Rapport du projet de système d'exploitation

9 avril 2011

Encadrant : Martin Yannick

Elèves : Lissy BARRO , Nada, Yassin, Python, Houda

ENSEIRB-MATMECA i2 2010/2011 semestre n°4

1 Introduction

Le projet consiste à réaliser un compilateur rudimentaire à l'aide des outils de génération d'analyseurs lexicaux et syntaxiques que sont *LEX* et *Yacc*. Ce compilateur devra permettre de passer d'un code source conforme à une grammaire que nous fournissons en annexe à du code3@[1].

Dans ce rapport, nous détaillons les 4 principales phases de notre travail :

- ✦ L'analyse du problème : contraintes et buts qui ressortent du cahier de charge.
- ✦ Les améliorations apportées : fonctionnalités qu'on apu apporter en plus de celles definies par le cahier de charge.
- ✦ La phase de conception qui consistera à parler des structures et des techniques mises en oeuvre dans la réalisation du compilateur.
- ✦ Enfin nous fournissons des exemples de code 3@ obtenus à partir d'un ensemble de codes sources pour simuler le comportement du compilateur réalisé.

2 Analyse du problème

Nous définissons ici les contraintes imposées par le cahier de charge ainsi que le but recherché :

2.1 Contraintes

- ✦ Un typage minimum : le type **int** pour les entiers, **float** pour les flottants, **bool** pour les booleens obeïssant à la hierarchie **bool** < **int** < **float**.
- ✦ Un mécanisme de déclarations locales avec prise en compte du masquage et de la portée des variables.
- ✦ Gestion des structures de contrôle à savoir les conditions et les boucles (`if, while, repeat`).

2.2 But

- ✦ Un code cible3@.
- ✦ Les deĉlarations de variables et de registres utilisés doivent être en tête de code ensuite viennent les instructions.
- ✦ Minimiser le nombre de registres utilisés.

3 Améliorations apportées

- ✦ Ajout de pointeurs sur les trois types avec la possibilité d'être double ou plus. Dans l'exemple d'un entier on peut avoir : **int***, **int****, **int***** ··· Mais pour des déclarations de la sorte **int* x,y ;** ou **int *x,y ;** seul **x** est consideré comme pointeur sur entier et **y** un entier. Pour que **y** soit aussi consideré comme un pointeur, écrire plutôt **int* x,*y** ou **int *x,*y**.
- ✦ Gestion des commentaires par le compilateur. Dans un fichier source on pourrait mettre des commentaires qui ne seront pas pris en compte dans la génération du code3@.
- ✦ Gestion d'erreurs avec renvoie d'un message et du numero de bloc de l'erreur. Le bloc 0 est le bloc principal.

[1] code3@ : code à trois adress, les operations dans le code demandes au plus trois adresses.

4 La phase de conception

4.1 Structuration du code du compilateur

Pour réaliser notre compilateur nous avons plutôt adopté une approche modulaire comme le decrit ci-dessous l'enumeration des fichiers sources du compilateur.

- `blocLabel.c` : ensemble des fonctions dont nous nous servons pour numeŕoter et compter les blocs et les labels du code entrant.
- `typage.c` : Module comprenant des opérations élémentaires sur les types, les fonctions de typage des expressions et des pointeurs ainsi que celles des conversions.
- `stack.c`, `HashTable.c`, `Table_des_chaines.c` et `Table_des_symboles.c` : ces modules servent à l'implémentation de notre table des symboles.
- `registres.c` : regroupe l'ensemble des fonctions se rapportant aux registres.
- `erreur.c` : regroupe l'ensemble des fonctions servant à l'affichage d'erreurs ou d'avertissements.

4.2 Techniques mis en en oeuvre pour l'analyse sémantique

Dans un code source les contenues des variables sont continuellement manipulés. On a donc besoin de les stocker avec toutes les informations necessaires à la réalisation du code3@ et de pouvoir les mettre à jour au besoin. Pour ce faire on a utilisé une table des symboles[2]. A chaque niveau du code source, la table contient les symboles déclarés dont on pourrait avoir besoin dans la generation du code3@. Les symboles déclarés non susceptibles d'être utilisées sont supprimés de la table.

