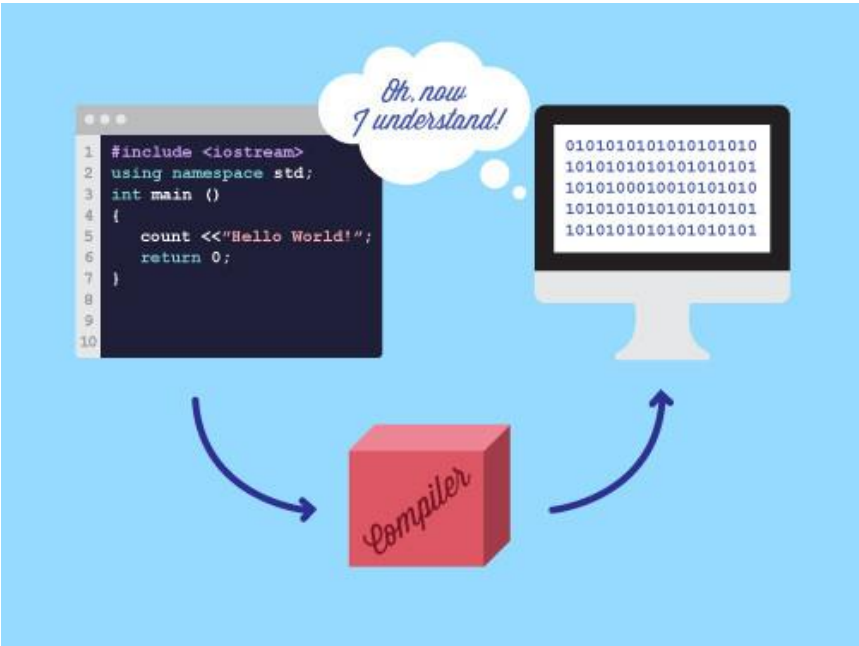
Compilateur



SOMMAIRE

Introduction

Dans le cadre des séances de travaux pratiques de la matière Compilation, au deuxième semestre de la 3e année de licence, nous avons été amené à développer un petit compilateur. Assez proche de l'idée du compilateur C (mots clés, principe, ...) , ce petit compilateur permet de reconnaître un langage C simplifié (variables, expressions, fonction, conditionnel, boucle, ...). Il fait ensuite la traduction en langage assembleur. Ce compilateur fonctionne conjointement à SIPRO, un émulateur de processeur et sur la base du langage assembleur associé. Avec cela, on peut simuler des calculs de processeur et voir le résultat de nos programmes. Ce compilateur est codé en flex et bison. Dans ce manuel nous aborderons la manière dont on utilise ce programme et les spécificités techniques le concernant.

Manuel Utilisateur

- Ouvrez un terminal dans le dossier asipro-master/asm et exécuter la commande “make”
- Ouvrez un terminal dans le dossier asipro-master/emul et exécuter la commande “make”
- Ouvrez un terminal dans le dossier src et exécuter la commande “make”
- Dans ce même terminal 2 options :
    - Taper soi même son code :
        - **./ex1 >../asipro-master/asm/file.asm**
        - Tapez votre code puis Ctrl + D pour finir la saisie
    - Envoyer un fichier de code source (.txt) en entrée :
        - **./ex1 < code.txt > ../asipro-master/asm/file.asm**
- Dans le terminal du dossier asipro/asm :
    - **./asipro file.asm ../emul/a.out**
- Dans le terminal du dossier asipro/emul :
    - **./sipro a.out**

Un code source doit contenir une fonction main qui retourne forcément un entier et de prototype :

int main () { <instructions> return <valeur> ; }

Elle sera appelée en début d'exécution du programme juste après les différentes déclarations de variables globales. Le programme s'arrête en affichant la valeur de retour de cette fonction.

Exemples

Ce petit compilateur est capable d'analyser la plupart des instructions d'un langage de programmation simple.

En effet, il peut reconnaître tout d'abord des expression au sens de **la valeur**, à savoir des calcules **d'entiers** (**+, - , * , /**), des **comparaisons** (**==, !=, >, >=, <, <=**), des calcules **booléens** (||, &&, !,). Il sait également interpréter la notion de **parenthésage**, les mots clés **true** et **false** et les **évaluer** dans des calculs.Il est capable de comprendre un **nom de variable au sens de sa valeur**. Il reconnaît aussi les **commentaires**.

