Compilateur de site web

Projet d'analyse syntaxique

Labbe Emeric
Lahaye Valentin
Parpaite Thibault
Sarrabayrouse Alexis

Licence informatique (S6)
M. Dorbec Paul & M. Kardos Féri
& M. Salvati Sylvain & M. Zeitoun Marc
Groupe IN601A1
29 avril 2016

Résumé

Ce document a pour but de décrire le déroulement de notre projet d'analyse syntaxique réalisé en langage C au cours de notre semestre 6 de licence informatique.

L'objectif de ce dernier est d'implémenter un compilateur de site web. Plus exactement il s'agit ici de créer un langage de programmation qui permettra de décrire les pages web et d'éviter la redondance d'information, en séparant le contenu de la présentation.

Pour cela, nous avons tout d'abord écrit un analyseur lexical mis en corrélation avec un analyseur syntaxique qui nous ont permis de remplir une structure arborescente adéquate permettant de modéliser le langage.

En complément de ce travail, nous avons mis en place l'outil CMake pour permettre une compilation dynamique, portable et simplifiée de notre projet. Des jeux de test ont été mis en place afin de tester ce que nous avons réalisé, ceux ci sont décris dans la partie tests unitaires. Est aussi jointe à ce dernier, une documentation au format HTML Doxygen de l'ensemble des fonctions, macros et autres structures de données implémentées tout au long du développement.

Table des matières

1 Analyse lexicale

1.1 Reconnaître une étiquette et un symbole

Tel qu'indiqué dans le sujet, une étiquette (ou label en Anglais) ne doit pas commencer par la séquence **xml**. De plus, ce label doit être immédiatement accolé soit à : [{ ou bien /.
Si une de ces conditions n'est pas respectée, alors il s'agit d'un symbole
Etudions les deux règles suivantes :

```
a(?!b) { printf("a%c\n", yytext[0]); }
.      { printf("%c\n", yytext[0]);  }
```

Dans cet exemple, le contenu affiché sur la sortie standard varie suivant si le caractère **a** est suivi d'un caractère **b** ou non.

Résultat produit après analyse de la séquence **ac** :

```
ac
```

Résultat produit après analyse de la séquence **ab** :

```
a
b
```

Ce mécanisme, pourtant très puissant, n'étant pas implémenté dans FLEX, nous avons choisi de retourner un token illicite (note : par la suite il s'est asi nous détectons une étiquette commençant par la chaîne de caractères **xml**. Voici les différentes règles mises en place pour la gestion d'étiquettes et symboles (yylval n'a pas été mis ici par soucis de lisibilité).

```
LABELSET ([[:alnum:]'._])

%%

[[:alpha:]]{LABELSET}*|_{LABELSET}+/[{\[\/] { return LABEL; }
[[:alpha:]]{LABELSET}*|_{LABELSET}          { return SYMBOL; }
(?i:xml){LABELSET}*                         { return SYMBOL; }
```

1.2 La gestion des chaînes de caractères

La syntaxe du langage impose que les chaînes de caractères soient délimitées par des guillemets. Ce choix soulève un problème majeur, à savoir la possibilité de placer un guillemet dans une chaîne de caractère sans que celui-ci ne soit fermant. Pour cela, nous utilisons le caractère \ pour dé-spécialiser le caractère ". Cette solution n'est malheureusement pas suffisante. En effet, le problème se pose de nouveau dès lors que l'on souhaite écrire la chaîne suivante : \".

