

Compilation

Rapport de Projet

Equipe:
ALLEMAND Fabien
LEBOT Samuel

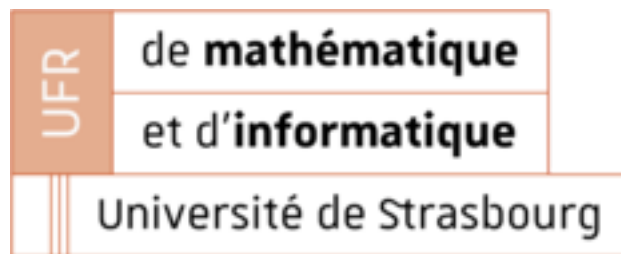


Table des matières

Liste des figures	1
1 Introduction	2
2 Analyse Lexicale	3
3 Analyse Syntaxique	4
4 Génération de Code MIPS	6
5 Conclusion	7
Bibliographie	8

Liste des figures

1	Exemple d'action Flex pour le symbole <code>+</code> et le mot clé <code>if</code>	3
2	Exemple d'action Flex pour les commentaires et les entiers	3
3	Exemple d'action Bison pour la multiplication de deux entiers	4
4	Déclaration du type <code>expr_val</code> dans le fichier <code>Bison</code>	4
5	Actions réalisées pour initialiser un programme lors de l'analyse syntaxique	5
6	Fonction effectuant la traduction d'un quadruplet <code>Q_ECHO</code>	6

1 Introduction

La compilation consiste à traduire un code source lisible par un humain en un code exécutable par un ordinateur. C'est à dire transformer un fichier texte contenant des instructions écrites dans un langage de programmation en un fichier binaire.

Ce projet consiste à écrire un compilateur pour un langage de programmation impératif simple appelé SoS qui utilise une syntaxe et des fonctionnalités issues d'un sous-ensemble de langage shell unix (Sous-Shell).

Écrit en C et en utilisant les outils Flex et Bison, le compilateur est capable de traduire un programme écrit en SoS en une suite d'instructions MIPS pouvant être exécutées à l'aide d'un simulateur.

Dépôt Git: <https://github.com/FABallemmand/ProjetCompilation>

2 Analyse Lexicale

La première étape de la compilation consiste à analyser les unités lexicales contenues dans un programme, c'est à dire découper le programme en blocs de taille la plus petite possible selon la syntaxe du langage de programmation.

L'analyse lexicale est réalisée à l'aide de **Flex**. Cet outil permet de définir des unités lexicales sous formes d'expressions rationnelles et d'associer une action à chacune d'elles.

On peut donc définir les unités lexicales utilisées dans un programme écrit en SoS. Dans le fichier `fsos.lex`, on définit tout d'abord les unités lexicales réservées au langage:

- Les symboles (+, -, *, ()...)
- Les mots clés (**if**, **for**, **test**...)

Puis les unités lexicales définies par l'utilisateur:

- Chaines de caractères
- Nombres
- Identifiants de variables ou de fonctions
- Commentaires
- Espaces et tabulations

On associe ensuite une action à chaque unité lexicale.

Pour les unités lexicales réservées au langages, cela consiste à renvoyer un *TOKEN*, c'est à dire une valeur numérique correspondant à une unité lexicale.

```
\+ return PLUS;
if return IF;
```

Figure 1: Exemple d'action Flex pour le symbole + et le mot clé if

Les unités lexicales ayant une valeur définie par l'utilisateur doivent être ignorées ou transmises (à l'aide de `yylval`, `yytext` ainsi qu'un *TOKEN*).

```
#[^\n]*\n ;
(((1-9)[0-9]*)|0) {yylval.val = strdup(yytext); return INTEGER;}
```

Figure 2: Exemple d'action Flex pour les commentaires et les entiers

Remarque | Les unités lexicales qui ne sont pas reconnues par l'analyseur lexical sont considérées incorrectes pour un programme SoS et mettent fin à la compilation.

Les valeurs renvoyées par l'analyseur lexical **Flex** sont transmises à l'analyseur syntaxique.

