UV 4.5
Compilation

Neau Guillaume Pierre Martin | SPID CI 2017 | Année 2016

Introduction

Le projet compilateur proposé par Monsieur Le Lann nous amène à proposer notre propre compilateur pour un langage permettant de simuler les instructions machine d'un processeur MIPS-X.

En nous inspirant de la syntaxe assembleur nous avons ainsi crées un langage intuitif générant les instructions machines exécutables.

Par ailleurs le compilateur réécrit le texte sous une forme indenté et génère une page html avec des codes couleurs et les instructions machine en vis-à-vis sur chaque ligne.

Compilateur

Cahier des charges

Le compilateur doit pouvoir compiler les 19 instructions génériques du MIPS-X en binaire.

Le format de compilation doit respecter la nomenclature d'instruction du microprocesseur :

5 bit opcode – 5 bits registre r1 – 1 bit pour le flag – 16 bits de paramètre – 5 bits registre r2

Le compilateur doit permettre un reporting d'erreur efficace pour la programmation

Le compilateur doit pouvoir afficher le code de manière 'jolie', que ce soit en réécrivant le code source une fois celui-ci correctement écrit.

Nous affichons une page synthèse avec des codes couleurs et les instructions machine correspondantes.

Fonctionnement

Le compilateur scan un fichier « entrée» et stock les mots dans une pile qu'il va utiliser. Les séparateurs sont les espaces ainsi que les retours à la ligne.

Les tokènes disponibles :

- Registre
- Entier
- Label
- Balise de début de code
- Balise fin de code
- Séparateur.

Le compilateur repose sur une syntaxe par ligne proche de l'assembleur. Le retour à la ligne joue donc un rôle crucial dans notre langage.

Le fonctionnement du compilateur tel que lisible dans le 'main' est le suivant :

Ouverture du fichier source.

Lexer -> Reporting d'erreur

Si pas d'erreur :

Parser + Analyse lexicale -> Reporting d'erreur

Si pas d'erreur :

Création du code + Prettyprint

En cas d'erreur on affiche les erreurs.

Fonctionnement Lexer

Le lexer lis chacun des mots contenu dans le fichier texte et leurs associe un code les déclarant en tant que tokènes. Puis stock le mot et son code dans une liste chaînée.

En cas d'erreur une autre liste chaînée est instanciée permettant de faire un reporting d'erreur efficace.

Fonctionnement Parser

Le Parser effectue l'analyse syntaxique et contextuelle. Il se base sur l'algorithme d'analyse syntaxique vue en cours.

Son fonctionnement est le suivant :

Expect(balise de début){

Pointe sur un mot de début de ligne

Expect syntaxte(mot)

Expect(balise de fin) }

Il effectue un reporting d'erreur en fin de traitement.

Un exemple de fonction expect est fournis en annexe.

Création du code

L'algorithme de création des codes suivant :

- HTML pretty
- Sources pretty
- Binaires

Est fondé sur le même que l'analyse syntaxique, on va pointer sur chaque début de ligne et utiliser des fonctions expect(mot) permettant de créer les codes.

Le html utilise un Template css afin de faire la mise en page.

Syntaxe générale (BNF)

Nous avons souhaité coller au maximum aux instructions tel que décrit dans la datasheet du MIPS-X.

1. Début de programme

   @Mips

2. Syntaxe générique

   <Opérande> <espace><Arg1><,><Arg2><,><Arg3>

   Exemple : add r1,0,r2

3. Syntaxte label

   <Label+' :'>

Résultats

Nous avons ainsi crée un nouveau langage très proche de l'assembleur et le plus intuitif possible permettant de générer des instructions pour un processeur MIPS-X. Nous avons, pour vérifier le bon fonctionnement de notre compilateur, testé différents jeux d'instruction : différents jeux de load/add/sub/mult et un programme plus complet et complexe : facto.txt.

Nous avons testé le reporting d'erreur pour les plus grossières et critique, celui-ci est fonctionnel.

Nous avons réussis à créer un langage de programmation et un compilateur parfaitement fonctionnel.

Annexe

Facto.txt

```
add 10,r1

add r1,0,r2


fact:

sub   r2,1,r2

braz r2,end:

mult r2,r1,r1

jmp   fact:


end:

stop
```

En vert : Opérande

En violet : Label

En noir : Arguments entier et registre.

En dorée : la virgule, séparateur d'arguments

Extrait de codes

```
if(ExpectDebutligne(tab->Codeword,ERR)==0){

    switch (tab->Codeword){
        /* Operande */
        case 2:
            ExpectSyntOperande(tab,ERR);
        break;

        /* Label */
        case 3:
            ExpectSyntLabel(tab,ERR);
        break;
    }
}
```

Figure 1_Analyse syntaxique premier niveau

```
if(classOperande>0 && classOperande<=13){

    cpt=CreateSeq(tab,seq);

    if(cpt>5){
        fprintf(logfichier,"Syntax error : Line overflow @%s.\n",tab->mot);
        sprintf(ERR->mot,"Syntax error : Line overflow @%s.",tab->mot);
        ERR->suivant=malloc(sizeof(Tableau));
        ERR=ERR->suivant;
    }

    if(cpt!=3 && cpt!=5){

        fprintf(logfichier,"Syntax error : wrong construction of line @%s.\n",tab->mot);
        sprintf(ERR->mot,"Syntax error : wrong construction of line @%s.",tab->mot);
        ERR->suivant=malloc(sizeof(Tableau));
        ERR=ERR->suivant;
    }

    if(ExpectSequence14(seq)==1){
        fprintf(logfichier,"Syntax error : wrong use of operande @%s.\n",tab->mot);
        sprintf(ERR->mot,"Syntax error : wrong use of operande @%s.",tab->mot);
        ERR->suivant=malloc(sizeof(Tableau));
        ERR=ERR->suivant;
    }

}
```

Figure 2_Vérification de la syntaxte pour des operandes simples