst}. On cherche principalement ? traiter le problËme des rËgles permettant de reconnaÓtre un nombre variable d'arguments (une liste de signaux dans notre cas).

\begin{enumerate} \itemsep=-.5ex

\item dÈfinir les rËgles de grammaire correspondant ? la dÈclaration des signaux VHDL

en analysant le contenu du fichier \texttt{signal.ex} qui vous est fourni, et implanter ces rËgles dans le fichier vst.y.~;

\item Dans le fichier vst.y, complÈter la dÈclaration des diffÈrents

token utilisÈs par le parser. On dÈclarera tous les token susceptibles d'Ítre reconnus par le scanner.~;

\item Comme nous n'allons pas effectuer d'actions pour ces

rËgles, nous utiliserons le mode \emph{debug} de bison pour vÈrifier que l'analyse se passe bien. Pour activer ce mode, il faut utiliser l'option \texttt{-t} de bison et positionner la variable \texttt{yydebug} ? une valeur autre que zÈro dans la fonction main()~;

\item Il faut par ailleurs que \texttt{bison} fournisse la liste des

\emph{tokens} ? \texttt{flex}. Pour cela, il faut utiliser l'option \texttt{-d}, pour demander ? bison de gÈnÈrer un fichier \texttt{vst.tab.h} contenant ces dÈfinitions. Il faut donc modifier le fichier vst.l pour qure le fichier \texttt{vst.tab.h} soit inclus dans le fichier vst.l et supprimer les dÈfinitions de token existant par ailleurs dans vst.l.

\item il faut Ègalement modifier la fonction \texttt{main}, qui est maintenant dÈfinie

dans le fichier vst.y, et doit appeller la fonction \texttt{yyparse()}.

\end{enumerate}

Voici une Èbauche du fichier \texttt{vst.y} que vous devez modifier et complÈter:

%\scriptsize \begin{verbatim} %{ #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;

} \end{verbatim} \normalsize

Nous rappellons que la compilation s'effectue grosso-modo ainsi, ? charge pour vous d'intÈgrer cela dans le \texttt{Makefile} dÈj? constituÈ pour \texttt{lex}. Nous vous rappellons que la compilation de \texttt{lex} doit dÈpendre du fichier d'entÍte gÈnÈrÈ par \texttt{yacc}.

%\small %\scriptsize \begin{verbatim} bison -t -d vst.y gcc -Wall -Werror -o parser vst.tab.c vst.yy.c -lfl \end{verbatim} \normalsize

\section*{Etape 4 : suite et fin de l'Ècriture du \emph{parser}}

L'idÈe est de construire le \emph{parser} de maniËre incrÈmentale. La syntaxe de la dÈclaration des signaux ayant ÈtÈ dÈfinie, vous allez, dans cet ordre~: \begin{enumerate} \itemsep=-.5ex \item dÈfinir les rËgles de dÈclaration des ports en partant

de l'exemple \texttt{port.ex} qui vous est fourni~;

\item dÈfinir les rËgles de dÈclaration du composant en partant

de l'exemple \texttt{component.ex} qui vous est fourni~;

\item dÈfinir les rËgles de dÈclaration de l'entitÈ en partant

de l'exemple \texttt{entity.ex} qui vous est fourni~;

\item dÈfinir toutes les rËgles nÈcessaires ? la lecture du fichier

\texttt{exemple.vst}.

\end{enumerate}

Vous pouvez faire des validations intermÈdiaires simplement en dÈclarant un nouveau symbole de dÈpart gr?ce ? %\small \begin{verbatim} %start regle \end{verbatim} \normalsize

Vous pouvez considÈrer que le TME est terminÈ lorsque le parser construit ? partir des deux fichiers vst.l et vst.y est capable de lire et de reconnaitre la totalitÈ du fichier exemple.vst.

\small \subsubsection*{signal.ex} \begin{multicols}{2} \begin{verbatim} signal opc : bit; signal read : bit; signal lock : bit; signal a : bit; signal d : bit; signal ack : bit; signal tout : bit; signal it_0 : bit; signal req0 : bit; signal req1 : bit; signal gnt0 : bit; signal gnt1 : bit; signal sel1 : bit; signal sel2 : bit; signal sel3 : bit; signal sel4 : bit; signal sel5 : bit; signal sel6 : bit; signal sel7 : bit; signal sel8 : bit; signal sel9 : bit; signal snoopdo: bit; \end{verbatim} \end{multicols} \subsubsection*{port.ex} \begin{multicols}{2} \begin{verbatim}

```
port (

        clk : in bit; resetn : in bit; ireq : out bit; ignt : in bit; dreq : out bit; dgnt : in bit; opc
        : out bit; lock : out bit; read : inout bit; a : inout bit; d : inout bit; ack : in bit; tout :
        in bit; it_0 : in bit; it_1 : in bit; it_2 : in bit; it_3 : in bit; it_4 : in bit; it_5 : in bit;
        snoopdo: in bit

);
```