3 Analyse Syntaxique

Après avoir déterminé les blocs de taille minimale composant le programme, il faut vérifier s'ils sont assemblés de façon correcte. C'est à dire si le programmeur a écrit des instructions correctes et agencées convenablement selon la grammaire du langage de programmation.

L'analyse syntaxique est réalisée à l'aide de **Bison**, un outil permettant de définir la grammaire d'un langage de programmation et de définir des actions à effectuer pour chaque règle rencontrée dans le programme.

L'exemple de règle ci-dessous correspond à la multiplication de deux entiers. On y trouve:

- Une instruction qui permet d'afficher la règle lorsqu'elle est utilisée (utilisé pour du debugage)
- La création d'une opérande de type variable utilisable dans une instruction de code intermédiaire
- Un appel à la fonction `genCode` qui permet de générer le code intermédiaire (code à trois adresses) correspondant.
- Des affectations de valeurs de l'élément de droite à l'élément de gauche de la règle selon leur type

```
produit_entier
: produit_entier STAR operande_entier
{
    if (DEBUG)
        printRule(" produit_entier STAR operande_entier");
    $$ .firstquad = $1.firstquad;
    struct quadop result = quadop_var(newtemp());
    genCode(quad_new(Q_MUL, $1.result, $3.result, result));
    $$ .result = result;
}
```

Figure 3: Exemple d'action Bison pour la multiplication de deux entiers

La première partie du fichier `bsos.y` permet de définir des propriétés de la grammaire (priorité des opérations) ainsi que des outils pour l'analyse syntaxique (TOKEN et types d'éléments de règles).

Les types définis dans la section `%union` permettent d'utiliser chaque élément de règle comme une structure pour y stocker des informations à propager lors de la compilation.

```
%union{
    struct {
        size_t firstquad;
        struct quadop result;
    } expr_val;
}

%type <expr_val> produit_entier
```

Figure 4: Déclaration du type `expr_val` dans le fichier Bison

Dans l'exemple précédent on utilise `dollar dollar` pour accéder aux champs de `produit_entier` et y affecter les valeurs de l'adresse du premier *quadruplet* (code à trois adresses) correspondant à la multiplication et le résultat de cette multiplication.

La partie la plus importante de fichier Bison (la deuxième) contient toutes les règles de la grammaire et leurs actions. À cette étape de la compilation, on génère du code intermédiaire grâce à la fonction `genCode`. Les différents types de quadruplets (`Q_ADD`, `Q_EQUAL_STRING`, `Q_GOTO`...) et types d'opérandes de quadruplets (`QO_CST`, `QO_VAR`, `QO_UNKNOWN`...) sont définis dans le fichier `quad.h`.

L'adresse d'une instruction correspond à sa position dans un tableau de quadruplets contenant toutes les instructions du programme. Certaines instructions, notamment les `Q_GOTO`, contiennent des opérandes qui ne sont pas connues au moment où elles sont compilées. Il faut donc les enregistrer dans le quadruplet comme `QO_UNKNOWN`, les enregistrer dans le membre gauche de la règle dans une liste d'opérandes à compléter (`next`, `ltrue` ou `lfalse`) et les compléter par la suite grâce à la fonction `complete`.

C'est aussi à cette étape de la compilation que la table des symboles est créée. Cette structure a pour but d'enregistrer les variables déclarées dans le programme SoS à compiler ainsi que leur portée. On utilise pour cela une pile de contextes (voir `symbol_table.h` et `symbol_table.c`). Dans l'exemple ci-dessous, on constate que la fonction `pushContext` est utilisée pour empiler le contexte global.

```
initialisation
: %empty
{
    if (DEBUG)
        printRule("empty (initialisation)");
    pushContext();
    newName(S_GLOBAL, status, VAR, 0);
    genCode(quad_new(Q_AFFECT, quadop_cst(zero), quadop_empty(), quadop_var(status)));
}
```

Figure 5: Actions réalisées pour initialiser un programme lors de l'analyse syntaxique

On remarque aussi dans l'exemple ci-dessus l'appel à la fonction `newName` qui permet de créer une nouvelle variable dans la table des symboles, ici pour créer la variable ? (identifiant invalide pour un utilisateur) qui contient le status (le code de retour du programme). Une stratégie similaire est utilisée pour le code retour d'une fonction.

