Rapport Projet Système Informatique

1. Partie Compilateur
2. Lex

La partie Lex utilise les expressions régulières pour reconnaitre les noms terminaux dans notre code, mais vu que le code est simplifié, cette partie ne présente pas de grande difficulté, ce qui est important c’est de connaitre le format de Lex et quelques conventions, comme les codes suivants :

yylval.nombre =atoi(yytext);

yylval.variable = yytext;

En utilisant l’union dans la partie de Yacc, il nous permet de retourner les noms de variables et les valeurs des entiers (ou bien exponentielle).

2. Table de symboles

Pour réaliser les opérations arithmétiques, il nous faut une table de symbole pour gérer les variables et les valeurs temporaires. Pour ce faire, nous avons créé le fichier symbole.c pour gérer ces parties. On a donc définit le structure de symbole :

**typedef** **struct**{

**char**\* name;

**enum** types type;

**bool** isInitialised;

**bool** isConstant;

} symbol;

On peut voir que nous avons défini le symbole avec un nom, un type (initialement pour identifier float ou INT, mais float n’est pas implémenté), un booléen pour identifier s’il est initialisé (ici on distingue si cette variable est seulement déclaré ou on l’a déjà donnée une valeur), et un booléen pour identifier s’il est constant. Les deux booléen nous permet de réaliser les tests d’erreur dans les parties suivantes.

Gestion de symbole :

* int pushSymbol(char \*name, bool isConstant, bool isInitialised, enum types type)
* void popSymbol();
* int findSymbol(char\* name);
* bool setInitialised (char\* name);

Chaque fois on voit un nouvel symbole (ce qui ne peut pas être trouvé par la fonction findSymbol), on le push dans la table. Comme ça on peut mémoriser tous les noms de variables dans la table, et on peut les mettre à jour en utilisant la fonction setInitialised.

Gestion de valeurs temporaires :

* int pushTmp( )
* int popTmp( )

La gestion de valeurs temporaires est plus facile que la gestion de symboles, car il suffit de connaitre dans quel registre que nous allons mettre la valeur temporaire. On utilise donc tout simplement une variable globale pour indiquer la position de la pile pour les valeurs temporaires. Ici on a choisi la stratégie classique, qui est de mettre les valeurs temporaires à la fin (le registre 256). Et on décrémente la position si on ajoute une nouvelle valeur temporaire, on incrémente si on fait une opération pop().

3. Assembleur

Pour traduire code C en assembleur et de les mettre dans un fichier, nous avons créé assembleur.c. Ses fonctionnalités sont simples , on a un tableau instructions qui stock tous les code assembleurs, une fonction add\_instruction pour réaliser les ajouts de code assembleurs.

Une fois Yacc termine l’analyse syntaxique, on appelle les fonctions

void write\_all\_instructions\_for\_human() ;

void write\_all\_instructions\_for\_machine();

Ces deux fonctions dessus permet d’écrire les code assembleur dans fichier asm\_human.txt et asm\_machine.txt. Le fichier asm\_human.txt est structuré pour une meilleure lisibilité, cela est utilisé pour débugger. Le fichier asm\_machine.txt suit les conventions pour l’interpréteur fournie sur moodle, cela est utilisé pour tester notre code assembleur.

4. Yacc

La partie Yacc décrit la grammaire de notre code C simplifié, nous avons réalisé cette partie en suivant ces étapes :  
 1. On décrit la grammaire de la manière générale. On test cette partie en lui donnant des simples code C, on voit s’il est capable de détecter les fautes syntaxiques.

文本

描述已自动生成 2. On ajoute les printf pour afficher les tokens détecté, le but c’est d’afficher les tokens trouvé comme un pile de opérations à réaliser.