Pour répondre à cette problématique, nous avons choisi de réduire les anti-slash deux à deux. Chaque paire est réduite en un seul anti-slash. Pour chaque paire constituée, le lexer retourne le token CHARACTER.

```
<STRING>\\\\ { return CHARACTER; }
```

Afin de ne produire aucun conflit avec les autres règles du lexer, nous avons ajouté un état exclusif STRING dans lequel ce dernier entre lorsque le caractère ” est matché dans l'état INITIAL.

```
\" { BEGIN STRING; return '"'; }
```

A l'issu de la réduction des anti-slash, trois cas sont possibles :

— Il reste un anti-slash suivi d'un guillemet : Le lexer retourne le token CHARACTER

```
<STRING>\\["] { return CHARACTER; }
```

— Il reste un anti-slash seul : Le lexer retourne le token CHARACTER

```
<STRING>[^"]  { return CHARACTER; }
```

— Il reste un guillemet seul : Le lexer retourne le caractère ” et sort de l'état STRING

```
<STRING>["]   { BEGIN INITIAL; return '"'; }
```

— Pour tout autre cas, le lexer retourne le token CHARACTER

1.3 Le problème des espaces

Notre première intuition fut de regrouper au maximum toutes les séries d'espaces rencontrées par le lexer puis, pour chaque groupement, faire retourner par ce dernier le token SPACES. Le traitement des espaces optionnels avait donc lieu dans la grammaire à l'aide de la règle suivante :

```
spaces : SPACES
       | %empty
       ;
```

Cette gestion des espaces nous a très rapidement conduit à de nombreux conflits SLR(1) *shift/reduce*. Ces derniers apparaissaient notamment lorsque deux non-terminaux construit à avec des espaces facultatifs se suivaient. Face à ce problème, nous avons choisi de changer de paradigme et de ne retourner que les espaces essentiels à la validité du langage à savoir, les espaces illicites. Un espace est illicite entre une étiquette et l'accolade, le crochet ou l'anti-slash suivant cette étiquette.

Nous avons implémenté ce paradigme à l'aide d'un nouvel état exclusif nommé RETSPACES dans le lexer. L'analyseur lexical doit impérativement y entrer à

la suite d'une étiquette où de la sortie d'une chaîne de caractères (élément le plus à droite d'un attribut). Cet état est composé d'une unique règle regroupant les espaces et retournant le token SPACES s'il y en a. En dehors de cet état, les espaces sont consommés sans être retournés. La rencontre de tout autre caractère différent d'un espace dans RETSPACES provoque le retour du lexer dans l'état initial. Le caractère en question est alors renvoyé sur l'entrée standard pour être ré-analysé.

```
<RETSPACES>[[:space:]]+ { BEGIN INITIAL; return SPACES; }
<RETSPACES>.            { BEGIN INITIAL; yyless(0);     }
```

Dans la suite du projet, après être tombé sur de nombreux conflits et cas particuliers où les espaces étaient génants, nous avons décidé de supprimer cet état et de ne retourner aucun espace (sauf dans les chaînes de caractères). Cela a été possible grâce au mécanisme de symbole d'avance de lex pour retourner des tokens illicites dans des cas non valides.

1.4 Importer un fichier

Une autre fonctionnalité demandée par le sujet était de pouvoir importer des variables, fonctions et autres arbres construits dans un autre fichier. La syntaxe à mettre en place était la suivante :

```
$a->f
```

Importe l'élément f du fichier a disponible dans le répertoire courant.

```
$.a->f
```

Importe l'élément f du fichier a disponible dans le répertoire du dessus.

```
$a/b->f
```

Importe l'élément f du fichier b disponible dans le répertoire a

```
$.a/b->f
```

Importe l'élément f du fichier b disponible dans le répertoire a. Ce dernier étant accessible à la racine du répertoire courant.
De manière générale :

- Le nombre de point correspond au nombre de fois que l'on doit remonter dans l'arborescence
- La chaîne de caractères à droite des points (ou du dollar s'il n'y a pas de points) et comprenant le dernier slash correspond au répertoire
- Celle située entre le dernier slash et la flèche au nom du fichier
- Enfin, l'étiquette à droite de la flèche correspond à l'élément à importer

De nouveau, nous avons fait le choix de déclarer un nouvel état exclusif dans le lexer pour ne pas créer de conflits avec les règles implémentées précédemment. Cet état nommé IMPORT comporte les règles suivantes :

```
[$] { BEGIN IMPORT; return '$' }
```