4.2.1 Implémentation de la table des symboles

Pour implementer la table des symboles nous avons utilisé une table de hachage dynamique de sorte à ne pas limiter le nombre de blocs dans un code source. Chaque ligne de la table correspond à un bloc et contient les eventuels eventuelles declarées dans ce bloc. La table de hachage est en quelque sorte une pile dynamique contenant des piles dynamiques. Pour ce faire nous sommes partis d'une pile generique dynamique implementée avec un tableau dont on augmente la taille avec la fonction `realloc` au besoin et on a ajouté une fonction supplementaire qui permet d'acceder en temps constant à un element de la pile connaissant son indice.
Ce choix de la table de hachage va beaucoup aidé dans le stockage, suppression des symboles et la mise à jour du contenu des variables de même que dans la manipulation des blocs autrement dit dans le masquage et la portée des variables.

4.2.2 stockage, suppression et mis à jour du contenu des variables

Seuls les symbole déclarés sont stockés et ceci est garanti si on fait le stockage qu'au niveau de la règle de grammaire suivante :

```
1 id_aff : id | id EQ const ;
```

Et les mises à jour ne se font qu'après une affection. On obéit à cette règle si on fait la mise à jour qu'au niveau de la règle de grammaire suivante :

```
1 affect : id EQ exp ;
```

Les fonctions de stockage et de mise à jour sont respectivement : `new_symb_value` et `update_symb_value`.
Dans le code source à chaque fois qu'on rencontre le caractère "**{**" on definit un nouveau bloc, notre table grandit d'une ligne. Et on supprime la derniere ligne de notre table au niveau de la règle :

```
1 inst : bloc ;
```

Ainsi on supprime toutes les variables qui ne doivent plus être utilisées.

4.2.3 Masquage et portée des variables

Pour un nom de variable donné, on utilise toujours le symbole correspondant[3] ayant la déclaration la plus récente. Vue qu'à chaque déclaration on stocke le symbole correspondant et qu'à chaque fois qu'on sort d'un bloc on supprime de la tables tous les symboles rélatifs aux variables ayant étée déclarées dans ce bloc et en considerant le fait qu'on cherche un symbole correspondant à un nom de variable en parcourant la table de son dernier élément vers son premier élément ceci assure à la fois le masquage et la portée des variables. Pour mettre

[2]Nous appellons symbole, une variable et son contenu.
[3]Un symbole correspondant est un symbole dans la table ayant le même nom que la variable

en evidence ces deux phénomènes de masquage et de portée des variables lors de la génération du code3@ on renome les variables en leur nom concatené au numéro du bloc dans lequel elles ont été déclarées. On accède à toutes ces information sur la variable via la structure d'un symbole.

4.2.4 Représentation d'un symbole

La valeur d'un symbole correspond à la structure `symb_value` suivante :

```
typedef enum type {    |   struct symb_value {
  my_INT,              |      type type;   //correspond au type de la variable
  my_FLOAT,            |      int bloc;    //le numero de bloc de declaration
  my_BOOL,             |      int pointeur; //type de pointeur
  STAR_INT,            |   }symb_value;
  STAR_FLOAT,          |
  STAR_BOOL,           |   pointeur = 0         ; pour les variables non pointeur
}type;                 |   pourteur = 1,2 ou plus ; pour les pointeurs simples, doubles ou plus
```

Autrement dit dans la generation du code3@ on a juste besoin comme informations sur la variable, sont type, son numéro de bloc de déclaration et si c'est un pointeur, de son ordre. Le symbole est répresenté par la structure `elem` que voici :

```
struct elem {
  sid symbol_name;              //le nom de la variable que le symbole represente
  symb_value_type symbol_value; //la valeur associee a la variable
};
```

Et nous avons défini comme suit le type des attributs en conformité avec la représentation des symboles :

```
typedef union YYSTYPE {
  sid sid_val;               //pour les attributs correspondant a un nom de variable
  type type;                  //pour les attributs correspondant a un type
  float float_val;           //pour les attributs correspondant a un flottant
  int int_val;               //pour les attributs correspondant a un entier
  struct info_complex{
    sid name;
    symb_value_type val;
  }complex;                   //pour les attributs correspondant a un symbole
} YYSTYPE;
```

4.2.5 Gestion des registres

Nous avons défini pour chaque type une variable globable entière correspondant au numero de registre qu'on incremente à chaque fois qu'on utilise un registre et qu'on décremente à chaque fois qu'on libère un registre.