\end{verbatim} \end{multicols} \subsubsection*{component.ex} \begin{multicols}{2} \begin{verbatim} component PIR3000

```
port (

        clk : in bit; resetn : in bit; ireq : out bit; ignt : in bit; dreq : out bit; dgnt : in bit; opc
        : out bit; lock : out bit; read : inout bit; a : inout bit; d : inout bit; ack : in bit; tout :
        in bit; it_0 : in bit; it_1 : in bit; it_2 : in bit; it_3 : in bit; it_4 : in bit; it_5 : in bit;
        snoopdo: in bit

);
```

end component; \end{verbatim} \end{multicols} \subsubsection*{entity.ex} \begin{verbatim} entity EXEMPLE is

```
port (

        clk : in bit; resetn : in bit

);
```

end exemple; \end{verbatim} \scriptsize \newpage \subsubsection*{expemple.vst} \begin{multicols}{2} \begin{verbatim} end exemple; entity EXEMPLE is

```
port (

        clk : in bit; resetn : in bit

);
```

end exemple;

architecture structural of system is component PIR3000

```
port (

        clk : in bit; resetn : in bit; ireq : out bit; ignt : in bit; dreq : out bit; dgnt : in bit; opc
        : out bit; lock : out bit; read : inout bit; a : inout bit; d : inout bit; ack : in bit; tout :
        in bit; it_0 : in bit; it_1 : in bit; it_2 : in bit; it_3 : in bit; it_4 : in bit; it_5 : in bit;
        snoopdo: in bit
```

```
        );

end component;

component PIRAM

        port (

                clk : in bit; resetn : in bit; sel : in bit; opc : in bit; read : in bit; a : in bit; d : inout
                bit; ack : out bit; tout : in bit

        );

end component;

component PITTY

        port (

                clk : in bit; resetn : in bit; sel : in bit; opc : in bit; read : in bit; a : in bit; d : inout
                bit; ack : out bit; tout : in bit

        );

end component;

component PIBCU

        port (

                clk : in bit; req0 : in bit; req1 : in bit; gnt0 : out bit; gnt1 : out bit; sel1 : out bit;
                sel2 : out bit; sel3 : out bit; sel4 : out bit; sel5 : out bit; sel6 : out bit; sel7 : out bit;
                sel8 : out bit; sel9 : out bit; snoopdo : out bit; resetn : in bit; opc : in bit; read : in
                bit; lock : in bit; a : in bit; d : inout bit; ack : inout bit; tout : out bit; it : out bit

        );

end component;

signal opc : bit; signal read : bit; signal lock : bit; signal a : bit; signal d : bit; signal ack : bit; signal tout : bit;

signal it_0 : bit; signal it_1 : bit; signal it_2 : bit; signal it_3 : bit; signal it_4 : bit; signal it_5 : bit;

signal req0 : bit; signal req1 : bit;

signal gnt0 : bit; signal gnt1 : bit;

signal sel1 : bit; signal sel2 : bit; signal sel3 : bit; signal sel4 : bit; signal sel5 : bit; signal sel6 : bit; signal sel7 : bit;
signal sel8 : bit; signal sel9 : bit; signal snoopdo: bit;

begin

        r3000 : Pir3000 port map (
```

```
        clk => clk, resetn => resetn, ireq => req0, ignt => gnt0, dreq => req1, dgnt =>
        gnt1, opc => opc, lock => lock, read => read, a => a, d => d, ack => ack, tout =>
        tout, snoopdo => snoopdo, it_0 => it_0, it_1 => it_1, it_2 => it_2, it_3 => it_3,
        it_4 => it_4, it_5 => it_5

);

rst : piram port map (

        clk => clk, resetn => resetn, sel => sel1, opc => opc, read => read, a => a, d => d,
        ack => ack, tout => tout

);

exc : piram port map (

        clk => clk, resetn => resetn, sel => sel2, opc => opc, read => read, a => a, d => d,
        ack => ack, tout => tout

);

bcu : pibcu port map (

        clk => clk, req0 => req0, req1 => req1,

        gnt0 => gnt0, gnt1 => gnt1,

        sel1 => sel1, sel2 => sel2, sel3 => sel3, sel4 => sel4, sel5 => sel5, sel6 => sel6,
        sel7 => sel7, sel8 => sel8, sel9 => sel9,

        resetn => resetn, opc => opc, read => read, lock => lock, a => a, d => d, ack =>
        ack, tout => tout, snoopdo=> snoopdo, it => it_1

);

ins : piram port map (

        clk => clk, resetn => resetn, sel => sel3, opc => opc, read => read, a => a, d => d,
        ack => ack, tout => tout

);

dat : piram port map (

        clk => clk, resetn => resetn, sel => sel4, opc => opc, read => read, a => a, d => d,
        ack => ack, tout => tout

);

tty : pitty port map (

        clk => clk, resetn => resetn, sel => sel5, opc => opc, read => read, a => a, d => d,
        ack => ack, tout => tout
```

```
        );

        ttz : pitty port map (

                clk => clk, resetn => resetn, sel => sel9, opc => opc, read => read, a => a, d => d,
                ack => ack, tout => tout

        );

end structural; \end{verbatim} \end{multicols}

\end{document}
```