On entre dans l'état IMPORT lorsque l'on rencontre un *$*.

```
<IMPORT>[.] { return '.'; }
```

Retourne un point lorsque l'on match un point.

```
<IMPORT>[^.][^/]*[/]([^/]+[/])* { return DIRECTORY; }
```

Toute chaîne de caractères ne commençant pas par un point et se finissant par un slash est un répertoire. Le lexer retourne alors le token DIRECTORY.

```
<IMPORT>[^./][^/]*-[>] {
                          BEGIN INITIAL;
                          yyless(yyleng - 2);
                          return DOCUMENT;
                        }
```

Toute chaîne de caractères ne commençant pas par un point, se terminant par une flèche et n'étant pas composée de slash est un nom de fichier. Dans ce cas, l'analyseur lexical retourne le token DOCUMENT. On sort de l'état IMPORT et la flèche est renvoyée sur l'entrée standard pour être ré-analysée.

2 Analyse syntaxique

2.1 Choix d'implémentation

2.1.1 Nos différentes règles

Règles initiales

Tout d'abord nos forçons notre grammaire à commencer par la règle **start** avec la déclaration de symbole de départ :

```
%start start
```

Ainsi nous ne pouvons commencer que par la règle **root** qui sera notre arbre complet une fois toutes les réductions effectuées.

Un fichier est composé d'un **header** contenant les déclarations globales (let suivi d'un point virgule) et des actions (emit), puis une suite **d'expression-partielle**.

Expression

Dans notre projet, nous considérons que "tout est expression". C'est-à-dire qu'une expression peut-être :

— Une expression arithmétique
— Une expression booléenne
— Une expression conditionnelle
— Une affectation
— Une fonction anonyme
— Un match
— Un import
— Une expression-partielle (notion de priorité, voir ci-dessous)

La notion de priorité dans les expressions

Nous avons défini plusieurs règles afin d'imposer une priorité à nos expressions (nous avons repris le même principe que celui de la calculatrice).

Une expression-partielle (un haut niveau de priorité) peut être :
— Une forêt
— Un content, c'est-à-dire une chaîne de caractères
— Une expression parenthésée (par exemple une expression arithmétique parenthésée)

Les expressions de priorités maximales sont les **expression-ari-f** qui sont :
— Expression-partielle

— Un nombre
— Une application
— Un symbole (variable)

Le niveau de priorité en-dessous est **expression-ari-e** qui peut être une multiplication (*) ou une division. Pour cette dernière nous avons utilisé le token DIVIDE (DIVIDE étant un double slash : //) afin de la différencier du backslash après un LABEL, puis les textbfexpression-ari-e (addition et soustraction). Une expression-ari-e pouvant être une expression-ari-t, on va chercher à obtenir une expression-ari-t afin de la réduire en expression-ari-e, les opérations de multiplication et de division seront donc effectuées en premier, la priorité est donc bien respectée.

Viennent ensuite les **expression-booleenne-t** qui sont des opérations de comparaison (inférieur, inférieur ou égal, supérieur, supérieur ou égal, égal) qui elles comparent deux expression-ari-e.
Enfin nous avons la règle **expression-booleenne-e** qui correspond aux deux opérateurs logiques OU/ET.
Ainsi dans une expression de type :

```
4 * 8 + 3 > 2 + 2 * 6 || f 1 2 \leq 6 - 1
```

on réalise d'abord les applications (s'il y en a), puis les multiplications, les additions et soustractions, ensuite les comparaisons et enfin le ET/OU. L'ordre est donc respecté.

Dans les expressions nous avons aussi défini la règle **expression-conditionnelle**, celle ci est de la forme IF expression-booleenne-e THEN expression ELSE expression, notre condition est donc une expression-booleenne-e (donc réduite au maximum avec les priorités), les statements quand à eux sont des expressions.

