Rapport du projet d’analyse syntaxique

Sommaire:

Introduction

Dans le cadre du projet en L3 Informatique en Analyse Syntaxique, nous devions réaliser un analyseur syntaxique en utilisant **flex** et **bison** pour un sous-ensemble du langage C, le TPC. **Un fichier bison .y a été fourni**, définissant la grammaire du langage et nous devions définir le lexique associé dans **un fichier flex .lex**, ainsi que le tracé de l'arbre abstrait pour un fichier bien écrit en utilisant le module **tree**.

Choix d'implantation

1) Définition du lexique

La grande majorité des lexèmes étant évident, leur explication pour la définition n'est pas nécessaire. Concernant les commentaires, nous avons repris ce qui a été fait durant les TPs en y ajoutant les commentaires sur une ligne, distingués par **"//"**.
Pour le lexème **IDENT**, il suffit de préciser que le premier caractère est soit une lettre de **A a Z**, soit un **"_"** puis de préciser qu'il peut être suivi par d'autres caractères légaux, y compris les chiffres avec la syntaxe **[...]*** (avec les bons caractères à la place des **"…"**). La définition du lexème **NUM** suit cette même logique, ainsi que **CHARACTER** en faisant bien attention qu'un caractère en **TPC** est entre guillemet simple comme **'p'**.

Au-delà de la définition du lexique, nous utilisons **yylval** afin de pouvoir transférer les lexèmes lus au fichier bison associé, permettant ainsi d'ajouter des informations à l'arbre abstrait.

2) Construction de l'arbre abstrait et les informations représentées

Pour construire l'arbre abstrait, nous utilisons le module **tree** fourni pour un TP. Au départ, tous les nœuds contenant des terminaux n'étaient pas aussi expressifs: il n'affiche que le rôle du terminal en question, si c'était un identifiant (ident) ou bien un type.
Pour ajouter plus d'informations, **nous récupérons les textes lus par le lecteur avec yylval**. Il suffisait ensuite d'ajouter un champ "name" a la structure "node", initialisé à NULL lors de la création d'un noeud, et d'utiliser la fonction **"strdup"** afin d'allouer un espace de bonne taille ainsi qu'une spécification plus précise au lexème associe (on souhaitait avoir par exemple **"cmpt"** pour **"int cmpt"** au lieu de **"ident"**).

Pour que l'affichage fasse le bon choix, nous avons changé la fonction **"printtree"** en lui faisant distinguer le cas ou le champ name avait reçu un nom par **"strdup"**. En effet, en l'utilisant à **NULL**, **il est facile de distinguer les nœuds qui possèdent un nom comme les terminaux et ceux qui n'en possèdent pas.**

Ainsi, un noeud contenant un terminal aura pour label_t son rôle en tant que terminal et un nom pour le rendre plus expressif, à l'exception des terminaux “statiques” comme les conditions **“if”** et **“else”** ou bien la boucle **“while”** dont nous avons bien pris le soin de synchroniser avec le tableau StringFromLabel en utilisant le lien vers stackoverflow précisant **une solution pour synchroniser plus facilement** le tableau et le type énuméré **label_t**.

3) Résolution de l'ambiguïté décalage/réduction

L'ambiguïté vient des deux règles définissant deux séquences de if: une sans else, et l’autre avec. En effet, lors qu’on arrive à cette situation:
Instr:

1. IF ‘(‘ Exp ‘)’ Instr ·
2. | IF ‘(‘ Exp ‘)’ Instr · ELSE Instr

En prenant cette suite d’instruction:

```
if (1){
      if (1)
            return
      else
            return
}
```

Il y a deux choix en conflit, le premier est d’utiliser d’abord **1** puis **2**. L’autre serait d’utiliser d’abord **2** et d’avoir utilisé **1** dans le Instr après le if du **2**, qui ne signifie pas la même chose. En effet, dans le premier, on serait dans la bonne lecture, relier le else au if le plus proche. Le deuxième serait donc mauvais car il repousserait l’empilement du else a un if moins proche, ce qui n’est pas le cas ici.
Pour régler ceci, on peut utiliser les syntaxe **%left, %right, %prec** disponible en bison pour définir des priorités au parseur.
Ici, on souhaite **prioriser l’action d’empiler le else au if le plus proche**. On l'indique donc en définissant **%right ELSE**. Cela n’est pas suffisant car dans la règle 1, on serait toujours à la fin de la séquence, il faut donc ajouter a la fin de la règle un **%prec symbole terminal** afin que dans la règle 1, un choix doit être fait après Instr. Ici, en indiquant ``**right THEN ELSE (voir le fichier .y)**, on définit les deux terminaux comme ayant la même priorité et par le **%right**, on spécifie la volonté de vouloir empiler.
Ainsi, en arrivant à la fin de la règle **1**, ou a la rencontre du else dans la règle **2**, le parseur va choisir d’**empiler au plus proche**, réglant le soucis.