En ce qui concerne les **instructions**, il possède **l'instruction vide**, **l'affichage** de valeur (**int, bool, string**), **réaffectation** (**ID** = nouvelle valeur;). Il sait aussi reconnaître et utiliser les **if**, **if-else** et **while**. Il connaît les **return** vide et **return valeur** pour les fins de fonction. Il connaît aussi la notion de **bloc d'instructions** (suite de déclarations suivie d'instructions le tout entre **accolades**).

Il connaît la syntaxe d'une **définition de fonction** (**type ID (param0 , … , paramN) bloc d'instruction**). Il peut aussi reconnaître les **paramètres** (**type ID**). Il gère les **appels de fonction** (**ID (expr0, … exprN);** avec N -1 le nombre de **paramètres**).

Pour le traitement des "suite de …" Il sait reconnaître des **suites d'expressions, de paramètres, d'instructions et de déclarations.**

Analyseur Lexicale

L'analyseur lexical, fichier flex(.l) permet de faire le lien entre ce qui est lu sur l'entrée et le fichier bison (.y) sur la base du maitre-esclave.

Il permet de reconnaître différentes unitées lexicales, comme les mot clé de notre langage C simplifié :

- **int / bool / true / false / return / print / if / else / while**

Il permet aussi de reconnaître différents opérateur booléens :

- **&& , || , ! , == , != , < , > , <= , >=**

Il permet également de reconnaître des nombres en les envoyant au fichier bison par l'intermédiaire du type union (entier) .

Même principe pour les identifiants (nom de variables, paramètres, fonctions) et les chaînes de caractère mais cette fois çi avec le type chaîne du type union.

Il fait donc certaines opérations comme l'allocation des différentes chaînes etc.. puis envoie au fichier bison un token représentant l'unité lue.

Il consomme les retours à la ligne / caractères blanc et renvoie sur la sortie standart les caractères non identifiés.

liste des token : **IF, ELSE, WHILE, PRINT, RETURN, INT, BOOL, TRUE, FALSE, AND, OR, NEG, EQ, NEQ, GEQ, LEQ, GT, LT, COMMENT, ID, STRING, NUMBER**

Analyseur Syntaxique

L'analyseur syntaxique permet de traiter les éléments de l'entrée via des token envoyées par l'analyseur lexical.

Pour la suite n est un entier > 0.

Le non terminal de base est ***lignes*** , il représente les lignes d'un programme, ce que l'on peut mettre sur une ligne.

Ensuite, ***instr*** qui représente les instructions possibles (règles du non terminale instr) . Ces instructions sont :

- instruction vide (expression ;)
- affichage sur la sortie (print expression ;)
- réaffectation (identifiant* = expression ;)
- instruction conditionnelle (if et if/else)
- boucle (while (expr) { instructions; })
- retour de fonction (return ; / return expression ;)
- blocs d'instructions ({ n declarations, n instruction })

* identifiant représente un variable (locale ou globale) / un paramètre

Un bloc d'instruction c'est 0 à n déclaration(s) suivie de 0 à n instruction(s)

Maintenant passons aux déclarations, avec ***decl*** qui permet de déclarer une variable (globale ou locale) de la forme **type ID = expr ;** avec *type* un autre non terminal valant soit **INT** soit **BOOL** pour entier et booléens et **ID**, l'identifiant (nom) de cette nouvelle variable.

Nous avons ensuite les expressions, avec le non terminal ***expr*** et ses règles qui représentent toutes les expressions possibles dans notre cadre. Voici ces expressions:

- addition (expr1 **+** expr2)
- soustraction (expr1 **-** expr2)
- division (expr **/** expr2) *expr2 ne vaut pas 0
- multiplication (expr1 ***** expr2)
- négation (**!** expr)
- ET Logique (expr1 **&&** expr2)
- OU Logique (expr1 **||**expr2)
- égalité (expr1 **==** expr2)
- différence (expr1 **!=** expr2)
- plus grand que (expr1 **>** expr2)
- plus grand ou égal (expr1 **>=** expr2)
- plus petit que (expr1 **<** expr2)
- plus petit ou égal (expr1 **<=** expr2)
- vrai (**true**)
- faux (**false**)
- identifiant (au sens de la valeur de la variable)
- chaine de caractère (variable globale non mutable)
- appel de fonction (sa valeur est la valeur du **return**)
- parenthésage (**(expr)**)
- nombre (repéré par le token **NUMBER**)
-

Autre élément important, la définition de fonction. Cette dernière est traitée par le non terminal ***deffun*** de la grammaire. La syntaxe est de la forme :

- type ID (paramètres) blockinstr

Dernier point, les suites de paramètres , d'instructions et de déclarations sont gérées par des non terminaux qui modélisent ce format de suite. Ils sont :

- **sparams , sinstrs, sdecls, sexpr**

Il y a aussi les commentaires qui sont gérés par ***comment*** par l'intermédiaire du token **COMMENT**.