Les affectations (let)

La règle **let** peut permet de réaliser une affectation dans un sous environnement (les règles sont analogue pour leurs homologues récursifs) :

```
LET SYMBOL affect IN expression
'(' expression WHERE SYMBOL affect ')'
```

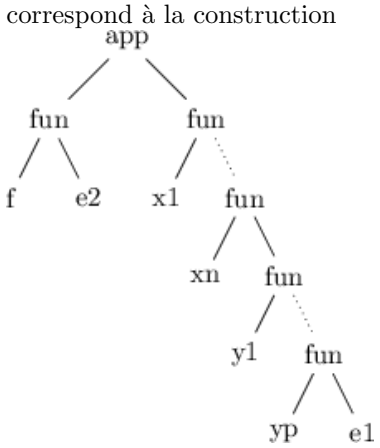
Nous avons choisi le faire de parenthéser les where car sinon il peut y avoir une apparition d'ambiguïté. Par exemple dans l'expression suivante, quel parenthésage est correct ?

```
 x where x = 3 * 2
(x where x = 3) * 2   ou bien   x where x = (3 * 2)
```

Une affectation doit être construite avec une associativité à droite, car le symbole le plus proche du = est associé à l'expression, etc. Par exemple

```
let f x1 ... xn = fun y1 ... yp -> e1 in e2
```

correspond à la construction



Par conséquent, nous avons décidé d'utiliser la règle suivante :

```
affect : SYMBOL affect
       | '=' expression
       | ARROW expression
       ;
```

Construction d'un mot

La règle **word** définit un mot, qui sera composé d'un ou plusieurs caractères (token CHARACTERE), word ne peut donc pas être vide.

La règle **word-list** comprend un ou plusieurs word, chacun séparé par un espace (token SPACES) obligatoire.

La règle **content** définit tout ce qui peut se trouver entre double quote (""), on peut donc avoir une word-list précédée par des espaces optionnels (règle spaces).

Les attributs

La règle **attribute** commencera forcement par un LABEL (définit dans le lexer) suivi d'espaces optionnels, du symbole = et d'un content. On aura alors

quelque chose de la forme : href=”www.labri.fr”

La règle **attribute-list** a été faite sur le même modèle que word-list, elle peut donc contenir un ou plusieurs attribute, chacun séparé par un espace obligatoire.

La règle **attributes** définit une attribute-list compris entre crochet : [href=”www.labri.fr” id=”lien”], on peut aussi avoir une empty-list donc deux crochets contenant ou non des espaces : [ ]

2.1.2 Les forêts

La règle **label** commencera toujours par un LABEL (pas de LABEL_XML), sera ensuite suivit par :

- un backslash, on aura alors : br/
- la règle attributes suivit d'un backslash : a[href=”www.labri.fr” id=”lien”]/
- un block (définit en dessous)
- la règle attributes suivie d'espaces optionnels et d'un block

La règle **body** peut avoir 5 formes différentes :

- être vide :

```
%empty
```

- une application
- une application suivit d'une virgule obligatoire et d'un autre body
- un content suivit d'espaces optionnels et d'un autre body
- un set puis un body

La règle **block** est un body entre accolades.

La règle **set** peut être soit un block soit un label, elle commencera donc toujours par une accolade ouvrante ou un LABEL. Cela correspond à une forêt.

Opérations sur les arbres et filtrage

Les opérations sur les arbres se font par la règle **match** qui se dérive par un token **TMATCH** suivi d'une **expression** suivi d'un token **WITH** puis d'un **patterns** et se fini par un token **END**.

La règle **patterns** permet de réduire la liste des opérations associées à ce match en se dérivant de la façon suivante : ” pforest ARROW expression patterns

```
patterns : '|' pforest ARROW expression patterns
         | '|' pforest ARROW expression
```

La règle **pforest** permet de réduire la forêt de filtre (pattern) associée à l'opération.

