

# TME4 : Ecriture d'un parser lofig

1. Introduction
2. Etape 1 : Communication entre le scanner et le parser
3. Etape 2 : Affichage du fichier complet
4. Etape 3 : Construction de la structure lofig

## Introduction

Ce TME constitue la suite du TME3 sur lex et yacc. Cette seconde partie a pour but de vous faire construire en mémoire la structure de données `\emph{lofig}` présentée en cours. A chaque règle importante de la grammaire correspond la création d'un objet de la structure de données `\emph{lofig}`.

Nous faisons l'hypothèse que le scanner et le parser du TME4 sont achevés. Vous êtes invités à utiliser vos propres fichiers `vst.l` et `vst.y`, mais vous pouvez également utiliser les fichiers `vst.l` et `vst.y` fournis avec ce TME. ces fichiers contiennent cependant quelques erreurs que vous devrez corriger afin de vous obliger à les comprendre.

## Etape 1 : Communication entre le scanner et le parser

Dans le TME3, portant sur l'analyse syntaxique du format `.vst`, le scanner transmettait au parser des numéros de tokens définissant le type des tokens reconnus. cela a permis de vérifier que les règles de grammaire étaient correctement écrites, et que les constructions grammaticales du format `.vst` étaient correctement reconnues.

Chaque fois que le scanner reconnaît un token, il peut également transmettre au parser une valeur associée à ce token, en utilisant la variable globale `yyval`. Le type de la valeur associée à un token peut être différent suivant le token (ce peut être une valeur numérique, ou un pointeur sur chaîne de caractères, ou autre chose). Il faut donc définir, pour chaque token auquel on souhaite associer une valeur, le type de la valeur qui sera stockée dans la variable `yyval`.

Dans cette étape, on ne va pas directement construire la structure de données `\emph{lofig}`, mais on va se contenter d'introduire dans le parser les `\emph{actions de compilation}` lui permettant d'afficher sur le terminal (en utilisant `printf`) le texte correspondant aux différentes constructions grammaticales reconnues. Dans cette première étape, on se limitera même à afficher les constructions grammaticales correspondant à la partie `entity` du fichier analysé. Le but ici est simplement de vérifier le mécanisme de transmission de valeurs entre le scanner et le parser.

Il faut pour cela modifier les deux fichiers `vst.l` et `vst.y`.

- Déterminez quels tokens retournent une valeur significative

(les tokens n'ont pas tous une valeur). Dans le fichier `vst.y`, typez les tokens qui doivent l'être en utilisant la construction `\verb+%token <type> TOKEN+`.

- On rappelle que pour chaque token reconnu par le parser, la chaîne de caractères

correspondant à ce token est stockée dans un buffer pointé par la variable `yytext`, et que le scanner réutilise ce même buffer pour chaque nouveau token. Quand on souhaite que le scanner transmette cette chaîne au parser, il faut que le scanner recopie cette chaîne de caractères dans un autre buffer et transmette au parser un pointeur sur ce nouveau buffer. Il faut donc modifier le fichier `vst.l` en

conséquence.

- Le type de la variable `yyval` contenant la valeur du token est défini

dans le fichier `vst.y`, en utilisant le mot clé `%union`. Modifier le fichier `vst.y` en conséquence.

- Pour permettre aux différentes règles du parser de communiquer entre elles, il faut également

définir le type de la variable associée à chacune des règles du parser qui renvoient une valeur. On rappelle que la valeur associée à un token dans une règle est stockée dans la variable `$i` (où `i` est l'index du token dans la règle), et que la valeur du token défini par la règle (c'est à dire la valeur du membre de gauche) est stockée dans la variable `$$`. Modifier le fichier `vst.y` pour définir le type des token associés aux règles du parser en utilisant la construction `%type <type> règle`.

- Ajouter les affichages, c'est à dire les actions de compilation associées aux règles

qui interviennent dans la partie "entity" du fichier `exemple.vst`. On pourra pour cela définir dans le fichier `vst.y` un objet de type `port_t`, permettant de représenter un ensemble de ports sous forme de liste chaînée. Chaque objet `port_t` comporte un champs "NAME" définissant son nom, un champs "TYPE" définissant sa direction, et un champs "NEXT" permettant de construire une liste chaînée. On introduira dans les règles "port" et "ports" les actions permettant de construire la liste des ports, et on utilisera dans la règle "entity" une boucle `for` pour parcourir cette liste et afficher les ports.

*Avertissement : Bison émet un avertissement "type clash on default action" pour certaines règles, lorsque l'action de compilation n'est pas définie. En effet, il effectue par défaut l'opération `{ $$ = $1 }`, et il proteste lorsque les deux tokens n'ont pas le même type. Il faut définir une action de compilation vide `{ }` pour éviter ce problème.*

## Etape 2 : Affichage du fichier complet

Dans cette seconde partie, on souhaite compléter les actions de compilation définies dans le fichier `vst.y`, de façon à afficher la totalité du fichier `exemple.vst`, sous une forme qui respecte la syntaxe du format `.vst`. Il faut donc traiter les constructions grammaticales correspondant à la partie "architecture" du fichier.