- **getRegistre** : Prend en parametre un type et renvoie une chaine de caractère de la forme `R_fn`, `R_in` ou `R_bn` pour respectivement un type `float`, `int` ou `bool` où n est le numéro de registre correspondant et incremente ensuite le numero de registre correspondant. Dans le cas d'un pointeur le caractère `R` dans la chaîne est remplacé par le caractère `Pm` où m est l'ordre du pointeur.
- **freeRegistre** : prend en paramètre un type on décrémente le numéro de registre correspondant.

Pour les registres des types `float`, `int` et `bool` nous avons une optimisation parfaite du nombre de registre mais nous optimisons pas le nombre de registre pour les pointeurs.

4.2.6 Typage des expressions arithmétiques

Pour ce qui est des opérations entre types simples que sont `float`, `int` et `bool`, le résultat est typé avec le type le plus restrictif en respectant la hiérarchie `float` < `int` < `bool`. Pour réaliser cela on a donner un type `complex`[4] aux attributs des expressions arithmétiques permettant de faire remonter les informations dont on a besoin pour générer le code3@ lors des actions sémentiques.

- Pour les opérateurs logiques `AND`, `OR` et `NOT` : Les membres de l'opération doivent être de type booléen sinon une conversion s'en suit et le resultat de l'operation est de type booléen.
- Pour les opérateurs de comparaisons : Les membres de l'opération doivent être de même type sinon une conversion en le type le plus grand selon la hiérarchie `float` < `int` < `bool` s'en suit et le resultat de l'operation est de type booléen.

[4]complex est une structure identique à celle représentant les symboles

- ✦ Pour les autres opérateurs : le traitement est le même que precedemment mais le type du résultat est le type le plus grand des membre de l'opération.
- ✦ Pour les pointeurs, le traitement est un peu différent : Si l'un des membres est un pointeur alors l'autre est necessairement soit un pointeur ou un entier. Si le deuxième membre est un entier alors le resulat est du type du pointeur, sinon c'est à dire si les deux membres sont de type pointeur, le traitement est le même que pour les non-pointeurs et selon la même hiérarchie de type.

4.2.7 Les structures de contrôles

Les conditions

IF exp THEN inst En fonction des règles utilisées dans la construction de l'arbre pendant l'analyse sémentique nous effectuons des actions sementiques consistant à la génération du code3@.

```
a-    GENERATION DE Code3@ ====> On execute le code de exp                | code de exp;
    - la valeur est dans un un registre.                                   | if !$2.name goto $$;
b- if_exp_then : IF exp THEN;                                              | code de inst
    - Si $2 n'est pas de type booleen alors il est caster en bouleen.      | $1:
    - On definit un label qu'on stocke dans $$ {$$ = label;}.              ----------------------------
    - GENERATION DE Code3@ ====> if !$2.name goto $$;
    - On libere le registre contenant la valeur de l'expression.
c-    GENERATION DE Code3@ ====> On execute le code de inst
d- if_exp_then_inst : if_exp_then inst;
    - On fait remonter le label cree precedemment {$$ = $1;}
e- cond : if_exp_then_inst
    - GENERATION DE Code3@ ====> $1: (on affiche le label qu'on a fait remonter).
```

IF exp THEN inst1 ELSE inst2

```
a-    GENERATION DE Code3@ ====> On execute le code de exp                | code de exp;
    - la valeur est dans un un registre.                                   | if !$2.name goto $$;
b- if_exp_then : IF exp THEN;                                              | code de inst2
    - Si $2 n'est pas de type booleen alors il est caster en bouleen.      | goto $2;
    - On definit un label qu'on stocke dans $$ {$$ = getLabel();}.         | $1: code de inst2
    - GENERATION DE Code3@ ====> if !$2.name goto $$;                      | $1:
    - On libere le registre contenant la valeur de l'expression.          ----------------------------
d-    GENERATION DE Code3@ ====> On execute le code de inst1
e- if_exp_then_inst : if_exp_then inst;
    - On fait remonter le label cree precedenment {$$ = $1;}
f- else : ELSE
    - On definit un nouveau label qu'on stocke dans $$ {$$ = getLabel();}
g- if_exp_then_inst_else : if_exp_then_inst else
    - GENERATION DE Code3@ ====> goto $2: ($2 correspond au deuxieme label cree).
    - GENERATION DE Code3@ ====> $1: (on affiche le premier label cree).
    - On fait remonter le deuxieme label cree: {$$ = $2}
h-    GENERATION DE Code3@ ====> On execute le code de inst2
i- cond : if_exp_then_inst_else inst
    - GENERATION DE Code3@ ====> $1: (on affiche le deuxieme label qu'on a fait remonter).
```