Et enfin, la règle **pattern** permet de réduire les différents filtres applicables à l'expression du match.

2.2 Le module *word*

Ce module nous permet de constituer des mots et de gérer les caractères spéciaux HTML. Son fonctionnement est relativement simple. Tel que décrit précédemment, l'analyseur lexical retourne les caractères un à un après traitement des anti-slash. Ce module s'occupe donc simplement de les concaténer. A chaque concaténation, ce dernier détermine si le caractère à concaténer doit être transformé, c'est à dire si l'on doit ajouter à la chaîne courante le caractère ou son code HTML équivalent. Le sujet préconisait de se référencer à l'adresse `https://dev.w3.org/html5/html-author/charref` pour obtenir une liste exhaustive des caractères spéciaux HTML. Malheureusement, la solution proposée sur cette page n'est pas satisfaisante. En effet, chaque caractère spécial possède un code unique dont la valeur ne possède aucune relation directe avec les autres code présentés. Ainsi, l'implémentation de cette solution nécessiterait l'écriture d'un immense tableau permettant d'associer un caractère à son code HTML. Pour remédier à ce problème, nous avons trouvé une écriture équivalente standardisée permettant de décrire une entité HTML. Voici l'écriture en question :

```
{caractère} <=> &#{code ASCII sur trois digit};
```

Ainsi pour le caractère . , nous obtenons le code suivant :

```
. <=> &#046;
```

Nous avons implémenté l'algorithme en question à l'aide de la fonction *sprintf* :

```
sprintf(buf, "&#%03d;", c);
```

La grammaire pour la gestion des mots est la suivante :

```
content : '"' spaces word-list '"'
        | '"' empt-list '"'
        ;

word-list : word SPACES word-list
          | word SPACES
          | word
          ;

empt-list : SPACES
          | %empty
          ;
```

Une liste de mots est entourée de guillemets. Elle est soit vide, soit composée d'une suite de mots séparée par des espaces. Cette dernière peut être précédée d'espaces optionnels. Le dernier mot de la liste peut être suivi d'espaces ou non.

```
word : word CHARACTER
     | CHARACTER
     ;
```

Un mot est composée à minima d'un caractère.

2.3 La fonction emit

Pour l'implémentions de la fonction emit dans le fichier *machine.c* nous nous sommes appuyé sur la fonction **tree_to_xml()** réaliser au début du projet pour créer une fonction **ast_to_xml** qui écrit dans le FILE*, en paramètre, la représentation de l'arbre en paramètre.

Pour un rendu plus agréable nous avons ajouté l'indentation au fichier créé à l'aide du paramètre *depth* de la fonction qui représente le niveau de profondeur dans la hiérarchie du nœud actuel. Puis la fonction **indent** prend cette profondeur en paramètre avec un FILE* et écrit autant de tabulation que de niveau de profondeur.

3 Compilation et tests unitaires

3.1 Mise en place de l'outil CMake

Dans un soucis de portabilité et d'extensibilité, nous avons choisi d'utiliser l'outil CMake pour la compilation du projet. En effet, ce dernier permet une compilation séparée de l'ensemble des modules développés y compris ceux dépendants de FLEX et BISON. L'ajout de la commande **make doc** permet de générer la documentation du projet via l'outil Doxygen. Pour compiler le projet : *A la racine :*

```
mkdir build
cd build
```

Dans le dossier build :

```
cmake .. [OU] cmake -DCMAKE_BUILD_TYPE=Release ..
[OU] cmake -DCMAKE_BUILD_TYPE=Debug ..
```

Les deux premières options permettent de compiler en **-O3**. La dernière quant à elle active l'option **-g** permettant l'utilisation d'un debugger tel que **gdb**.

