Nom 1 : TEKPA
Prenom : Max
Groupe TP : 1

Nom 2 : TRAN
Prenom : Thien-Loc
Groupe TP : 1

Projet d'Analyse syntaxique

Licence d'Informatique

--2023-2024--

Rapport : TPC

1. Modularisation

Nous avons décidé de découper le projet en quatre modules.

- flex.lex : Ce module contient l'analyseur lexical. Il définit les règles permettant de traiter le fichier d'entrée lexème par lexème, reconnaît les lexèmes et pour chaque lexème, lance une action. Toutes les règles définies dans ce fichier sont autorisées par le parseur.

- Parser.y : Ce module contient l'analyseur syntaxique. Il contient la grammaire du langage ainsi que des actions associées à chaque règle de grammaire. L'objectif du parser (analyseur syntaxique) est de prendre en entrée une séquence de lexèmes générés par l'analyseur lexical (flex.lex ci-dessus), et de construire un arbre syntaxique qui représente la structure grammaticale du programme source. Si les séquences de lexèmes générés par le flex ne sont pas définies, le parser renvoie un message d'erreur.

- Tree.h : Ce module contient des fonctions du langage C permettant de dessiner l'arbre l'abstrait de la structure grammaticale du programme source. Il est inclut dans parser.y.

- Main.c : Contient le point d'entrée de l'analyseur, gère les options et les messages d'erreurs. Il inclut parser.y et tree.h.

2. Choix d'implémentations

Nous avons commencé par la construction de l'analyseur lexical (flex.lex) en faisant les règles qui recouvrent l'ensemble des lexèmes définis par la grammaire de base. Pour pouvoir reconnaître les

commentaires sur plusieurs lignes, nous avons fait une conditions de démarrage. Après avoir fait les règles reconnaissables par la grammaire de base, nous avons ensuite commencé l'extension du langage avec les tableaux. Cependant, nous avons rapidement constaté que notre première version était mal construite, ne parvenant pas à identifier correctement l'ensemble des tableaux du langage. Nous avons du modifier quelques fois les règles permettant l'identification des tableaux.
Exemple : La version ci-dessous est mal construite, elle ne reconnaît pas tous les tableaux du langage et génère de nombreux conflits.

Dans la partie declarateurs

```
Declarateurs:
      Declarateurs ',' IDENT After_id
   | IDENT
   | TableauDeclarateur
   ;

TableauDeclarateur:
      IDENT '[' NUM ']' After_id
   ;
After_id:
   '[' NUM ']' After_id
   |
   ;
```

Dans la partie paramètre de fonction

```
ListTypVar:
      ListTypVar ',' TYPE IDENT After_param
   | TYPE IDENT
   | TableParametre
   ;
TableParametre:
      IDENT '[']' After_param
   ;
After_param:
      '[']' After_param
   |
   ;
```

Après avoir réussi la création des tableaux selon nos critères, nous avons entamé la construction de l'arbre abstrait conformément aux indications du projet. Pour cette tâche, nous avons opté pour les structures suivantes :

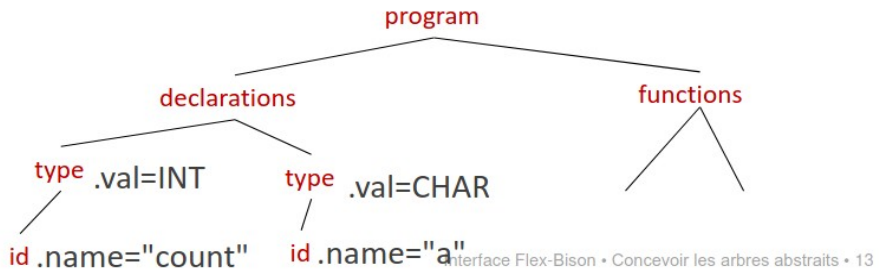

Figure 1: **Structure représentant l'ensemble de l'arbre**
Auteur : Eric Laporte.
Cours 7 d'Analyse syntaxique.
https://elearning.univ-eiffel.fr/pluginfile.php/387112/mod_resource/content/33/as7-bison-avance.pdf

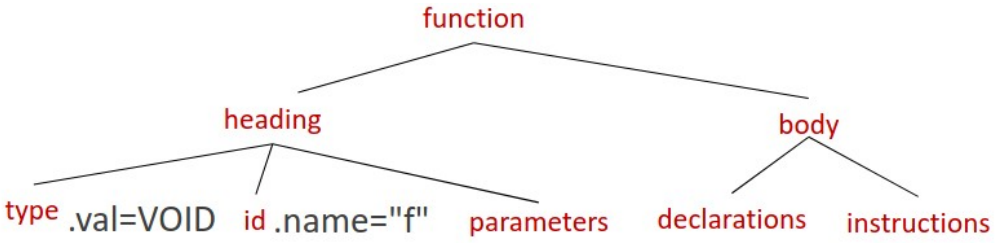

Figure 2: **Structure représentant une fonction dans l'arbre**
Auteur : Eric Laporte.
Cours 7 d'Analyse syntaxique.
https://elearning.univ-eiffel.fr/pluginfile.php/387112/mod_resource/content/33/as7-bison-avance.pdf

Choix de construction de l'arbre :
Dans la section de déclaration de variables, lorsque plusieurs variables partagent un même type, par exemple : int un, deux, trois;
Dans l'arbre abstrait, le type 'int' sera le nœud parent, avec les trois variables en tant que nœuds enfants de 'int'. En revanche, si plusieurs types sont utilisés avec des variables distinctes, chaque type aura sa propre variable en tant que nœud enfant. Nous estimons que ces deux représentations sont plus intuitivement compréhensibles.

Pour la représentation des tableaux dans l'arbre, nous les distinguons des autres identificateurs par le fait qu'ils aient des nœuds fils qui sont soit la taille du tableau dans la partie déclaration, soit leur expression dans le corps de la fonction, en revanche il y a aucune façon de distinguer les tableaux des autres identificateurs dans les paramètres d'une fonction.

Dans la grammaire de base, il y avait un conflit de décalage/réduction, pour enlever l'avertissement, nous avons fait une déclaration %expect 1 (dans parser.y). Dans notre cas 1 représente le nombre de conflits, Bison a bien trouvé ce nombre de conflits et il ne nous fait plus d'avertissement.

Difficultés rencontrées:

En ce qui concerne les difficultés rencontrées, il y avait la création des tableaux, nos règles ne reconnaissaient pas l'ensemble des tableaux et produisaient des conflits. Nous avons également rencontré des problèmes dans la reconnaissance de l'ensemble des éléments pouvant constituer des caractères littéraux en TPC. Par exemple, notre analyseur lexical ne parvenait pas à reconnaître les caractères de tabulation ('\t') et de saut de ligne ('\n').

De plus, lors de l'utilisation de la fonction `getopt()` pour gérer les options, nous avons constaté une limitation. Nous ne parvenions à reconnaître que les options simples telles que "-t" et "-h", tandis que les options longues telles que "--tree" et "--help" n'étaient pas correctement identifiées. En conséquence, nous avons opté pour une approche sans utiliser `getopt()` pour les options.

Conclusion:

Nous avons fait de notre mieux pour respecter scrupuleusement les directives énoncées dans le projet. En ce qui concerne la répartition des tâches au sein de notre binôme, nous avons commencé par collaborer pour écrire l'analyseur lexical, identifiant de manière exhaustive tous les lexèmes du sujet. Par la suite, nous avons réparti équitablement le reste des tâches, veillant à une répartition des responsabilités la plus équilibrée possible.