TME4 : Ecriture d'un parser ".vst" vers "lofig"

Introduction

Ce TME constitue la suite du TME3 sur *flex* et *bison*. Cette seconde partie a pour but de vous faire construire en mémoire la structure de données *lofig* présentée en cours. A chaque règle importante de la grammaire correspond la création d'un objet de la structure de données *lofig*.

Nous faisons l'hypothèse que l'analyseur lexical et l'analyseur syntaxique réalisés dans le TME3 sont achevés et opérationnels. Vous êtes donc 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.

Attention : ces fichiers contiennent délibérement quelques erreurs, que vous devrez corriger, afin de vous obliger à les comprendre.

Etape 1 : Communication entre le scaner et le parser

Dans le TME3, portant sur l'analyse syntaxique du format ".vst", le scaner 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 scaner reconnait un token, il peut également transmettre au parser une valeur associée à ce token, en utilisant la variable globale `yylval`. 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 `yylval`.

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

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

- Commencez par déterminer quels sont les tokens qui retournent une valeur significative, car les tokens n'ont pas tous besoin de transmettre une valeur. En pratique, il n'y a qu'un type de token qui doit transmettre une valeur. Pour définir le type de la valeur transmise dans le fichier *vst.y*, utilisez la construction

```
%token  <type> TOKEN
```

- On rappelle que pour chaque token reconnu par le scaner, 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 (dans la variable `yylval`). Il faut donc mofifier le fichier vst.l en conséquence.

- Les valeurs transmises dans la variable `yylval` ont des types différents, qui dépendent du type du token reconnu. On utilise donc une `union` pour définir le type génétal de la vatiable `yylval` dans le fichier *vst.y*, qui doit donc être modifié en conséquence.

```
%union {
            char* string;
            void* pointer;
            int  index;
}
```

- 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 le membre de droite de 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 enfin les *actions de compilation* associées aux règles qui interviennent dans la partie "entity" du fichier *exemple.vst*. Pour chaque règle, cette *action de compilation* consiste simplement à afficher le texte de la règle en utilisant la fonction printf(). On pourra pour cela définir dans le fichier *vst.y* un objet de type `port_t`, permettant de représenter un ensemble de ports par une 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 la 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.

```
typedef struct port {
        struct port     *NEXT;
        char            *NAME;
        int             TYPE;
} port_t;
```

Avertissement : Bison émet un avertissement "type clash on default action", 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 complêter 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 toutes les constructions grammaticales correspondant à la partie "architecture" du fichier.

- On commencera par introduire dans la règle "architecture" les actions de compilations permettant d'afficher le début et la fin de la section "architecture".
- Introduire dans les règles "component" et "signal" les actions de compilation permettant d'afficher les components et les signaux.
- 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`.
- 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 de données *lofig* fait appel à des fonctions spéficiques. 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` permet de construire des listes chaînées.

La structure *chain* (parfois appelée *bip*, comme *bipointeur*) contient deux pointeurs, et permet de construire des ensembles d'objets de même type, sous forme de listes chaînées.

```
typedef struct chain {
                struct chain     *NEXT;
                void             *DATA;
} chain_list;
```

Les différentes fonctions dont vous aurez besoin sont:

- `mbkenv()` pour initialiser `Alliance`.
- `addlofig()` pour créer l'`entity` et les `component`s;
- `getlofig()` qui renvoie un pointeur vers une figure désignée par son nom;
- `addlocon()` pour créer les `port`, aussi bien dans les figures que dans les instances;
- `addlosig()` pour créer les `signal`s;
- `addloins()` pour créer les `instance`s;
- `addchain()` pour créer les bipointeurs;
- `reverse()` pour «renverser» l'ordre des éléments dans une liste de type chain;
- `freechain()` pour libérer une liste créée par `addchain`;
- `namealloc()` pour insérer tous les noms dans un dictionnaire.

La fonction `mbkenv()` ne prend aucun paramètre et ne rend rien mais elle intialise un certain nombre de variables globales et elle lit l'environnement unix concernant Alliance. Elle doit etre invoquée au début de la fonction main().

Grâce à la fonction `namealloc()` différents pointeurs représentant la même chaîne de caractère (et donc le même nom) pointent sur une unique zone de mémoire, permettant de tester l'identité entre deux noms par un simple test d'égalité des pointeurs. 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 *lofig*, on est amené à manipuler plusieurs structures de type *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. créez la figure;
2. créez les ports de la figure et les signaux externes;

3. créez les components;
4. créez les signaux internes;
5. créez les instances.

Vous ferez l'hypothèse simplificatrice que les connecteurs de l'instance (construction `port map`) sont dans le même ordre que les connecteurs du modèle (construction `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 pour éviter d'écraser le fichier initial, sans oublier d'utiliser la fonction `namealloc()` .

Compte-Rendu

Vous n'avez pas de compte-rendu écrit à rendre pour ce TME4. Il vous sera demandé une démonstation du parser *.vst* vers *lofig* au début du TME5.