Compilation

Rapport de Projet

Equipe:
ALLEMAND Fabien
LEBOT Samuel

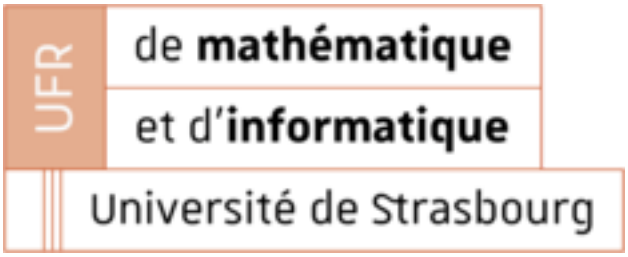

Table des matières

Liste des figures **1**

1 Introduction **2**

2 Analyse Lexicale **3**

3 Analyse Syntaxique **4**

4 Génération de Code Intermédiaire **6**

5 Génération de Code MIPS **7**

6 Conclusion **8**

Bibliographie **9**

Liste des figures

1 Exemple d'action Flex pour le symbole + et le mot clé **if** 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 expr_val dans le fichier Bison 4
5 Actions réalisées pour initialiser un programme lors de l'analyse syntaxique 5

1 Introduction

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 debuggage)
- 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 information à porpager 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'addresse du premier *quadruplet* (code à trois adresses) correspondant à la multiplication et le resultat 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 fichiers 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 un 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 portées. 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 le table des symboles qui ont elles aussi un identifiant invalide pour un utilisateur afin d'éviter tout conflit entre varibles.
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 Intermédiaire

5 Génération de Code MIPS

6 Conclusion

Bibliographie