On aurait pu "résoudre le problème" en acceptant la solution par défaut de Bison, malheureusement celle-ci faisait le choix de réduire par défaut, ce qui ne correspondait pas à la solution voulu pour ce cas.

4) Extension du langage

L'extension du langage propose de **ne pouvoir déclarer et initialiser uniquement les variables locales**. En particulier, nous ne pouvons pas écrire `int tmp =` … dans une variable globale en **TPC**.

Pour cela, il fallait remarquer que **la grammaire fournie initialement ne faisait pas de distinctio**n entre les variables globales et locales. Nous avons donc créé trois règles pour la **déclaration des variables locales ainsi que sa déclaration associée** et l'affectation à une expression, en faisant bien attention de changer aussi la règle utilisée dans la règle **"Corps"**.
Dans cette nouvelle définition de déclaration de variable locale, a la différence de celle globale, possède une **affectation** après le terminal **IDENT**, ou **Affectation** est est une règle ou il y a soit une affectation avec **'=' Exp**, ou bien rien.

Ainsi, nous pouvions affecter uniquement les variables locales.

5) Gestion des options à l'exécution et des messages d'erreurs

Nous avons utilisé le module **getopt** pour définir les différentes options disponibles. Nous avons fait un module **Format** utilisant donc le module **getopt** pour définir une fonction **read_option**, qui va lire le buffer argv du main et identifier les différents arguments et identifier si ceux-ci sont légaux (option avec - suivi d'un caractère ou bien - - avec un mot).
En cas d'option valides lues, ici **-h, ou --help et -t ou --tree,** on changeait certaines variables globales pour dire qu'une certaine option a bien été lue. Nous avons aussi pris le soin d'arrêter la lecture des arguments de argv en cas d'arguments/options invalides ou bien de l'option -h lue.

En cas de sélection de **l'option -t, une variable globale, valant initialement 0, passe à 1** pour indiquer que l'on veut afficher l'arbre. On utilise ensuite cette variable dans **la toute première règle définie,** afin d'accéder a print tree ou non.

En ce qui concerne les erreurs de parsing, nous affichons **un message indiquant la ligne et environs la colonne dans le code qui provoque cette erreur**. Nous utilisons donc les variables lineno et errorinline, défini dans le fichier flex .lex, ou nous modifions leur valeur selon les lexèmes lues par le lexeur (un retour a la ligne incrémenter de 1 la variable lineno).

Difficultés rencontrées

Nous avons eu quelques soucis liés au type énuméré **label_t** car nous n'avions tout simplement pas consulter la page spécifier en commentaire.

Le choix de l'affichage de certains nœuds pour l'arbre abstrait a aussi posé quelques soucis car nous n'étions pas à l'aise avec la syntaxe utilisée pour "se déplacer" dans une règle à l'aide des $$, $1, $2 etc…
Mais une fois que cela était moins obscur, il a été plus rapide de construire cet arbre.

Nous avions aussi eu le problème de **rendre les étiquettes de nœuds plus expressives:** Nous étions confronté au problème du **type énuméré label_t** qui est écrit en préprocesseur. Nous avons donc essayé de chercher du cote des chaînes de caractères mais **nous ne voulions pas changer le prototype de la fonction makeNode**. Pour cela, nous avons défini un champ name, contenant une étiquette plus spécifique pour un nœud, et nous l'utilisons lorsque nous avons besoin de rendre un terminal plus expressif dans l'affichage du noeud.

Conclusion

Le projet peut lire un programme lit dans le sous langage tpc, elle permet la construction d'un arbre abstrait par rapport à un programme, et elle nous avons gérer l'ambiguïté décalage/réduction, les supports de cours et l'utilisation des certains travaux realises en TP nous a grandement aidé à faciliter la conception de ce projet.

Nous avons apprécié travailler sur le projet, avec ce que nous avons produit, nous sommes assez convaincus de partir sur de bonnes bases pour la suite du projet en compilation.

Annexe: Compiler et exécuter le projet

Pour compiler le programme, effectuer au répertoire racine

make

Il existe une règle pour nettoyer le répertoire de ses fichiers objets et exécutables, il faut faire

make clean

Vous pouvez utiliser un script bash présent à la racine pour déployer le programme et effectuer les tests automatiquement qui se trouvent dans le répertoire test

L'exécutable nommé **tpcas** se trouve dans le répertoire bin et peut s'exécuter sans fichier spécifié.
Dans ce cas là, vous devez écrire votre programme sur le terminal.

En cas de besoin, il y a l'option -h ou –help pour avoir plus d'informations sur comment utiliser le programme.

Vous pouvez soit renseigner le chemin vers le fichier a lire en argument, ou alors en écriture sur la sortie standard avec **"tpcas < chemin"**

SYNOPSIS:

Usage ./tpcas [-options] [filepath]

OPTIONS:

-t, --tree :
print the respective abstract tree of your current tpc file.
It is made to have a better understanding of what this syntaxical analyser can deal with.

-h, --help :
helps using this program easier. You can learn how to execute it.