1. On commencera par introduire dans la règle "architecture" les actions de compilations permettant d'afficher le début et la fin de la section "architecture".
2. Introduire dans les règles "component" et "signal" les actions de compilation permettant d'afficher les composants et les signaux.
3. Introduire dans les règles "instances", "links" et "link" les actions de compilation permettant d'afficher les instances. On pourra pour cela définir dans le fichier `.vst` un objet de type `link_t`, permettant de représenter un ensemble de links sous forme d'une liste chaînée. On pourra s'inspirer de ce qui a été fait dans la première étape pour l'objet `port_t`.
4. Vérifier que le fichier texte généré par votre parser respecte la syntaxe du format `.vst`, en utilisant le parser pour analyser le fichier qu'il a généré.

## Etape 3 : Construction de la structure lofig

La construction de la structure mémoire `lofig` fait appel à des fonctions spécifiques. Une page de manuel existe pour chacun des objets de cette structure et chacune de ces fonctions. Nous vous invitons à les consulter.

Les différents objets dont vous aurez besoin sont:

- `lofig` pour représenter l'entity et les component;s;
- `losig` pour représenter les signal;s;
- `loins` pour représenter les instance;s;
- `locon` pour représenter les port;s;
- `chain` est une structure utilitaire qui contient deux pointeurs.

`\end{itemize}`

La structure `chain` permet en particulier de construire des ensembles d'objets de même type sous forme de listes chaînées (ensembles de signaux possédant une caractéristique commune par exemple).

Les différentes fonctions dont vous aurez besoin sont: `\begin{itemize}\itemsep=-.4ex \item mbkenv` pour initialiser le fonctionnement des bibliothèques d'Alliance (cette fonction ne prend aucun paramètre et ne rend rien mais elle initialise un certain nombre de variables globales et elle lit l'environnement unix concernant Alliance. Elle doit être invoquée au début de la fonction `main()`). `\item addlofig` pour créer l'entity et les component;s; `\item getlofig` qui renvoie un pointeur vers une figure désignée par son nom; `\item addlocon` pour créer les port;s, aussi bien dans les figures que dans les instances; `\item addlosig` pour créer les signal;s; `\item addloins` pour créer les instance;s; `\item addchain` pour créer les bipointeurs; `\item reverse` pour «retourner» l'ordre des éléments dans une liste de type `chain`; `\item freechain+` pour libérer une liste créée par `\verb+\addchain`; `\item namealloc` pour insérer tous les noms (chaînes de caractères) dans un dictionnaire. Grâce à cette fonction,  $n$  pointeurs représentant  $n$  chaînes de caractères identiques pointent sur une unique zone de mémoire, permettant de tester l'égalité des pointeurs (par `==`) en lieu et place de l'égalité de chaque caractère (par `strcmp`). `\end{itemize}`

Un fichier utilisant ces types et fonctions doit inclure le fichier `alliance.h`, et l'édition de liens doit comporter `-lalliance`. Ces fichiers référencés se trouvent sous `~encadr/cao/include` et `~encadr/cao/lib`. Modifiez vos sources et votre `Makefile` en conséquence.

Indication : Pour construire la structure de données, on est amené à manipuler plusieurs structures `lofig` : On a une structure `lofig` correspondant à l'entity dont on souhaite construire la net-list, mais on est amené à construire une structure `lofig` pour chacune des figures instanciées (c'est à dire pour chacun des component). Les structures `lofig` associées aux component sont des boîtes "vides", qui ne contiennent que la liste de ports, mais pas d'instances et pas de signaux. Vous utiliserez deux variables globales, l'une pointant sur la figure représentant l'entity, et l'autre pointant sur le component en cours d'analyse, et vous construirez la structure de donnée dans l'ordre suivant:

1. ajoutez la figure;
2. ajoutez les ports de la figure et les signaux externes;
3. ajoutez les components;
4. ajoutez les signaux internes;
5. ajoutez les instances. Vous ferez l'hypothèse simplificatrice que les connecteurs de l'instance (port map) `\emph{ sont dans le même ordre } que les connecteurs du modèle (component);`

Pour vous aider à déboguer et vérifier que la construction est correcte, vous pouvez utiliser la fonction `viewlofig`. Vous pouvez également sauvegarder sur disque la structure de données construite en mémoire en utilisant la fonction `savelofig`. Il est préférable de changer le nom de la figure avant d'effectuer la sauvegarde, sans oublier d'utiliser la fonction `namealloc`.