Les boucles

WHILE exp DO inst

```
a- while : WHILE;                                                          | $$ : code de exp
   - On definit un label qu'on stocke dans $$ et on l'affiche{$$ = getLabel();} | if !$0.name goto $$;
   - GENERATION DE Code3@ ====> $$: (on affiche le label qu'on vient de creer). | code de inst
b-   GENERATION DE Code3@ ====> On execute le code de exp                  | goto $1;
c- do : DO;                                                                | $3:
    - Si $0 n'est pas de type booleen alors il est caster en bouleen.      ----------------------
    - On definit un label qu'on stocke dans $$ {$$ = getLabel();}.
    - GENERATION DE Code3@ ====> if !$0.name goto $$;
d-    GENERATION DE Code3@ ====> On execute le code de inst
e- loop : while exp do inst;
    - GENERATION DE Code3@ ====> goto $1; ($1 correspond au premiere label cree).
    - GENERATION DE Code3@ ====> $3: (on affiche le deuxieme label).
```

REPEAT inst UNTIL exp

```
a- repeat : REPEAT                                                              | $$: code de inst
   - On definit un label qu'on stocke dans $$ et on l'affiche{$$ = getLabel();} | code de exp
   - GENERATION DE Code3@ ====> $$: (on affiche le label qu'on vient de creer). | if !$4.name goto $1;
b-   GENERATION DE Code3@ ====> On execute le code de inst                      ----------------------
c-   GENERATION DE Code3@ ====> On execute le code de exp
b- loop : repeat inst UNTIL exp;
   - Si $4 n'est pas de type booleen alors il est caster en bouleen.
   - GENERATION DE Code3@ ====> if !$4.name goto $1; ($1 correspond au seul label cree et qu'on a fait
       remonter)
```

Des exemples

```
bool a;              |  bool a;           |  bool a;             | bool a;
if(a) then a = false |  while (a) do      |  repeat a = true     | if(a) then {
else a = true        |    a = false       |  until(a == true)    |    if(a) then {a=false;}}
---------------------|--------------------|----------------------|---------------------------
bool a_0;            |  bool a_0;         |  bool a_0;           | bool a_0;
bool R_b1;           |  bool R_b1;        |  bool R_b1;          | bool R_b1;
R_b1 = a_0;          |  l0:               |  bool R_b2;          | R_b1 = a_0;
a_0 = R_b1;          |  R_b1 = a_0;       |  l0:                 | if !R_b1 goto l0;
if !R_b1 goto l0;    |  if !R_b1 goto l1; |  R_b1 = true;        | R_b1 = a_0;
R_b1 = false;        |  R_b1 = false;     |  a_0 = R_b1;         | if !R_b1 goto l1;
a_0 = R_b1;          |  a_0 = R_b1;       |  R_b1 = a_0;         | R_b1 = false;
goto l1;             |  goto l0;          |  R_b2 = false;       | a_0 = R_b1;
l0:                  |  l1:               |  R_b1 = R_b1 == R_b2 | l1:
R_b1 = true;         |                    |  if !R_b1 goto l0;   | l0:
a_0 = R_b1;          |                    |                      |
l1:                  |                    |                      |
nop                  |                    |                      |
```

4.2.8 Gestion d'erreurs

En plus des erreurs liées au non respect de la grammaire, notre compilateur détecte et signale d'autres erreurs :