Voici un petit exemple pour le code :

Int a ;

Int b = -1 ;

Int c = 5 ;

3. Dans cette étape, nous commençons à implémenter les opérations arithmétiques, avec l’étape 2, on est capable de considérer l’opération  a = b + a   comme :

文本

描述已自动生成

Ici pour a = b + a, on lui considère comme ‘Var tEGALE Calcul tPOINT\_VIRGULE’, donc ‘b + a’ présente dans la partie Calcul. Les entiers et les variables présente dans les opérations arithmétiques n’ont pas de même comportement que ceux qui présente ailleurs que la partie calcul. Pour la partie calcul, si on voit une variable ou un entier, on copie sa valeur (ou affecter la valeur pour un entier) dans la pile des valeurs temporaire. Pour le premier‘a’(qui n’est pas dans la partie calcul), on conserve que sa position dans la table de symbole.

Quand on voit l’opération arithmétique +, on pop deux valeurs temporaires et on les ajoute, on remet le résultat dans la pile temporaire. Quand on voit l’affectation avec calcul, on fait pop et on met la valeur de Tmp (ce qui stock la valeur de résultat a + b) dans Tmp+1(ce qui stock l’adresse de la variable a).

Voici donc le principe des opérations arithmétiques, la difficulté est de bien identifier quel opération pour les tokens trouvés, et souvent les même tokens n’ont pas de même comportement.

4. Dans cette étape, nous avons intégré la partie assembleur dans Yacc, ce qui est facile à réaliser. Nous utilisons tout simplement la fonction add\_instruction (dans le assembleur.h) pour ajouter les codes assembleur.

5.Implémentation de if et while, les principes de cette partie est bien expliqué dans le sujet. Nous avons choisi la stratégie de patcher les lignes à sauter :

void patch(int from, int to){

label[from] = to ;

}

Nous avons réalisé cette fonction avant la présentation de M. Eric Alata, donc on n’a pas utiliser $, qui est une méthode plus judicieux. On a construit une table de label[], qui stock les saute à faire, c’est-à-dire les lignes d’instruction dans le fichier qui stock les code assembleur qu’on est obligé de patcher à cause des if et while. De plus, on a créé une variable index\_label, qui comporte comme le sommet de table label[] (label[] se comporte comme un pile pour nous - last in first out). L’idée de notre implémentation est donc :

int line =get\_line\_asm();

patcher( label[index\_label], line);

//patcher la table avec l'index de cette ligne +1, on a un décalage de 1 déjà car get \_line compte les lignes à partir de 1

index\_label -- ;

//décrementer index\_label, c'est à dire que pour cet index , on a patché, plus rien à inquieter

Lors de l’implémentation de cette partie, nous avons constaté que le comportement de patch n’est pas la même pour if et if-else, car on doit sauter une ligne en plus pour if-else. Pour un moment, notre Yacc n’arrive pas à différencier if et if-else :

tIF tPARENTHESE\_OUVRANTE Condition { Code } tPARENTHESE\_FERMANTE Body

tIF tPARENTHESE\_OUVRANTE Condition { Code } tPARENTHESE\_FERMANTE Body tELSE

Car on a mis le code à exécuter au milieu, ce problème peut-être facilement résolu en mettant le code à la fin, mais on a perdu du temps pour identifier ce problème.

6. Traitement de l’erreur, avec la structure de symbole, on est capade détecter plusieurs erreurs. On a vu précédemment que dans la structure de symbole, on a un booléen isConstant et un champ isInitialiazed. Donc on est capable de détecter plusieurs erreurs :

-- Déclaration d’une variable déjà déclarée.

-- Opération arithmétique avec les variables non initialisées

-- Tout opérations (sauf déclaration) avec les variables non déclarées

-- Modification de la valeur d’une constante

5.Makefile et utilisation

Pour compiler la partie compilateur, veuillez utiliser makefile. Un exécutable nommé ‘exec’ sera généré.

Pour parser et convertir le code C en assembleur, utiliser la commande :

./exec < test.c

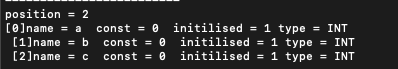
L’exécutable va afficher les informations sur la terminale :

Première partie : code requis

文本

描述已自动生成

Deuxième partie : table de symboles



L’exécutable va générer deux fichiers : asm\_machine.txt et asm\_human.txt. Pour vérifier, on peut utiliser l’interpréteur fourni sur Moodle en utilisant la commande ./interpréteur < asm\_machine.txt