En ce qui concerne les problèmes rencontrés, certains conflits réduction/décalage, notamment entre le if simple et le if/else ou encore sur le while avec les morceaux de code au milieux des différentes productions. Pour régler ce problème il fallait jouer sur les priorité (if - if/else) et ajouter des non terminaux "fictifs". Dans notre code ces non terminaux sont ***avinstr*** et ***whileexpr.***

Analyseur Sémantique

L'analyseur sémantique est un élément qui permet de contrôler le sens des lignes de codes. Comme par exemple la gestion du typage, que ce soit entre les opérateurs, à l'affectation, à la réaffectation, sur le type des valeurs de retours etc…

Pour certains cas, il fallait user de stratégie pour arriver à un résultat convaincant. Je pense notamment au codage des booléens. Il a été défini que 0 serait la valeur false et qu'un entier supérieur à 0 serait true. En effet, cette astuce permet de coder par la suite assez simplement les ET / OU Logiques à l'aide respectivement de la multiplication, de l'addition et de leurs éléments neutres respectifs.

En ce qui concerne les types, il y a premièrement un type énuméré qui possède les valeurs (**T_INT, T_BOOL, T_STRING, ERR_TYPE**) qui sont les valeurs que peuvent synthétiser les différents éléments de la grammaire. Il y a aussi l'utilisation d'une table des symboles (liste chaînée C) qui références des entrées pouvant êtres variable global, variable locale (ou paramètre) et fonctions. Cette liste donne différentes informations comme un nom (**ID**), une classe (**GLOBAL_VARIABLE, LOCAL_VARIABLE, FUNCTION**), un numéros (seulement les variables locales/paramètres) et un type (**VOID_T, INT_T, BOOL_T, STRING_T**). De plus, pour les fonctions il y a un nombre de paramètres, de variables global et les types des paramètres qui sont renseignées. Toutes ces informations permettent ainsi la bonne gestion de la sémantique, c'est-à-dire la cohérence des différents éléments du code entre eux. Exemple simple, on teste si les deux expressions entre l'opérateur "**+**" sont bien du type **T_INT.** Autre exemple, on teste si la valeur de l'expression d'un return est bien du même type que celui de la fonction.

Un booléens permet de vérifier que la fonction principale (appelée en premier par le programme) renvoie bien une valeur, entière pour pouvoir afficher cette valeur en fin d'exécution.

Dernière chose, un pointeur sur la fonction courante a été mis en place pour savoir si on est en train d'analyser une fonction ou pas. Cela permet par exemple de choisir entre variable globale ou local lors de l'ajout d'une nouvelle entrée dans la table des symboles. Cette ajout permet aussi d'avoir constamment l'entrée de la table contenant les informations de la fonction courante pour effectuer différents test lié encore une fois au typage ou autres. On peut donc tester si lors de l'analyse d'un **return** on se trouve bien dans une fonction ou non et réagir en conséquence.

Code Assembleur

Le code assembleur est envoyé vers la sortie standard (ici un fichier **.asm**) et est formaté de manière à créer un programme assembleur correct.

Avant même de débuter l'analyse (fonction **yyparse**), du code est généré en amont. Ce dernier permet d'initialiser les variables global du programme comme des chaînes de caractère important (message d'erreur de la division par 0, chaine true et false, chaine de fin de programme...). De plus, il initialise les registre **bp** et **sp** qui participent au bon fonctionnement de la **pile** et termine par un saut inconditionnel à l'étiquette repérant le début de programme.