La fonction `newName` est aussi utilisée pour créer des variables temporaires dans la table des symboles qui ont elles aussi un identifiant invalide pour un utilisateur afin d'éviter tout conflit entre variables.

Par la suite, on peut vérifier si une variable est dans le contexte voulu grâce à la fonction `lookUp` et en précisant le niveau de contexte souhaité (courrant et/ou global).

4 Génération de Code MIPS

Après avoir analysé le programme en SoS et construit le code intermédiaire qui lui correspond il reste une dernière étape afin d'obtenir un programme qui fonctionne: traduire le code intermédiaire en code exécutable par une machine. Dans ce projet, les fichiers `translator.h` et `translator.c` permettent de traduire le code intermédiaire en code MIPS.

La traduction se déroule en deux étapes: on traduit en premier ce qui correspond au segment data puis ce qui correspond au segment text.

Pour le segment data, il faut enregistrer toutes les constantes utilisées dans le programme et afin de simplifier la traduction on ajoute l'adresse de la pile du contexte global et l'adresse de la pile du contexte précédent.

En ce qui concerne le segment text, il faut parcourir le tableau `global_code` qui contient l'ensemble des quadruplets constituant le programme et pour chaque quadruplet écrire le code MIPS correspondant dans le fichier de sortie.

Dans l'exemple ci-dessous, on reconnaît dans les `fprintf` les instructions MIPS pour charger l'adresse de l'unique argument de la fonction `echo_string`, convertir cet argument en entier, le placer en argument de fonction et appeler la fonction `echo_string`.

```
size_t echo_(int i, size_t nb_used_const, struct stack_frame *current_frame_list,
             size_t nb_nested_declaration)
{
    if (DEBUG)
        printCall("echo.");
    if (global_code[i].op1.kind == QO.CST)
    {
        fprintf(output_file, "jal $a0, const_%ld\n", nb_used_const++); // Charger l'
        // adresse de l'unique argument d'echo
    }
    else
    {
        printError("Argument invalide pour echo");
        exit(1);
    }

    fprintf(output_file, "jal string_to_int # Convertir le nombre de chaine en entier\n");
    fprintf(output_file, "move $a0, $v0 # Placer le nombre de chaine en argument de
    // fonction\n");
    fprintf(output_file, "jal echo_string # Appeler la fonction echo_string\n");

    return nb_used_const;
}
```

Figure 6: Fonction effectuant la traduction d'un quadruplet `Q_ECHO`

Il est important de créer des fonctions MIPS permettant d'effectuer les opérations, les comparaisons, les tests, `echo` et `read` qui pourront être appelées dans le programme MIPS résultat. Dans l'exemple ci-dessus la fonction `echo_string` est une fonction écrite en MIPS.

Cette bibliothèque de fonctions a été créée dans le fichier `string.asm`.

5 Conclusion

Le compilateur que nous avons écrit (bien qu'assez peu performant) permet de compiler correctement des programmes écrits dans le langage SoS.

L'analyseur lexical et l'analyseur syntaxique permettent de générer du code intermédiaire. Nous avons accordé une attention particulière à la table de symboles et à la gestion des contextes.

La traduction en code MIPS est assurée par un programme en C qui traduit le code intermédiaire en MIPS et une bibliothèque de fonctions MIPS.

La compilation est encore de nos jours un domaine de recherche en plein essor. De nouvelles contraintes liées à la performance mais aussi à l'efficacité énergétique du code généré donnent lieu à de nouvelles méthodes d'optimisation parfois très complexes.

Cet aspect de la compilation n'a malheureusement pas pu être abordée lors de ce projet. Sans doute de nombreuses améliorations peuvent être apportées à ce compilateur!

Bibliographie