Variable non déclarée : lorsqu'on ne retrouve pas dans la table un symbole correspondant à un nom de variable. Ceci s'observe lorsqu'on utilise une variable non déclarée ou une variable declarée dans un bloc profond dans un bloc supérieur ou dans un autre bloc qui n'est ni un bloc superieur ou un bloc plus profond. La compilation s'arrête.

```
int a; a = c;        |   int a_0;
                     |   error: c is not declared
                     |   Generation de code3@ interrompue!!
-------------------------------------------------------------
int a;               |   int a_0; int c_1;
{int c; c = 4;};     |   error: c is not declared
a = c;               |   Generation de code3@ interrompue!!
-------------------------------------------------------------
int a;               |   int a_0; int c_1; int d_1;
{int c; c = 4;};     |   error: c is not declared
{int d; d = c;};     |   Generation de code3@ interrompue!!
```

Redéclaration d'une variable : en stockant un symbole on verifie dans la ligne correspondant au bloc courant si un autre symbole ne porte pas le même nom au quel cas on envoie un message de rédeclaration. La compilation continue et seule la dernière déclaration est prise en compte.

```
float a;             |  float a_0; int a_0;
int a;               |  Warning: a is redeclared twice
a = 7.6;             |  float R_f1;  int R_i1;
                     |  R_f1 = 7.6;  R_i1 = (int)R_f1;
                     |  a_0 = R_i1;
```

Expression mal typée : par exemple si dans "`if exp then inst`" exp n'est pas booléen un message d'avertissement est envoyé et une conversion s'en suit automatiquement mais la compilation continue.

```
float a;              |  float a_0;
if a then a = 1.;     |  float R_f1; bool R_b1;
                      |  R_f1 = a_0;
                      |  Warning: R_f1 non booleenne
                      |  R_b1 = (bool)R_f1;
                      |  if !R_b1 goto l0;
                      |  R_f0 = 1; a_0 = R_f0;
                      |  l0:
```

Incompatibilité des types pendant l'affectation : Situation qui s'observe si on affecte à une variable simple un pointeur ou à un pointeur un pointeur d'ordre differente. La compilation compilation continue.

```
int *a,b;             |  int *a_0; int b_0;
                      |  error: typage : affectation
                      |  Generation de code3@ interrompue!!
```

4.2.9 Gestion des pointeurs

L'ajout d'information suplementaire `int pointeur`[5] dans la structure representant les symboles nous a permis de gerer les pointeur en tenant compte de leur ordre.

```
--------Code source----|----------------------Code3@---------------------------
int ***a;              |   int ***a_0;   |  int R_i1;        |  c_0 = R_i1;
int **b;               |   int **b_0;    |  int** P2_i3;     |  P3_i2 = a_0;
int c;                 |   int c_0;      |  P2_i1 = b_0;     |  P2_i3 = *P3_i2
c = **b;               |   int** P2_i1;  |  P1_i2 = *P2_i1;  | b_0 = P2_i3;
b = *a;                |   int* P1_i2;   |  R_i1 = *P1_i2    |
```

4.2.10 Gestions des commentaires

On a ajouté dans le fichier `projet.l` pour l'analyse lexicale des instructions supplémentaires permettant d'inserer des commentaires comme dans l'exemple ci-dessous :

```
--------Code source-----------|-------------------Code3@--------------------
int x;                        |           int x_0;
//bool y; est en commentaire  |           int R_i1;
/* {int i;                    |           R_i1 = 3;
   i = 9;}                    |           x_0 = R_i1;
ce bloc est en commentaire    |
*/                            |
x = 3;                        |
```

5 Conclusion

Ce projet nous a permis de mieux comprendre le fonctionnement d'un compilateur. Nous avons rencontré des difficultés mais notre compilateur répond finalement au cahier de charge et nous avons ajouté d'autres fonctionnalités comme la gestion des pointeurs et la prise en compte des commentaitres dans un code source. Cependant pour ce qui est de l'arithmetique des pointeurs nous ne l'avons pas poussé assez loin par manque de temps. Ce que nous retenons de ce projet est l'ensemble des petites subtilités, notamment le fontionnement de la pile d'arbre lors des actions sémantiques qui forcent ainsi notre compréhension du problème.

[5] pointeur = 0 pour les types simples et vaut l'ordre du pointeur pour les pointeurs.