3.2 Écriture de tests unitaires

Tel que le sujet le suggérait, nous avons développé, tout au long du projet, des batteries de tests pour tester la robustesse et la fiabilité de notre implémentation. Dans un soucis de cohérence et dans le but de simplifier l'écriture des tests, nous avons commencé par écrire une API très simple devant être implémentée pour mettre en place un test unitaire. L'interface de l'API en question est la suivante :

```
typedef struct ut_s *ut_t;

typedef enum { UT_FAILED, UT_PASSED } ut_status_t;

extern ut_status_t ut_run (ut_t ut);
```

Cette interface déclare l'existence d'une fonction **ut_run**, prenant en paramètre un pointeur sur une structure **ut_s** et retournant les constantes UT_FAILED ou UT_PASSED suivant le résultat du test. La définition des champs de la structure ainsi que la fonction **ut_run** est laissée libre au programmeur selon les besoins et le fonctionnement du test à implémenter.

Exemple d'implémentation de l'API pour les tests unitaires du lexer :

```
struct ut_s {
    char     *s_input;
    tokens_t  sa_output[61];
    int       size;
};

ut_status_t ut_run (ut_t ut) {
    YY_BUFFER_STATE input = yy_scan_string(ut->s_input);
    int i = -1;
    while ((++i < ut->size) && ((tokens_t)yylex() == ut->sa_output[i]));
    ut_status_t status = ((i == ut->size) && (yylex() == 0))
        ? UT_PASSED : UT_FAILED ;
    yy_delete_buffer(input);
    return status;
}
```

Dans ce contexte, la structure **ut_s** contient un champs mémorisant la chaîne de caractère à tester, un tableau contenant la séquence de tokens attendue en sortie du lexer ainsi que la longueur de cette séquence. La fonction **ut_run** quant à elle se charge de vérifier que la séquence de tokens retournée par le lexer respecte et ne dépasse pas la longueur de celle attendue.

Enfin, nous avons simplifié au maximum la compilation des tests unitaires en implémentant une macro nommée **add_ut** au sein de CMake. Cette dernière permet de compiler un test et de l'ajouter à la liste des tests unitaires lancés par CMake à l'issu de la commande **make test**. Cette macro lève aussi un flag à la compilation pouvant être utilisé pour changer le comportement d'un module suivant si celui-ci est compilé dans le cadre d'un test unitaire ou non. Ce flag permet par exemple de ne pas inclure le fichier *parser.h* dans les sources du lexer suivant le contexte de compilation de ce dernier. En effet, il nous importe peu de considérer le parser lorsque l'on souhaite uniquement tester le bon fonctionnement du lexer.

```
#ifndef UT_LEXER
#include <string.h>
#include <tree.h>
#include <parser.h>
#else
#include <tokens.h>
#endif
```

Autre exemple d'utilisation du flag relatant d'un test unitaire dans les sources du lexer. Le parser n'étant pas inclus dans un tel contexte, la variable **yylval** n'existe pas. L'utilisation de ce flag permet donc d'éviter une erreur importante à la compilation.

```
<STRING>[^"] {
                #ifndef UT_LEXER
                    yylval.c = yytext[0];
                #endif
                return CHARACTER;
              }
```

Exemple d'appel à la macro **add_ut** pour la compilation du test vérifiant le traitement des chaînes de caractères par le lexer.

```
add_ut(lexer string "${PROJECT_BINARY_DIR}/src/lexer.c" "${FLEX_LIBRARIES}" DEFAULT)
```

4 Conclusion

Les objectifs principaux de ce projet ont été atteints, c'est-à-dire mettre en relation un analyseur lexical et un analyseur syntaxique afin de créer un langage de programmation permettant de décrire du contenu web. En particulier à partir d'un fichier nous pouvons l'analyser et reconnaître des erreurs de syntaxes tout en remplissant une structure abstraire permettant de l'analyser par la suite. Cependant la partie de mise en corrélation avec la machine abstraite n'a pas pu être aboutie par manque de temps, ce qui est plutôt dommage.

Ce travail nous a permis de mettre en application nos connaissances récemment acquises en analyse syntaxique tout en consolidant nos acquis de C.