

TME3 : Réalisation d'un parser de fichiers pour le format ".vst "

1. Objectifs

1. 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 donc développer un *parser* pour le sous-ensemble de la grammaire du langage VHDL utilisé par la chaîne de CAO Allied 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.

1 : Écriture du `\emph{scanner}` }

On rappelle qu'on appelle `\texttt{token}` une suite de caractères lus dans le fichier qu'on cherche à analyser, et correspondant à une expression régulière : mot-clef, identificateur, etc... Vous devez écrire le fichier `\texttt{vst.l}` définissant les expressions régulières correspondant au format .vst.

On rappelle qu'en VHDL, les lettres majuscules et minuscules ne sont pas différenciées. Cherchez dans le manuel de `\texttt{flex}` une manière simple de produire un `\emph{scanner}` qui ne fasse pas la différence entre majuscules et minuscules.

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

```
\item Établir la liste des mots clefs du format .vst, et en déduire les
```

```
expressions régulières en langage \texttt{lex} permettant de les reconnaître~;
```

```
\item Établir la liste des opérateurs du langage et les règles
```

```
permettant de les reconnaître. On utilisera une classe pour les opérateurs possédant un seul caractère~;
```

```
\item un identificateur du VHDL commence par une lettre, suivie
```

```
optionnellement d'autant de lettres, chiffres ou \emph{underscore}s que l'on veut, avec la contrainte que l'on ne doit pas avoir 2 \emph{underscore}s successifs. Établir la règle de reconnaissance d'un identificateur~;
```

\item un commentaire commence par \texttt{-\hspace{.2ex}-} et s'achève en fin de

ligne. Etablir la règle permettant d'absorber ces commentaires.

\item Etablir la règle permettant d'absorber les caractères d'espacement et les

caractères de tabulation.

\end{enumerate}

On donne ci-dessous une partie du fichier \texttt{vst.l} que vous devez écrire. Modifiez ce fichier en introduisant les règles que nous vous avons demandé d'établir, et définissez les commandes d'affichage qui permettent d'afficher la chaîne de caractères correspondant à chaque token reconnu.

```
%\small %\scriptsize \begin{verbatim} %{ #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);} reglen {printf("TOKEN3: %s\n",
yytext);} %% int main(void) {
```

```
yylex(); return 0;
```

```
} \end{verbatim} \normalsize
```

Nous rappelons que la compilation s'effectue grosso-modo ainsi, à charge pour vous d'intégrer cela dans un \texttt{Makefile}.

```
%\small %\scriptsize \begin{verbatim} flex -t vst.l > vst.yy.c gcc -Wall -Werror -o scanner vst.yy.c -lfl
\end{verbatim} \normalsize
```

\texttt{lex} déclare une macro non utilisée dans le cas présent, et il y a donc un avertissement justifié au moment de la compilation. Pour éviter cela, il faut définir dans le prologue du fichier vst.l la macro \texttt{YY_NO_UNPUT}.

```
\section*{Etape 2 : modification du \emph{scanner}}
```

On cherche maintenant à modifier le scanner 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~:

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

\item 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. ~;

\item Modifier les actions associées aux règles afin qu'elles retournent

```
le type du \emph{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 rappelons que `\texttt{yylex}` retourne 0 lorsque la fin de fichier est lue~;

`\end{enumerate}`

`\section*{Etape 3 : début de l'écriture du \emph{parser}}`

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

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

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

`\item` définir les règles de grammaire correspondant ? la déclaration des signaux VHDL

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

`\item` Dans le fichier `vst.y`, compléter la déclaration des différents

token utilisés par le parser. On déclarera tous les token susceptibles d'être reconnus par le scanner.~;

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

règles, nous utiliserons le mode `\emph{debug}` de bison pour vérifier 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 zéro 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 générer un fichier `\texttt{vst.tab.h}` contenant ces définitions. Il faut donc modifier le fichier `vst.l` pour que le fichier `\texttt{vst.tab.h}` soit inclus dans le fichier `vst.l` et supprimer les définitions de token existant par ailleurs dans `vst.l`.

`\item` il faut également modifier la fonction `\texttt{main}`, qui est maintenant définie

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

`\end{enumerate}`

Voici une ébauche du fichier `\texttt{vst.y}` que vous devez modifier et compléter:

```
%\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 rappelons que la compilation s'effectue grosso-modo ainsi, ? charge pour vous d'intégrer cela dans le `Makefile` déjà constitué pour `lex`. Nous vous rappelons que la compilation de `lex` doit dépendre du fichier d'entête généré par `yacc`.

```


\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'écriture du \emph{parser}}`

L'idée est de construire le `\emph{parser}` de manière incrémentale. La syntaxe de la déclaration des signaux ayant été définie, vous allez, dans cet ordre~: `\begin{enumerate}` `\itemsep=-.5ex` `\item` définir les règles de déclaration des ports en partant

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

`\item` définir les règles de déclaration du composant en partant

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

`\item` définir les règles de déclaration de l'entité en partant

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

`\item` définir toutes les règles nécessaires ? la lecture du fichier

`\texttt{exemple.vst}`.

`\end{enumerate}`

1. Objectifs

Vous pouvez faire des validations intermédiaires simplement en déclarant un nouveau symbole de départ grâce ?
`%\small \begin{verbatim} %start regle \end{verbatim} \normalsize`

Vous pouvez considérer que le TME est terminé lorsque le parser construit ? partir des deux fichiers vst.l et vst.y est capable de lire et de reconnaître la totalité 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; igt : 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; igt : 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; igt : 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, igt => 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}
```