Lors de l'analyse (exécution de la fonction **yyparse**) du code assembleur est généré pour chaque production de manière à organiser dans le code assembleur les différentes actions. Si l'on prend le cas des **expressions entières** (opération entre deux expressions) , le code généré **calcul la valeur** puis la **place en sommet de pile**. Grâce à cela, la prochaine production qui se servirait d'une expression, comme une déclaration de variable par exemple, peut générer du code assembleur qui **récupère la dernière valeur** calculée (**pop registre**) et la **stocke** à l'endroit approprié (nous en parlerons par la suite). Le fonctionnement des expressions booléennes est similaire, on calcule le résultat (comparaison par exemple) et on empile le résultat (**0** ou **1** ou un entier supérieur à zéro dans le cas d'un **OU Logique** par exemple).

Concernant les instructions, certaines vont proposer des codes assembleurs différents en fonction du **contexte**. C'est le cas du **print** qui pour un entier va simplement effectuer un **callprintfd** sur le sommet de pile, pour un booléen va tester le sommet de pile et afficher (**callprintfs**) en conséquence la chaine de caractère **true** ou **false**. Pour les chaînes de caractère, **l'adresse** de cette chaîne à

été stockée en **sommet de pile** donc on la récupère simplement (**pop registre**) pour l'afficher. L'instruction vide (**expr ;**) va juste dépiler le sommet de pile.

Pour tout ce qui tourne autour des **variables globales**, on se sert de **label unique** (généré par la fonction **create_label**) pour stocker les variables (**storew**) et pouvoir récupérer la valeur à tout moment. Ces labels sont aussi utilisés pour les **if**, **if-else**, et **while**. Il permettent de réaliser des saut (**jmp** ou **jmpc**) afin de passer un partie de code que l'on ne veut pas exécuter (cas des **if / if-else**) ou bien de revenir en arrière (cas d'une boucle **while**).

La définition de fonction va générer du code qui place un **label** de début de **routine**, servant à l'appeler via une instruction **call** puis empiler la valeur des différents registre pour sauvegarder le contexte d'exécution. De plus, on change de position **bp** (fond de pile) pour des calculs de **position de variable** (abordé par la suite). Les codes des différentes **déclarations / instructions** sont générés automatiquement grâce aux productions citées précédemment. Pour terminer, lors de l'appel à **return**, on met le résultat de la fonction dans le registre ax et dépile dans tous les autres registres pour reprendre l'ancien contexte d'exécution au retour de fonction en terminant par l'instruction **ret** qui revient à **l'ip** suivant directement le **call** de l'appel à cet fonction.

En ce qui concerne l'appel de fonction, on simule un "environnement" pour la gestion des variables locales, paramètres et adresse. On **empile** donc x fois la valeur **0** pour **réserver de la place**, x étant le **nombre de variables locales** connue grâce à l'entrée de la **table des symboles** de même nom que la fonction appelée. On continue en appelant la fonction avec un **call registre** (registre contenant le label de début de fonction) empilant donc l'adresse pour le **ret**. Ensuite c'est le code de la **définition de fonction** qui prend le relais. A la fin, on **dépile** autant de fois qu'il y a de **variables locales** et de **paramètres** pour dépiler cette **simulation de contexte**.

Dernière subtilité, pour les **variables locales**, **paramètres**, que ce soit pour l'accès en **lecture** ou en **écriture** (déclaration, récupération de la valeur, réaffectation...) , tout se base sur le nouveau **bp** et l'on calcule via une formule, **l'adresse de l'emplacement** de cette variable (encore une fois grâce aux informations de la **table des symbole**).

Pour finir, après exécution de **yyparse,** une fin de code est produite. Cette dernière permet d'afficher la **valeur de retour** de la fonction **main** (contenue dans le registre **ax**), de terminer le programme (instruction **end**) et de créer la pile d'entier (**label de pile** + **@int 0**).

Conclusion

En conclusion, je suis globalement satisfait des possibilités de mon compilateur, ce travail a été très instructif pour comprendre le fonctionnement des vrai compilateurs. J'aurais cependant voulu, avec un peu plus de temps, pouvoir régler les différents problèmes liés aux fonctions lors de l'exécution. En particulier concernant les passages de valeurs en paramètre et les déclarations locales sans doute lié à une utilisation erronée de la simulation de contexte vu avant. De plus j'aurais aimé pouvoir ajouter des types supplémentaires comme l'utilisation plus poussée des chaine de caractère (**string**) avec des chaine mutable par exemple ainsi que le type **void** pour utiliser le **return** simple sans valeur**.** Dernier point, je pense qu'il aurait ou être possible d'optimiser le compilateur en retirant les push, pop successifs et utiliser directement la valeur encore contenue dans le registre dans certaines conditions le permettant.